EAI/Springer Innovations in Communication and Computing

Antonio Virdis
Michael Kirsche *Editors*

# Recent Advances in Network Simulation

## The OMNeT++ Environment and its Ecosystem

EAI
RESEARCH MEETS INNOVATION

Springer

# EAI/Springer Innovations in Communication and Computing

**Series editor**
Imrich Chlamtac, European Alliance for Innovation, Gent, Belgium

## Editor's Note

The impact of information technologies is creating a new world yet not fully understood. The extent and speed of economic, life style and social changes already perceived in everyday life is hard to estimate without understanding the technological driving forces behind it. This series presents contributed volumes featuring the latest research and development in the various information engineering technologies that play a key role in this process.

The range of topics, focusing primarily on communications and computing engineering include, but are not limited to, wireless networks; mobile communication; design and learning; gaming; interaction; e-health and pervasive healthcare; energy management; smart grids; internet of things; cognitive radio networks; computation; cloud computing; ubiquitous connectivity, and in mode general smart living, smart cities, Internet of Things and more. The series publishes a combination of expanded papers selected from hosted and sponsored European Alliance for Innovation (EAI) conferences that present cutting edge, global research as well as provide new perspectives on traditional related engineering fields. This content, complemented with open calls for contribution of book titles and individual chapters, together maintain Springer's and EAI's high standards of academic excellence. The audience for the books consists of researchers, industry professionals, advanced level students as well as practitioners in related fields of activity include information and communication specialists, security experts, economists, urban planners, doctors, and in general representatives in all those walks of life affected ad contributing to the information revolution.

## About EAI

EAI is a grassroots member organization initiated through cooperation between businesses, public, private and government organizations to address the global challenges of Europe's future competitiveness and link the European Research community with its counterparts around the globe. EAI reaches out to hundreds of thousands of individual subscribers on all continents and collaborates with an institutional member base including Fortune 500 companies, government organizations, and educational institutions, provide a free research and innovation platform.

Through its open free membership model EAI promotes a new research and innovation culture based on collaboration, connectivity and recognition of excellence by community.

More information about this series at http://www.springer.com/series/15427

Antonio Virdis • Michael Kirsche
Editors

# Recent Advances in Network Simulation

## The OMNeT++ Environment and its Ecosystem

Springer

EAI
RESEARCH MEETS INNOVATION

*Editors*
Antonio Virdis
University of Pisa
Pisa, Italy

Michael Kirsche
Brandenburg University of Technology
Cottbus-Senftenberg
Cottbus, Brandenburg, Germany

ISSN 2522-8595 ISSN 2522-8609 (electronic)
EAI/Springer Innovations in Communication and Computing
ISBN 978-3-030-12841-8 ISBN 978-3-030-12842-5 (eBook)
https://doi.org/10.1007/978-3-030-12842-5



This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

# Foreword

I see a simulator has two main functions. A simulator is first of all a convenient tool for assessing performance and capabilities of a target system—whether the system already exists, partially exists, or is still entirely on a design board. A simulator can be used to help understand the operation of the system and obtain performance-related results (such as throughput, latency, stability, robustness, and scalability), potentially with different design alternatives, under different configurations, in different operating conditions and runtime scenarios.

Another function of the simulator is to provide a common platform for designing and studying the target system. Most complex engineering systems contain many moving parts. It would be unrealistic to study such systems by focusing on only one specific part in isolation. The software components of a network system, for example, are organized into protocols and services, more or less in a hierarchical fashion. Encapsulation allows the upper-layer protocols or applications to operate using the functions and services of lower-layer protocols. A network system should also be seen as an ecosystem: a new protocol must operate in real network traffic situations where multiple applications and protocols coexist and interoperate. In such cases, a simulator can provide a common platform for developing and testing models for various components of the target system. By doing so, one can focus on a specific component of the system (e.g., a protocol in a network simulation) and rely on existing models built by other developers for the rest of the system. A robust community can thus be built on such premise, and modeling becomes a community effort. As a common platform, a simulator should not only provide methods for composing models of sorts but also support code verification and model validation by offering standard test harnesses, common use scenarios, and possibly additional capabilities, such as graphical user interface and visualization.

The first successful open-source network simulator with a broad community support is the one with the eponym *network simulator (ns)*, mostly known by its second version *Network Simulator 2 (ns-2)*. *ns* was developed in the mid-1990s and has been used predominantly in several research areas, such as Transmission Control Protocol (TCP) congestion control and wireless ad hoc routing. *ns* has

brought tremendous impact to network research, although many also saw potential alternatives in the simulator's software design in order to improve usability and performance.

OMNeT++, which stands for Open Modular Network Testbed in C++, is an outstanding example of such alternatives. As its name suggests, OMNeT++ features a modular design that separates the discrete-event simulation engine (aka the simulation library or framework) from the network models built on top of it. Discrete-event simulation refers to the technique of modeling systems as a series of events at discrete time instances that represent the potential change to the system's state. In addition to supporting discrete-event simulation, OMNeT++ provides an Eclipse-based Integrated Development Environment (IDE) and additional tools for model configuration, data analysis, and visualization. Network models built on top of the OMNeT++ framework represent detailed network operations, including sending and receiving data between simulated entities (hosts, servers, routers, switches, mobile stations, and so on) over either wired or wireless transmission media. OMNeT++ consists of many network models, including various protocols for the traditional TCP/Internet Protocol (IP) network as well as other types of networks, including peer-to-peer/overlay networks (e.g., BitTorrent), cellular networks (e.g., Long Term Evolution (LTE)), vehicular networks, satellite networks, Mobile Ad Hoc NETworks (MANETs), storage networks in parallel/distributed file systems, and so on.

The development effort of OMNeT++ dated back in the early 2000s, and the simulator quickly became a very successful one. One can measure the success of an open network simulator in three interdependent aspects. First, the simulator needs to have a broad user base. As such, one can find models developed by various users as the building blocks for implementing, testing, and validating new network research. Second, the simulator needs to have many active projects with ongoing development efforts. Significant activities can ensure that simulator be constantly maintained and documented with continued support. Last, the simulator needs to support experimentation and generate results in research publications. Standard modules and common test scenarios can help cross-validation of and ensure reproducibility for published results. By all measures, I see that OMNeT++ is a successful network simulator. As a case in point, the simulator has so far generated about 500 research papers according to Google Scholar. It is certainly indicative to the level of success the simulator has achieved. There is an active research community at work.

This book provides a comprehensive introduction to the OMNeT++ simulation environment and its network models. It is in my view a must-have for those who need to use OMNeT++ for their daily research. Especially to the newcomers of OMNeT++, I would like to welcome them to the ever-growing community of users and developers. I hope they will enjoy the ride on the success of the simulator. As a part of a vibrant community, I hope everyone will be able to make his or her contributions.

Florida International University, Miami, FL, USA Jason Liu
19 June 2018

# Preface

This book testifies the joint efforts of the OMNeT++ community in continuously extending and improving the OMNeT++ ecosystem of frameworks and simulation modules. It finds its roots within the *OMNeT++ Community Summit* events, where OMNeT++ researchers and developers meet annually to present their results and discuss the latest developments in the field. The first summit took place in 2014, replacing the conference co-located workshop format of OMNeT++ events initiated in 2008.

OMNeT++—the Open Modular Network Testbed in C++—provides a versatile environment to conduct all kinds of simulative experiments that follow the Discrete Event-based Simulation (DES) principle. The number and heterogeneity of research works and simulation tools that are publicly available for OMNeT++ are large enough to be seen as a *Simulation Ecosystem*, i.e., a set of connected yet independent software projects, all sharing the same working ground, the OMNeT++ engine, which allows them to coexist and to be potentially interoperable. This offers a great potential to researchers who can build complex simulation scenarios just by tapping into this simulation ecosystem.

So far, the only ways to discover these independent projects were to either dig into the various forums and websites covering the frameworks, models, and topics or to follow the OMNeT++ community initiatives. This book aims to provide researchers and users with a detailed overview of the OMNeT++ ecosystem, describing both large frameworks and smaller building blocks, which contribute to the same simulation environment. Each book chapter offers the reader a tutorial-centric presentation of a specific simulation topic, still formally presenting the overall research scenario but focusing on the practical aspects that will reduce the time to simulation. Moreover, each chapter will be connected to a website, providing links to the project code, additional examples, and tutorial and teaching material that will significantly ease the learning process.

This book is divided into three distinct parts. The first part introduces the OMNeT++ simulation framework, accompanied by a number of practical examples. The basics of OMNeT++ simulations and special topics are introduced here. The second part of the book presents the OMNeT++ ecosystem by covering the

major extension frameworks being used and actively maintained today. Six individual chapters cover the frameworks INET, INETMANET, SimuLTE, RINASim, SimuLTE, Veins, and SEA++ . These frameworks range from generic ones like INET, which includes simulation models for the TCP/IP communication stack, to application-specific ones like SimuLTE, which includes simulation models for LTE communication use cases. Each framework chapter introduces the individual framework and gives background information as well as details for practical usage. The third book part collects eight different chapters that introduce novel research topics and recent developments actively pursued by the OMNeT++ community. Chapter 8 discusses the topic of reproducibility of OMNeT++ simulations. Chapter 9 introduces a remote interface to control OMNeT++ simulations and modify simulation parameters during runtime. In Chap. 10, simulation models for in-vehicular networks are introduced along with a domain-specific description language to design automotive networks. The topic of simulating vehicular mobility is covered in Chap. 11 with an introduction of the LIMoSim framework. Vehicular communication is also the topic in Chap. 12 where Artery is introduced, a framework that covers vehicular communication that complies to European specifications. A combination of said Artery framework with the SimuLTE framework (introduced in Chap. 5) is discussed by the authors of Chap. 13 to combine vehicular communication simulation with LTE communication simulation. Chapter 14 discusses the simulation of Opportunistic Networks (OppNets) and, in particular, the topics of data dissemination models and evaluation metrics for OppNets. Chapter 15 presents a simulation model for the Deterministic and Synchronous Multi-Channel Extension (DSME), a Medium Access Control (MAC) protocol extension for the popular IEEE 802.15.4 standard that is used in resource-constrained wireless networks. The wide range of topics in this third part of the book reflects the versatility of OMNeT++ as a simulation tool.

We want to thank a number of persons that helped with the creation of this book: first and foremost, we thank all the authors of the 15 individual chapters who provided their ideas and input, their expertise, and their hard work, both in writing and in reviewing, to fill this book with life. We would also like to thank all the researchers involved in the OMNeT++ community, who contribute to the ecosystem by sharing their ideas and insights and taking part in the community events. Finally, we want to say thank you to our families for their patience during the writing and editing process when free time was often scarce.

Pisa, Italy — Antonio Virdis
Cottbus, Germany — Michael Kirsche
October 2018

# Typographic and Stylistic Conventions

The following list of typographic conventions apply to the formatting of text and the use of markups throughout this book.

| Characteristic | Definition/Meaning | Examples |
|---|---|---|
| `TypeWriter` | A `fixed width font` is used in source code listings and for all source-code-related markups (e.g., to mark function or class names, constants, variables, and likewise) | "The class `TrafficGen` is" "Use `copy(&msg, bool)`" |
| *Cursive* | *Cursive markups* are used to emphasize all non-source-code-related terms (e.g., scientific terms, special topics, adjectives) | "The *service primitive* is" |
| **Program** | Words written in **boldface** indicate individual computer programs or product names | "The **opp_run** program" |
| *file.xyz* | File names use a *slanted italic* font | "The *omnetpp.ini* file" |
| [*KeyName*] | Cursive names in squared brackets represent keys on the keyboard | "Press [*Ctrl*] or [*Alt*]" |
| [*Key1*]+[*Key2*] | Keys combined with a "+" sign are meant to be pressed simultaneously | "Press [*Ctrl*]+[*Shift*]+[*W*]" |
| *Term*→*Term* | *Cursive terms* in combination with arrows (→) mark command sequences in user interface elements, dialogs, pop-up, or pull-down menus | "*Menu*→*File*→*Save As...*" |
| . . . | Three dots inside tables and source code listings indicate left out or abbreviated parts | |
| (✉) | A letter symbol marks the corresponding author | |

# Support Material

Support material (e.g., links to project websites, model and framework source code, additional examples, tutorials, teaching material) for each chapter is available online at the following companion website:

https://omnetpp.github.io/omnetpp-ecosystem-book

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| 6TiSCH | Internet Protocol Version 6 over the Time-Slotted Channel Hopping mode of IEEE 802.15.4e |
| ACC | Adaptive Cruise Control |
| ACK | Acknowledgment |
| ACT | Allocation Counter Table |
| ADAS | Advanced Driver Assistance System |
| ADSL | Asymmetric Digital Subscriber Line |
| AE | Application Entity |
| AEI | Application Entity Instance |
| AEI-id | Application Entity Instance Identifier |
| AEN | Application Entity Name |
| AFDX | Avionics Full-Duplex Switched Ethernet |
| AM | Acknowledged Mode |
| AMC | Adaptive Modulation and Coding |
| ANDL | Abstract Network Description Language |
| ANI | Application Naming Information |
| ANSA | Automated Network Simulation and Analysis |
| AODV | Ad Hoc On-Demand Distance Vector |
| AP | Application Process |
| API | Application Programming Interface |
| API-id | Application Process Instance Identifier |
| APN | Application Process Name |
| ARP | Address Resolution Protocol |
| ARQ | Automatic Repeat-reQuest |

| | |
|---|---|
| ASE | Attack Simulation Engine |
| ASI | Attack Specification Interpreter |
| ASL | Attack Specification Language |
| ASN.1 | Abstract Syntax Notation One |
| AVB | Audio Video Bridging |
| BAG | Bandwidth Allocation Gap |
| BATMAN | Better Approach To Mobile Adhoc Networking |
| BDD | Behavior-Driven Development |
| BER | Bit Error Rate |
| BGP | Border Gateway Protocol |
| BLE | Bluetooth Low Energy |
| BLER | Block Error Rate |
| BMBF | German Federal Ministry of Education and Research |
| BO | Beacon Order |
| BPSK | Binary Phase Shift Keying |
| BSD | Berkeley Software Distribution |
| BSR | Buffer Status Report |
| BSS | Basic Service Set |
| BTP | Basic Transport Protocol |
| C2C | Car-to-Car |
| C2I | Car-to-Infrastructure |
| C-V2X | Cellular Vehicle-to-Everything (V2X) |
| CA | Cooperative Awareness |
| CACC | Cooperative Adaptive Cruise Control |
| CACE | Common Application Connection Establishment |
| CAM | Cooperative Awareness (CA) Message |
| CAN | Controller Area Network |
| CAP | Contention Access Period |
| CBR | Constant Bit Rate |
| CBS | Credit-Based Shaper |
| CC | Cruise Control |
| CCA | Clear Channel Assessment |
| CDAP | Common Distributed Application Protocol |
| CDP | Cisco Discovery Protocol |
| CEP-id | Connection Endpoint Identifier |

| | |
|---|---|
| CFP | Contention-Free Period |
| CLI | Command Line Interface |
| CoAP | Constrained Application Protocol |
| CoMP | Coordinated MultiPoint |
| CoMP-CS | Coordinated MultiPoint (CoMP) Coordinated Scheduling |
| CoMP-JT | CoMP Joint Transmission |
| CoRE4INET | Communication over Real-time Ethernet for INET |
| COTS | Commercial Off-The-Shelf |
| CPU | Central Processing Unit |
| CQI | Channel Quality Indicator |
| CRC | Cyclic Redundancy Check |
| CSMA | Carrier Sense Multiple Access |
| CSMA/CA | Carrier Sense Multiple Access with Collision Avoidance |
| CSMA/CD | Carrier Sense Multiple Access with Collision Detection |
| CSV | Comma-Separated Values |
| CTS | Clear to Send |
| D2D | Device-to-Device |
| DA | Distributed Inter-Process Communication Facility Allocator |
| DAF | Distributed Application Facility |
| DAN | Distributed Application Name |
| DAP | Distributed Application Process |
| DCC | Decentralized Congestion Control |
| DDoS | Distributed Denial-of-Service |
| DEN | Decentralized Environmental Notification |
| DENM | Decentralized Environmental Notification (DEN) Message |
| DES | Discrete Event-based Simulation |
| DGA | Deterministic Gossip Algorithm |
| DHCP | Dynamic Host Configuration Protocol |
| DIF | Distributed Inter-Process Communication Facility |
| DL | Downlink |
| DRF | Data Run Flag |
| DRR | Deficit Round Robin |
| DSCP | Differentiated Services Code Point |
| DSDV | Destination-Sequenced Distance Vector |
| DSL | Domain-Specific Language |

| | |
|---|---|
| DSME | Deterministic and Synchronous Multi-Channel Extension |
| DSR | Dynamic Source Routing |
| DSRC | Dedicated Short-Range Communication |
| DSSS | Direct-Sequence Spread Spectrum |
| DTCP | Data Transfer Control Protocol |
| DT-PDU | Data Transfer Protocol Data Unit |
| DTLS | Datagram Transport Layer Security |
| DTN | Delay-Tolerant Network |
| DTP | Data Transfer Protocol |
| DYMO | DYnamic MANET On-demand |
| E2E | End-to-End |
| ECN | Explicit Congestion Notification |
| ECU | Electronic Control Unit |
| EDCA | Enhanced Distributed Channel Access |
| EFCP | Error and Flow Control Protocol |
| EFCPI | Error and Flow Control Protocol Instance |
| EIGRP | Enhanced Interior Gateway Routing Protocol |
| EMC | Electromagnetic Compatibility |
| eNB | evolved Node B |
| EPC | Evolved Packet Core |
| ERP-OFDM | Extended Rate Physical Orthogonal Frequency Division Multiplex |
| ETSI | European Telecommunications Standards Institute |
| FA | Flow Allocator |
| FAI | Flow Allocator Instance |
| FCS | Frame Check Sequence |
| FDMA | Frequency Division Multiple Access |
| FEL | Future Event List |
| FES | Future Event Set |
| FHSS | Frequency-Hopping Spread Spectrum |
| FiCo4OMNeT | Fieldbus Communication for OMNeT++ |
| FIFO | First In First Out |
| FLoRa | Framework for Long Range |
| FOT | Field Operational Test |
| GEM | Global Environment Model |

| | |
|---|---|
| GEP | Global Event Processor |
| GIS | Geographic Information System |
| GN | GeoNetworking |
| GPS | Global Positioning System |
| GPSR | Greedy Perimeter Stateless Routing |
| GTS | Guaranteed Time Slot |
| GUI | Graphical User Interface |
| H-ARQ | Hybrid Automatic Repeat-reQuest |
| HIL | Hardware-in-the-Loop |
| HLA | High-Level Architecture |
| HR-DSSS | High-Rate Direct-Sequence Spread Spectrum |
| HSS | Home Subscriber Server |
| HT | High Throughput |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| HWMP | Hybrid Wireless Mesh Protocol |
| IAS | Intersection Assistance System |
| ICMP | Internet Control Message Protocol |
| ICMPv6 | Internet Control Message Protocol for Internet Protocol Version 6 |
| ICT | Information and Communications Technology |
| ID | Identifier |
| IDE | Integrated Development Environment |
| IDM | Intelligent Driver Model |
| IEEE | Institute of Electrical and Electronics Engineers |
| IGMP | Internet Group Management Protocol |
| IKEv2 | Internet Key Exchange Version 2 |
| IoT | Internet of Things |
| IP | Internet Protocol |
| IPC | Inter-Process Communication |
| IPCP | Inter-Process Communication Process |
| IPsec | Internet Protocol security |
| IPv4 | Internet Protocol Version 4 |
| IPv6 | Internet Protocol Version 6 |
| IR | Infrared |
| IRM | Inter-Process Communication Resource Manager |

| | |
|---|---|
| IS-IS | Intermediate System to Intermediate System |
| ISM | Industrial, Scientific and Medical |
| ISO | International Organization for Standardization |
| ITS | Intelligent Transportation System |
| ITS-S | Intelligent Transportation System-Station |
| IVC | Inter-Vehicle Communication |
| LAN | Local Area Network |
| LCID | Logical Connection Identifier |
| LDM | Local Dynamic Map |
| LDP | Label Distribution Protocol |
| LEM | Local Environment Model |
| LENA | LTE-EPC Network simulAtor |
| LEP | Local Event Processor |
| LER | Label Edge Router |
| LGPL | GNU Lesser General Public License |
| LIB | Label Information Base |
| LIGO | Laser Interferometer Gravitational-Wave Observatory |
| LIMoSim | Lightweight Information and Communications Technology-centric Mobility Simulation |
| LIN | Local Interconnect Network |
| LLC | Logical Link Control |
| LLDN | Low Latency Deterministic Network |
| LLDP | Link Layer Discovery Protocol |
| LoRa | Long Range |
| LoRaWAN | Long Range Wide Area Network |
| LOS | Line of Sight |
| LP | Logical Process |
| LSP | Label-Switched Path |
| LSR | Label-Switching Router |
| LTE | Long Term Evolution |
| LTE-A | Long Term Evolution Advanced |
| lwIP | lightweigh Internet Protocol |
| LWM2M | Lightweight Machine-to-Machine |
| M2M | Machine-to-Machine |
| MAC | Medium Access Control |

| | |
|---|---|
| MANET | Mobile Ad Hoc NETwork |
| MATSim-T | Multi-Agent Transport Simulation Toolkit |
| MaxC/I | Maximum Carrier-over-Interference |
| MCPS | Medium Access Control Common Part Sublayer |
| MCS | Modulation and Coding Scheme |
| MEC | Multi-access Edge Computing |
| MIB | Management Information Base |
| MIMO | Multiple Input Multiple Output |
| MIP | Mobile Internet Protocol |
| MIPv6 | Mobile Internet Protocol version 6 |
| MLME | Medium Access Control Sublayer Management Entity |
| MME | Mobile Management Entity |
| mmWave | millimeter Wave |
| MO | Multi-Superframe Order |
| MOBIL | Minimizing Overall Braking Induced by Lane change |
| MOST | Media Oriented Systems Transport |
| MPDU | Medium Access Control Protocol Data Unit |
| MPEG | Moving Picture Experts Group |
| MPEG-4 | Moving Picture Experts Group Layer-4 Video |
| MPL | Maximum Packet Lifetime |
| MPLS | Multiprotocol Label Switching |
| MPP | Mobile Peer-to-Peer Protocol |
| MRIP | Multiple Replications In Parallel |
| MSDU | Medium Access Control Service Data Unit |
| NACK | Negative-Acknowledgment |
| NED | Network Topology Description |
| NETA | NETwork Attacks Framework for OMNeT++ |
| NFS | Network File System |
| NIC | Network Interface Card |
| ns-2 | Network Simulator 2 |
| ns-3 | Network Simulator 3 |
| NSC | Network Simulation Cradle |
| NSM | Namespace Management |
| OCB | Outside the Context of a Basic Service Set |
| ODD | Organic Data Dissemination |

| | |
|---|---|
| OEM | Original Equipment Manufacturer |
| OFDM | Orthogonal Frequency Division Multiplex |
| OLSR | Optimized Link State Routing |
| OMNeT++ | Open Modular Network Testbed in C++ |
| ONE | Opportunistic Network Environment |
| OPEN | One-Pair Ether-Net |
| OpenGL | Open Graphics Library |
| OppNet | Opportunistic Network |
| OPS | Opportunistic Protocol Simulator |
| OS | Operating System |
| OSCORE | Object Security for Constrained RESTful Environments |
| OSG | OpenSceneGraph |
| OSI | Open Systems Interconnection |
| OSM | OpenStreetMap |
| OSPF | Open Shortest Path First |
| OTV | Overlay Transport Virtualization |
| P2P | Peer-to-Peer |
| PAN | Personal Area Network |
| PASER | Position-Aware Secure and Efficient Mesh Routing Protocol |
| PCAP | Packet Capture |
| PCAPng | Packet Capture next generation |
| PDCCH | Physical Downlink Control Channel |
| PDCP | Packet Data Convergence Protocol |
| PDES | Parallel Discrete-Event Simulation |
| PDR | Packet Delivery Ratio |
| PDU | Protocol Data Unit |
| PDUFG | Protocol Data Unit Forwarding Generator |
| PER | Packet Error Rate |
| PF | Proportional Fair |
| PGW | Packet Data Network Gateway |
| PHB | Per-Hop Behavior |
| PHY | Physical Layer |
| PID | Proportional, Integral, and Derivative |
| PIM | Protocol-Independent Multicast |
| PIM-DM | Protocol-Independent Multicast-Dense Mode |

| | |
|---|---|
| PIM-SM | Protocol-Independent Multicast-Sparse Mode |
| PoA | Point of Attachment |
| POSIX | Portable Operating System Interface |
| PPP | Point-to-Point Protocol |
| PRACH | Physical Random Access Channel |
| PRNG | Pseudo-Random Number Generators |
| PUCCH | Physical Uplink Control Channel |
| QAM | Quadrature Amplitude Modulation |
| QoS | Quality of Service |
| QPSK | Quadrature Phase Shift Keying |
| RAl | Resource Allocator |
| RAC | Random Access |
| RAN | Radio Access Network |
| RB | Resource Block |
| RED | Random Early Detection |
| RFC | Request for Comments |
| RIB | Resource Information Base |
| RIBd | Resource Information Base Daemon |
| RINA | Recursive InterNetwork Architecture |
| RIP | Routing Information Protocol |
| RLC | Radio Link Control |
| RLWE | Receiver's Left Window Edge |
| RMT | Relaying and Multiplexing Task |
| RNG | Random Number Generator |
| ROHC | RObust Header Compression |
| ROI | Region of Interest |
| RPC | Remote Procedure Call |
| RRC | Radio Resource Control |
| RRS | Randomized Rumor Spreading |
| RRWE | Receiver's Right Window Edge |
| RSS | Received Signal Strength |
| RSSI | Received Signal Strength Indicator |
| RSTP | Rapid Spanning Tree Protocol |
| RSVP | Resource Reservation Protocol |
| RSVP-TE | Resource Reservation Protocol-Traffic Engineering |

| | |
|---|---|
| RTP | Real-time Transport Protocol |
| RTS | Request to Send |
| RTT | Round-Trip Time |
| SAB | Slot Allocation Bitmap |
| SACK | Selective Acknowledgment |
| SAE | Society of Automotive Engineers |
| SAP | Service Access Point |
| SAORS | Socially-Aware Opportunistic Routing System |
| SAS | Smart Antenna System |
| SCADA | Supervisory Control and Data Acquisition |
| SCF | Store-Carry-Forward |
| SCTP | Stream Control Transmission Protocol |
| SDK | Software Development Kit |
| SDN | Software-Defined Networking |
| SDU | Service Data Unit |
| SGW | Serving Gateway |
| SHB | Single-Hop Broadcast |
| SIM | Subscriber Identity Module |
| SINR | Signal-to-Interference-plus-Noise-Ratio |
| SLWE | Sender's Left Window Edge |
| SN | Sequence Number |
| SNMP | Simple Network Management Protocol |
| SNR | Signal-to-Noise Ratio |
| SNIR | Signal-to-Noise-plus-Interference Ratio |
| SO | Superframe Order |
| SON | Self-Organizing Network |
| SRWE | Sender's Right Window Edge |
| SSH | Secure Shell |
| STL | Standard Template Library |
| STP | Spanning Tree Protocol |
| SUMO | Simulation of Urban MObility |
| SV | Summary Vector |
| SVG | Scalable Vector Graphics |
| SWIM | Small Worlds in Motion |
| TAP | Terminal Access Point |

| | |
|---|---|
| TAS | Time-Aware Shaper |
| TB | Transport Block |
| TCP | Transmission Control Protocol |
| TDD | Test-Driven Development |
| TDMA | Time Division Multiple Access |
| TED | Traffic Engineering Database |
| TLS | Transport Layer Security |
| TM | Transparent Mode |
| TraCI | Traffic Control Interface |
| TRILL | Transparent Interconnection of Lots of Links |
| TSCH | Time-Slotted Channel Hopping |
| TSN | Time-Sensitive Networking |
| TTC | Time To Collision |
| TTI | Transmission Time Interval |
| TTL | Time To Live |
| TUN | TUNnel |
| TXOP | Transmit Opportunity |
| UAV | Unmanned Aerial Vehicle |
| UDG | Unit Disk Graph |
| UDP | User Datagram Protocol |
| UE | User Equipment |
| UI | User Interface |
| UL | Uplink |
| ULA | Uniform Linear Array |
| UM | Unacknowledged Mode |
| UML | Unified Modeling Language |
| URI | Uniform Resource Identifier |
| URLLC | Ultra-Reliable Low-Latency Communication |
| UWB-IR | Ultra-Wideband Impulse Radio |
| V2I | Vehicle-to-Infrastructure |
| V2V | Vehicle-to-Vehicle |
| V2X | Vehicle-to-Everything |
| VANET | Vehicular Ad Hoc Network |
| VBR | Variable Bit Rate |
| VDP | Vehicle Data Provider |

| | |
|---|---|
| Veins | Vehicles in Network Simulation |
| VHT | Very High Throughput |
| VLAN | Virtual Local Area Network |
| VLC | Visible Light Communication |
| VoD | Video on Demand |
| VoIP | Voice-over-IP |
| VPN | Virtual Private Network |
| WAMP | Web Application Messaging Protocol |
| WAN | Wide Area Network |
| WAVE | Wireless Access in Vehicular Environments |
| W-CDMA | Wideband Code Division Multiple Access |
| WGS84 | World Geodetic System 1984 |
| WLAN | Wireless Local Area Network |
| WSA | Wave Service Advertisement |
| WSM | Wave Short Message |
| WSN | Wireless Sensor Network |
| XML | Extensible Markup Language |

# Part I
# The OMNeT++ Simulation Environment

# Chapter 1
# A Practical Introduction to the OMNeT++ Simulation Framework


**Andras Varga**

## 1.1 Introduction

OMNeT++[1] is often quoted as a network simulator, but it is really a generic simulation framework for the research and development of complex distributed systems. During the many years it has been available, countless simulation models and model frameworks have been written on top of OMNeT++ by researchers in diverse areas: queuing, resource modeling, internet protocols, wireless networks, switched local area networks, peer-to-peer networks, media streaming, mobile ad-hoc networks, mesh networks, wireless sensor networks, vehicular networks, networks-on-chip, optical networks, high-performance computing systems, cloud computing, storage area networks, and more. Most of these model frameworks are open source, developed as independent projects and follow their own release cycles.

One of most useful and largest model frameworks is the INET Framework, or INET for short. It provides protocols, agents, and other models for working with communication networks. INET is especially useful when designing and validating new protocols, or exploring new or exotic scenarios.

Several other simulation frameworks take INET as a base and extend it into specific directions, such as vehicular networks (e.g., Veins which is introduced in Chap. 6), real-time Ethernet communication (e.g., CoRE4INET which is introduced in Chap. 10), overlay/peer-to-peer networks (e.g., OverSim), or Long Term Evolution (LTE) (e.g., SimuLTE which is introduced in Chap. 5). Veins, for example,

[1] OMNeT++ community website: http://www.omnetpp.org.

A. Varga (✉)
Opensim Ltd, Budapest, Hungary
e-mail: andras@omnetpp.org



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_1

offers a comprehensive suite of models for intervehicle communications. To that end, it uses components from INET and relies on co-simulation with a 3rd-party road traffic simulator.

OMNeT++ provides a solid foundation and rich facilities for writing simulation frameworks like INET. The C++ simulation kernel provides support to structure and parametrize models, use event scheduling, add and control random numbers, collect statistical results, support graphics and animation, and much more. OMNeT++ comes with an Integrated Development Environment (IDE) that provides a comfortable environment for developing simulation models. Using the IDE is optional: model files can be edited in any text editor, and nearly all functionalities of the IDE are also accessible using command-line utilities. Simulations can be run under a graphical runtime environment (*Qtenv*) that supports 2D/3D animation and also allows one to explore the internal state of the model. A console-based runtime environment (*CmdEnv*) is provided for batch execution and other use cases that do not need a Graphical User Interface (GUI).

While many network simulators have a more-or-less fixed architecture for representing network nodes, OMNeT++ provides a generic component architecture and leaves it up to the model designer to decide how to map concepts such as network nodes, network interfaces, protocols, and applications to model components. Model components are termed *modules*, and, if well designed, modules can be used in a variety of different environments and can be combined in various ways like LEGO$^{TM}$ blocks. Modules primarily communicate via message passing, either directly or via predefined connections. Messages may represent events, packets, commands, jobs, or other entities depending on the model domain.

OMNeT++ can be extremely versatile. There are extensions for real-time simulation, network emulation, parallel distributed simulation, co-simulation, **SystemC** integration, federated simulations (High-Level Architecture (HLA)), database connectivity, running simulations in computing clouds, and several other functions. At the time of writing, adding **Python** support is underway, mainly for result analysis and simulation control.

The rest of this chapter will give a brief and practical overview of OMNeT++ and its capabilities. We start from simple things like running an existing simulation and gradually move on to more advanced topics such as how to implement new protocol models or how to set up network emulation.

## 1.2 Getting Started

We will start our journey by running an example simulation from the INET Framework and observing the simulation results. We will use the simulation IDE for our work. The IDE contains everything one needs for developing, running, and analyzing simulation models. There is a dual-mode (graphical/text) editor for designing modules and networks, a simulation configuration editor, C++ editing and build support, a simulation launcher also capable of running simulation batches, a

result plotting and analysis tool, a tool that visualizes simulation execution on a sequence chart, and other utilities like a **Doxygen**-based documentation generator.

### 1.2.1 Starting the IDE

If you have a complete OMNeT++ installation, the IDE is part of it by default. You can start it by typing the `omnetpp` command in the terminal.

### 1.2.2 Installing INET

The first time you start it, the IDE will offer downloading and installing the current (matching) version of the INET Framework. Accept it and wait until the download and the subsequent C++ build finishes. If you miss the initial dialog, you can also find it in the menu under *Help*→*Install*→*Simulation Models*.

While INET builds, you can explore the IDE (see Fig. 1.1). In the top left part of the window, you can find an area labeled *Project Explorer*. In the IDE, the workspace contains *projects*, and projects can contain files and folders. Find the *inet* project in the *Project Explorer* and open it. You will see several folders under

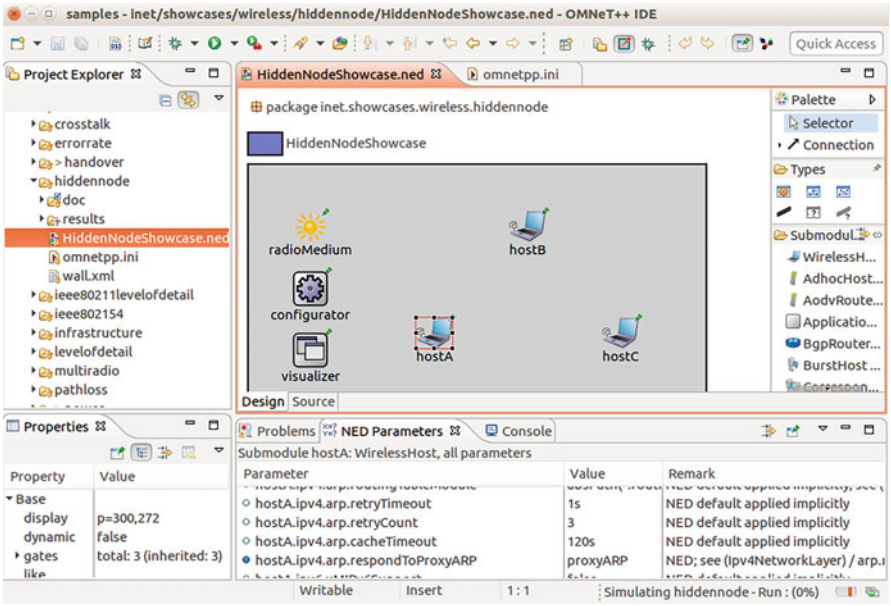



**Fig. 1.1** The OMNeT++ Integrated Development Environment (IDE)

*inet*, the most important folders are *src*, *examples*, *showcases*, and *tutorials*. The *src* folder holds the source files of various protocols and other components provided by the INET Framework, while the latter three contain example simulations and tutorials.

### 1.2.3 Exploring an Example Simulation

Let us take a look at one of the example simulations. Go into the */inet/showcases/-wireless/hiddennode/* folder in the *Project Explorer* and look at the files it contains. You will find a file with the *.ned* extension (*HiddenNodeShowcase.ned*), one called *omnetpp.ini*, and some others.

The Network Topology Description (NED) file contains the structure of the network to be simulated. You can view its contents by double-clicking it to open an editor inside the IDE that displays the network both in graphical and in textual source form. You can edit and change either representation, the two will be kept consistent.

The *omnetpp.ini* file contains model parameters and simulation kernel settings. It is read automatically by the simulation program when it starts. *INI* configuration files with other names can also be used, but they must be explicitly added to the command-line arguments. Double-clicking the file opens it in an editor where you can view or edit it either in text mode or using forms. Hovering over entries or lines in the *INI* editor will display help.

The most frequently used configuration options are `network=...` to select which network to simulate and `sim-time-limit=...` to set a limit for the time span to be simulated. Most simulations would run indefinitely if there was no simulation time limit, CPU time limit, or some other stopping criteria set.

### 1.2.4 Running the Simulation

To run the simulation, select the simulation's folder or the *omnetpp.ini* file in the *Project Explorer* and click the green *Run* button on the toolbar. A new application window titled *OMNeT++/Qtenv* should appear, displaying the simulated network and various controls on its toolbar. The window belongs to Qtenv, OMNeT++'s graphical runtime interface for simulations. The primary function of Qtenv is to let you interactively execute the simulation (run, pause/resume, step, and restart) and observe the sequence of events and the state of the simulation in a convenient way. To run the simulation, click the *Run* button on the toolbar. This will cause the simulation to run slowly with full animation, as shown in Fig. 1.2. The slider on the toolbar will allow you to control the animation speed. You can pause the simulation with the *Stop* toolbar button at any time, and inspect the log or the objects the model

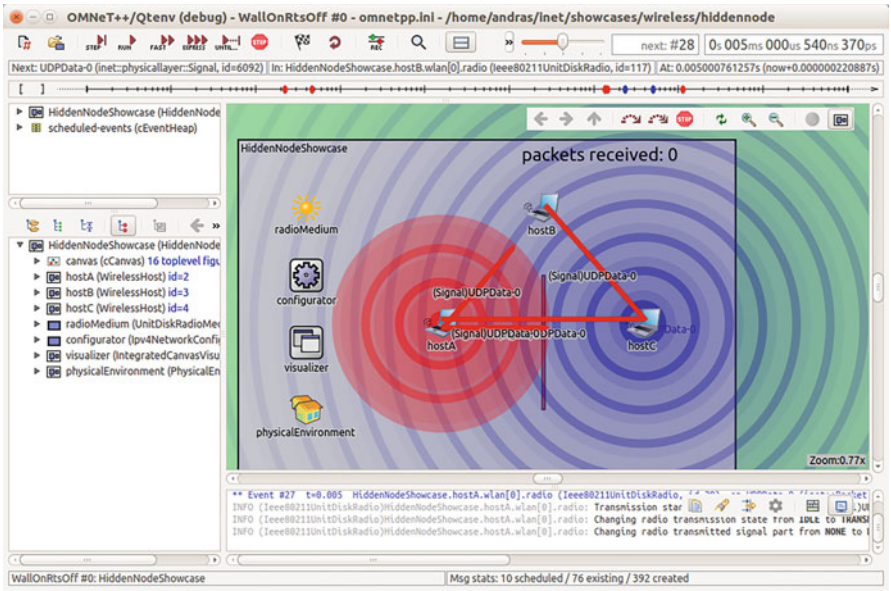



**Fig. 1.2** A wireless simulation running under the Qtenv

is composed of. To run the simulation at full speed, choose the *Express* button. If the simulation runs to completion or you want to restart it at any time, you can do so with the *Restart* toolbar button.

Run the simulation to completion and exit Qtenv to return to the OMNeT++ IDE.

### *1.2.5 Looking at the Simulation Results*

By default, result files are created in the *results* folder under the simulation's working directory. Open this folder and double-click on a file with the *.sca* or *.vec* extension. A dialog will come up, offering the creation of an analysis file (*.anf* extension). After accepting the dialog, the *Analysis Editor* will open.

The *Analysis Editor* lets you work with the contents of the *.sca* and *.vec* result files listed on its *Inputs* page. The *.sca* files store results that are scalar in nature, such as scalar values, summary statistics, histograms, etc. The *.vec* files, on the other hand, store output vectors (basically time series data). The *Browse Data* page lets you browse the contents of the files.

Try the *Analysis Editor* by double-clicking on a vector on the *Browse Data* page to plot it. The vector will open on a separate page (depicted in Fig. 1.3). The resulting chart can be zoomed, panned, and its graphical properties changed. It can also be exported in various formats.

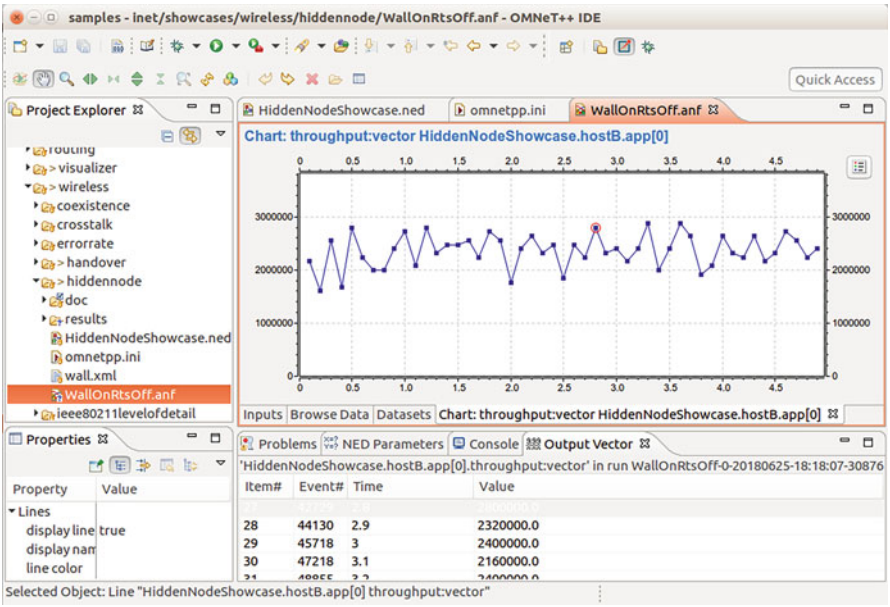



**Fig. 1.3** Plotting a result vector

### *1.2.6 Visualizing a Sequence Chart*

Simulations may contain complex interactions among components. The OMNeT++ IDE can visualize these interactions on a sequence chart drawn from a log recorded from the simulation. A sequence chart can be used to explore the sequence of events and timing relationships.

The input of a sequence chart is an eventlog file recorded by the simulation. To record an eventlog, rerun the simulation with eventlog recording enabled. There are several ways to do that; one of the easiest ones is to use *Record Eventlog* button on the Qtenv toolbar.

To open the sequence chart, find the file with the *.elog* extension in the *results* folder, and double-click it. The initial view might be a little overwhelming because every module is visualized on its own axis and all types of interactions are shown, resulting in too many axes and arrows. However, after some filtering (e.g., show only the top-level modules, `hostA`, `hostB`, and `hostC`) and playing with the display options, one can arrive at the display shown in Fig. 1.4.

The three horizontal axes represent three hosts. Time increases from left to right. The blue strips represent frame transmissions. Note that time is not linear: large intervals are shrunk and small intervals are blown up, in order to make better use of the finite screen area.

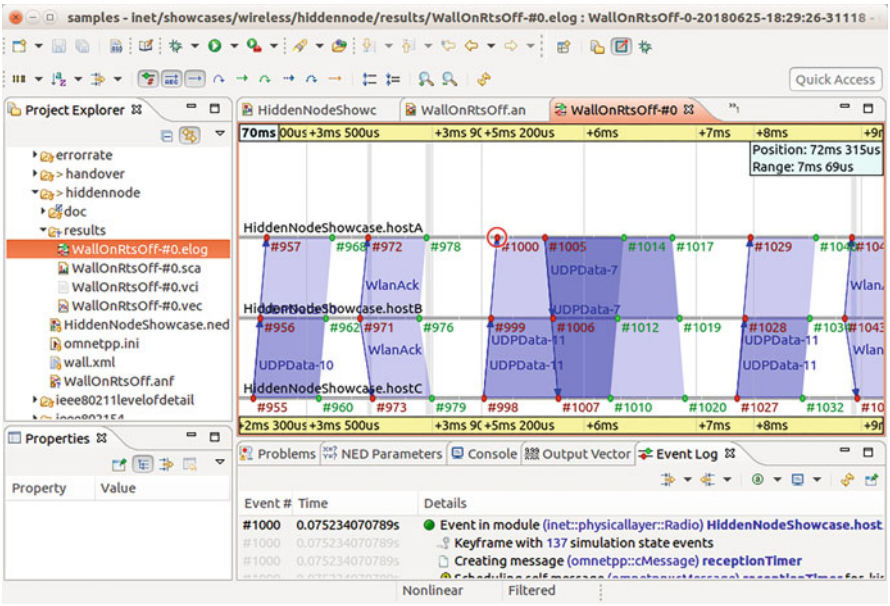



**Fig. 1.4** Sequence chart example

It is not difficult to interpret the chart: `hostA` and `hostC` transmit frames to `hostB`. Some transmissions are successful (are acknowledged by `hostB`), but others are not, because the two transmitting hosts do not know of each other and their transmissions overlap at `hostB`, producing a collision.

## 1.3 Assembling and Running Simulations

In this section, we move on from simply being consumers of existing simulation models and learn how to assemble new models from components provided by INET or other frameworks, how to parametrize them, and how to run large simulation campaigns. But first, we need to learn about the component model of OMNeT++.

### *1.3.1 The Component Model*

The OMNeT++ component model consists of four key components: *simple* and *compound modules*, *connections*, and *parameters*. The following subsections give a short introduction to each key component.

#### 1.3.1.1 Simple Modules and Compound Modules

It has been mentioned that an OMNeT++ model consists of modules that communicate with message passing. The active modules are termed *simple modules*; they are implemented in C++, using the simulation class library. Simple modules can be grouped and structured to form *compound modules*. Compound modules may also contain other compound modules; the number of hierarchy levels is not limited.

In network simulations, simple modules may represent user agents, traffic sources and sinks, exemplary protocol entities like Transmission Control Protocol (TCP), network devices like an IEEE 802.11 Network Interface Card (NIC), data structures like routing tables, or user agents that generate traffic. Simulation-related functions such as controlling the movement of mobile nodes or auto-assigning Internet Protocol (IP) addresses in a network are also often cast as simple modules. Network nodes such as hosts and routers are typically compound modules assembled from simple modules. Additional hierarchy levels are occasionally used above node level (to represent subnetworks) or within nodes (i.e., to group simple modules representing individual protocols of the Internet Protocol Version 6 (IPv6) family into an IPv6 compound module).

Both simple and compound modules are instances of *module types*. While describing the model, the user defines module types; instances of these module types serve as components for more complex module types. The network to be simulated is an instance of a module type. When a module type is used as a building block, there is no distinction whether it is a simple or a compound module. This allows the user to transparently split a module into several simple modules within a compound module, or do the opposite, re-implement the functionality of a compound module in one simple module, without affecting existing users of the module type. The feasibility of model reuse is proven by simulation model frameworks like INET and its various extensions.

#### 1.3.1.2 Connections

Modules communicate with messages, which, in addition to predefined attributes such as timestamps, may contain arbitrary data. Simple modules typically send messages via *gates*, but it is also possible to send them directly to their destination modules. There are *input*, *output*, and *inout* gates. An input and an output gate or two inout gates may be linked with a connection. Connections are defined as part of a compound module, and may connect two submodules, a submodule with the parent, or two gates of the parent module. Connections spanning across hierarchy levels are not permitted, as it would hinder model reuse. Due to the hierarchical structure of the model, messages typically travel through a chain of connections, to start and arrive in simple modules. Compound modules act as “cardboard boxes” in the model, transparently relaying messages between their inside and the outside world. Properties such as propagation delay, data rate, and bit error rate can be assigned to

connections. One can also define connection types with specific properties (termed *channels*) and reuse them in several places.

#### 1.3.1.3 Parameters

Modules can have *parameters*, which are used to pass configuration data to simple modules and to help define the model topology. Supported parameter types are string, integer, double, boolean, and XML, the latter being used to access custom XML-based configuration files. Parameters may have default values, units of measurement, and other attributes attached to them. Parameters may also be *volatile*, enabling them to be re-evaluated every time the simulation code reads their values. For example, when a volatile parameter has the expression `exponential(2.0)` assigned to it, the simulation code will get a different random number from the exponential distribution every time it reads the parameter. Volatile parameters are commonly used to pass stochastic input to modules.

### *1.3.2 Setting Up a New Project*

When you work from the IDE, a new project can be created with *File→New→ OMNeT++ Project*. There are several project templates (with or without example content) to choose from. The usual project layout is to have a *src* folder for components (simple modules, etc.) and a separate *simulations* folder for the actual simulations.

Once the project is created, almost all of its configuration settings can be found in the *Project Properties* dialog, available from the *Project* menu or via right-clicking the project in the *Project Explorer*. This dialog has several pages; here, we review some of the practically most important ones.[2]

If your simulation will use components from another project, say INET, you will need the *Project References* page. It contains a list of all projects with a check box in front of each. Make sure the projects that your project depends on are checked.

The *OMNeT++→NED Source Folders* page allows you to specify where your NED files are. You only need to specify the root folder(s) of the NED package hierarchy, not each and every folder that contains a NED file. Usually, they will be the *src* and the *simulations* (or *examples*) folders. If your simulation uses extra icons, put them into the *images* folder inside your project.

[2]More information is available in the OMNeT++ documentation, namely in the *User Guide*.

### 1.3.3 The NED Language

OMNeT++ has its own domain-specific language (DSL), called the Network Topology Description (NED), for describing the above component model. Typical ingredients of NED descriptions are simple module declarations, compound module definitions, and network definitions. To support simulation in-the-large (the INET Framework contains over 500 module types), the NED language supports inner types, component inheritance, parametric submodule types and module/channel interfaces, metadata annotations called *properties*, a package system inspired by Java, and a documentation system not unlike **Doxygen** or **Javadoc**.

Simple module declarations include the description of the external interface of the module, including the list of module interfaces it conforms to, and the names and types of its gates and parameters. Compound module definitions consist of the declaration of the module's external interface (module interfaces, gates, and parameters), plus the definition of its internal structure. The latter includes the list of submodules and their interconnection. It is possible to declare a submodule to be an array of fixed or parametric size (*submodule vectors*) (see Listing 1.3). For the connections part, limited programming constructs (loop, conditional) are available in order to allow for creating parametric topologies, such as a router with an arbitrary number of ports, or a hexagonal mesh with parametric dimensions.

Network definitions are compound modules that qualify as self-contained simulation models. Examine the subsequent simple example of a NED module definition in Listing 1.1. The NED definition of a compound module is given in Listing 1.2.

**Listing 1.1** An exemplary simple module NED description

```
1  // Models the network layer.
2  simple Routing
3  {
4     parameters:
5        @display("i=block/switch");
6     gates:
7        input in[];
8        output out[];
9        input localIn;
10       output localOut;
11 }
```

**Listing 1.2** An exemplary compound module NED description

```
1  // A "Node" consists of a Routing module, an App module,
2  // and one L2Queue per port.
3  module Node
4  {
5     parameters:
6        int address;
7        string appType;
8        @display("i=misc/node_vs,gold");
9     gates:
10       inout port[];
11    submodules:
12       app: <appType> like IApp {
```

```
            address = address;
            @display("p=140,60");
        }
        routing: Routing {
            @display("p=140,130");
          gates:
            in[sizeof(port)];
            out[sizeof(port)];
        }
        queue[sizeof(port)]: L2Queue {
            @display("p=80,200,row");
        }
    connections:
        routing.localOut --> app.in;
        routing.localIn <-- app.out;
        for i=0..sizeof(port)-1 {
          routing.out[i] --> queue[i].in;
          routing.in[i] <-- queue[i].out;
          queue[i].line <--> port[i];
        }
}
```

**Listing 1.3** Parametric submodule types

```
moduleinterface IFilterNode {
    gates: input in; output out;
}

module DelayerNode like IFilterNode {
    gates: input in; output out;
}

network Example {
    submodules:
        filter: <default("DelayerNode")> like IFilterNode;
        //...
}
```

In order to be prepared for unforeseen use cases, the NED language supports adding metadata annotations to types, parameters, submodules, connections, and other items. So far, metadata annotations have been used to store graphics attributes (position, icon, etc.); to override the default choice of the underlying C++ class name for simple modules and channels; to denote the C++ namespace; to mark gates that are expected to remain unconnected; to declare measurement units and prompt text for parameters; to label gates for automatic matching by the graphical editor's *connect* tool; to denote compound modules that represent physical network nodes in the INET Framework; and for other purposes.

#### 1.3.3.1 Editing NED

The IDE provides a graphical editor for editing NED files. Files can be edited in source as well, and the two representations are always kept in sync. The **opp_nedtool** utility can be used to convert or pretty-print NED files from the command line.

### 1.3.4 Configuring Simulations

*INI* files contain model parameters and configurations for the simulation. They are simple text files that follow the conventional structure composed of sections and key-value pairs. An example is given in Listing 1.4. Keys may denote configuration options, model parameters, and per-object configuration options. Configuration options are easy to recognize because they do not contain a dot character. We have already seen two, `network` and `sim-time-limit`, but there are about a hundred of them at the time of the writing of the book. They configure execution limits, Random Number Generators (RNGs), verbosity (for non-GUI execution), result recording, C++ debugging support, logging, and other features. Other configuration options allow one to replace parts of the simulation runtime with custom components.

Keys that specify model parameters consist of the hierarchical name of the module plus the parameter name, using dots as separators. In order to be able to set the parameters of several modules together, the key may contain wildcards (`*`, `**`, `?`) and numerical ranges for vector indices and numbers embedded in names. The double asterisk (`**`) is able to match several components in the path (e.g., modules at multiple hierarchy levels), while the single asterisk is not. If several keys match the same model parameter, the first match is used. Parameters that have a fixed value assigned from NED will not be looked up in the *INI* file, only the ones that have just a default value or no value at all.

Per-object configuration keys look similar to parameters, but the part after the last dot is the name of a configuration option, and it contains a hyphen. There are nearly twenty per-object options, and they configure the per-module logging, the per-module RNG settings, the per-item result recording options, and so on. One can get a list of the available global and per-object configuration options by running any simulation or **opp_run** with the `-h configdetails` option.

*INI* files may contain several named configurations. Configurations can build upon each other, adding new settings or overriding existing ones. This is practically multiple inheritance. The base is defined with the `extends` configuration option; sections that do not have it implicitly extend the special section named `[General]`. *INI* files can also define parameter studies; this will be covered in a later section.

**Listing 1.4** An exemplary *omnetpp.ini* with several configuration options

```
1 [General]
2 network = Aloha
3 Aloha.numHosts = 20
4 Aloha.txRate = 9.6kbps
5 Aloha.host[*].pkLenBits = 952b
6
```

```
 7 [Config PureAloha] # extends [General]
 8 description = "pure Aloha, overloaded"
 9 Aloha.host[*].iaTime = exponential(2s)
10
11 [Config SlottedAloha] # extends [General]
12 description = "slotted Aloha, overloaded"
13 Aloha.slotTime = 100ms
14 Aloha.host[*].iaTime = exponential(0.5s)
```

#### 1.3.4.1 Editing INI Files

The simulation IDE has a dual-mode (form and text) editor and several associated views for setting up and editing simulation options, parameter settings, and other configuration information. The editor provides wizards, syntax highlighting, content assist, on-the-fly validation, and other convenience features.

### *1.3.5 Launching Simulations*

When you launch a simulation by clicking the green *Run* button in the IDE for the first time, you create a *launch configuration*. A launch configuration is an editable piece of configuration that stores which program to launch and with what options, so it can be used to launch the same simulation later again.

Launch configurations can be edited in the *Run Configurations* dialog. The dialog can be opened in several ways, e.g., by [*Ctrl*]-clicking the *Run* button, via the local menu of the *Run* button, or from the *Run* menu. The dialog offers many options for controlling how you want to launch the simulation, one of the most important ones of them being the combo titled *User Interface*.

You can select *Cmdenv* or *Qtenv*. Qtenv, the graphical runtime environment, has been mentioned before. Cmdenv is command-line runtime user interface that runs simulation without user interaction. Using Cmdenv is more convenient than Qtenv if (when) you no longer want to follow what happens in the simulation, and you are only interested in the results. Cmdenv is also much more convenient for batch execution of, e.g., parameter studies.

When you run the simulation under Cmdenv, it will run without user interaction. You can follow its progress using the *Progress* view or the *Console* view.

#### 1.3.5.1 Running Simulations from the Command Line

Simulations compiled to an executable are self-running, so they can be run like any other program on your system. Simulations compiled to a shared library, for example INET in its default setup, can be run using the **opp_run** or **opp_run_dbg**

utilities which are part of OMNeT++.[3] Qtenv and Cmdenv are not separate programs, but libraries that are part of the simulation program. They can be activated using the `-u Cmdenv` or `-u Qtenv` command-line options. Other controls in the launch configuration dialog translate to further command-line options.

The **opp_runall** utility lets one run simulation campaigns, making use of several CPUs. This tool is covered in Sect. 1.3.7.4.

### *1.3.6 Interactive Execution Using Qtenv*

The primary function of Qtenv is to let you interactively execute the simulation (run, pause/resume, step, and restart) while visualizing and letting you inspect the events and the internal state of the simulation as much as possible. To this end, it has advanced visualization and animation capabilities, lets you inspect the log and the internals of simulation objects, can record an event log for later analysis, and has many more features. Because of this, Qtenv is not only valuable when you are developing your own model or want to demonstrate it, but also when you are trying to understand a model written by someone else.

You can explore Qtenv by trying out the *Run* buttons on the toolbar. The run modes they represent (*normal*, *fast*, *express*) differ in the amount of animation and frequency of display updates they perform. Express mode reduces User Interface (UI) overhead to almost zero, so the simulation can execute at full speed. If the simulation runs to completion or you want to restart it, you can do so with the *Restart* button.

At any time, you can examine the model's state. The current simulation time and serial number of the current simulation event are displayed in the top-right corner.

You can explore the components and other objects the model is composed of in the top-left and bottom-left areas, named *Object Tree* and *Object Inspector*. Double-clicking one of the hosts or another icon in the graphical network view will go into that component and let you view its internals.

The bottom part of the window displays the log from the currently displayed part of the model. It has two modes. In *messages* mode, it shows the history of the message flow between model components, and in *log* mode, the informational and debug log produced by the components. Both can be extremely useful when one needs to figure out what is going on inside the simulation.

Qtenv provides network animation. Animation is automatic, that is, the simulation code does not need to be instrumented with animation requests. The animation is also generic, that is, not specific for network simulation: it works equally well for queuing network simulations, process chain simulations, and other simulations. Qtenv lets the user open a graphical inspector (animation canvas)

[3] The difference between **opp_run** and **opp_run_dbg** is that the former is used with release-mode binaries and the latter with debug-mode ones. Debug and release builds are covered in Sect. 1.4.3.1.

for any compound module or several compound modules. The canvas shows the submodules (network nodes, protocols, etc.) and their interconnections. Positions, icons, background image, and other graphics attributes come from the module (or channel) display strings. At the startup, Qtenv automatically opens a canvas for the top-level compound module which represents the network.

During the simulation, Qtenv animates as messages or packets travel between modules and animates method calls between modules as well. The simulation author can affect the animation by manipulating display strings (for example, updating coordinates of a mobile node, or changing the coloring of a protocol module depending on its state) during simulation.

Qtenv provides *live* animation, as opposed to *playback* provided by the ns-2 *nam* tool. Compared to playback, live animation has its advantages (all objects can be examined in detail at any time and it can be combined with C++ debugging) and disadvantages (backward play or replay of the history is not possible) as well.

## *1.3.7 Parameter Studies*

Simulations are often used to explore a parameter space, for example, to gather data on how certain protocols or networks behave under different conditions or with different parameter settings. In this section, we look at how such parameter studies can be defined and carried out using OMNeT++.

### 1.3.7.1 Defining a Parameter Study

*INI* files offer a convenient syntax for defining parameter studies (cf. Listing 1.5). If an *INI* file configuration contains one or more iteration variable of the form `${...}`, for example `${numHosts=1..5,10,20,50}`, the simulation runtime will take the Cartesian product of the sequences and generate a simulation run for each. For example, the above `numHosts` iteration variable together with a `${pkLen=100,200,500}` variable will generate $8 \times 3 = 24$ simulation runs. If not all combinations are needed, the user can specify a constraint to filter out the unwanted ones. For example, the constraint `$numHosts>10 || $pkLen==500` would mean that, e.g., a run with $numHosts = 20$ and $pkLen = 100$ will be left out.

**Listing 1.5** An exemplary parameter study, defined in the *omnetpp.ini* configuration file

```
1 [Config ThroughputTest]
2 **.numHosts = ${numHosts=1..5,10,20,50}
3 **.pkLen = ${pkLen=100,200,500}
4 constraint = $numHosts>10 || $pkLen==500
5 repeat = 10
```

It is also possible to specify that each run has to be repeated 10 times with different random number generator seeds, yielding $24 \times 10 = 240$ runs. They are numbered from 0 through 239, and the user can tell the simulation program (via command-line options) to execute, say, run #146 of configuration `ThroughputTest` in the specified configuration file. The seeds for these runs are generated automatically (but in a configurable way) from the run number and/or the repetition counter. It is also possible to specify seeds manually, but this is rarely needed or desired.

#### 1.3.7.2 Running Parameter Studies

Although the easiest way to run parameter study is from the IDE, for a better understanding, let us first see how running it from the command line would look like. First of all, we would use Cmdenv because the Qtenv UI is not needed for batch execution. If we just specify the name of the *INI* configuration file containing the parameter study with the `-c` option, Cmdenv will perform all simulations one after another. The `-r` allows to pass a run filter expression as well. An example is as follows:

```
$ ./aloha -u Cmdenv -c PureAlohaExp -r '$numHosts>15'
```

This command will do the job and execute the desired simulation runs, but it is not optimal for a number or reasons. For example, if one run crashes, the program will be terminated and the rest of the runs will not be executed at all. Also, it will only use one CPU core.

We could execute each simulation in its own process by querying the list of run numbers matching the filter first (`-q runnumbers`), and running them one by one:

```
$ ./aloha -u Cmdenv -c PureAlohaExp -r '$numHosts>15' -s \
-q runnumbers
28 29 30 31 32 33 34 35 36 37 38 39 40 41
$ ./aloha -u Cmdenv -c PureAlohaExp -r 28
$ ./aloha -u Cmdenv -c PureAlohaExp -r 29
$ ./aloha -u Cmdenv -c PureAlohaExp -r 30

...

$ ./aloha -u Cmdenv -c PureAlohaExp -r 40
$ ./aloha -u Cmdenv -c PureAlohaExp -r 41
```

To make use of multiple CPUs, we could first launch as many simulations as we have CPUs, then launch new ones every time one of them finishes, to keep all CPUs busy. This is exactly what both the IDE and the **opp_runall** command line do.

#### 1.3.7.3 Using the IDE

The easiest way to run a parameter study is to use the launcher in the IDE. To try it, simply open the *Run Configurations* dialog, select or create a launch configuration for your simulation, and select the *INI* configuration file containing the parameter study from the *Config Name* combo box. Make sure Cmdenv is selected as the user interface (it is more suitable for this purpose than Qtenv), then click *Run*. The IDE will run all simulations of the parameter study in the background. You will be able to track progress in the *Progress* view, and also terminate the batch if you wish.

If you want to use more than one CPU core, make sure to select the *Allow Multiple Processes* checkbox in the dialog. This option will divide the series of simulation runs into smaller batches of a fixed size (e.g., 2 or 5 runs each), and launch a separate Cmdenv instance for each batch. When one Cmdenv instance (one batch) finishes, the IDE will launch another one, keeping the specified number of CPU cores busy.

#### 1.3.7.4 Using opp_runall

The **opp_runall** program is an OMNeT++ utility that supports running simulation batches from the command line. Its feature set is similar to the IDE launcher. The following command will run the *./aloha* simulation program with the `PureAlohaExp` *INI* configuration file for the `$numHosts>15` condition using 4 CPU cores:

```
$ opp_runall -j4 ./aloha -c PureAlohaExp -r '$numHosts>15'
```

### *1.3.8 Analyzing the Results*

Simulation results are normally written to the *results* folder under the simulation's working directory. Open this folder and double-click on a *.sca* or *.vec* file to create an *ANF* (analysis) file. *ANF* files describe the analysis, which can be examined in the *Analysis Editor*, integrated in the OMNeT++ IDE.

The analysis editor is a multipage editor, where one can switch between pages using the tabs at the bottom of the editor area. The *Inputs* page is used for selecting the result files that serve as input for the analysis. The *Browse Data* page lets you view the simulation results in the files specified on the *Inputs page*. The *Charts* page contains chart specifications as labeled icons. Double-clicking on a chart icon will cause the computations to be performed and the resulting chart to be opened in a

separate page.[4] Charts are live, so they allow zooming and panning, and may also be interactive. Some chart properties can be edited using a *Properties* dialog.

Although charts can be created as easily as double-clicking items in the *Browse Data* page, the heart of each chart is a **Python** script that relies on packages like **Numpy**, **Scipy**, and **Pandas**. The script can be modified by the user to include arbitrary computations and chart types, so the possibilities are practically limitless. The results can plotted using **Matplotlib** or the analysis tool's built-in chart types. The built-in chart types are more limited in features, but usually scale significantly better than **Matplotlib** charts.

**opp_scavetool** can be used to query simulation result files (*.sca* and *.vec* files) and export them in Comma-Separated Values (CSV) and other formats, possibly after filtering. CSV can be directly imported into **Python /Pandas**, **GNU R**, or other programs, e.g., spreadsheets for further (interactive or scripted) processing.

### *1.3.9 Eventlog and Sequence Charts*

Sequence charts are for exploring the sequence of events and timing relationships. OMNeT++ simulations can optionally create an eventlog file that records simulation events such as: message creations and deletions, event scheduling and cancellation, message sends and packet transmissions, model topology changes, display string changes, debug log messages from simple modules, etc. Message and packet fields may also be captured in the event log file at a configurable detail level; this feature relies on reflection information generated by the message compiler.

The simulation IDE can visualize the log using an interactive sequence diagram, which significantly facilitates the verification of protocol models. The chart can be panned and zoomed, there are several ways (linear, nonlinear, step, etc.) to map simulation time and events to the $x$-axis, and the chart can be filtered by modules and by various other criteria. Tooltips show the properties of events and messages/packets in detail, and it is also possible to browse the detailed log of actions by simulation event. The sequence chart can also be exported in the Scalable Vector Graphics (SVG) format. During operation, the tool only keeps parts of the file in memory, so it is feasible to view event log files of several gigabytes with it. **opp_eventlogtool** can be used to query and filter eventlog (*.elog*) files from the command line.

[4]This text describes the *Analysis Editor* from OMNeT++ version 6.0, which was under development at the time of writing. Older versions (4.x, 5.x) differ significantly.

## 1.4 Model Development

This section provides an introduction into the practice of writing OMNeT++ models, that is, simple modules involving C++ code.

Developing new simulation components in C++ is not unlike developing any C++ program or library. One needs to set up a project with a suitable build system, followed by the *edit-build-run* cycle. Debugging is often inevitable, as are other tools that provide higher-level information about the model's operation. Tests to uncover problems and regressions are often a must, especially for larger and longer-lived projects. The following subsections provide some practical guidance for these topics.

### *1.4.1 Editing C++ Source Files*

The OMNeT++ IDE has special editors for *NED*, *MSG*, and *INI* files. C++ editing is provided by the CDT (C/C++ Development Tools) project of Eclipse. In addition to the usual editor functions like syntax highlighting, folding, undo/redo, find/replace, and incremental search, the C++ editor also offers several "smart" features like autocompletion and content assist, various ways to explore and navigate the code (e.g., go to declaration, find references, or show type hierarchy), and refactoring operations like rename, extract local variables, or extract functions. Most "smart" features require the project to be indexed by Eclipse.

### *1.4.2 Version Control*

We strongly recommend to keep source files under version control. We recommend the use of Git. If you intend to make your project public, create a repository on GitHub. However, even small personal projects with only a local Git repository benefit from being able to review changes.

The OMNeT++ IDE has built-in Git support. When you import a project under version control (has a *.git* folder), the IDE automatically picks that up without any extra configuration. Git-related functions such as commit, push, and pull are available in the *Team* submenu of the project's or file's context menu. Managing a project repository by mixing the IDE and external tools like **gitk** or **git-gui** is also possible.

### *1.4.3 C++ Build*

To build, choose *Build Project* from the *Project* menu, or hit [*Ctrl*]+[*B*].

C++ build settings can be configured on the *OMNeT++→Makemake* page of the *Project Properties*. The page shows the folder structure of your project and allows you to choose for each folder whether you want to generate a makefile in that folder, and if so, with what options. Usually, you only want to have a makefile in the *src* folder. If your project depends on another project or on external C/C++ libraries, this page is also the place where you can specify that you want to link with those libraries. The generated makefiles file will be updated before each build.

#### 1.4.3.1 Debug vs Release Build

In order to be able to debug the simulation in a meaningful way, the compiled code must contain debug information. OMNeT++ allows models to be compiled in one of two modes: *release* and *debug*. The former one is the default mode, and it will instruct the compiler to emit optimized code; the latter is meant for debugging. In the IDE, you can switch between the two modes by selecting the *Build Configurations → Set Active* item from the project's context menu.

When working from the command line, use the `make MODE=debug` command to build the model in debug mode. Debug mode builds produce binaries that have the *_dbg* suffix in their names, to allow debug and release mode binaries to co-exist in the same directory.

#### 1.4.3.2 Project Features

It takes a while to compile INET from source. You can reduce both the compilation time and the size of the binaries if you exclude parts you do not need for your simulation study from the build process. To do that, go into the *Project* menu and choose *Project Features* from it. A dialog will pop up where you can select which protocols and components you need. After making your choice you need to recompile INET (choose *Build Project* from the *Project* menu, or hit [*Ctrl*]+[*B*]).

#### 1.4.3.3 Command-Line Usage

If you do not use the IDE for some reason, you can use the **opp_makemake** utility with the appropriate options to generate a makefile. One of the most useful options is `--deep`, which instructs **opp_makemake** to generate a makefile that covers files in the whole source tree, not just in the current folder. The `-make-so` option (or `-s` for short) sets the target type to shared library (the default is building an executable), and `-o` can be used to specify the base name of the output. For example,

the following command will create a makefile that builds a shared library named *libFoo.so* or *libFoo_dbg.so* on Linux, and *Foo.dll* or *Foo_dbg.dll* on Windows:

```
$ opp_makemake -f --deep --make-so -o Foo
```

**opp_makemake** is fully compatible with the makefile generator in the IDE.

Once a makefile is present, use the `make -j4` command to perform the build (replace "4" with the number of CPU cores you wish to use). To perform a debug build, enter the `make -j4 MODE=debug` command (as the default mode in the generated makefiles is `release`.)

The **opp_featuretool** program lets you query, enable, and disable project features from the command line.

### *1.4.4 Debugging*

OMNeT++ simulations are essentially C++ programs, so C++ debuggers can be used to find the source of crashes and facilitate tracking down other problems. This section deals with debugging simulations. However, debugging is not always necessary or practical, and tools that provide a higher-level view of the simulation are sometimes of more help. Two such tools are the Qtenv runtime and the *Sequence Chart* tool in the IDE. Qtenv has three facilities relevant here: *animation*, *inspection*, and *logs*.

In addition to animation, Qtenv lets the user inspect objects and variables in the model. For example, you can view the list of scheduled events, examine module parameters, state variables, contents of queues, messages and packets, and look at the current values of statistics. You can also search among all objects in the simulation by various filter criteria. Qtenv also displays debug log output by modules as well as the packet flow. The latter feature is comparable to a network analyzer like **Wireshark**. Note that Qtenv runs as part of the simulation program, so if the simulation crashes, it will bring down the Qtenv UI as well.

#### 1.4.4.1 Debugging in the IDE

The OMNeT++ IDE contains an integrated C/C++ debugger that comes from the Eclipse CDT project. Beyond the basics (single-stepping, stack trace, breakpoints, watches, etc.), it also offers several convenience and advanced functionalities such as inspection tooltips and smart breakpoints. Currently, CDT uses the GNU Debugger (**gdb**) as the underlying debugger; **LLDB** support is underway.

Launching a simulation in debug mode is very similar to running it, only you have to select the *Debug* toolbar icon or menu item instead of *Run*. When debugging, the IDE will switch to an alternative layout (the *Debug* perspective) better suited for debugging, and specialized views like *Debug*, *Breakpoints*, and *Expressions* will

appear. After you are done with debugging, you can switch back to the *Simulation* perspective using the perspective switching controls in the top-right corner.

Debugging a simulation is by and large the same as debugging any C/C++ program, so we do not go into detail about it in this chapter. However, we will cover some OMNeT++ features that facilitate debugging.

#### 1.4.4.2 Just-in-Time Debugging

When the simulation stops with a runtime error, being able to debug may help in identifying the cause of the error. To allow such *just-in-time debugging*, enable the *debug-on-errors* option before running the simulation, e.g., by selecting the appropriate checkbox in the launch configuration. It will cause the program to be stopped in the debugger when the error occurs, and you can look at the stack trace and examine variables.

### *1.4.5 Sanitizing*

When your simulation starts to have random crashes or display other nondeterministic behavior, it is time to sanitize it. Unfortunately, C++ is a language that, in addition to allowing you to create very efficient code, also lets you shoot yourself in the foot: use of uninitialized variables, array over-indexing, buffer overflows, heap use-after-delete, stack use-after-return, memory leaks, and the list goes on and on. These types of errors are difficult to track down using a debugger. Instead, the number one tool to turn to is **Valgrind** under Linux (or comparable tools like **DrMemory** under Windows).

**Valgrind** is a suite of tools for debugging and profiling code on Linux. **Valgrind** can automatically detect various memory access and memory management bugs and perform detailed profiling of your program. The IDE supports launching your simulation under **Valgrind** (use the *Profile* button on the toolbar, next to *Run* and *Debug*), or you can do the same from the command line. One downside is that programs run quite a bit slower (e.g., 20×) under **Valgrind**; however, it produces a report that is both comprehensive and reliable. It is a good idea to "valgrind" your simulation from time to time, even if it seemingly works correct.

An alternative to **Valgrind** are sanitizers, a family of dynamic testing tools built into the Clang and GCC C++ compilers. **AddressSanitizer** finds memory errors such as use-after-free, buffer overflows, and leaks; **MemorySanitizer** finds uses of uninitialized memory; **UndefinedBehaviorSanitizer** finds other kinds of undefined behavior, such as use of incorrect dynamic type, shift by illegal amount, and many others. To use these sanitizers, you need to compile the simulation (and possibly the OMNeT++ libraries as well) with certain flags such as `-fsanitize=address` to produce an instrumented binary that will print diagnostic messages when it is run. The need for a special build is an inconvenience compared to **Valgrind** which

is able to detect errors in the unmodified program, but the simulation will execute significantly faster than under **Valgrind** (expect 2×–3× slowdown instead of 20×).

Code quality can also be improved using static analysis tools like **Coverity**, **PVS Studio**, or **CppCheck**. One upside of static analysis tools is that they can find errors in all parts of the program, not only those that were exercised during execution; a downside is that the reports produced tend to contain false positives.

### *1.4.6 Profiling*

Many simulations need to be run for several hours or longer to obtain meaningful simulation results, or are run several hundred or thousand times as part of a simulation study. Thus, optimizing them for speed is often vital. Profiling answers the question of how much time is spent executing various parts of the program, which helps you to figure out which parts of the program are worthwhile to optimize.

The simplest way of profiling, which requires no special tool, is to manually stop the program from time to time in the debugger, each time checking the stack trace. If it frequently stops in the same part of the program, it is a strong indication that the program spends a lot of its time there, and therefore it is a good candidate for optimizations. One class of profilers called statistical profilers utilizes and extends this idea. Other classes of profilers include instrumentation-based and hypervisors/instruction set simulators.

**Valgrind** is also useful here. While **Valgrind** has become synonymous with **Memcheck**, its memory sanitizer tool, it contains other tools too, like the **Callgrind** profiler. Running the simulation program under **Callgrind** will produce a trace file that can be viewed using **KCacheGrind**, an excellent graphical frontend. Other profiling tools are **GNU gprof**, **gperftools** (originally Google Performance Tools), and the system-wide Linux profilers **Sysprof** and **OProfile**.

### *1.4.7 Validation and Verification*

Correctness of the simulation model is a primary concern of the developers and users of the model, because they want to obtain credible simulation results. Verification and validation are activities conducted during the development of a simulation model with the ultimate goal of producing an accurate and credible model. Verification of a model is the process of confirming that it is correctly implemented with respect to the conceptual model, that is, it matches specifications and assumptions deemed acceptable for the given purpose of application. During verification, the model is tested to find and fix errors in the implementation of the model.

Validation checks the accuracy of the model's representation of the real system. Model validation is defined to mean "substantiation that a computerized model

within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model" [6]. A model should be built for a specific purpose or set of objectives and its validity determined for that purpose.

Of the two, verification is essentially a software engineering issue, so it can be assisted with tools used for software quality assurance, for example testing tools. Validation is not a software engineering issue.

### *1.4.8 Fingerprints*

When an existing simulation model is refactored or extended with new features, it is important that changes do not accidentally break existing functionality. Regression tests are widely employed to ensure that. There are many ways to write regression tests, but a very practical and easy-to-use technique is built into OMNeT++: *fingerprints*. Fingerprint tests are a low-cost but effective tool for regression testing of simulation models. A fingerprint is a hash computed during the simulation from various properties of simulation events, messages, and statistics. The hash value is continuously updated as the simulation executes, and thus, the final fingerprint value is characteristic of the simulation's trajectory.

For regression testing, one needs to compare the computed fingerprints to that from a reference run—if they differ, the simulation trajectory has changed. In general, fingerprint tests are useful to ensure that a code change (e.g., refactoring, a bugfix, or a new feature) did not break the simulation. The fingerprint implementation allows to select what to include in the hash value. Changing the ingredients allows one to make the fingerprint sensitive for certain changes while keeping it immune to others.

Syntactically, a fingerprint is a string of hex digits that represent a 32-bit number, followed by a slash character and a group of letters that indicate the ingredients of the fingerprint. Each ingredient is identified with a letter. The *t*, for instance, stands for the simulation time. The *omnetpp.ini* source code excerpt in Listing 1.6 means that a fingerprint needs to be computed with the simulation time (*t*), the module full path (*p*), the received packet's bit length (*l*), and the extra data (*x*) included for each event. The result is expected to be `53de-64a7`.

**Listing 1.6** Specifying the expected fingerprint

```
1 fingerprint = 53de-64a7/tplx
```

OMNeT++ contains a script for automated fingerprint tests as well. The script runs either all or selected simulations defined in a CSV file (with columns like the working directory, the command to run, the simulation time limit, and the expected fingerprints) and reports the results. The following excerpt from a CSV file prescribes the fingerprint tests to run:

```
examples/aodv/, inet -c Static,   50s,  4c29-95ef/tplx
examples/aodv/, inet -c Dynamic,  60s,  8915-f239/tplx
examples/dhcp/, inet -c Wired,    800s, e88f-fee0/tplx
examples/dhcp/, inet -c Wireless, 500s, faa5-4111/tplx
```

### 1.4.9 Documenting

When a simulation model is expected to be shared with other people, documenting it is usually a good idea. OMNeT++ includes support for generating documentation from NED file and message definition comments. The generated documentation lists modules, channels, messages, etc. and presents their details including description, gates, parameters, assignable submodule parameters, and syntax-highlighted source code. The documentation also includes network diagrams, usage diagrams, and inheritance diagrams. The documentation tool integrates with **Doxygen**, meaning that it can hyperlink simple modules and message classes to their C++ implementation classes in the **Doxygen** documentation. Documentation is embedded in normal C/C++ code comments (//). An example is given in Listing 1.7.

**Listing 1.7** Exemplary NED documentation

```
1 // Ethernet MAC. Performs transmission and reception of Ethernet frames. [...]
2 //
3 // @see ~EtherEncap, ~EtherFrame
4 simple EtherMac like IEtherMac {
5   parameters:
6     string address = default("auto"); // MAC address (hex string) or "auto"
7     bool duplex = default(true); // full duplex (true) or half duplex (false)
8     bool promiscuous = default(false); // if true, receive all packets
9 }
```

## 1.5 Writing Components

This section will show you how to develop new simulation components in C++. It covers discrete event simulation fundamentals and several Application Programming Interfaces (APIs) of the OMNeT++ simulation kernel, from random number generation, timers, and packets to statistical result collection, implementing animation, and many more topics.

### 1.5.1 Discrete Event Simulation

Before starting to write protocol models in OMNeT++, we need to get familiar with the concepts of Discrete Event-based Simulation (DES). Readers already familiar with DES are unlikely to find much new in this subsection and may skip it.

#### 1.5.1.1 Discrete Event Systems

A *discrete event system* is a system where state changes (events) happen at discrete instances in time, and events take zero time to happen. It is assumed that nothing (i.e., nothing interesting) happens between two consecutive events, that is, no state change takes place in the system between the events. This is in contrast to *continuous* systems where state changes are continuous. Systems that can be viewed as discrete event systems can be modeled using discrete event simulation, DES. Computer networks are usually viewed as discrete event systems, where events represent things like the start or end of a frame transmission, expiry of a retransmission timeout, or an application opening/closing a socket.

The time when events occur is often called the *event timestamp*; with OMNeT++ we use the term *arrival time* (because in the class library, the word "timestamp" is reserved for a user-settable attribute in the event class). Time within the model is typically called *simulation time*, *model time*, or *virtual time*, to distinguish it from real time or CPU time which refer to how long the simulation program has been running and how much CPU time it has consumed.

#### 1.5.1.2 The Discrete Event-Based Simulation Algorithm

A discrete event simulator maintains a set of future events in a data structure called the Future Event List (FEL), a.k.a. the Future Event Set (FES). Such simulators usually work according to the following pseudocode from Listing 1.8.

**Listing 1.8** Pseudocode of the DES event loop

```
initialize() # insert initial events into FES, set now = 0

while not isEmpty(FES) and not completed:
   event = removeFirst(FES)
   now = timestampOf(event)
   process(event) # may insert events into FES or remove scheduled ones

finalize() # write statistical results, etc.
```

The initialization step usually builds the data structures representing the simulation model, calls any user-defined initialization code, and inserts initial events into the FES to ensure that the simulation can start. Initialization strategies can differ considerably from one simulator to another. The subsequent loop consumes events from the FES and processes them. Events are processed in strict timestamp order to maintain causality, that is, to ensure that no current event may have an effect on earlier events. Processing an event involves calls to user-supplied code. For example, using the computer network simulation example, processing a "timeout expired" event may consist of re-sending a copy of the network packet, updating the retry count, scheduling another "timeout" event, and so on. The user code may also remove events from the FES, for example when canceling timeouts. The simulation stops when there are no events left or when it is not necessary for the simulation

to run further because the model time or the CPU time has reached a given limit, or because the statistics have reached the desired accuracy. At this time, before the program exits, the user will typically want to record statistics into output files.

In practice, the FES often contains events with the same timestamp value, and it is important to decide how they should be ordered relative to each other. In OMNeT++, this issue is resolved with two simple rules: first, an event scheduled (i.e., inserted into the FES) earlier will also be processed earlier. Second, the effect of scheduling order can be overridden by a priority field in the event.

#### 1.5.1.3 Simulation Time

When implementing a discrete event simulator, one would be tempted to represent simulation time with double-precision floating-point numbers (C/C++ `double`). This is generally not a good idea, because such simulation time would have diminishing precision as simulation time advances. OMNeT++ represents simulation time with a `simtime_t` type which is a 64-bit fixed-point number with a base-10 exponent. The exponent is stored in a global variable to eliminate the need for normalization, and to conserve memory. The range provided by 64 bits is more than enough for practical simulations (approx. ±292 years with nanosecond precision or ±107 days with picosecond-precision).

### *1.5.2 Modules, Messages, and Events*

Simple modules are implemented as C++ classes, derived from the library class `cSimpleModule`. Listings 1.9 and 1.10 present two simple examples of a source and sink module that demonstrate the subsequently described basic tasks. Message sending and receiving are the most frequent tasks in simple modules. Messages are represented with the `cMessage` class. A module can send messages to another module via output gates and connections using the `send()` call, or directly to other modules using the `sendDirect()` function.

The simulation kernel sends messages to the `handleMessage(cMessage*)` method of the module; module authors primarily need to override this method to add functionality. The alternative to `handleMessage()` is a process-style description, where users have to override the `activity()` method, and messages are delivered as the return value of blocking *receive* calls. In general, using `activity()` is not recommended because it does not scale due to the underlying coroutine stacks, but there are situations when it is extremely useful, such as when porting a process-based program into a simulation.

Timers and timeouts are implemented with normal messages that the module sends to itself.[5] These self-messages are sent with a `scheduleAt()` call and are delivered back to the module in the same way as messages arriving from other modules. Self-messages can also be canceled. Note that there is no separate event class, its role is fulfilled by `cMessage`.

The programmer can provide code to execute during module initialization and finalization by overriding dedicated methods of the module class (`initialize()`, `finish()`). Finalization takes place on successful simulation termination only, and its code is most commonly used to record summary simulation results. OMNeT++ also supports multi-stage initialization, and it has proven essential for large models like the INET Framework as a tool for managing complex initialization interdependencies across various model components. Listing 1.11 gives an example of a multi-stage initialization process.

**Listing 1.9** A simple module C++ class that implements a source

```
1  class Source : public cSimpleModule {
2   protected:
3     virtual void initialize() override;
4     virtual void handleMessage(cMessage *msg) override;
5  };
6
7  Define_Module(Source);
8
9  void Source::initialize() {
10    scheduleAt(simTime(), new cMessage("timer"));
11 }
12
13 void Source::handleMessage(cMessage *msg) {
14    send(new cMessage("generatedMsg"), "out");
15    scheduleAt(simTime() + 1, msg);
16 }
```

**Listing 1.10** A simple module C++ class implementing a sink

```
1  class Sink : public cSimpleModule {
2   protected:
3     int numReceived;
4     virtual void initialize() override;
5     virtual void handleMessage(cMessage *msg) override;
6     virtual void finish() override;
7  };
8
9  Define_Module(Sink);
10
11 void Sink::initialize() {
12    numReceived = 0;
13 }
14
15 void Sink::handleMessage(cMessage *msg) {
16    EV << "received message '" << msg->getName() << "'\n";
```

[5] Other simulators implement timers as callbacks invoked from the simulation kernel. OMNeT++ prefers the *handleMessage* approach, because we have found that it results in code that is easier to understand and maintain.

```
17    numReceived++;
18    delete msg;
19 }
20
21 void Sink::finish() {
22    EV << "received total " << numReceived << " messages\n";
23 }
```

**Listing 1.11** Multi-stage initialization

```
1  class MultiStage : public cSimpleModule {
2   protected:
3    virtual int numInitStages() const override { return 4; }
4    virtual void initialize(int stage) override;
5    //...
6  };
7
8  void MultiStage::initialize(int stage) {
9    switch (stage) {
10      case 0: /*...*/; break;
11      case 1: /*...*/; break;
12      case 2: /*...*/; break;
13      case 3: /*...*/; break;
14   }
15 }
```

### *1.5.3 Accessing Parameters*

Module parameters declared in NED files can be accessed with the `par()` method of the module/channel. `par()` expects the name of the parameter as an argument and returns a reference to the parameter object which has the type `cPar` (cf. Listing 1.12). The `cPar` value can be read with methods that correspond to the parameter's NED type: `boolValue()`, `intValue()` (returns `intpar_t`, an integer at least 64 bits wide), `doubleValue()`, `stringValue()` / `stdstringValue()` (they return `const char *` and `std::string`, respectively), and `xmlValue()` (returns `cXMLElement*`). There are also overloaded type cast operators for the corresponding types (`bool`; all integer primitive types like `int`, `long`, etc.; `double`; `const char *`; and `cXMLElement *`).

A parameter can be declared `volatile` in the NED file. The `volatile` modifier indicates that a parameter is re-read every time a value is needed during simulation. Volatile parameters are typically used for things like random packet generation interval (cf. Listing 1.13) and are assigned values like `exponential(1.0)` (numbers drawn from the exponential distribution with mean 1.0). In contrast, nonvolatile NED parameters are constants and reading their values multiple times is guaranteed to yield the same value. When a nonvolatile parameter is assigned a random value like `exponential(1.0)`, it is evaluated once at the beginning of the simulation and replaced with the result, so all reads will get the same (randomly generated) value. The typical use of nonvolatile parameters

is to read them once in the `initialize()` method of the module class and store the values in class variables for easy access later when necessary.

**Listing 1.12** Reading a parameter in a module

```
class JobSource : public cSimpleModule {
 protected:
   long numJobs;
   virtual void initialize();
   //...
};

void JobSource::initialize() {
   numJobs = par("numJobs");
   //...
}
```

`volatile` parameters need to be re-read every time the value is needed. For example, a parameter that represents a random packet generation interval may be used like in the example from Listing 1.13:

**Listing 1.13** Reading a volatile parameter

```
void Source1::handleMessage(cMessage *msg) {
   //...
   scheduleAt(simTime() + par("interval").doubleValue(), timerMsg);
   //...
}
```

The code above looks up the parameter by name every time. This name lookup can be avoided by storing the parameter object's pointer in a class variable, resulting in the following code from Listing 1.14:

**Listing 1.14** Sparing the per-read lookup of a volatile parameter

```
class Source2 : public cSimpleModule {
 protected:
   cPar *intervalp;
   cMessage *timerMsg;
   virtual void initialize() override;
   virtual void handleMessage(cMessage *msg) override;
   //...
};

void Source2::initialize() {
   intervalp = &par("interval");
   //...
}

void Source2::handleMessage(cMessage *msg) {
   //...
   scheduleAt(simTime() + intervalp->doubleValue(), timerMsg);
   //...
}
```

### 1.5.4 Random Numbers

If you run the same discrete event simulation multiple times, it will always produce the same results. This is because "random" numbers consumed by simulations are not really random, but rather produced by deterministic algorithms, called Pseudo-Random Number Generators (PRNG) (or RNG for short). The benefit of using PRNGs is that simulations are repeatable (in contrast to emulations or field tests). PRNGs generate the "random" sequence from a *seed* (start) value. To generate a different sequence of "random" numbers, one needs to supply a different seed start value. By default, OMNeT++ takes care of assigning seed values. For example, by specifying `repeat=10` in the *INI* file you request 10 repetitions of the same simulation; all of them will run with different seed values. Listing 1.15 demonstrates how random numbers are generated in the source code.

**Listing 1.15** Generating random numbers

```
1 unsigned long k = intrand(10); // random integer from [0,9]
2 double d = dblrand(); // random double integer from [0,1)
```

OMNeT++ primarily uses the *Mersenne Twister* PRNG [5] for the generation of random numbers.[6] A configurable number of global random number streams are provided to the simulation. Global random number streams are mapped to module-local ones; module parameters and module source code consume random numbers from these module-local streams. The mapping from global streams to module-local ones can be configured/parametrized in a flexible way, allowing the use of variance reduction techniques and other "tricks" without the need to change anything in the actual simulation model.

Seeding is automatic, but it is also possible to use manually selected seeds in the configuration. The simulation requires as many seeds as the configured number of global RNG streams. Due to the practically infinite cycle length of the Mersenne Twister, overlapping RNG streams is not an issue in OMNeT++.

Random variate generation builds on top of the above-described random number architecture. Several distributions are available (cf. Listing 1.16 for an excerpt). Continuous ones include *uniform*, *exponential*, *normal*, *truncated normal*, *gamma*, *beta*, *Erlang*, *chi-square*, *Student-t*, *Cauchy*, *triangular*, *lognormal*, *Weibull*, and *Pareto*; discrete ones include *uniform*, *Bernoulli*, *binomial*, *geometric*, *negative binomial*, and *Poisson* distributions.

[6]The *Mersenne Twister* is the RNG class selected by default in OMNeT++, but two others are available as well (the LCG-32, a.k.a. the "default standard" RNG, and another one wrapping random numbers from the **Akaroa** library). It is also possible to write other RNG classes and select them from the *INI*-file, without changing anything in the simulation framework.

**Listing 1.16** Generating random variates

```
1 double x = exponential(10); // draw from exponential distribution with mean=10
2 double y = normal(0,1); // draw from the unit normal distribution
3 double z = normal(0,1,2); // same, using RNG 2 of the containing module
```

It is possible to add new distributions programmed by the user and make them available in the NED language and in the configuration (see Sect. 1.3.4). It is also possible to dynamically load distributions defined as histograms.

## *1.5.5 The Simulation Library*

### 1.5.5.1 Classes

Most classes in the OMNeT++ simulation library represent various parts of the component model: *modules*, *channels*, *gates*, *module parameters*, *objects*, and so on. Messages and packets are represented by `cMessage` class and its subclass, `cPacket`. A frequently used container class is `cQueue`, which can also be set up to operate as a priority queue. The simulation library contains a topology discovery class, `cTopology`, which can extract a network topology from the model according to the user's specification, make it available as a graph, and support algorithms such as *Dijkstra's* shortest path.

### 1.5.5.2 Ownership Tracking

Instances of several classes in the OMNeT++ class library, most notably `cMessage`, maintain pointers back to their owners. The owner is usually the module which has created or received the given message, a queue or other container object in a module, or the simulation kernel (more precisely, the future events list). The owner pointer allows the simulation kernel to catch common mistakes such as sending the same message object twice, sending out a message while it is sitting in a queue, or accessing a message which is being held by another module.

Ownership management is transparent for most of the time. The most frequent case when it needs manual help is when a module passes a message object to another module by means of a C++ method call; then, the target module explicitly needs to *take* the object from its current owner. Modules are *soft owners* and will yield to such requests, but if the owner is a queue for example, it is a *hard owner* and will raise an error instead.

Since modules maintain a list of owned objects, it is possible to recursively enumerate all objects in the simulation in a generic way, that is, without using pointer fields declared in simple module subclasses. This mechanism makes it possible for the user to inspect the simulation in the graphical runtime environment on object level, and to find leaked objects.

### *1.5.6 Representing Network Packets*

An important aspect of network simulation is the representation of network packets. In OMNeT++, packets are C++ classes derived from `cPacket`, which is in turn a subclass of `cMessage`. `cPacket`'s fields include the length of the packet, an error flag used to signal a corrupted packet, and a pointer to the encapsulated packet. The latter is used by the packet's `encapsulate()` and `decapsulate()` methods that are used when a message is passed up or down between protocol layers. These methods automatically update the length of the outer packet. The *encapsulated packet* pointer also gives an opportunity to OMNeT++ to reduce the number of packet object duplications by performing reference counting and copy-on-access on the encapsulated packet.

#### 1.5.6.1 The Message Compiler

In OMNeT++, messages and network packets are represented with C++ classes. With getter and setter methods for each field, a copy constructor, assignment operator, and a virtual `dup()` function (network packets are often copied or duplicated during simulation), plus hand-written reflection information needed for displaying packet contents in the graphical runtime Qtenv, it would be a time-consuming and tedious task to implement packet classes in plain C++. OMNeT++ takes the burden off the simulation programmers by providing a simple language (not unlike C structs with metadata annotation support) to describe messages, and the build system automatically generates C++ classes from them during the build process. Generic classes and `structs` may also be generated this way, not only packets and messages. If customizations are needed, the message compiler can be asked (via metadata annotations) to generate an intermediate base class only, from which the programmer can derive the final packet class with the necessary customizations. The success of the concept is proven by the fact that in modern OMNeT++ models practically all packet classes are generated. An exemplary message description is given in Listing 1.17:

**Listing 1.17** An exemplary *MSG* message description

```
1 // Represents a packet in the network.
2 packet SamplePacket
3 {
4    int srcAddr;
5    int destAddr;
6    int hopLimit = 32;
7 }
```

The **opp_msgtool** utility can be used to translate *MSG* files to C++, to convert, or to pretty-print them from the command line.

#### 1.5.6.2 Control Info

In OMNeT++, protocol layers are usually implemented as modules that exchange packets. However, communication between protocol layers often requires sending additional information to be attached to packets. For example, when a TCP implementation sends down a TCP packet to IP, it needs to specify the destination IP address and possibly other parameters. When IP passes up a packet to TCP after decapsulating it from an IP datagram, it will want to let TCP know at least the source and the destination IP addresses. This additional information is represented by *control info* objects in OMNeT++. Control info objects are attached to packets.

### *1.5.7 Wired Packet Transmission*

When modeling wired connections, packets are sent from the transmitter (e.g., the Medium Access Control (MAC) or Physical Layer (PHY) module) of one network node to the receiver of another node, via a connection path that contains exactly one channel object. Like modules, channels are programmable in C++ as well, and they are responsible for modeling propagation delay, calculating and modeling transmission duration, and performing the transmission error modeling. The default channel model, the `DatarateChannel`, performs a simple Bit Error Rate (BER) and/or Packet Error Rate (PER)-based error modeling. Error modeling sets a flag in the packet. It is then the responsibility of the receiver module to check this flag and act accordingly (e.g., drop a packet).

Normally, the packet object is delivered to the receiver module at the simulation time that corresponds to end of the reception of the packet. However, the receiver module may request that packets are delivered to it at the beginning of their reception, by "reprogramming" the receiver gate with an appropriate API call. The last transmission duration is available in a field of the packet object, and may be used by the receiver to determine how long the channel is to be considered busy.

### *1.5.8 Wireless Packet Transmission*

Due to diverse and often conflicting requirements, such as the level of detail versus performance tradeoffs, the OMNeT++ simulation kernel does not have a built-in mechanism for modeling the wireless channel and wireless packet transmissions. Instead, it is left to simulation model frameworks to implement such functionality using facilities of the OMNeT++ simulation kernel.

Wireless transmissions are usually implemented with sending the packets directly to the wireless nodes within range, using the aforementioned `sendDirect()` call. Usually, there is a separate dedicated module (for instance, the *channel controller*) for keeping track which nodes are within range of each

other, and which frequency they occupy. The packet (frame) may be encapsulated into a conceptual *air frame* which contains the physical properties of the radio transmission. The (air) frame object needs to be duplicated for each receiving node.[7] The task of modeling the wireless channel and the radio reception is shared between the channel controller and the actual destination node(s).

### 1.5.9 Dynamic Module Instantiation

Usually, all modules of a simulation are instantiated statically as part of the network setup procedure before the simulation begins, and destroyed when the simulation is terminated. However, it is also possible to dynamically instantiate (or delete) modules while the simulation is running. Of course, such operations may only be initiated programmatically, i.e., from a simple module or another active component (cf. Listing 1.18). Both simple and compound modules may be created at runtime, the latter will have its complete internal structure (submodules, connections) built out automatically as well.

Dynamic instantiation can be useful in a number of cases, for instance when the network topology is loaded from a file or generated by an algorithm at runtime, or when the network is changing dynamically (e.g., mobile devices are constantly arriving or leaving the playground, network links are repeatedly being added or cut).

**Listing 1.18** All-in-one module instantiation

```
1 // find a module type, and instantiate it as a child of our parent module
2 cModuleType *moduleType = cModuleType::get("nodes.WirelessNode");
3 cModule *mod = moduleType->createScheduleInit("node", getParentModule());
```

### 1.5.10 Signals

The OMNeT++ simulation library contains a built-in notification mechanism, which allows for publish-subscribe style communication between simulation components among many other uses. Components (modules and channels) can emit notifications termed *signals*, and other modules (or arbitrary pieces of code) can subscribe to them. Signals are identified by names, but for efficiency, calls use dynamically assigned numeric signal identifiers. Names and identifiers are globally valid in the whole simulation. Listing 1.19 shows an exemplary signal definition:

[7] Duplicating all protocol layers encapsulated in the frame would be a waste of CPU cycles because in a wireless network, most frames are immediately discarded by the receiver due to incorrect reception or wrong destination MAC address. Hence, OMNeT++ uses reference counting on encapsulated packets and only duplicates them if needed, that is, when they actually get decapsulated in a higher layer protocol module.

**Listing 1.19** Declaring signals in the NED language

```
module Queue {
    @signal[queueLength](type=long);
    @signal[queueingTime](type=simtime_t);
}
```

Signals propagate upwards in the module hierarchy. At any level, one can register callbacks called *listeners* which will be notified (called back) whenever a signal is emitted. Compare Listing 1.20 for an exemplary registering process and Listing 1.21 for the opposed listening process.

**Listing 1.20** Registering and emitting a signal in the C++ source

```
simsignal_t queueLengthSignal; // class member
queueLengthSignal = registerSignal("queueLength"); // into initialize()
emit(queueLengthSignal, queue.getLength()); // into handleMessage()
```

**Listing 1.21** Listening on a signal

```
class NoisyListener : public cListener { // listener class
protected:
  virtual void receiveSignal(cComponent *source, simsignal_t signalID,
                       long value, cObject *details) override
  {
    const char *signalName = cComponent::getSignalName(signalID);
    EV << "Received " << value << " on signal " << signalName << endl;
  }
};
// into initialize():
simsignal_t queueLengthSignal = registerSignal("queueLength");
subscribe(queueLengthSignal, new NoisyListener());
```

The significance of upwards propagation is that listeners registered at a certain module will receive signals from all components in that submodule tree. Listeners registered at the top level will receive signals from the whole simulation. Since a module can register listeners at any other module, it can get notified about events anywhere it wishes. For example, a simple module representing a routing protocol may register a listener for the hypothetical *INTERFACE_UP* and *INTERFACE_DOWN* signals at the parent compound module that represents the router and initiate action to update the routing tables accordingly.

When a signal is emitted, it can carry a value with it. The value can be of a basic type (e.g., long, double, string, etc.) or a pointer to an arbitrary object. Objects can be already existing objects, or ones specially crafted for the purpose of emitting the signal. Computing the signal value or propagating the signal may cost valuable CPU cycles, so the signal mechanism was implemented in a way that helps avoid emitting or further propagating signals for which there are no listeners.

Simulation signals can be used for several purposes, in particular:

- to implement publish-subscribe style communication among modules; it is advantageous when the producer and consumer of the information do not know

about each other, and there is, possibly, a many-to-one or a many-to-many relationship between them;
- when some module needs to be notified about simulation model changes such as module creation and deletion, connection creation and deletion, parameter changes, and so on. Such signals, both pre- and post-change ones, are emitted by the OMNeT++ simulation kernel, with attached objects that contain the details of the change;
- for emitting variables to be recorded as simulation results, for example queue lengths, packet drops, or End-to-End (E2E) delays. Then, it is up the simulation framework to add listeners which record the selected data in some form;
- for emitting animation primitives or auxiliary information that can be used by an animation engine;
- for emitting Packet Capture (PCAP) traces that can be captured and written into a file by a dedicated module(s) or the simulation framework.

## *1.5.11 Statistical Result Collection*

### 1.5.11.1 Scalar and Vector Results

OMNeT++ distinguishes three types of results: *scalars*, *vectors*, and *statistics*. A scalar is a single number; vectors are timestamped time series; and statistics are records composed of statistical properties (mean, variance, minimum, maximum, etc.; possibly also histogram data) of time series. Recording of individual vectors, scalars, and statistics can be enabled or disabled via the configuration (*INI* file), and it is also the place to set up the recording intervals for vectors. Simulation results are recorded into textual, line-oriented *scalar files* (which actually hold statistics results as well) and *vector files*. The advantage of a text-based format is that it is very accessible for 3rd party tools. The file format is well specified, extensible, and open for other simulators to adapt. There are standalone implementations available for recording and for loading the format.

Result files are self-describing: they contain many attributes of the simulation run: the network, experiment-measurement-replication labels, iteration variables, time/date, host, process ID of the simulation, etc. By default, one file contains data from only one run. Vectors are recorded into a separate file for practical reasons: vector data usually consume several magnitudes more disk space than others. The vector file contains data clustered by vectors indexed for efficient access. This allows for extracting certain vectors from the file, and even near random access within vectors, without having to read the full contents of the vector file even once.

#### 1.5.11.2 Declarative Result Recording

Declarative result recording is based on two pillars: signals and NED properties. Signals, emitted from modules, act as data source, and NED properties contain the declarations of which signals to record and in what form (cf. Listing 1.22). One is able to record a particular variable as a vector, as a statistic (mean, variance, histogram bins, etc.), or to record only a single property of the variable (mean, time average, count, maximum value, etc.) as a scalar. The signal framework also allows to implement aggregate statistics (e.g., total number of packet drops in the network) and warmup periods (ignoring an initial time interval when computing scalars or statistics). Signals also allow the user to employ dedicated statistics collection and aggregation modules in the simulation, without the need to change existing modules.

**Listing 1.22** Declared statistics

```
@signal[qlen](type=long);
@signal[qtime](type=simtime_t);
@statistic[queueLength](title="queue length";
                  source=qlen; record=vector,timeavg,max;
                  interpolationmode=sample-hold);
@statistic[queueingTime](title="queueing time";
                  source=qtime; record=histogram,vector?;
                  unit=s; interpolationmode=none);
```

#### 1.5.11.3 Programmatic Result Recording

An alternative to using signals and `@statistic` is to collect and record results from the C++ model code. Then, one would keep intermediate results in class variables inside modules and record them during the finalization phase (`finish()` method) using C++ calls (cf. Listing 1.23 for an example). Counts or totals can be simply collected in plain `int`, `long`, or `double` variables. If summary statistics such as mean, standard deviation, min/max, etc. are needed, a `cStdDev` object can be used. To also record a histogram, the `cHistogram` class is provided by OMNeT++. `cHistogram` is highly configurable, but most of the time it does not need to be configured at all (cf. Listing 1.24), because it automatically creates a good histogram due to built-in techniques such as *precollection*, *auto-extension*, and *bin merging*. Alternatives to `cHistogram` are `cPSquare` and `cKSplit`, both giving up some accuracy for being able to produce a higher quality histogram without having a priori information about the distribution. Output vectors (time series) can be recorded using `cOutVector` objects.

**Listing 1.23** Recording scalars, vectors, and histograms from C++ code

```
class ResultRecording : public cSimpleModule {
   int packetCount = 0;
   cHistogram histogram;
   cOutVector outputVector;
 protected:
   virtual void initialize() override;
```

```
7     virtual void handleMessage(cMessage *msg) override;
8     virtual void finish() override;
9  };
10
11 void ResultRecording::initialize() {
12    histogram.setName("packetLength");
13    outputVector.setName("packetLength");
14 }
15
16 void ResultRecording::handleMessage(cMessage *msg) {
17    packetCount++;
18    int64_t packetLength = check_and_cast<cPacket*>(msg)->getByteLength();
19    histogram.collect(packetLength);
20    outputVector.record(packetLength);
21    delete msg;
22 }
23
24 void ResultRecording::finish () {
25    recordScalar("packetCount", packetCount);
26    histogram.record();
27 }
```

**Listing 1.24** Exemplary collection of data into a histogram

```
1 cHistogram hist("histogram");
2 for (int i = 0; i < 1000000; i++) {
3    double value = normal(0,1);
4    hist.collect(value);
5 }
6 hist.recordAs("demoHistogram");
```

## *1.5.12 Graphics and Animation*

### 1.5.12.1 Display Strings

Display strings are a simple way to customize the default OMNeT++ animation. They are `@display` NED properties that can be placed in module definitions and in submodules as well. The basic use case is to assign icons and positions to submodules, like it is demonstrated in the following code example:

**Listing 1.25** Assigning an icon to a module type, and position it as a submodule in the network

```
1 module Router {
2    @display("i=abstract/router"); // "i" tag sets the icon
3    //...
4 }
5
6 network Network {
7    submodules:
8       router1: Router { @display("p=50,50"); } // "p" tag sets the position
9       //...
```

Other possibilities include using a geometric shape (rectangle or oval) instead of an icon, displaying additional "decoration" elements like a small status icon or a status text near the icon, colorizing the icon, or drawing a circle or disc around the icon to indicate the transmission range. Display strings may also be manipulated at runtime. The following C++ source code example alters the module's display string to make the module icon 50% red.

**Listing 1.26** Programmatically updating a display string

```
1 getDisplayString().setTagArg("i", 1, "red");
2 getDisplayString().setTagArg("i", 2, "50");
```

#### 1.5.12.2 The Canvas

When you reach the limits of display strings, the next level is to use the *Canvas API*. The Canvas API is the figure-based 2D drawing API of OMNeT++. It facilitates enriching simulations with graphical elements when viewed under a graphical user interface like Qtenv. Every module has one canvas by default (the same area where submodules are also visualized). Additional ones can also be created, so there can be a virtually unlimited number of canvases.

Canvases hold items called *figures*. Many figure types are available, including various shapes, text, and image, including an SVG-like path item (a generalized polygon/polyline, with arcs and Bezier curves). Transformations (scaling, rotation, and skewing) are also supported, as well as transparency. Figures can be nested or grouped to create compound images that can be moved, scaled, or rotated together as one unit. Figures can also be created statically from NED files using `@figure` properties as well as dynamically from C++ code. Statically created figures may also be accessed and dynamically manipulated at runtime.

**Listing 1.27** Adding a rounded, filled, semitransparent, rotated rectangle to the canvas via a definition in the NED description

```
1 network Canvas {
2    @figure[greenRect](type=rectangle;bounds=50,10,160,100;cornerRadius=5;
         fillColor=green;fillOpacity=0.5;transform=rotate(15));
```

When creating figures from C++ code, the key classes are `cCanvas` and `cFigure`. `cFigure` is an abstract base class, and concrete figure types are subclasses of `cFigure`. The following code in Listing 1.28 shows how to create and set various graphical attributes of a figure from within the C++ source code. The subsequent result is shown in Fig. 1.5.

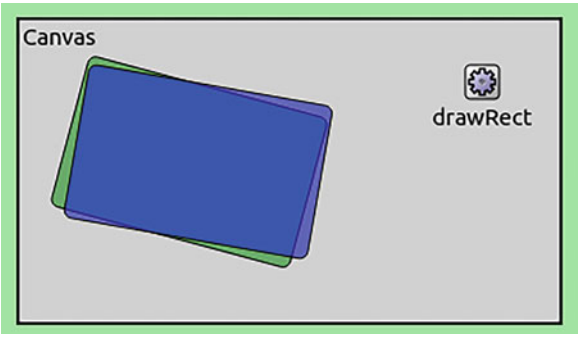



**Fig. 1.5** The resulting two rectangles

**Listing 1.28** Adding another rounded rectangle, this time via C++ source code

```
cRectangleFigure *rect = new cRectangleFigure("blueRect");
rect->setBounds(cFigure::Rectangle(50,20,160,100));
rect->setCornerRadius(5);
rect->setFilled(true);
rect->setFillColor(cFigure::BLUE);
rect->setFillOpacity(0.5);
rect->rotate(10 * M_PI/180);
getSystemModule()->getCanvas()->addFigure(rect);
```

Canvas drawing use cases are limitless. For example, in mobile and wireless simulations, the canvas API can be used to draw the scene including a background (like a street map or a floor plan), mobile objects (vehicles, people), obstacles (trees, buildings, and hills), antennas with orientation, and extra information like a connectivity graph, movement trails, visualization of individual transmissions, and much more. In other simulations, the canvas API can be used to display textual annotations, status information, live statistics in the form of plots, charts, gauges, counters, and so on.

#### 1.5.12.3 Refreshing

When programming a simulation with dynamic visualization, one might wonder what is the best place for code that updates display strings and maintains canvas figures and 3D scene(s). `handleMessage()` is usually a poor choice because it does not know when and how often the graphical user interface actually refreshes the application window, so it would often work needlessly. A better option is the `refreshDisplay()` method.

`refreshDisplay()` is expressly intended to serve as a container for visualization-related code. It is invoked from graphical user interfaces like Qtenv on demand, whenever GUI contents need to be refreshed. Cmdenv does not invoke this method at all. Moving display string updates and canvas figures maintenance into `refreshDisplay()` can result in significant performance gain and, in some cases, also in more consistent information being displayed.

#### 1.5.12.4 Smooth Animations

When running a simulation in Qtenv, events are usually processed as fast as possible, and execution only pauses momentarily to give built-in animation effects, like message sending, time to play out. However, it is also possible to have the simulation run at (scaled) real-time and meanwhile perform smooth animation, regardless when and how often simulation events occur. This features allows one to properly animate the movement of a mobile node or a signal wavefront, regardless of how many simulation events are there in those periods.

The key to enable such smooth animation is to set an *animation speed* on the canvas, using its `setAnimationSpeed()` method. This will put the GUI into the "scaled real-time" operation mode (where the scale is the animation speed), and causes `refreshDisplay()` to be called repeatedly at a reasonable rate to allow code in it to render frames. The actual playing speed of the animation can be modified interactively using the speed slider on the Qtenv toolbar.

**Listing 1.29** An example of a smooth animation of a sun symbol moving along a sine wave, independent of simulation events

```
1  void AnimatorModule::initialize() {
2     cCanvas *canvas = getSystemModule()->getCanvas();
3     imageFigure = new cImageFigure(); // imageFigure should be be a class member
4     imageFigure->setImageName("misc/sun");
5     canvas->addFigure(imageFigure);
6     canvas->setAnimationSpeed(10, this); // animate at a speed 10x real time
7     scheduleAt(simTime(), new cMessage()); // start events too (not shown)
8  }
9
10 void AnimatorModule::refreshDisplay() const {
11    double t = simTime().dbl();
12    double x = fmod(10*t,500), y = 100 + 50*sin(t/2);
13    imageFigure->setPosition(cFigure::Point(x, y));
14 }
```

#### 1.5.12.5 3D Graphics

OMNeT++ allows simulations to be animated using advanced 3D-based graphics. Support for 3D graphics is facilitated through the inclusion and use of the open source OpenSceneGraph (OSG)[8] and osgEarth[9] libraries.

OpenSceneGraph is an OpenGL-based high-performance 3D graphics toolkit, used by application developers in fields such as visual simulation, games, virtual reality, scientific visualization, and modeling. osgEarth, in turn, is a geospatial Software Development Kit (SDK) and terrain engine built on top of OpenSceneGraph, not quite unlike Google Earth. osgEarth can use a variety of map providers, satellite imaging providers, and elevation data sources. It can also download such data for

[8] OpenSceneGraph (OSG) project website: http://www.openscenegraph.org/.

[9] osgEarth website: http://osgearth.org/.

**Fig. 1.6** Visual output of the hang-glider animation example

offline use, which is especially useful if the simulation itself depends on elevation or other data obtained from osgEarth.

Using OSG and osgEarth, one can visualize terrain, roads, urban street networks, indoor environments, satellites, and more. In a vehicular network simulation, people, cars, and aircraft can be visualized (cf. Fig. 1.6). For simulations of wireless networks, one can create a scene that, in addition to the faithful representation of the physical world, also displays the transmission range of wireless nodes, their connectivity graph, various statistics, and shows individual wireless transmissions with an animation effect.

OMNeT++ basically exposes the OpenSceneGraph API. One needs to assemble an OSG scene graph in the model, and give it to OMNeT++ for display. The rendered scene will appear in the Qtenv main window. The scene graph can be updated at runtime, and changes will be reflected in the display. When a scene graph has been built by the simulation model, it needs to be given to a `cOsgCanvas` object to inform the OMNeT++ GUI about it. `cOsgCanvas` wraps a scene graph, plus hints for the GUI on how to best display the scene (e.g., the default camera position). In the GUI, the user can use the mouse to manipulate the camera to view the scene from various angles and distances, look at various parts of the scene, etc.

The following two code fragments exemplify the basic use of the OSG and osgEarth APIs. The result of running the simulation under Qtenv is shown in Fig. 1.6.

**Listing 1.30** Example of displaying a 3D interactive earth globe with OpenStreetMap

```
1 // #include <osgDB/ReadFile>
2 cOsgCanvas *osgCanvas = getSystemModule()->getOsgCanvas();
3 osgCanvas->setViewerStyle(cOsgCanvas::STYLE_EARTH);
4 osg::Node *scene = osgDB::readNodeFile("openstreetmap.earth");
5 osgCanvas->setScene(scene);
```

**Listing 1.31** Example continued: a huge hang-glider above Budapest

```
// #include <osgEarth/MapNode>
// #include <osgEarthUtil/ObjectLocator>
double lat = 47.500, lon = 19.047;
auto mapNode = osgEarth::MapNode::findMapNode(scene);
auto locatorNode = new osgEarth::Util::ObjectLocatorNode(mapNode->getMap());
auto modelNode = osgDB::readNodeFile("glider.osgb.50.scale");
locatorNode->addChild(modelNode);
mapNode->getModelLayerGroup()->addChild(locatorNode);
locatorNode->getLocator()->setPosition(osg::Vec3d(lon, lat, 100));
locatorNode->getLocator()->setOrientation(osg::Vec3d(0, 0, 0));
osgCanvas->setEarthViewpoint(cOsgCanvas::EarthViewpoint(lon, lat, 50, 0, -90,
      500));
```

## 1.6 Advanced Usage

Next to the basic functionalities introduced in the previous sections, OMNeT++ provides support for a number of advanced functions like cloud computing support, debug checkpointing, and parallel and co-simulation. These and other advance functions are introduced in the following subsections.

### *1.6.1 Swarms and Cloud Computing*

With parameter studies, it is easy to reach the point of combinatorial explosion, and then even counting with short simulation runs, one easily arrives at a point where the total simulation campaign would require days or weeks to complete on a single computer. However, CPU power is available nowadays in abundance in cloud computing services, at very affordable prices. There are numerous cloud services (e.g., Amazon AWS, Microsoft Azure, DigitalOcean, Google Cloud Platform, etc.). These services, following a registration, allow users to run their code on a high number of CPUs, often at surprisingly low prices.

At the same time, virtualization, and especially container technology like Docker, has matured tremendously. Docker, via Docker images, makes packaging and deployment of software, including simulation programs, very convenient on cloud nodes. Docker Swarm Mode, which acts as a layer above Docker that adds cluster management and orchestration features, is also very useful for running simulation campaigns, and can be used in cloud services like AWS and also to make use of local computing resources.

Docker and cloud computing significantly lower the entry barrier compared to grid computing characterized by middleware such as **HTCondor** and **Oracle Grid Engine** and offer a lot more power than traditional setups like cross-mounted Network File System (NFS) directories combined with Secure Shell (SSH) execution.

Significant work has been done to allow simulations and simulation campaigns to be easily deployable in cloud services like AWS and in local clusters using Docker Swarm. At the time of writing, after a relatively painless setup procedure, INET simulations can be run with a single command analogous to **opp_runall** on a local Docker Swarm or on AWS, and on completion, simulation results appear in the local *results* directory. The code and instructions are available on the OMNeT++ website as a tutorial.[10] Further work is expected in this area, and it is possible that similar features will become integral parts of the OMNeT++ software as well.

### *1.6.2 Checkpointing*

Debugging long-running simulations can be challenging, because one often needs to run the simulation for a long time just to get to the point of failure and be able to start debugging. Checkpointing can facilitate debugging such error cases. It is a technique that basically consists of saving a snapshot of the application's state, and being able to resume execution from there, even multiple times. OMNeT++ itself contains no checkpointing functionality, but it is available via external tools. It depends on the tool whether it is able to restore GUI windows (usually not the case). The following list contains checkpointing software available on Linux:

- Berkeley Lab Checkpoint/Restart (BLCR),
- DMTCP (Distributed MultiThreaded Checkpointing),
- CRIU is a user space checkpoint lib,
- Docker and its underlying technology have a checkpoint and restore mechanism.

### *1.6.3 Running Multiple Replications in Parallel*

A specialized way to use clusters is **Akaroa** [2, 8]. It is an implementation of the Multiple Replications In Parallel (MRIP) principle, which can be used to speed up steady-state simulations. **Akaroa** runs multiple instances of the same simulation program (but with different seeds) simultaneously on different processors, e.g., on nodes of a computing cluster, and a central process monitors certain output variables of the simulation. When **Akaroa** decides that it has enough observations to form an estimate of the required accuracy of all variables, it halts the simulation. When using $n$ processors, simulations need to run only roughly $1/n$ times the required sequential execution time. **Akaroa** support is integrated into OMNeT++.

[10] OMNeT++ tutorial website: https://docs.omnetpp.org/.

### 1.6.4 Emulation, Real-Time, and Hardware-in-the-Loop Support

OMNeT++ provides a facility to replace the event scheduler class with a custom one, which is the key for many features including co-simulation, real-time simulation, network or device emulation, and distributed simulation. The event scheduler's job is to always return the next event to be processed by the simulator. The default implementation returns the first event in the FEL. For real-time simulation, this scheduler is replaced with one augmented with *wait* calls (e.g., `usleep`) that synchronize the simulation time to the system clock. Different options are available when the simulation time has fallen behind: one may re-adjust the reference time, leave it unchanged and hope to catch up later, or stop with an error message.

For emulation, the real-time scheduler is augmented with code that captures packets from real network devices and inserts them into the simulation. The INET Framework contains an emulation scheduler and uses PCAP to capture packets, and raw sockets to send packets to the real network device. Emulation in INET also relies on *header serializer* classes that convert between protocol headers and their C++ object representations used within the simulation. The emulation feature has been successfully used to test the interoperability of INET's Stream Control Transmission Protocol (SCTP) model with real-life SCTP implementations [9].

### 1.6.5 Co-simulation Support: HLA, SystemC, and TraCI

OMNeT++ supports distributed simulation using HLA[11] (IEEE 1516 [4]) as well. The OMNeT++ scheduler plays the role of the HLA *Federate Ambassador*, is responsible for exchanging messages (interactions, change notifications, etc.) with other federates, and performs time regulation. OMNeT++ also supports mixing SystemC (IEEE 1666-2005 [3]) modules with OMNeT++ modules in the simulation. When this feature is enabled, there are two FELs in the simulation, OMNeT++'s and SystemC's, and a special scheduler takes care that events are consumed from both lists in increasing timestamp order. This method of performing mixed simulations is orders of magnitude faster and also more flexible than letting the two simulators execute in separate processes and communicate over a pipe or socket connection.

Another example of co-simulation is Veins, a framework for simulating intervehicle communication (see Chap. 6). Veins integrates the SUMO road traffic simulator with OMNeT++: SUMO simulates the movement of vehicles and an OMNeT++ model is responsible for simulating the communication among vehicles. With Veins, each simulation is performed by executing the two simulators in parallel, in separate

[11] The source code for the HLA and SystemC integration features is not open source. Nevertheless, it is available for researchers free of charge on request.

processes. The SUMO process and the OMNeT++ simulation process communicate over a TCP socket, using a protocol known as Traffic Control Interface (TraCI). TraCI enables the bidirectionally coupled simulation of road traffic and network traffic. The movement of vehicles in SUMO is reflected as node movement in the OMNeT++ simulation. Nodes can then interact with the running road traffic simulation, e.g., to simulate the influence of the communication on the road traffic.

### *1.6.6 Parallel Simulation Support*

Parallel Discrete-Event Simulation (PDES) refers to the execution of a single discrete event simulation on a parallel computer. Both shared-memory multiprocessors and computing clusters can be used for PDES, but the former are preferred because of the sensitivity of the PDES algorithm to the messaging latency among CPUs.

For parallel execution, the model is to be partitioned into several Logical Processes (LPs) that will be simulated independently on different processors. Each LP will have its own local FES and thus will maintain its own local simulation time. The main challenge of parallel simulations is keeping LPs synchronized in order to avoid violating the causality of events. Without synchronization, a message sent from one LP could arrive in another LP when the simulation time in the receiving LP has already passed the timestamp of the message, breaking the causality of events in the receiving LP. There are two broad categories of parallel simulation algorithms that differ in the way they handle causality problems outlined above. *Conservative* synchronization algorithms prevent incausalities from happening, while *optimistic* ones allow incausalities to occur but detect and repair them.

OMNeT++ implements conservative parallel simulation via the *Chandy-Misra Null Message* algorithm [1, 7]. For parallel simulation, the scheduler is modified to listen for messages arriving from other LPs, and inserts them into the simulation. The scheduler also blocks the simulation when it is not safe to execute the next event due to potential causality violation, until clearance arrives from other LPs to continue in the form of a null message.

Due to messaging and especially synchronization overhead, parallel simulation is not guaranteed to provide any speedup. In unfortunate cases, PDES might execute magnitudes slower than sequential simulation. Success depends on two factors: the messaging latency among processors and the parallelism available in the model (lookahead). For example, wireless simulations usually perform poorly under PDES. A formula [10] has been developed to help determining the chance of success for parallelization of a particular simulation. The formula uses easy-to-measure quantities as input. It is documented in the OMNeT++ manual.

### 1.6.7 Custom Result Recording

The OMNeT++ simulation kernel allows to replace the standard way of recording simulation results into traditional OMNeT++ *.sca* and *.vec* files with custom, user-defined recorders. Alternative recorders which produce **SQLite** database files are provided with recent OMNeT++ releases, but given the user requirements, anyone can implement their own custom recorders.

### 1.6.8 Embedding the Simulation Kernel

The OMNeT++ simulation kernel is a C++ library that can be added into custom applications to provide simulation capabilities. This has been demonstrated in real-life applications several times. Models can be developed and tested in the simulation IDE, then compiled into the application in an unchanged form. The OMNeT++ core itself is modular, and you can choose which parts to keep and which parts to replace. For example, you can choose whether to keep *INI* files as means to configure simulations, or implement a custom configuration provider.

### 1.6.9 IDE Extensibility

The simulation IDE is based on **Eclipse**. **Eclipse** is most known as a programming language IDE, but it is also an integration platform for all sorts of developer-oriented applications. In the simulation IDE, the **Eclipse** platform provides the workbench infrastructure and workspace handling (projects, etc.). Additional plug-ins written by the OMNeT++ team provide simulation-related capabilities. C++ development and Git support come from **Eclipse** projects. Other features (Unified Modeling Language (UML) modeling, bug tracker integration, LaTeX editing, **Python** development, database access, etc.) can be installed into **Eclipse** from the **Eclipse Marketplace**.[12]

In the same fashion, it is also possible for 3rd party developers to extend the OMNeT++ IDE and add features like project-specific topology generation, network scenario generation, and the like. For this reason, the OMNeT++ **Eclipse** plug-ins are well documented, as they provide a public API and expose extension points where new functionality can be plugged in or existing functions can be customized. To further simplify the deployment of plug-ins, the simulation IDE loads plug-ins not only from the installation directory but from user projects as well. This makes it possible to bundle simulation framework-specific plug-ins (such as the INET

[12] Eclipse Marketplace website: http://marketplace.eclipse.org/.

Framework) with the simulation frameworks, so when a user imports the simulation framework into the IDE, the framework-specific UI contributions will immediately appear in the IDE.

The IDE also makes it possible to write wizards without any Java or C++ programming, using an XML-based UI description language and a template language for content generation. This feature offers a relatively quick and painless way to add topology generators and file importers into the IDE.

## 1.7 Conclusion

With the presented features of OMNeT++ you have a versatile toolset for developing your own discrete event-based simulations, with or without the support of existing frameworks such as the ones presented subsequently in Part II in this book.

## References

1. Chandy, M., Misra, J.: Distributed simulation: a case study in design and verification of distributed programs. IEEE Trans. Softw. Eng. **SE-5**(5), 440–452 (1979). https://doi.org/10.1109/TSE.1979.230182
2. Ewing, G., Pawlikowski, K., McNickle, D.: Akaroa2: exploiting network computing by distributing stochastic simulation. In: Proceedings of the European Simulation Multiconference (ESM'900), pp. 175–181. International Society for Computer Simulation, San Diego (1999)
3. IEEE Standards Association: IEEE Standard System C Language Reference Manual. IEEE Std 1666-2005, IEEE Standards Association (IEEE-SA), Piscataway (2006). https://doi.org/10.1109/IEEESTD.2006.99475
4. IEEE Standards Association: IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules. IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000), IEEE Standards Association (IEEE-SA), Piscataway (2010). https://doi.org/10.1109/IEEESTD.2010.5553440
5. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Trans. Modeling Comput. Simul. **8**(1), 3–30 (1998)
6. SCS Technical Committee: Terminology for model credibility. Simulation **32**(3), 103–104 (1979). https://doi.org/10.1177/003754977903200304
7. Sekercioglu, Y.A., Varga, A., Egan, G.K.: Parallel simulation made easy with OMNeT++. In: Proceedings of the 15th European Simulation Symposium (ESS) (2003)
8. Sroka, S., Karl, H.: Using Akaroa2 with OMNeT++. In: Proceedings of the 2nd International OMNeT++ Workshop, pp. 43–50 (2002)
9. Tüxen, M., Rüngeler, I., Rathgeb, E.P.: Interface connecting the INET simulation framework with the real world. In: Proceedings of the 2nd Conference on Simulation Tools and Techniques. ICST (2008)
10. Varga, A., Sekercioglu, Y.A., Egan, G.K.: A practical efficiency criterion for the null message algorithm. In: Proceedings of the 15th European Simulation Symposium (ESS) (2003)

# Part II
# The OMNeT++Ecosystem

# Chapter 2
# INET Framework


**Levente Mészáros, Andras Varga, and Michael Kirsche**

## 2.1 Introduction

The INET Framework is one of the oldest and the single largest collection of simulation models for OMNeT++. Frameworks, in general, provide a number of simulation models for specific application fields to extend the generic and application-independent OMNeT++ discrete-event simulator. INET can be seen as a standard communication protocol library that facilitates the simulation of communication network scenarios. It has a long and intertwined history with OMNeT++: from its starting point—the OMNeT++ Internet Protocol Suite—back in 2000, over its renaming and recreation as the INET Framework in 2004, to the releases of INET versions 4 and 4.1 covered in this book chapter. INET has grown and pushed the evolution of OMNeT++ just like OMNeT++ has constantly influenced INET's development.

INET is designed to be suitable for research and experimentation in the field of communication network simulation. It is especially useful when exploring new scenarios or designing and validating new communication protocols from all layers of the protocol stack. INET's target audience includes academic researchers, post-doctorates, Ph.D. as well as undergrad students. The framework is particularly well applicable to educational purposes as it is easy to learn and experiment with. INET is widely used by many reputable universities throughout the world. INET

L. Mészáros (✉) · A. Varga
Opensim Ltd, Budapest, Hungary
e-mail: levy@omnetpp.org; andras@omnetpp.org

M. Kirsche
Brandenburg University of Technology Cottbus-Senftenberg, Cottbus, Brandenburg, Germany
e-mail: michael.kirsche@b-tu.de



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_2

is furthermore also used by many large and widely known industrial users for commercial purposes.

INET contains an extensive number of models. The project has started out as a model suite for the Internet protocol stack (e.g., IPv4, TCP, and UDP), and has gradually expanded to other areas, in particular: Ethernet, IEEE 802.11, QoS mechanisms, MPLS, DiffServ, Internet routing protocols, node mobility, MANET routing protocols, further wireless MAC protocols, IPv6, and SCTP, just to name a few. Recent advancements include physical layer and physical medium modeling, advanced visualization features, and improved network emulation capabilities. INET is useful for the simulation of a wide range of networks, from backbones to data centers, mobile ad hoc to (wireless) sensor networks. Later in this chapter, separate sections are devoted to the simulation of various classes of networks.

INET is an open-source project with many contributors and a large and active community. Other frameworks like INETMANET (see Chap. 3), SimuLTE (see Chap. 5), or Veins (see Chap. 6) take INET as a base for their own developments in specific application areas. The INET source code is freely available on Github[1] for anyone to modify and use. The license of most components is GNU Lesser General Public License (LGPL), which is particularly suitable in academic settings.

## 2.2 Assembling Simulations

INET is an extensive and complex framework that uses OMNeT++'s inherent modularity. The majority of INET's components and models can be combined in various ways and adjusted/parameterized according to the requirements of the simulation scenario. Assembling a simulation scenario with INET is therefore feasible and complex at the same time because of the diversity and pure size of the model library. Users have a wide range of models available with a large number of adjustable parameters. On the other hand, the sheer amount of model and configuration options and the incorporation of all components into a running simulation can be overwhelming for new users. Luckily, INET facilitates the construction of simulation scenarios without requiring programming experience if changes to existing components are only made in terms of parametrization. Users can start to assemble small configurations and scenarios and extend or reuse them to build large-scale network simulations by composing the existing models. This section discusses the process of assembling a simulation scenario. The range of available simulation modules in INET is introduced along the various steps of creating a complete full-stack simulation scenario.

A first step for new INET users is to get accustomed with the directory structure of the framework. The source code of the actual simulation models is contained within the *src/* folder. Example simulation scenarios (located in the *examples/*

[1] INET Github repository: https://github.com/inet-framework/inet.

directory), showcases (located in the *showcases/* directory), and tutorials (located in the *tutorials/* directory) provide a wide range of starting points for the user's own simulations. Documentation is found in the *doc/* folder, but it can also be browsed online.

Inside *src/*, the source code of components is organized into folders that correspond to Network Topology Description (NED) packages. For example, the *src/inet/node/inet/* folder corresponds to the `inet.node.inet` package. Packages are organized along Open Systems Interconnection (OSI) layers (physical, data link, network, transport, and application), with a few extra folders for components that do not fit into the OSI model (mobility models, energy modeling, physical environment, visualization, and common utility components, among others).

### 2.2.1 Network Definition

Every OMNeT++ (and thus every INET) simulation requires a special compound module (cf. Chap. 1) that represents the actual network to be simulated. This network definition assembles all simple and compound modules that are present in the simulation. An example of a wired INET network is depicted in Fig. 2.1. It contains different network nodes (router, switch, client, and server), a support module to automate the network configuration, and cable connections between the nodes. Complex networks can be structured and organized through compound modules, nesting smaller networks into hierarchical larger ones. OMNeT++ also provides wizards to generate topologies or to import them from various file formats. Such networks can be assembled using the OMNeT++ Integrated Development Environment (IDE), using the graphical editor or editing the NED source shown in Listing 2.1.

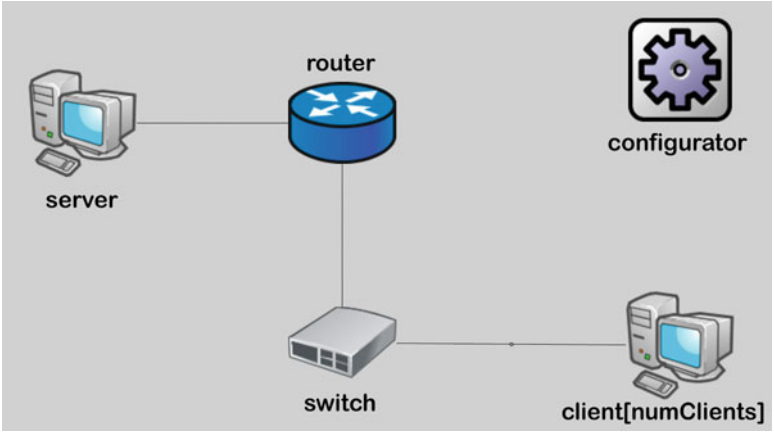



**Fig. 2.1** A simple wired INET network with hosts, router, switch, and network configurator module

**Listing 2.1** NED source code of the wired INET network example

```
import inet.networklayer.configurator.ipv4.Ipv4NetworkConfigurator;
import inet.node.inet.Router;
import inet.node.inet.StandardHost;
import inet.node.ethernet.EtherSwitch;
import ned.DatarateChannel;

network WiredNetworkExample
{
    parameters:
        int numClients; // number of clients in the network
    submodules:
        configurator: Ipv4NetworkConfigurator;
        server: StandardHost;
        router: Router;
        switch: EtherSwitch;
        client[numClients]: StandardHost;
    connections: // network level connections
        router.pppg++ <--> { datarate = 1GBps; } <--> server.pppg++; // PPP
        switch.ethg++ <--> Eth1G <--> router.ethg++; // ethernet
        for i=0..numClients-1 {
            client[i].ethg++ <--> Eth1G <--> switch.ethg++; // ethernet
        }
}
```

The NED source defines a network named `WiredNetworkExample`. Inside the definition, the `submodules` section lists the elements of the network (e.g., switches, and routers) and support modules (e.g., configurators, and visualizers). `StandardHost`, `Router`, and `EtherSwitch` are some of the pre-assembled module types that INET provides. The content of the `connections` section describes the "cables" between the wired hosts in the example. The model above uses various parameterized instances of OMNeT++'s existing `DatarateChannel` channel model. Other elements of the NED file import the necessary types (`import` lines at the top), define the number of client nodes as a parameter for the network (`parameters` section), and define a channel type for local use within the network (`types` section).

The above network needs to be configured before it can be simulated. The configuration is given in *INI* files, usually named *omnetpp.ini*. The *INI* file contains the name of the network to be simulated as well as its parameters. Although practically all modules in INET have parameters, most have suitable defaults, so usually only a small subset of parameters need to be explicitly given in the *omnetpp.ini* file. A suitable configuration for the above network is given in Listing 2.2.

**Listing 2.2** Wired network configuration example

```
network = WiredNetworkExample
*.numClients = 10 # number of clients in network
*.client[*].numApps = 1 # number of applications on clients
*.client[*].app[0].typename = "TcpSessionApp" # client application type
*.client[*].app[0].connectAddress = "server" # destination address
*.client[*].app[0].connectPort = 1000 # destination port
*.client[*].app[0].sendBytes = 1MB # amount of data to send
*.server.numApps = 1 # number of applications on server
*.server.app[0].typename = "TcpEchoApp" # server application type
*.server.app[0].localPort = 1000 # TCP server listen port
```

Another example shows a wireless network. Wireless networks are somewhat different, because wireless nodes do not use OMNeT++ channels. Instead, an additional module which represents the transmission medium needs to be included in the network. The NED code in Listing 2.3 shows an exemplary IEEE 802.11 network with two hosts and an access point.

**Listing 2.3** An INET wireless network example (imports omitted)

```
network WirelessNetworkExample
{
    submodules:
        configurator: Ipv4NetworkConfigurator; // network autoconfiguration
        radioMedium: Ieee80211ScalarRadioMedium; // shared wireless medium
        host1: WirelessHost { @display("p=200,100"); } // infrastructure wifi
        host2: WirelessHost { @display("p=500,100"); }
        accessPoint: AccessPoint { @display("p=374,200"); } // wifi router
}
```

In general, there are many ways to combine INET modules, as it will be shown in the subsequent sections, not only inside network nodes but also on the network level. For example, you could have several wireless transmission mediums in a network, each with its own physical environment, representing distant locations, which do not really affect each other, and which are connected via a wired network.

### 2.2.2 Network Participants

INET's ready-to-be-used network elements are located in the *src/inet/node/* folder, sorted according to use cases. `StandardHost`, used in Listing 2.1, is one of the most prominent network node types. `StandardHost` represents a widely configurable general-purpose network host with protocols of the Internet Protocol (IP) suite and several network interfaces. Figure 2.2 shows the graphical representation of a host instance in Qtenv. (Some optional components are missing from the screenshot, because they were not used in the particular simulation.) Other, pre-assembled node types take `StandardHost` and specialize it for other use cases. `WirelessHost`, for instance, provides wireless network interfaces pre-configured in infrastructure mode, while `AdhocHost` has wireless interfaces pre-configured for ad hoc operation.

Further models represent various other network devices, such as `EtherSwitch` that implements an Ethernet switch, `AccessPoint` that models a WiFi access point, `Router`, which is a generic IP router with routing protocols like Open Shortest Path First (OSPF) or Border Gateway Protocol (BGP), and `LdpMplsRouter` and `RsvpMplsRouter`, which are Multiprotocol Label Switching (MPLS) routers with Label Distribution Protocol (LDP) and Resource Reservation Protocol-Traffic Engineering (RSVP-TE) as signaling protocol.

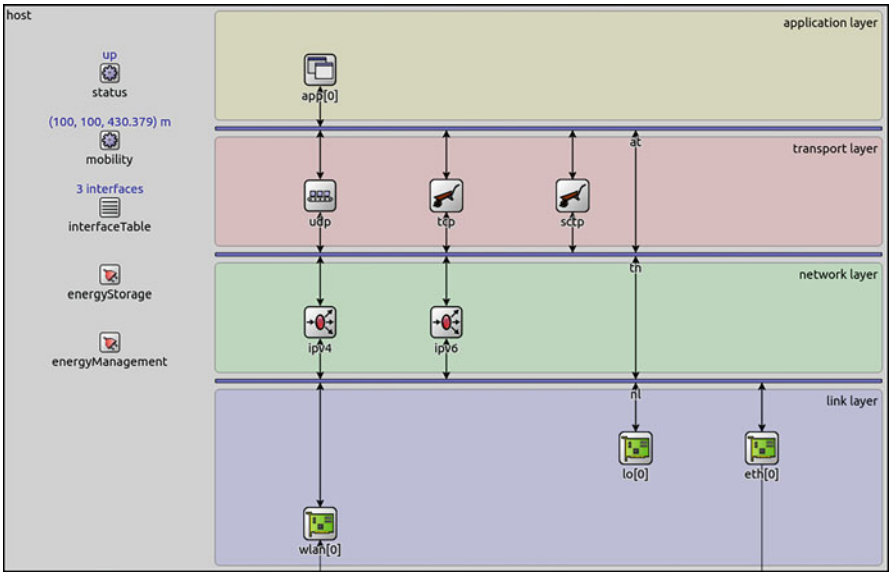



**Fig. 2.2** A `StandardHost` instance in the Qtenv GUI

Most of these node types are widely configurable, but nevertheless, they are not universal. Think of them as examples of how protocol models can be combined to form network nodes. Sometimes, it can be a good idea (or may even be necessary) to assemble purpose-built network node models for particular simulation scenarios.

**Internal Structure** `StandardHost` and other nodes are composed of protocol models and other components. Protocol models are typically implemented as OMNeT++ simple modules, although some complex ones like `Ieee80211Mac` contain further submodules for structuring and flexibility.

OMNeT++ connections are used to interconnect the modules and represent the communication channels between different layers and protocols. As seen in Fig. 2.2, `StandardHost` is organized roughly by OSI layers. At places where many-to-many or many-to-one communication is needed, typically between layers, dispatcher modules are inserted. (Dispatchers appear in the graphics as elongated thin blue rectangles.) Dispatchers are zero-configuration components which discover and learn about their environment, and allow messages to "find their way" through the communication stack. In simpler node architectures where only one-to-one communication exists between layers, dispatchers can be omitted.

**Applications** Traffic generators and other application-layer modules can be added into `StandardHost` via configuration. Applications are in `StandardHost`'s `app[]` submodule vector, where both the number and the individual types of the submodules are configurable.

```
app[numApps]: <> like IApp;
```

To add applications, one needs to specify the number of submodules in the `numApps` parameter, and types of the individual submodules using the `typename` syntax, and then set the parameters of the applications themselves.

```
**.hostA.numApps = 2 # number of applications
**.hostA.apps[0].typename = "UdpBasicApp" # NED module type
**.hostA.apps[1].typename = "PingApp" # NED module type
**.hostA.apps[1].destAddr = "host2" # further parameter setting
```

INET provides various applications that communicate via Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Stream Control Transmission Protocol (SCTP), or directly over lower-layer protocols. Many are generic and can be parameterized to generate constant (CBR) or variable bit rate (VBR) traffic, or emulate simple file transfer or request-response protocols. A sample of these applications:

- `PingApp`: periodic Internet Control Message Protocol (ICMP) echo requests,
- `TcpBasicClientApp`: generic client application model for use with TCP,
- `TcpGenericServerApp`: generic server model for TCP application,
- `TcpSinkApp`: consumes/discards incoming packets from TCP connections,
- `TcpSessionApp`: models a TCP application that opens a new connection, sends a configurable (scriptable) amount of data, and closes the connection,
- `UdpBasicApp`: generic UDP traffic generator,
- `UdpBasicBurst`: models UDP burst traffic,
- `UdpEchoApp`: "echoes" received packets back to the sender,
- `UdpSink`: consumes and discards UDP packets,
- `UdpVideoStreamClient` & `UdpVideoStreamServer`: client and server models for UDP-based video streaming,
- `SimpleVoipSender` & `SimpleVoipReceiver`: sender and receiver models for Voice-over-IP (VoIP) sessions with (alternating) *talkspurts* and silences,
- `VoipStreamSender` & `VoipStreamReceiver`: simulate a VoIP session that uses real-life audio data for the transmission,
- `HttpBrowser` & `HttpServer`: part of the *HttpTools* package, implement client and server Hypertext Transfer Protocol (HTTP) traffic models.

Most applications are network layer agnostic and work over both Internet Protocol Version 4 (IPv4) and Internet Protocol Version 6 (IPv6); the destination address decides which protocol will be used. The applications can be programmed using INET's socket classes that provide an API similar to Unix sockets.

**Network Interfaces** An important ingredient is network interfaces. The concept of network interfaces roughly coincides with that in operating systems, that is, a network interface in INET represents the combination of hardware and software elements that provide the network connectivity. Network interfaces occupy layer

1 and 2 (the physical and data link layers) of the OSI reference model. There are several pre-assembled network interface modules in INET:

- `LoopbackInterface` providing a link-local loopback in the node,
- `EthernetInterface` representing an Ethernet interface,
- `PppInterface` an interface for wired Point-to-Point Protocol (PPP) links,
- `WirelessInterface` and other, more specialized wireless interface types,
- `TunInterface` provides a tunneling for use by upper layers,
- `ExtInterface` for emulation and Hardware-in-the-Loop (HIL) simulations.

INET's network interfaces are compound modules that connect to the above layers. The most important component of a network interface is the implementation of the layer-2 protocol. For some interfaces such as `PppInterface`, this is a single module; for others like `EthernetInterface` and `Ieee80211Interface`, it consists of separate modules for Medium Access Control (MAC), Logical Link Control (LLC), and possibly other subcomponents.

The physical layer is not always present as a separate submodule. For example, `Ieee80211Interface` and other wireless interfaces contain a radio module, but it is implicit for PPP and Ethernet.

Other ingredients in a network interface are optional, and can be added via parameters in simulations where they are needed. For example, the output queue: it is absent by default, because most layer-2 protocol implementations already contain an internal queue which is more efficient to work with. The possibility to plug in an "external" queue module allows one to experiment with different queuing policies and implement Quality of Service (QoS), Random Early Detection (RED), etc. Another type of components that can be added are traffic conditioners. They allow packet classifiers/markers, and traffic shaping and policing elements to be added to the interface, for example to implement a DiffServ router.

Now that we have covered the basics, in the next sections we will delve into the details of simulating various types of networks using INET.

## 2.3 Simulating TCP/IP Networks

TCP/IP or the Internet Protocol suite is the foundation of the Internet and many other networks, so being able to simulate TCP/IP-based networks is essential. INET not only provides the necessary protocol models, but built-in network nodes like `StandardHost` and `Router` already contain TCP/IP support in their default configuration.

In network nodes, all protocol components are optional, and their presence can be controlled with parameters like `hasUdp`, `hasTcp`, `hasSctp`, `hasIpv4`, and `hasIpv6`. By default, UDP, TCP, and IPv4 are present, others can be added by setting corresponding parameter to `true`. For example, one can create a node with a dual IP stack (i.e., with both IPv4 and IPv6) by setting `hasIpv6` (an addition to `hasIpv4`) to `true`.

Network nodes also allow the protocol models to be replaced with alternative implementations. This especially makes sense with TCP, as INET already contains useful alternative implementations, but it also allows a protocol model to be replaced with a modified (tweaked) or mock version for experimentation or testing. Listings 2.4 and 2.5 illustrate the idea.

**Listing 2.4** Replacing TCP and UDP modules with alternative implementations

```
1 **.tcp.typename = "TcpLwip"
2 **.udp.typename = "MyTweakedUdp"
```

**Listing 2.5** Transport protocols in `StandardHost`

```
1 submodules:
2    udp: <default("Udp")> like IUdp if hasUdp;
3    tcp: <default("Tcp")> like ITcp if hasTcp;
4    sctp: <default("Sctp")> like ISctp if hasSctp;
```

**Internet Protocol Version 4** IPv4 is a network (or in the original terminology, internetwork) layer protocol that provides a hop-by-hop, unreliable, and connectionless delivery service for higher layers. IPv4 is implemented by the `Ipv4` module, which performs encapsulation and decapsulation, fragmentation and assembly, and forwarding of IP datagrams. Routes are kept in a separate module, `Ipv4RoutingTable`. `Ipv4` and other modules use C++ calls to query and update the routes.

In addition to interacting with other modules by exchanging packets, `Ipv4` also supports a Linux-like C++ Netfilter Application Programming Interface (API) that allows other modules to hook into the operation of `Ipv4`, intercept packets at various stages of processing, and change their default treatment. The Netfilter API is useful for implementing on-demand routing protocols without modifying the `Ipv4` module's code, for example.

`Ipv4`, `Ipv4RoutingTable`, and implementations of related protocols (`Icmp`, `Arp`, `Igmpv2`/`Igmpv3`, etc.) are assembled into a compound module called `Ipv4NetworkLayer`, and that module type is used as a building block in node types such as `StandardHost` and `Router`. Submodules are replaceable; for instance, Address Resolution Protocol (ARP) has an alternative implementation called `GlobalArp`, which looks up addresses in a shared global table without actually exchanging ARP protocol messages. It can be used in place of the `Arp` module in scenarios where simulating the actual ARP exchanges is not necessary.

Dynamic Host Configuration Protocol (DHCP) support is provided by two module types, `DhcpServer` and `DhcpClient`. They can be installed into nodes, formally into the application layer, when needed.

**Internet Protocol Version 6** IPv6 is the successor of IPv4. IPv6 not just lifts the limit on the number of available IP addresses by using larger, 128-bit addresses but also does many things differently from IPv4. For example, several features that were

added to IPv4 as an afterthought (often as separate protocols) have been integrated into IPv6 and are part of the base protocol.

Similarly to IPv4, IPv6 support is implemented in INET by several cooperating modules. The base protocol is in the `Ipv6` module, which relies on the `Ipv6RoutingTable` to store the routes. The `Ipv6NeighbourDiscovery` module implements all tasks associated with neighbor discovery and stateless address auto-configuration. The data structures themselves (destination cache, neighbor cache, and prefix list) are kept in `Ipv6RoutingTable`. The rest of the Internet Control Message Protocol for Internet Protocol Version 6 (ICMPv6) functionality, such as error messages and echo request/reply, is implemented in `Icmpv6`.

**Mobile IPv6** Mobile IPv6 support has been contributed to INET by the xMIPv6 project. The main module is `xMIPv6`, which implements Fast Mobile Internet Protocol version 6 (MIPv6), Hierarchical MIPv6, and Fast Hierarchical MIPv6 (thus, $x \in F, H, FH$). The binding cache and related data structures are kept in the `BindingCache` module.

**Alternative Network Protocols** Built-in network nodes like `StandardHost` and `Router` may also contain alternative network layer protocols instead of (or in addition to) IPv4 and IPv6. For example, the `NextHopForwarding` module is simple and straightforward implementation of the next-hop forwarding concept (without support for hierarchical routing and fragmentation, for example), and can use mere integers as addresses (as opposed to IPv4 or IPv6 addresses). It can be used for experimentation or for teaching purposes. `NextHopForwarding` is added to nodes as a `NextHopNetworkLayer` compound module that also contains a `NextHopRoutingTable`.

**Transport Layer Protocols** On the transport layer, INET currently provides support for the TCP, UDP, SCTP, and Real-time Transport Protocol (RTP) protocols.

INET contains three implementations of the TCP protocol. The `Tcp` module is the primary implementation. It is a complete implementation in the sense that it reproduces TCP features like the connection state machine, window-based flow control, persistence and keep-alive timers, adaptive retransmission timeout computation, delayed acknowledgment, Nagle's algorithm, selective acknowledgment (SACK), and SACK-based loss recovery. It also supports various congestion control schemes like Tahoe, Reno, New Reno, Westwood, and Vegas. The `Tcp` model was designed for readability, extensibility, and experimentation. The code is extensively commented, many features like Nagle and delayed acknowledgment can be enabled or disabled via parameters, and detailed logging and statistics are provided. Adding a new congestion control algorithm is as simple as writing and registering a C++ class.

The other two implementations, `TcpLwip` and `TcpNsc` wrap 3rd-party TCP implementations, are primarily provided for validating the main TCP model. The three implementations are drop-in replaceable, and can even be mixed in the same

network. `TcpLwip` is a wrapper around the lightweigh Internet Protocol (lwIP) library, a widely used open-source TCP/IP stack designed for embedded systems. `TcpNsc` wraps the Network Simulation Cradle (NSC), a library that allows real-world TCP/IP network stacks to be used inside a network simulator.

SCTP is a lesser known transport-layer protocol, which is regarded by many as the successor of TCP. SCTP is message oriented, and ensures reliable, in-sequence transport of messages with congestion control like TCP. It also provides multi-homing and redundant paths to increase resilience and reliability. The `Sctp` model has been contributed to INET and is being maintained by the team of Michael Tüxen at the FH Münster [6].

The RTP protocol is more specialized than previous ones, its purpose is multimedia streaming. The protocol is implemented by the modules `Rtp` and `Rtcp`. INET provides a separate node type, `RtpHost`, for modeling RTP traffic.

**Configuring Addresses and Routing** An IP network needs a substantial amount of configuration to be able to work: interfaces need IP addresses, routing tables need to be filled in, and so on. In real life, host IP addresses are often obtained dynamically via DHCP or a similar protocol, and routers' routing tables are filled in by routing protocols. However, for many simulation scenarios it is quite sufficient to statically assign addresses and fill in the routing tables at the beginning of the simulation.

Early versions of INET have required the user to provide all such information explicitly, in the form of configuration files. However, it has proven to be tedious and error prone for even small networks. Modern INET versions provide a configurator module which, when added into the network, configures the interfaces and routing tables at the start of the simulation.

`Ipv4NetworkConfigurator` supports both manual (when the user specifies each address and route) and automatic network configuration, plus any mix of them. The latter means that the user can provide a partial manual configuration, and the configurator will fill in the gaps automatically. The configurator takes subnets into account when assigning addresses and supports hierarchical routing. (Hierarchical routing can be set up by using only a fraction of configuration entries compared to the number of nodes.) The configurator also performs routing table optimization by merging routes, which reduces the size of routing tables in large networks.

A noteworthy architectural detail is that `Ipv4NetworkConfigurator` only computes and remembers IPv4 configuration information, but does not apply it. Applying the configuration is done by an additional, per-node module called `Ipv4NodeConfigurator`. This separation enables implementing simulated node crashes, shutdowns, and reboots in INET. After a simulated reboot, when the node starts with a clean slate, the node configurator can restore the node's configuration by reading it from the global configurator.

`Ipv4NetworkConfigurator` expects the specification of its job in an XML file. Here is an example configuration file that deals with address assignment and adds a manual route as well:

**Listing 2.6** `Ipv4NetworkConfigurator` XML configuration

```
<config>
  <interface among="host{0-2} router0" address="10.0.0.x"/>
  <interface among="host{3-5} router1" address="10.0.1.x"/>
  <interface among="host{6-8} router2" address="10.0.2.x"/>
  <interface hosts="router*" address="10.1.x.x"/>
  <route hosts="host0 host1" destination="10.0.2.0" netmask="255.255.255.0"
         gateway="router1" interface="eth0" metric="100"/>
</config>
```

**Routing Protocols** INET has models for several internet routing protocols, including Routing Information Protocol (RIP), OSPF, and BGP. The easiest way to add routing to a network is to use the `Router` NED type for routers. `Router` contains a conditional instance for each of the above protocols. These submodules can be enabled by setting the `hasRIP`, `hasOSPF`, and/or `hasBGP` parameters to true. There are also NED types called `RipRouter`, and `OspfRouter`, `BgpRouter`, which are all `Router`'s with the appropriate routing protocol enabled. Detailed configuration for the routing protocols can be provided via XML files.

Additional routing protocols such as Intermediate System to Intermediate System (IS-IS), Enhanced Interior Gateway Routing Protocol (EIGRP), and Babel are available from the Automated Network Simulation and Analysis (ANSA) project.[2]

**Multicast** INET has basic support for multicast networks. `Ipv4RoutingTable` and `Ipv4` are multicast-capable, and the Internet Group Management Protocol (IGMP) protocol is implemented with the `Igmpv2` and `Igmpv3` modules. For multicast based on shared trees, INET contains models of the Protocol-Independent Multicast-Sparse Mode (PIM-SM) and Protocol-Independent Multicast-Dense Mode (PIM-DM) protocols in the `PimSm` and `PimDm` modules. Many of the multicast-related protocol models were contributed by the ANSA project.

**Differentiated Services** Differentiated Services or DiffServ is a scalable mechanism primarily for providing Quality of Service (QoS) in IP networks. In DiffServ, packets are classified and marked as belonging to a specific class, and routers implement per-hop behaviors (PHBs) that are tied to the traffic classes.

INET's network interface modules, for example the `EthernetInterface` or the `PppInterface` module, have the possibility to install traffic conditioner elements into them to implement Differentiated Services in a network. Traffic conditioners have one input and one output gate, and can transform the incoming traffic by dropping or delaying packets. They can also set the Differentiated Services

[2] ANSA project website: https://ansa.omnetpp.org/.

Code Point (DSCP) field of the packet, or mark them other way, for differentiated handling in the queues.

Traffic conditioners perform the following actions:

- classify the incoming packets,
- meter the traffic in each class,
- mark/drop packets depending on the result of metering, and
- shape the traffic by delaying packets to conform to the desired traffic profile.

INET provides classifier, meter, and marker modules that can be composed to build a traffic conditioner as a compound module.

Network interfaces also contain an optional external queue component. (In the absence of an external queue module, layer-2 protocol modules use an internal drop-tail queue to buffer the packets while the line is busy.) A queue component has one input and one output gate, and implements a passive queue behavior: it only delivers a packet when the module connected to its output explicitly requests it to.

The elements of which traffic conditioners and output queues can be built are:

- *queue*: container of packets, accessed via First In First Out (FIFO) strategy.
- *dropper*: attached to one or more queue, it can limit the queue length below some threshold by selectively dropping packets.
- *scheduler*: decide which packet is transmitted first, when more packets are available on their inputs.
- *classifier*: classify the received packets according to their content (e.g., source/destination, address and port, protocol, and DSCP field of IP datagrams) and forward them to the corresponding output gate.
- *meter*: classify the received packets according to the temporal characteristic of their traffic stream.
- *marker*: marks packets by setting their fields to control their further processing.

## 2.4 Simulating MPLS Networks

MPLS is a "layer 2.5" protocol for high-performance telecommunication networks. MPLS directs data from one network node to the next based on numeric labels instead of network addresses, avoiding complex lookups in a routing table and allowing traffic engineering. The labels identify virtual links (label-switched paths or LSPs, also called MPLS tunnels) between distant nodes rather than endpoints. The routers that make up a label-switched network are called label-switching routers (LSRs) inside the network ("transit nodes"), and label edge routers (LERs) on the edges of the network ("ingress" or "egress" nodes). In each Label-Switching Router (LSR), the label information base (LIBs) table contains the output interface and label operations for incoming MPLS packets.

A fundamental MPLS concept is that two LSRs must agree on the meaning of the labels used to forward traffic between and through them. This common

understanding is achieved by using signaling protocols by which one LSR informs another of label bindings it has made. Such signaling protocols are also called label distribution protocols. The two main label distribution protocols used with MPLS are LDP and RSVP-TE. LDP and RSVP-TE keep their view of the network in the traffic engineering database (TED) and use it as input to determine what Label-Switched Paths (LSPs) they should build and maintain.

INET provides basic support for building MPLS simulations. It provides models for the MPLS, LDP, and RSVP-TE protocols and their associated data structures, and pre-assembled MPLS-capable router models: `Mpls` implements the MPLS protocol; `LibTable` holds the Label Information Base (LIB); `Ldp` implements the LDP protocol; `RsvpTe` implements the RSVP-TE protocol; and `Ted` contains the traffic engineering database. A further module, `LinkStateRouting`, implements a simple hypothetical routing protocol that provides network topology information for `Ted`. INET also contains configurable ingress classifier modules such as `RsvpClassifier`; the job of such classifiers is to assign labels to packets as they enter an MPLS domain.

INET provides two pre-assembled MPLS LSR models: `LdpMplsRouter` uses the LDP signaling protocol and `RsvpMplsRouter` uses RSVP-TE.

## 2.5 Simulating Ethernet Networks

Ethernet is the most popular wired Local Area Network (LAN) technology nowadays, and its use is also growing in metropolitan area and wide area networks. Today, switched Ethernet is prevalent, and most links operate in full duplex mode. INET contains support for all major Ethernet technologies and device types.

There are several node models that can be used in an Ethernet network. Node models such as `StandardHost` and `Router` are Ethernet-capable. `EtherSwitch` models an Ethernet switch, i.e., a multiport bridging device. `EtherHub` models an Ethernet hub or multiport repeater. `EtherBus` can be used to model legacy Ethernet networks that use coaxial cable (10BASE2 or 10BASE5 network segments). A minimal sample node to generate "raw" Ethernet traffic is also provided (`EtherHost`).

INET cables (fiber optic/twisted pair) are represented by OMNeT++ connections. Connections used in Ethernet LANs must be derived from `DatarateConnection` and should have their `delay` and `datarate` parameters set. The `delay` parameter can be used to model the distance between the nodes. INET has the following predefined channel types: `Ether10M`, `Ether100M`, `Ether1G`, `Eth10G`, `Eth40G`, `Eth100G`, and recently also `Eth200G` and `Eth400G`.

**Ethernet Interface** The `EthernetInterface` compound module implements the `IWiredInterface` interface. It complements `EtherMac` and `EtherEncap` with an output queue for QoS and RED support.

The Ethernet Medium Access Control (MAC) layer transmits the Ethernet frames on the physical media. This is a sublayer within the data link layer. Because encapsulation/decapsulation is not always needed (e.g., switches do not perform encapsulation/decapsulation), it is implemented in separate modules (`EtherEncap` and `EtherLlc`) that are part of the LLC layer.

Nowadays almost all Ethernet networks operate using full-duplex point-to-point links between hosts and switches. This means that there are no collisions, and the behavior of the MAC component is much simpler than in classic Ethernet that used coaxial cables and hubs. INET contains two MAC modules for Ethernet: the `EtherMacFullDuplex` is simpler to understand and easier to extend, because it supports only full-duplex links. The `EtherMac` module contains the implementation of Carrier Sense Multiple Access with Collision Detection (CSMA/CD), and it can operate both half-duplex and full-duplex mode.

**Switches** An essential component of Ethernet switches is the MAC relay unit. A relay unit has N gate pairs that connect it to the switch ports that are instances of `EthernetInterface`. The default relay unit type is `MacRelayUnit`.

Switches also contain a `MacAddressTable` module which stores the mapping between ports and MAC addresses. (`MacRelayUnit` has a parameter that holds the path to its associated `MacAddressTable` instance.) The address table is filled dynamically as the switch learns which addresses are reachable via each port. Entries are deleted if their age exceeds a certain limit. The table can also be preloaded from a text file while initializing the relay unit.

When the relay unit receives a data frame, it updates the table with the *(source address, input port)* pair. Then, it looks up the destination address in the address table. If it was found, the frame is sent out to the corresponding port. If the address was not in the table, the frame is broadcast to all ports except the one it was received from. (If the destination was on the subnet of the source port, it must have already received the original frame.)

**Spanning Tree Support** In Ethernet networks containing multiple switches, broadcast storms are prevented by using a spanning tree protocol (Spanning Tree Protocol (STP) or Rapid Spanning Tree Protocol (RSTP)) that disables selected links to eliminate cycles from the topology. Ethernet switch models in INET contain support for STP and RSTP.

`Ieee8021dRelay` is a MAC relay unit to be used instead of `MacRelayUnit` when STP or RSTP is needed.

The `Stp` module type implements STP. STP is a network protocol that builds a loop-free logical topology for Ethernet networks. The basic function of STP is to prevent bridge loops and the broadcast radiation that results from them. STP creates a spanning tree within a network of connected layer-2 bridges and disables those links that are not part of the spanning tree, leaving a single active path between any two network nodes.

`Rstp` implements RSTP, an improved version of STP. RSTP provides significantly faster recovery in response to network changes or failures.

**Related Projects** Several projects extend or complement the Ethernet models in INET. Communication over Real-time Ethernet for INET (CoRE4INET)[3] adds Virtual Local Area Network (VLAN) support, TTEthernet, and Audio Video Bridging (AVB) support. The ANSA project[4] provides models for layer-2 protocols like Link Layer Discovery Protocol (LLDP), Cisco Discovery Protocol (CDP), and Transparent Interconnection of Lots of Links (TRILL).

## 2.6 Simulating Wireless Networks

Wireless networks went through tremendous growth in the last two decades, and have become ubiquitous by today. INET contains models for simulating IEEE 802.11 networks, as well as several other wireless technologies; these models will be covered in later sections. In this section, we look at the physical layer and the related infrastructure INET provides for the detailed (or not-so-detailed, depending on the needs) simulation of wireless networks.

Simulating wireless communication is significantly more complex and challenging than wired communication for several reasons. First, the wireless channel is a shared medium that leads to more complicated MAC protocols than in most wired networks. More importantly, unlike wired networks where the physical layer can usually be abstracted away in simulation, the effects of the underlying physical reality cannot be ignored in wireless simulations. In wired networks, signal power usually does not attenuate significantly enough to cause high error rates by the time it reaches the receiver; in contrast, wireless signals suffer from path loss, obstacle loss, interference from background noise, and other effects, potentially resulting in garbled reception. These effects must be accounted for in the simulation. Transmission power, gains of the transmitter and receiver antennas (any or both of which may be a directional antenna), reception sensitivity, and other aspects also need to be taken into account. In addition, while in the wired case two signals overlapping at the receiver are normally a clear collision, in the wireless case the weaker signal may or may not make the stronger signal unrecognizable, depending on the Signal-to-Noise Ratio (SNR).

These challenges lead to an increased model complexity and increased runtimes. Scaling wireless simulations to a higher number of nodes is also more challenging than with wired networks, due to the shared nature of the wireless medium. Because of these reasons, it is crucial that the wireless infrastructure in the simulator provides flexibility and scalable level of detail. It must be possible to turn off the simulation of certain aspects for gaining performance if they are not important for a certain simulation. On the other hand, it must also be possible to increase the detail level of the simulation for scenarios that require it, even at the cost of consuming more CPU

[3] CoRE4INET website: https://core4inet.core-rg.de.

[4] ANSA website: https://ansa.omnetpp.org.

cycles. One extreme may be using a unit disk radio; the other extreme a symbol-level or sample-level simulation with detailed physical layer model (e.g., ray tracing).

The infrastructure that INET provides for wireless simulations has three ingredients: per-node *radio* (or more generally, transceiver) models that include the antenna as well; the global *transmission medium* that keeps track of ongoing transmissions; and the model of the *physical environment* (obstacles, terrain, etc.). We will examine them shortly, and we will see that each one consists of several smaller components. The subcomponents can be individually configured or replaced with another built-in or custom-developed component, providing a very flexible and versatile foundation for a diverse range of wireless simulations. INET also provides visualization features specific to wireless simulations, which can be helpful while developing or demonstrating simulations.

### *2.6.1 Signal Representation*

When describing the physical phenomenon of a wireless signal, the focus is on the analog domain representation. The reason is that other domains, the bit domain for example, where forward error correction, scrambling, and interleaving happen; or the symbol domain where modulation happens, are computationally very expensive to simulate, and thus they are often just approximated. In fact, the effect of these physical layer processes can be integrated into the error model of the receiver, which is based on the analog domain representation of the signal and the interference.

INET provides multiple analog domain representations, each having its own advantages and application areas.

**Scalar Representation** A reasonable approximation to characterize a wireless signal in the analog domain is to use a scalar *signal power* which stays constant over the signals's *frequency* interval and *time* interval. The scalar representation is mostly appropriate for narrowband wireless signals. It already allows using many path loss algorithms such as free-space path loss, log-normal shadowing, Rician fading, etc., while it can still be stored and calculated with efficiency.

Nevertheless, the scalar analog domain representation may be inappropriate for several cases, for instance:

- when simulating the coexistence of multiple different wireless technologies,
- when simulating the crosstalk between adjacent WiFi channels,
- or when simulating wideband wireless signals, because the interference between such wireless signals requires a more accurate representation in either the time and/or the frequency domains.

**Dimensional Representation** A more accurate but also computationally more expensive way to characterize a wireless signal in the analog domain is to use a *signal power* which changes over *time* and/or *frequency*. The dimensional

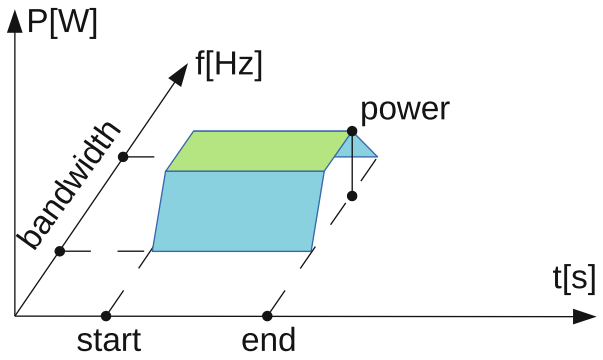



**Fig. 2.3** Changing signal strength along frequency

representation uses multiple *signal power* values for individual *time* and/or *frequency* coordinates and uses linear or sample-hold interpolation in between.

For example, when simulating crosstalk between adjacent WiFi channels, the signal power must be represented in the frequency domain more accurately than just a constant power over a frequency range, so the effect of cross-channel interference can be correctly computed. The real signal's power spectral density can be approximated using multiple signal power values at different frequency coordinates (cp. Fig. 2.3).

Another example where dimensional representation has proven to be useful is the simulation of IEEE 802.15.4's Ultra-Wideband Impulse Radio (UWB-IR) Physical Layer (PHY). This radio transmits wideband signals composed of short pulses with gaps in between. Here, the scalar representation which assumes constant power over the whole duration of the frame is clearly not sufficient. Hence, the `Ieee802154UwbIrRadio` model uses dimensional representation where the signal power varies over time to be able to describe the impulses. This approach allows for the effect of multipath self-interference to be taken into account correctly.

**Unit Disk** In contrast, for simulation studies that concentrate on higher layers, even the scalar signal representation may prove to be too detailed. In such cases, the *unit disk graph* model, a simplistic but fast and predictable physical layer model can provide a suitable alternative. This model is applicable if network nodes need to have a finite communication range, but physical effects of signal propagation are to be ignored. It has been widely used in the study of mobile wireless ad hoc networks, for example routing in such networks.

The INET implementation, `UnitDiskRadio`, allows three radii to be given as parameters: communication range, interference range, and detection range. Signals are correctly received within the communication range of the transmitter; they render other signals unreceivable within the interference range; and are detectable (as nonempty channel) by receivers within the detection range. As a bonus, one can

also turn off interference modeling (meaning that signals colliding at a receiver will all be received correctly), which is sometimes a useful abstraction.

**Bit- and Symbol-Domain Modeling** INET also contains several modules for the accurate simulation of the individual processing steps of the bit and symbol domains.

For example, the transmitter modules of the 802.11 Orthogonal Frequency Division Multiplex (OFDM) radio are the `Ieee80211LayeredOfdmTransmitter`, the `Ieee80211OfdmEncoder`, and the `Ieee80211OfdmModulator`. The encoder module further contains a `ConvolutionalCoder` for the forward error correction, an `Ieee80211OfdmInterleaver` for the interleaving, and an `AdditiveScrambler` for the scrambling. Similarly, the detailed receiver modules for the 802.11 OFDM radio are the `Ieee80211LayeredOfdmReceiver`, the `Ieee80211OfdmDecoder`, and the `Ieee80211OfdmDemodulator`. These modules together are capable of simulating the 802.11 OFDM PHY down to the symbol-level accuracy.

For another example, the detailed transmitter modules for a hypothetical amplitude and phase-shift keying radio are the `ApskLayeredTransmitter`, the `ApskEncoder`, the `ApskModulator`, and similarly to the receiver above the `ApskLayeredReceiver`, the `ApskDecoder`, and the `ApskDemodulator`.

### *2.6.2 Modeling the Radio*

The `Radio` module represents the physical device which is capable of transmitting and receiving signals on the transmission medium. The `Radio` module contains an antenna, a transmitter, a receiver, and an optional energy consumer submodule. The antenna is shared between the transmitter and the receiver. The energy consumer model is only used when the simulation of energy consumption is necessary.

INET contains several additional radio modules derived from `Radio`, some of which are specific to a particular physical layer. For example, `Ieee80211Radio` represents the IEEE 802.11 PHY, and the `ApskRadio` and the `UnitDiskRadio` are hypothetical radios.

The transmitter submodule represents the physical process which converts packets into electric signals. The receiver submodule represents the opposite physical process that converts electric signals into packets. There are many variants for both, for example `Ieee80211ScalarTransmitter` and `Ieee80211ScalarReceiver` are the ones which represent the signal with a scalar power in the analog domain.

Determining reception errors is the crucial part of the reception process. The receiver module contains another replaceable submodule called the error model. The error model describes how the Signal-to-Noise-plus-Interference Ratio (SNIR) affects the amount of errors at the receiver. INET contains protocol agnostic error

models such as the `StochasticErrorModel`, but there are also protocol-specific error models, the IEEE 802.11 error models, for instance, are discussed in Sect. 2.7.

In wireless networks, the position of the antennas is decisive. The position and orientation of antennas are provided by the network node's mobility by default. In some simulations, the antenna is not located at the same position as the network node, or its orientation is different from the node's, in such cases the antenna has its own mobility submodule. This feature allows, for example, the simulation of a moving vehicle with multiple radio interfaces, each placed at a specific location in the vehicle and each oriented differently than the vehicle itself.

**Directional Antennas** By default, INET radios use the `IsotropicAntenna`, a theoretical point source, which radiates with the same intensity in all directions. Nevertheless, the physical layer also supports directional antennas, and INET contains several models. For example, the `DipoleAntenna` represents a dipole, the `ParabolicAntenna` represents a parabolic antenna, the `CosineAntenna` is a hypothetical model, and the `AxiallySymmetricAntenna` is a generic model which can be used to describe any 3D axially symmetric antenna lobe.

### *2.6.3 Transmission Medium*

Wireless network models are required to contain a special module that represents the shared physical medium where the communication takes place. The *medium* module keeps track of transceivers, noise sources, ongoing transmissions, background noise, and other ongoing noises. It also contains models of signal propagation, path loss, obstacle loss, signal analog model, and background noise.

The transmission medium is modeled as an OMNeT++ compound module with several replaceable submodules. These submodules are used via C++ method calls and are responsible for modeling signal propagation, path loss, obstacle loss, signal analog model, and background noise. With the help of its submodules, the medium module computes when, where, and how signals arrive at receivers, including the set of interfering signals and noises. Additional submodules implement various caches for efficiency.

**Propagation Models** The propagation model describes how the wireless signal travels through space over time, and its main purpose is to compute when and where the signal's start and end arrive at various (potentially moving) receivers. By default, INET uses a model that represents constant speed propagation, with the speed of light (speed is configurable via a module parameter). This model ignores the movement of receivers during signal's travel which is suitable for radio communication, but a more sophisticated model might be necessary for simulating acoustic (e.g., underwater) communication, where signal speeds are much closer to the speed of transceivers.

**Path Loss Models** As a signal propagates through space, its power density decreases. This is called path loss, and it is the combination of many effects such as free-space loss, refraction, diffraction, reflection, and absorption. In INET, the main purpose of the path loss model is to compute the power loss for a given signal, but it is also capable of estimating the range for a given loss. The latter is useful, for example, to allow visualizing the communication range.

INET contains the implementations of a number of well-known path loss algorithms that differ in their parametrization and application area: free-space path loss (this is the default), breakpoint path loss (dual-slope free-space model with two separate path loss exponents), log-normal shadowing, two-ray ground reflection, Rician fading, Rayleigh fading, and Nakagami fading. The following example replaces the default free-space path loss model with log-normal shadowing:

**Listing 2.7** Replacing the default path loss model with log-normal shadowing

```
1 *.radioMedium.pathLoss.typename = "LogNormalShadowing" # module type
2 *.radioMedium.pathLoss.sigma = 1.1 # override default value of 1
```

**Obstacle Loss Models** When the signal propagates through space, it also passes through physical objects present in that space. As the signal penetrates physical objects, its power decreases when it is absorbed by the object's material and also when it reflects from surfaces. There are various ways to model this effect, which differ in the trade-off between accuracy and performance. In INET, the main purpose of an obstacle loss model is to compute the power loss based on the signal's path and the frequency.

INET contains two built-in obstacle loss models: an "ideal" obstacle loss model that regards obstacles to be fully opaque to all signals, and a dielectric obstacle loss model that takes the material of the obstructing physical object into account when computing power loss. These obstacle loss models use the physical environment model to determine the set of penetrated physical objects. (Statistical obstacle loss models are also possible but currently not included in INET.)

By default, the medium module does not contain any obstacle loss model. The following example shows how to select the dielectric obstacle loss model:

**Listing 2.8** Obstacle loss model configuration example

```
1 *.radioMedium.obstacleLoss.typename = "DielectricObstacleLoss" # module type
```

**Background Noise Models** Thermal noise, cosmic background noise, and other random fluctuations of the electromagnetic field affect the quality of the communication channel. This kind of noise does not come from a particular source, so it does not make sense to model it as a signal that propagates through space. In INET, the main purpose of a background noise model is to compute the analog representation of the background noise for a given space–time interval. For example, `IsotropicScalarBackgroundNoise` computes a background noise that is

independent of space–time coordinates, and its scalar power is determined by a module parameter.

The simplest background noise model can be configured as follows:

**Listing 2.9** Background noise model configuration example

```
1 *.radioMedium.backgroundNoise.typename = "IsotropicScalarBackgroundNoise" #type
2 *.radioMedium.backgroundNoise.power = -110 dBm # isotropic scalar noise power
```

**Analog Models** The analog signal is a complex physical phenomenon which can be modeled in many different ways. Choosing the right analog domain signal representation is the most important factor in the trade-off between accuracy and performance. The analog model of the transmission medium determines how signals are represented while being transmitted, propagated, and received.

In INET, an analog model is an OMNeT++ simple module. Its main purpose is to compute the received signal from the transmitted signal. The analog model combines the effect of the antenna, path loss, and obstacle loss models. Transceivers must be configured to transmit and receive signals according to the representation used by the analog model.

The most commonly used analog model, which uses a scalar signal power representation over a frequency and time interval, can be configured as follows:

**Listing 2.10** Analog model configuration example

```
1 *.radioMedium.analogModel.typename = "ScalarAnalogModel" # module type
```

**Filters and Caches** As a central component, the medium module influences performance to a large extent, so it also provides a couple of parameters for optimization. For example, the `rangeFilter` parameter controls how the set of affected receivers is determined based on their distance when the signal enters the medium.

The medium module also employs several caches (neighbor cache, medium limits cache, and communication cache) to improve its performance. These caches are replaceable, because different data structures may be the optimal choice under different circumstances. For example, the neighbor cache has three alternative implementations, based on neighbor lists, grid, and quad tree data structure.

### *2.6.4 Modeling the Physical Environment*

The transmission medium model heavily relies on the one that represents the physical environment. The main purpose of the *physical environment* model is to describe buildings, walls, furniture, vegetation, terrain, weather, and other physical objects and conditions that might have profound effects on the simulation.

In INET, the physical environment is modeled by the `PhysicalEnvironment` compound module. This module normally has one instance in the network, and it provides services for other parts of the simulation. It contains submodules to model physical objects and the ground as well as a few parameters for the physical properties of the environment.

**Physical Objects** The most important aspect of the physical environment is the objects which are present in it. The physical environment model stores the shape, position, orientation, and material of each object. (Objects are assumed to be homogeneous, which is a simplified description but still allows for a reasonable approximation of reality.) Listing 2.11 shows how to define physical objects using the XML syntax supported by the physical environment model:

**Listing 2.11** Defining physical objects

```
<environment>
  <!-- an object defined with an in-line shape and material -->
  <object position="min 1 2 0" orientation="90 -45 0" shape="cuboid 40 10 20"
          material="brick" line-color="0 0 0" fill-color="126 43 32"/>
  <!-- separately defined shapes and materials -->
  <shape id="11" type="cuboid" size="1 2 3"/>
  <shape id="12" type="sphere" radius="1000"/>
  <shape id="13" type="prism" height="100" points="0 0 2 0 2 2 0 2"/>
  <shape id="14" type="polyhedron" points="0 0 0 2 0 0 2 2 0 0 2 0 ..."/>
  <material id="11" resistivity="8" relativePermittivity="2.5"/>
  <!-- an object that uses a previously defined shape and material -->
  <object position="min 1 1 0" orientation="60 30 0" shape="13" material="11"/>
</environment>
```

The file must be passed to the physical environment model in a parameter:

**Listing 2.12** Physical objects configuration example

```
*.physicalEnvironment.config = xmldoc("objects.xml") # load physical objects
```

**Ground Models** In many outdoor simulation scenarios, the terrain has profound effects on signal propagation. For example, vehicles on the opposite sides of a mountain cannot directly communicate with each other. In INET, the ground model describes the 3D surface of the terrain. Its main purpose is to provide elevation data for locations. INET currently contains two grounds models: `FlatGround` which represents a flat surface parallel to the XY-plane at a certain height, and `OsgEarthGround` which uses the *osgEarth* API to provide elevation data from an external map data source.

**Geographic Coordinate System Models** Internally, INET uses a 3D Cartesian coordinate system, *scene coordinates*, to describe locations. In scenarios where it is important where the simulation scene is located on the surface of the earth (e.g., those involving *osgEarth*-based visualization or ground model), a geographic coordinate system module provides the mapping between scene coordinates and geographic coordinates. INET contains two geographic coordinate system models. `SimpleGeographicCoordinateSystem` implements a simple formula that

uses linear approximation, while `OsgGeographicCoordinateSystem` provides an accurate mapping on top of the *osgEarth* library.

Listing 2.13 shows how the geographic coordinate system module can be configured to place the simulation scene at a particular geographic location and orientation.

**Listing 2.13** Geographic coordinate system configuration

```
1 *.physicalEnvironment.coordinateSystemModule = "coordinateSystem" # reference
2 *.*.mobility.coordinateSystemModule = "coordinateSystem" # reference
3 *.coordinateSystem.sceneLongitude = -71.070421deg # scene origin
4 *.coordinateSystem.sceneLatitude = 42.357824deg # scene origin
5 *.coordinateSystem.sceneHeading = 68.3deg # scene orientation
```

**Object Cache** The physical environment model also contains an object cache to accelerate looking up obstacles in the signal propagation path, as it might affect performance quite heavily. Multiple cache implementations are provided, as different cache data structures may be optimal for different scenarios. For example, `GridObjectCache` organizes objects into a 3D spatial grid, while `BvhObjectCache` organizes them into a tree data structure based on a recursive 3D volume division.

## 2.7 Simulating IEEE 802.11 Networks

IEEE 802.11, also known as WiFi, is the most widely used and universal wireless networking standard. Specifications (e.g., [3]) are updated every few years, adding more features and ever increasing bit rates.

There are several node models that can be used for building a WiFi network. For example `WirelessHost` and `AdHocHost`: both are derived from `StandardHost`, and are pre-configured to have one `Ieee80211Interface`. By default, `WirelessHost`'s interface is set up for 802.11 infrastructure mode and `AdHocHost`'s for 802.11 ad hoc mode. Access points are represented with the `AccessPoint` node type, which also happens to have Ethernet interfaces and frame relaying functionality. Other nodes can also become WiFi-enabled by adding an `Ieee80211Interface` to them. WiFi networks also require a matching transmission medium module to be present in the network, which is usually an `Ieee80211ScalarRadioMedium`.

The number of wireless network interfaces is configurable in all built-in node types that support wireless networks (see Listing 2.14).

**Listing 2.14** Multiple wireless interfaces example

```
1 *.host[*].numWlanInterfaces = 2 # number of wireless network interfaces
2 *.host[*].wlan[0].agent.defaultSsid = "alpha" # connects to alpha network
3 *.host[*].wlan[1].agent.defaultSsid = "bravo" # connects to bravo network
```

**Network Interface** `Ieee80211Interface` is composed of several submodules. The `llc` submodule performs encapsulation and decapsulation for the network layer. The `mgmt` submodule implements authentication, association, beaconing, channel scanning, etc. The `agent` submodule initiates channel scanning and connecting to access points on behalf of the user. The `mac` submodule transmits and receives frames according to the 802.11 medium access procedure. The `radio` submodule transmits and receives 802.11 PHY signals.

Operation mode (infrastructure vs. ad hoc) is determined by the management component. It has several implementations which differ in their role and level of detail. `Ieee80211MgmtAdhoc` is for ad hoc mode stations, `Ieee80211MgmtSta` and `Ieee80211MgmtStaSimplified` for infrastructure mode stations, and `Ieee80211MgmtAp` and `Ieee80211MgmtApSimplified` for access points.

The "simplified" ones assume that stations are statically associated to an access point for the entire duration of the simulation (the scan-authenticate-associate process is not simulated), so they cannot be used in experiments involving handover.

The agent component is only needed for interfaces containing the component `Ieee80211MgmtSta`. `Ieee80211MgmtSta` does not take any action by itself, it requires an agent component to initiate actions. `Ieee80211AgentSta` is the default agent. By modifying or replacing the agent, one can alter the dynamic behavior of stations in the network, for example implement different handover strategies.

An example configuration that configures nodes for static access point attachment is as follows:

```
1 *.host*.wlan[0].mgmt.typename = "Ieee80211MgmtStaSimplified"
2 *.ap.wlan[0].mgmt.typename = "Ieee80211MgmtApSimplified"
3 *.*.wlan[0].agent.typename = ""
4 *.ap.wlan[0].mac.address = "10:00:00:00:00:00"
5 *.host*.wlan[0].mgmt.accessPointAddress = "10:00:00:00:00:00"
```

**Options for Configuring the PHY** `Ieee80211Radio`, just like other radios in INET, contains a `transmitter`, a `receiver`, an `energyConsumer`, and an `antenna` submodule. One can choose from several predefined radios with different levels of detail for the signal analog domain representation. The various radio types (with the matching transmission medium types in parentheses) are:

- `Ieee80211ScalarRadio` (`Ieee80211ScalarRadioMedium`),
- `Ieee80211DimensionalRadio` (`Ieee80211DimensionalRadioMedium`),
- and `Ieee80211UnitDiskRadio` (`UnitDiskRadioMedium`).

The radio submodules have numerous parameters, which allow them to be fully configured and used on their own, but in a typical setting the radio is configured by

the MAC module automatically. The following configuration snippet illustrates the PHY parameters:

```
**.wlan[*].radio.transmitter.power = 2mW
**.wlan[*].radio.receiver.sensitivity = -190dBm
**.wlan[*].radio.receiver.snirThreshold = 4dB
```

The radio modules supports several physical layers defined by the standard: OFDM, ERP-OFDM, HT, VHT, HR-DSSS, DSSS, FHSS, and IR. The various well-known WiFi operation modes (802.11 a/b/g/n/p/ac) are composed of the supported optional and mandatory data rates using the required PHY characteristics. One can define new WiFi operation modes by composing a different set of physical layers and data rates in C++. Both the 2.4 and 5 GHz bands are supported with multiple WiFi channels but experimenting with new configurations is also possible.

In INET radios, receivers contain an important module, called the error model. The error model is responsible for determining whether a received packet contains errors or not based on the SNIR. The 802.11 physical layer provides several alternative error models: `Ieee80211NistErrorModel`, `Ieee80211YansErrorModel`, `Ieee80211BerTableErrorModel`. The former two modules use predefined error rate probability functions (based on the SNIR), while the latter allows defining arbitrary mappings. Error models also have a `corruptionMode` parameter which determines the level of detail for how to represent corrupt packets. Refer to Sect. 2.13.3 for more details on the representation of packets.

For the OFDM PHY, INET also includes a bit-level transmitter and receiver model which carries out many radio functions such as forward error correction, scrambling, interleaving, and modulation in the bit and symbol domains. This model itself is implemented in `Ieee80211LayeredOfdmTransmitter`, in `Ieee80211OfdmEncoder`, in `Ieee80211OfdmModulator`, and in their corresponding receiver, decoder, and demodulator counterparts. The detailed 802.11 radio is slower than the default one, but it allows experimenting with many physical layer techniques in the context of a more complete network simulation stack.

**Configuring and Experimenting with the MAC** The `Ieee80211Mac` module type represents the IEEE 802.11 MAC, and it is arguably the most important and most complex part of the `Ieee80211Interface`. `Ieee80211Mac` receives data and management frames from the upper layers and transmits them according to the frame exchange sequences allowed by the coordination function. It also implements MAC data services such as aggregation and fragmentation, provides multiple access categories for QoS, implements block acknowledgment, and follows the Transmit Opportunity (TXOP) procedures according to the IEEE 802.11 MAC standard.

The most distinguishing feature of `Ieee80211Mac` is that it is extremely modular in order to facilitate experimenting with various aspects of the protocol. The immense number of features provided by the IEEE 802.11 standard are implemented

by numerous simple and compound modules cooperating via C++ methods calls. Each of these modules have their own parameters to control their behavior. The following configuration snippet illustrates some of the MAC parameters:

```
1 **.wlan[*].mac.dcf.channelAccess.cwMin = 7
2 **.wlan[*].mac.dcf.rateControl.interval = 1s
3 **.wlan[*].mac.dcf.rtsPolicy.rtsThreshold = 500B
4 **.wlan[*].mac.dcf.recoveryProcedure.shortRetryLimit = 15
5 **.wlan[*].mac.hcf.edcaTxopProcedures[1].txopLimit = 7ms
6 **.wlan[*].mac.hcf.originatorAckPolicy.blockAckReqTreshold = 12
7 **.wlan[*].mac.hcf.originatorMacDataService.msduAggregationPolicy.
      aggregationLengthThreshold = 10000B
```

The `Ieee80211Mac` compound module is assembled from these smaller components, using composition several levels deep (see Fig. 2.4), and most components are replaceable from the configuration. The practical implications are that users can experiment with various policies, features, and algorithms, etc., without the need to touch a single line of code in the existing implementation.

Policies, which are the most likely to be experimented with, are also extracted into their own modules. They are easy to implement, because they focus on in what conditions a certain procedure should be applied as opposed to how to actually execute the procedure. The model has replaceable built-in policies for the Acknowledgment (ACK) procedure, the Request to Send (RTS)/Clear to Send (CTS) procedure, the initiation and acceptance of block ACK agreements, for the decision for Medium Access Control Service Data Unit (MSDU) and Medium Access Control Protocol Data Unit (MPDU) aggregation, and similarly for fragmentation.

The many other separate components of `Ieee80211Mac` are derived from the IEEE 802.11 standard and have well-identified purposes. These components are often way more complicated than the aforementioned simple policies, but they allow deeper experimentation with many aspects of the standard. For example, they allow experimenting with more access categories, alternative backoff periods and contention mechanisms, new recovery procedures, modified block ACK agreements, different rate selection and rate control algorithms, new frame exchange sequences, alternative channel protection mechanisms and TXOP procedures, modified channel access methods and coordination functions, and new MAC data services.

As for the module structure, `Ieee80211Mac` contains two separate coordination functions as submodules, `Dcf` (depicted in Fig. 2.5) and `Hcf`. Both coordination functions are further divided into the corresponding channel access function, `Dcaf` for `Dcf`, and `Edca` having one `Edcaf` submodule per access category for `Hcf`.

Each channel access function has its own `Contention` submodule which executes the exponential backoff mechanism. The coordination functions contain further submodules, the originator and recipient MAC data services (responsible for data transformations), the rate selection and rate control algorithms (responsible for selecting the data rate on a per frame basis), the protection mechanism (which allows keeping the channel ownership), and several policies. The `Hcf` coordination function is even more complicated than `Dcf` because it includes multiple

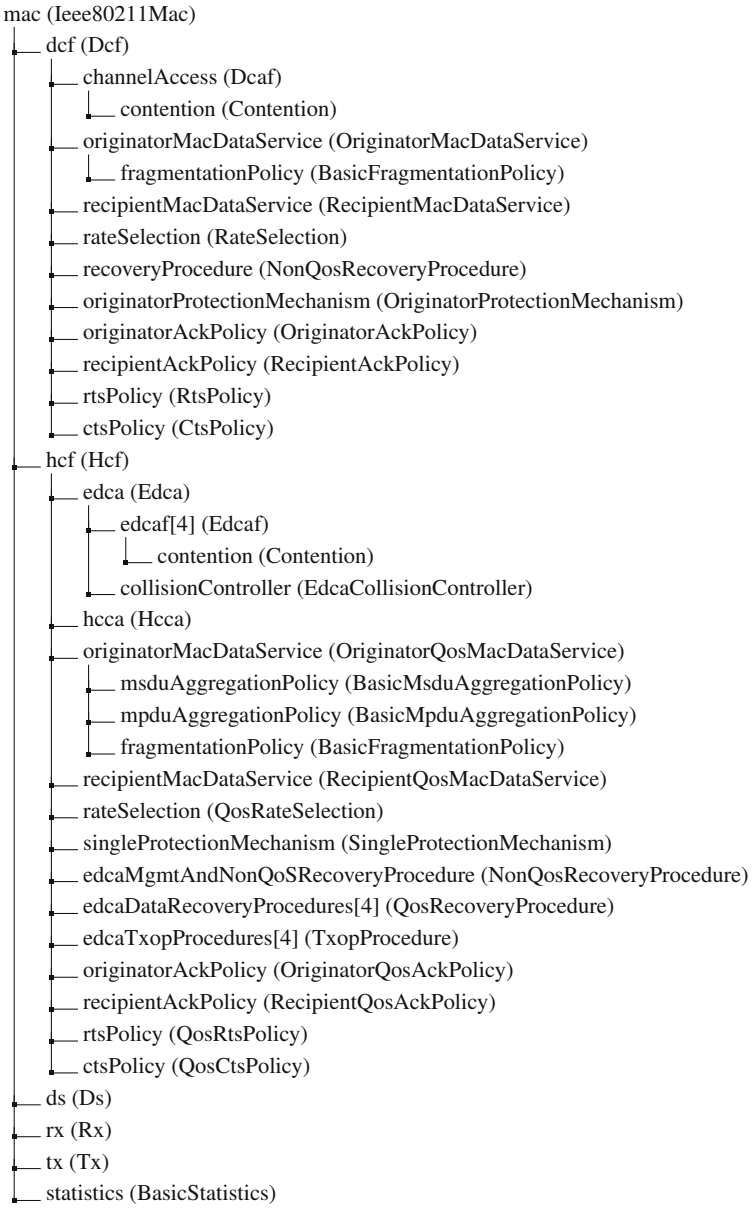



**Fig. 2.4** The submodule tree of `Ieee80211Mac` expanded. Any module can be replaced with a custom version

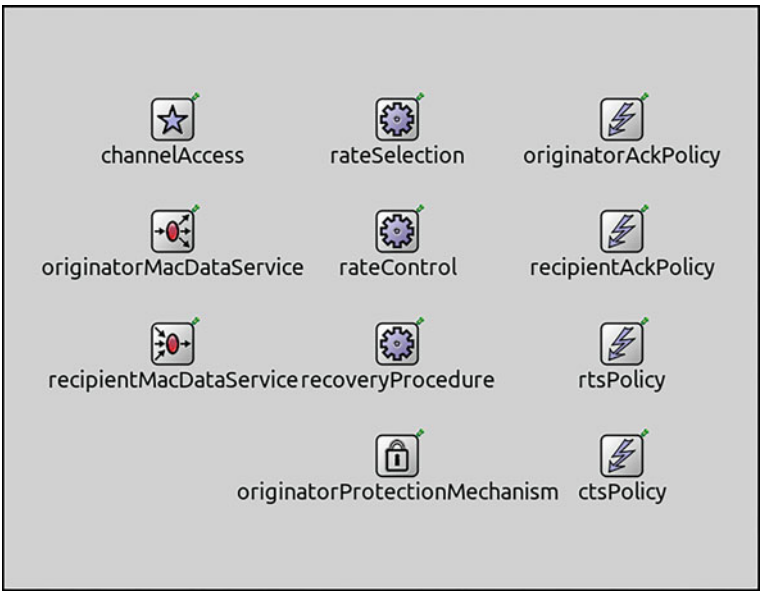



**Fig. 2.5** IEEE 802.11 distributed coordination function

`TxOpProcedure` and `QosRecoveryProcedure` submodules, and additional policy submodules for aggregation and block ACK agreements.

The presented loosely coupled structure makes it possible to build alternative, potentially nonstandard compliant, 802.11 MAC modules using standard OMNeT++ composition. For example, increasing the number of access categories with a few new ones having different parameters requires very little programming and configuration.

Additionally, any of the MAC submodules can also be replaced with alternative implementations as long as the corresponding C++ interface (having the same name as the module interface) is properly implemented. These features of the `Ieee80211Mac` module allows exploring research topics which go way beyond experimenting only with alternative policies. Refer to Sect. 2.14.3 for a complete rate control example.

### *2.7.1 WSN MAC Protocols*

The choice of wireless protocol for a Wireless Sensor Network (WSN) depends on the application requirements. IEEE 802.15.4 and IEEE 802.11 are commonly used, but there are also MAC protocols designed specifically for the requirements of wireless sensor networks: low data rate, low complexity, and low energy consumption.

B-MAC (Berkeley MAC) is a Carrier Sense Multiple Access (CSMA)-based protocol designed for low-traffic, low-power communication and is one of the most widely used protocols (e.g., it is part of the TinyOS operating system for WSNs). Its model in INET is the `BMac` module [2], but there is also a `BMacInterface`, which is a `WirelessInterface` with the MAC type set to `BMac`.

L-MAC (Lightweight MAC) is another energy-efficient MAC protocol, but it uses Time Division Multiple Access (TDMA) to give nodes in the WSN the opportunity to communicate collision-free. The network is self-organizing in terms of time slot assignment and synchronization. INET's model is `LMac` [7], and there is also an `LMacInterface`.

X-MAC is another CSMA-based protocol that reduces latency and improves energy consumption compared to B-MAC while retaining the advantages of low power listening, namely low power communication, simplicity and a decoupling of transmitter and receiver sleep schedules. INET's model is `XMac` [5], and there is also an `XMacInterface`.

The flexibility of INET network interfaces allows experimenting with various WSN MAC protocols and it also allows comparing the results for the same scenario.

### *2.7.2 Alternative Network Layer Protocols*

INET contains several network layer protocols in addition to the IPv4 and IPv6 standards. These network layers are not part of `StandardHost` and other network nodes by default, but they can be enabled and configured from INI files.

`Flooding`, for example, implements a plain packet flooding protocol which avoids re-broadcasting the same packet. The `ProbabilisticBroadcast` and the `AdaptiveProbabilisticBroadcast` modules provide stochastic data dissemination services. `WiseRoute` is a simple loop-free routing algorithm that builds a routing tree from a central network node. It is especially useful for wireless sensor networks and convergecast traffic. `NextHopForwarding` is a simplified variant of IPv4 which uses network interface module Identifiers (IDs) as addresses.

Some of these alternative network layer protocols only serve educational purposes, but they demonstrate in any case the flexibility and extensibility of INET.

### *2.7.3 Energy Modeling*

Sensor nodes are battery-powered and are often equipped with solar panels. Hence, the ability to model the storage and consumption of energy is crucial for the simulations of low-power WSNs and many wireless ad hoc networks, to be able to design and evaluate energy-aware and power saving routing and MAC protocols, and to evaluate the energy efficiency of network configurations in general.

INET's energy modules are capable of modeling processes and components to store, consume, generate, and manage energy. The models are grouped into these four basic types and either located in the */src/inet/power* folder or allocated to a specific layer (e.g., the physical layer). All models fall into one of the following two categories: those based on charge and current (abbreviated `Cc`), and those based on energy and power (abbreviated `Ep`). The category abbreviations are included in the names of each energy consumer/generator/management/storage model.

The energy modeling components are decoupled from the communication protocol stack and other models to increase interchangeability. Interface classes for the major types enable an easy exchange of models through parametric (sub-)module types.

**Energy Consumers** Energy is consumed in various components inside a communication network. CPUs, transceivers, displays, and many other components use energy when signals are processed and data packets are routed through the network. All of these components can be modeled in INET, if required for the specific use case and purpose of the simulation scenario. An INET energy consumer model is represented by a simple OMNeT++ module that implements the consumption of energy of a device or a (software) process over time. Typically, consumer models provide the current consumption or the power for the current simulation time (or a time span). INET includes two models: `AlternatingEpEnergyConsumer` as an exemplary statistical energy consumer model and `StateBasedEpEnergyConsumer` as a transceiver energy consumer model aiming at the different transceiver states (e.g., transmit or receive). The configuration excerpt in Listing 2.15 shows how the `StateBasedEpEnergyConsumer` model can be configured for the simulation of the power consumption in a wireless communication scenario. We can observe that transmitter and receiver power consumption amounts are configured here. The time that the transceiver spends in one of these states is counted to calculate the amount of consumed energy.

**Listing 2.15** Exemplary energy consumer configuration

```
1 *.host[*].wlan[*].radio.energyConsumerType = "StateBasedEpEnergyConsumer"
2 *.host[*].wlan[*].radio.energyConsumer.sleepPowerConsumption = 0.1mW
3 *.host[*].wlan[*].radio.energyConsumer.receiverIdlePowerConsumption = 2mW
4 *.host[*].wlan[*].radio.energyConsumer.receiverBusyPowerConsumption = 5mW
5 *.host[*].wlan[*].radio.energyConsumer.receiverReceivingPowerConsumption = 10mW
6 *.host[*].wlan[*].radio.energyConsumer.transmitterIdlePowerConsumption = 2mW
7 *.host[*].wlan[*].radio.energyConsumer. \
8   transmitterTransmittingPowerConsumption = 100mW # continue previous line
```

**Energy Generators** The generation of energy is also a crucial process that can be modeled with INET. Components like solar panels can provide energy resources for stationary and mobile network participants; a vital functionality for applications like WSNs. INET provides a basic module for the statistical generation of energy/power,

called `AlternatingEpEnergyGenerator`. It models the generation of energy using a physical phenomena over time, providing the current power generation level for the current simulation time. Listing 2.16 depicts a configuration example for the power generator. Note that the generator requires a so-called energy sink module where the generated energy is eventually stored (see line 2).

**Listing 2.16** Exemplary energy generator configuration

```
1 *.host[*].energyGeneratorType = "AlternatingEpEnergyGenerator"
2 *.host[*].energyGenerator.energySinkModule = "^.energyStorage" # module ref.
3 *.host[*].energyGenerator.powerGeneration = 1mW
4 *.host[*].energyGenerator.sleepInterval = exponential(10s) # random intervals
5 *.host[*].energyGenerator.generationInterval = exponential(10s)
```

**Energy Storage** Mobile, nonstationary network devices without a permanent connection to a power grid require an energy storage component. Batteries are typically employed in mobile devices, either as standalone solutions or connected to energy generators. INET provides several simple modules that take the energy generated by energy generators and model the physical phenomena used to store the produced energy. The built-in models are the `IdealEpEnergyStorage` model, an idealistic model with unlimited energy capacity, the `SimpleEpEnergyStorage` model, a nontrivial model that integrates the difference between the generated and the consumed power over time, and the `SimpleCcBattery` model, which provides a charge/current-based model of a simple battery. All storage models compute the amount of available energy at the current simulation time, to be further used in other simulation models. Listing 2.17 shows how to configure the `SimpleEpEnergyStorage` model.

**Listing 2.17** Exemplary energy storage configuration

```
1 *.host[*].energyStorageType = "SimpleEpEnergyStorage"
2 *.host[*].energyStorage.nominalCapacity = 0.05J # maximum capacity
3 *.host[*].energyStorage.initialCapacity = uniform(0J, this.nominalCapacity)
```

**Energy Management** The last of the four basic energy model types considers the task of monitoring energy storage models and managing node status. Estimation algorithms and control functionalities can be implemented here to make sure that the energy storage component is always operating in a safe state. Listing 2.18 depicts the parameters for the INET `SimpleEpEnergyManagement` module. It can be used together with the node failure modeling approaches provided by INET.

**Listing 2.18** Exemplary energy management configuration

```
1 *.host[*].energyManagementType = "SimpleEpEnergyManagement"
2 *.host[*].energyManagement.nodeStartCapacity = 0.025J # start threshold
3 *.host[*].energyManagement.nodeShutdownCapacity = 0J # shutdown threshold
```

## 2.8 Scripting

Simulation scenarios often contain complex setups that are impossible to describe by just using a static configuration provided at the simulation start. INET provides the user with scripting support so that users can express what actions shall be executed at which simulation time (e.g., change a connection or a parameter value, remove a route, and restart a router). The `ScenarioManager` module initializes and controls such scripted simulation experiments. Users can choose from actions such as creating/deleting a connection or a module, setting/changing a module or a channel parameter, and inducing a life cycle operation such as crash or reboot. Further commands are dispatched to other OMNeT++ simple modules (and executed there).

`ScenarioManager` expects the script in XML form. Listing 2.19 shows an exemplary scenario script that uses most of the built-in commands.

**Listing 2.19** Exemplary scenario manager XML script

```
<scenario>
   <set-param t="10" module="host[1].mobility" par="speed" value="5"/>
   <at t="12">
      <create-module type="inet.node.inet.Router" parent="."
                     submodule="routerX"/>
   </at>
   <at t="16">
      <set-param module="host[1].mobility" par="speed" value="25"/>
      <connect src-module="router1" src-gate="ethg[1]"
               dest-module="routerX" dest-gate="ethg[0]"
               channel-type="ned.DatarateChannel">
         <param name="delay" value="1us"/>
         <param name="datarate" value="1Gbps"/>
      </connect>
   </at>
   <at t="23">
      <shutdown module="routerX"/>
   </at>
   <at t="42">
      <disconnect src-module="routerX" src-gate="ethg[0]"/>
   </at>
   <at t="61">
      <delete-module module="routerX"/>
   </at>
</scenario>
```

The `t` attribute defines the simulation time when a command must be executed. The `<at>` command is used to collect several commands to be executed at the same time. `ScenarioManager` takes its script input directly inline in the configuration file (feasible for short scripts) or as a reference to an external XML file.

## 2.9 Recording PCAP Files

Packet Capture (PCAP) is a widely used file format for capturing, analyzing, and replaying network traffic. INET is capable of recording PCAP files from a simulation for further processing using 3rd-party tools such as **Wireshark**.

The `PcapRecorder` module is integrated into all network nodes, so capturing packet traces can be simply enabled from the INI configuration file as follows:

**Listing 2.20** PCAP recording configuration example

```
1 *.router.numPcapRecorders = 1
2 *.router.pcapRecorder[*].pcapFile = "results/router.pcap"
3 *.router.pcapRecorder[*].pcapNetwork = 1 # for ethernet
4 *.router.pcapRecorder[*].moduleNamePatterns = "eth[*]"
5 *.router.pcapRecorder[*].packetDataFilter="inet::Ipv4Header and timeToLive(3)"
```

PCAP recording captures traffic on the network level by default, but it also supports recording packets between protocols inside the network node. In order to reduce the PCAP file size and increase simulation performance, PCAP recording also supports filtering the recorded packets based on the data they contain.

## 2.10 Network Emulation

In contrast to the general definition (e.g., in [1]) and use of the term *emulation*, INET uses a broader definition to describe a system which is partially implemented in the real world and partially in the simulation.

There are several reasons to use emulation, here are the most prominent ones:

- Out of necessity, because some part of the system only exists in the real world or as a simulation model. For example, streaming video to a real media player program over a simulated network, running the real Linux Ad Hoc On-Demand Distance Vector (AODV) routing daemon over a simulated MANET, running a newly researched simulated TCP congestion algorithm over a real network.
- For validating a simulation model by replacing it with its real-world counterparts. For example, comparing the accuracy of the IEEE 802.11 MAC & PHY simulation models against a real-world Wifi card.
- To test the interoperability of a simulated model with its real-world counterparts. For example, simulating an OSPF router connected to a real network.
- As a means of implementing hybrid simulations. The real-world network may contain several network emulator devices or simulations running in emulation mode. Such a setup provides a relatively easy way for connecting heterogeneous simulators/emulators with each other, omitting the necessity of High-Level Architecture (HLA) or a custom interoperability solution.

Owing to the highly modular design of INET, there are many ways to mix the real-world elements with the simulated ones. INET allows mixing real and simulated subnetworks, network nodes, network interfaces, protocols, and applications.

The most important INET modules for emulation are the following:

- `ExtLowerEthernetInterface` represents an Ethernet network interface which has its upper part in the simulation and its lower part in the real world. This interface allows, for example, using a real network interface in a simulation. Packets received by the simulated network interface from above will be sent out on the underlying real network interface. Packets received by the real network interface (or rather, an appropriate subset of them) from the network will be received on the simulated network interface. This module requires a real or a virtual Ethernet interface on the host OS. The simulation sends and receives packets through this network interface using the OS socket API.
- `ExtUpperEthernetInterface` represents an Ethernet network interface which has its upper part in the real world and its lower part in the simulation. This interface allows, for instance, using a real routing protocol in a simulation. Basically, it works the opposite way of `ExtLowerEthernetInterface`. Packets received by the simulated network interface from the network will be received on the real interface. Packets received by the real network interface from above will be sent out on the underlying simulated network interface. This module requires a network Terminal Access Point (TAP) device in the host OS. The simulation sends and receives packets through the TAP device using the operating system's file API.
- `ExtUpperIeee80211Interface` represents an IEEE 802.11 wireless network interface which works similarly to `ExtUpperEthernetInterface`.
- `ExtLowerUdp` represents the UDP protocol which has its upper part in the simulation and its lower part in the real world. This module allows simulated applications and protocols using UDP (e.g., some routing protocols) to be run on the real network stack as standalone applications.

The host OS that runs the simulation program may need extra configuration such as adding virtual Ethernet interfaces, virtual Ethernet links, TAP and TUNnel (TUN) devices, configuring MAC and IP addresses, and adding routes. In order to operate properly, real network interfaces and their corresponding simulated parts must have the exact same configuration (e.g., MAC address, IP address).

Most often only packets are exchanged between the simulation and the real-world elements. To exchange packets, they must be converted to their binary representation, which requires enabling computing checksums for all involved protocols:

```
1 **.fcsMode = "computed" # enables FCS computation for MAC protocols
2 **.crcMode = "computed" # enables CRC computation for higher layer protocols
```

All network nodes found in INET (e.g., `StandardHost`, `Router`) can be configured to have external network interfaces and external protocols by simply configuring the appropriate module type parameter.

The following configuration shows how to replace a network interface with an external network interface connected to the `eth0` host OS device:

```
*.host1.eth[0].typename = "ExtLowerEthernetInterface" # replaces original type
*.host1.eth[0].device = "eth0" # device name on the host OS
```

In order to use emulation, the simulator must be run so that simulation time is synchronized with the real clock. One way to achieve this is to configure a special OMNeT++ event scheduler which takes care of executing events in a timely manner:

```
scheduler-class = "inet::RealTimeScheduler" # scheduler C++ class name
```

`RealTimeScheduler` also supports reading multiple file descriptors efficiently using the OS `select` Portable Operating System Interface (POSIX) [4] API. This approach works only if the relative speed of the simulation compared to real time (the simsec/sec value) is significantly larger than one.

Alternatively, for some emulations, the time perception of the participating real programs (e.g., applications or routing protocols) can be changed to be synchronized to the simulation time. The trick is to pre-load a library, when the application is started, which connects to the simulation and overrides all time-related functions of the underlying operating system. This approach does not have the above limitation but it may slow down real programs.

## 2.11 Exploring Simulations

When you need to explore, demonstrate, or troubleshoot a simulation, a good Graphical User Interface (GUI) can make this task significantly easier. Qtenv, the GUI-based simulation runtime of OMNeT++ is such a tool, and its capabilities go well beyond displaying animations.

When you run an INET simulation in Qtenv, it will display the network in a graphical form in the main area of the window. Each network node, represented by a submodule, will be shown as an icon. You can enter the node by double-clicking its icon. Doing so will reveal the submodules that represent applications, communication protocols, network interfaces, and other elements inside the node. Several submodules display basic statistics in text labels above the icons, to provide a quick overview of the network's past and current operation.

The routes of a network node can be inspected by opening a network layer (e.g., the `ipv4` submodule). Routes are contained by `ipv4`'s `routingTable` submodule (type `Ipv4RoutingTable`), and can be inspected in the *Properties* view by selecting the submodule and opening the fields `routes` or `multicastRoutes`. Network interface information such as IP or MAC address can be accessed by selecting a particular interface in the network node. It is also possible to peek into the detailed configuration of all network interfaces by selecting the `interfaceTable` submodule (of type `InterfaceTable`) and inspecting its contents in *Properties*.

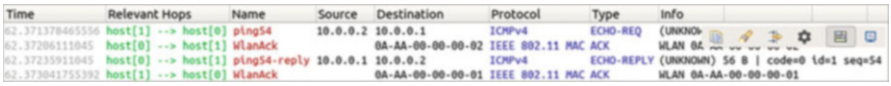



**Fig. 2.6** Packet log view of an IEEE 802.11 ad hoc ping exchange

The *Find Objects* functionality of Qtenv also makes accessing multiple objects easy, without the need to manually shift through the hierarchy of submodules. For example, using the `*.routingTable` object full path filter will find and list all routing tables within the network.

Qtenv also contains a *Packet Log* view, which, when inspecting the network module, displays the recently exchanged packets between network nodes. INET extends the default OMNeT++ *Packet Log* view with communication network simulation-specific columns. For example, INET automatically extracts source address, destination address, protocol, packet type, packet length, and additional protocol-specific information (see Fig. 2.6) from the captured packets.

When a packet is selected, the *Properties* view displays the contents of the packet. On the network level, a packet is always encapsulated into a physical signal. Inside network nodes, the physical signal is not used. The actual protocol data can be inspected further deep down in the protocol-specific data structures, in the individual chunks. Additionally, packets and chunks can be always inspected as raw hex bytes and raw bits.

The end result is that the INET *Packet Log* view allows the user to browse the recently exchanged packets similarly to the well-known protocol analyzer, **Wireshark**.

## 2.12 Visualization

When the simulation is running under Qtenv, INET is able to visualize a wide range of events and conditions in the network, for instance packet drops, data link connectivity, wireless signal path loss, transport connections, routing table routes, and many more. Visualization is implemented as a collection of configurable INET modules that can be added to simulations at will.

Since the Qtenv environment supports both 2D (via the Canvas API) and 3D (via OSG/osgEarth) graphics, there are two variants of each visualizer. For instance, there are separate `MediumCanvasVisualizer` and `MediumOsgVisualizer` modules, but there is also a `MediumVisualizer` which contains both.

The visualization features of the all visualizer modules can be used by adding an `IntegratedVisualizer` module to the network. This module contains all single-task visualizers, making all visualization features available in a simulation.

The customization of the visualization is possible via the visualizer modules' parameters. Most visualizers have a boolean "master switch" parameter which

enables the visualization in the first place. Other parameters define the scope, appearance, and other aspects of the visualization.

**Communication and Activity Visualizers** Several visualizers focus on the communication activity aspect of INET simulations. `PacketDropVisualizer`, for instance, indicates dropped packets with an animation; `MediumVisualizer` displays radio signals as well as communication and interference ranges of radios, `RadioVisualizer` indicates radio modes, transmitting and receiving states, and antenna directional characteristics; recent successful communication activity is displayed with the `PhysicalLinkVisualizer`, `DataLinkVisualizer`, `NetworkPathVisualizer`, and `TransportRouteVisualizer` for the respective OSI layer in the form of arrows. The arrows may pass through several network nodes for higher OSI layers.

**State Visualizers** A large group of visualizers display various communication-related states present in the simulation. For example, `RoutingTableVisualizer` indicates IP routes with arrows; `MobilityVisualizer` indicates mobility states, such as speed, movement trails, and orientation; `InterfaceTableVisualizer` displays information about network interfaces, such as interface name, IP and MAC address; `StatisticVisualizer` displays the last collected value of a chosen statistic; `SubmoduleInfoVisualizer` displays the internal state of a module or a submodule; `TransportConnectionVisualizer` indicates the endpoints of established transport connections; `QueueVisualizer` indicates queue length/queue utilization; `EnergyStorageVisualizer` displays the charge level of energy storages; `Ieee80211Visualizer` indicates members of an IEEE 802.11 network; `NetworkNodeVisualizer` displays a network node.

**Infrastructure Visualizers** Another class of visualizers display various auxiliary information. `PhysicalEnvironmentVisualizer`, for example, displays objects of the physical environment module; `SceneOsgEarthVisualizer` displays a map using **OsgEarth**; `SceneVisualizer` displays the simulation scene and the coordinate axes; and `TracingObstacleLossVisualizer` shows signal power loss in obstacles.

**Visualization Examples** For illustration, let us have a look at the effect of some visualizers. Fig. 2.7, for instance, is a screenshot from a wireless simulation where `MediumVisualizer` has been enabled. The circles in the screenshot indicate communication ranges of the wireless nodes. When the screenshot was taken, a radio signal was being transmitted by `host1` with a transmission power of −30 dBW. The signal wavefront had already passed `accessPoint1`, and the signal strength at that node was −111 dBW. The signal had not arrived at `router` yet.

The second example illustrates activity visualizers that indicate recent successful packet transmissions via arrows. Separate visualizers are used for the different OSI layers; for example, data link activity is shown by the `DataLinkVisualizer`.

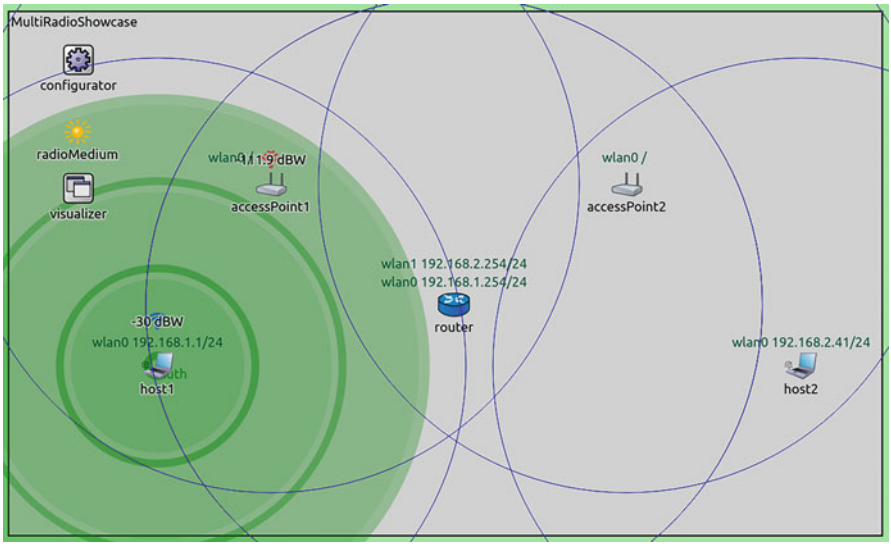



**Fig. 2.7** Visualization of wireless communication and the underlying wireless medium

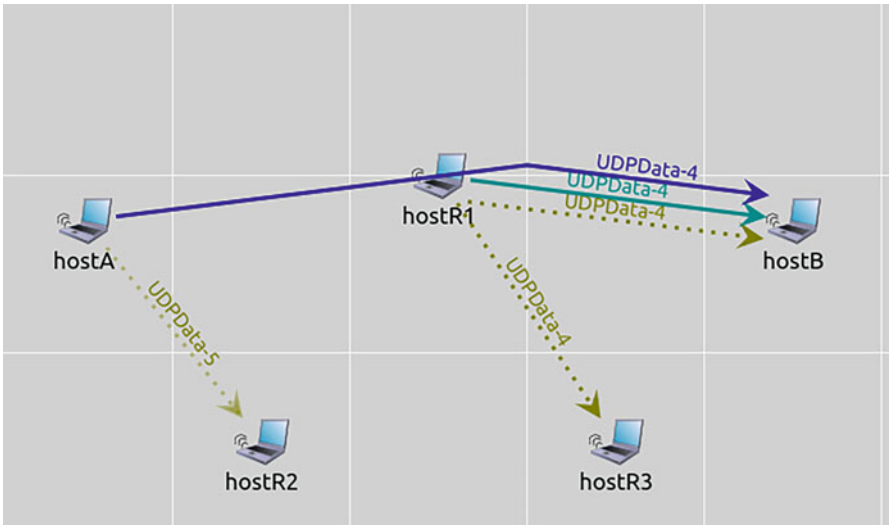



**Fig. 2.8** Visualization of communication links between different protocol layers

The screenshot in Fig. 2.8 was taken from a wireless simulation where multiple activity visualizers were enabled. In Fig. 2.8, physical link transmissions are indicated by dotted arrows, data link layer transmissions by green ones, and network layer transmission between `hostA` and `hostB` with a polyline.

The third example demonstrates the 3D visualization of INET using *OSG* and *osgEarth*. The map in Fig. 2.9 shows a small part of downtown Boston using

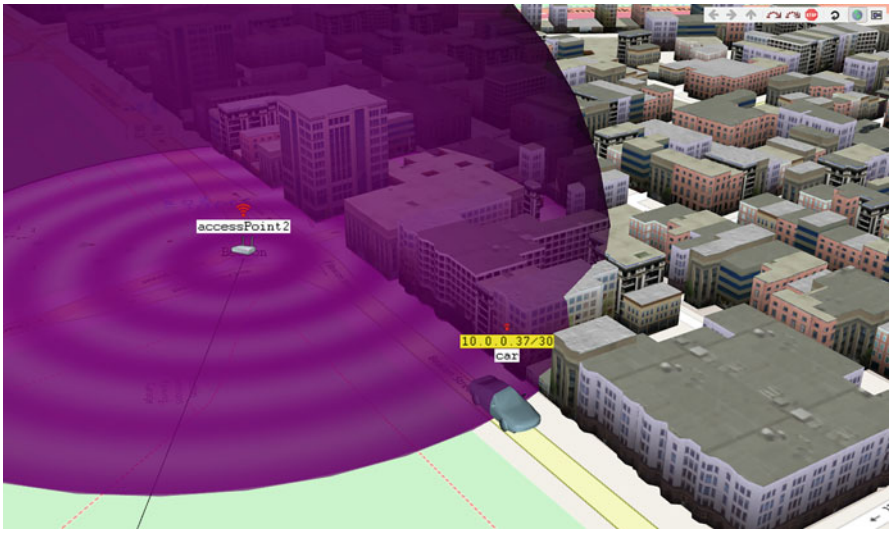



**Fig. 2.9** Visualization of a wireless network using *osgEarth* and *OpenStreetMap*

*OpenStreetMap* as data provider. The image shows a WiFi access point in a road junction and a car moving away from the access point. The figure also indicates the WiFi signal strengths above the network nodes as usual. The circles and the dome on the ground indicate an ongoing wireless transmission from the access point at the moment when the signal wavefront reaches the car.

## 2.13 Developing New Protocol Models and Other Components

Earlier sections have shown how to use the components shipped with INET to assemble and configure simulations of various types of networks. This was possible to do with only editing NED and INI files (and sometimes, additional configuration files of XML or other formats). In this section, however, we introduce the reader to some of the "secrets" of writing new INET components using C++.

For the most part, writing an INET component (protocol model, application model, or some support component) is not fundamentally different from writing any other OMNeT++ simple module. The difference mainly lies in the fact that INET is already a complex system, and in order to write code that extends it or properly interoperates with it, one needs to know about its internal APIs, mechanisms, and conventions. For example, new protocols have to be added to the list of known protocols, and they also have to be registered during initialization in order to allow the message dispatching mechanism to work.

It is impossible to cover all INET development topics within the limits of this chapter, so this section conveys only the most important concepts, and illustrate them with short code examples.

### 2.13.1 Initialization

All INET modules need to be initialized before the simulation starts. Module initialization comprises tasks like reading module parameters, initializing the internal state, scheduling timers, subscribing to signals, and processing XML configurations.

The initialization of a module often relies on information produced by the initialization of another module. For example, adding static routes to the routing table requires IP addresses to be assigned throughout the network. Therefore, INET initialization heavily relies on the OMNeT++ multistage initialization feature. INET uses one or more stages for each OSI layer, and several additional stages for mobility, power consumption, physical environment, and so on. Initialization generally occurs by OSI layers bottom up. Stages are identified in the code with constants like `INITSTAGE_LOCAL`, `INITSTAGE_PHYSICAL_LAYER`, and `INITSTAGE_LINK_LAYER`.

### 2.13.2 Life Cycle and Failure Modeling

Similar to the initialization process, INET provides a whole life cycle management of modules. Modules can support start up (representing the process of booting/starting a network node), shut down (the process of a regular/orderly shutdown), crash (a nongraceful shutdown event), and possible restart events to evaluate the effects on the network protocol or the application under examination. Life cycle operations are also executed in multiple stages to solve possible dependencies between modules, just like in the initialization process. The `LifecycleController` class provides an API to control the necessary operations.

The exact behavior of a crash or a shutdown event needs to be explicitly modeled and programmed into the affected network component. Effects like the loss of nonpersistent data, the tear-down of connections, the reset of protocol states, and the clearing of routing tables need to be considered. INET components are prepared for such life cycle events (e.g., the `Tcp` module closes all open sockets/connections, `Ipv4` clears its routing table, application modules reset their state, switches clear their MAC address tables). Components generally ignore and discard messages that are sent to them while they are down.

Life cycle events are usually triggered from scripts (cf. Sect. 2.8), but can also be done via C++ code. For example, energy management components can be configured to trigger shutdown or crash events when a device runs out of battery power.

### 2.13.3 Working with Packets

In INET, all packets are represented by the `Packet` class data structure. `Packet` is largely just a generic container, real data is stored in *chunks* inside the packet. Chunks may represent protocol headers (or parts of them), protocol trailers, and payload.

There are several kinds of chunks. The most common ones represent specific protocol headers. For example, the `Ipv4Header` chunk represents the IPv4 header. Such chunks subclass from `FieldsChunk` (which itself subclasses from `Chunk`), and have a data member for each protocol header field; for example, `Ipv4Header` contains fields like `srcAddress`, `destAddress`, and `timeToLive`.

INET also contains several generic (protocol-agnostic) chunks. `BytesChunk` stores raw bytes. There is also a `BitsChunk` for storing data that are not multiple of 8 bits. `ByteCountChunk` and `BitCountChunk` can be used when the actual values are not important.

Several chunks can be combined to form a larger chunk by adding them into a `SequenceChunk`. Another chunk type, `SliceChunk`, is particularly useful for implementing fragmentation. `SliceChunk` represents a slice of another chunk by wrapping it and additionally storing an offset and length.

Typically, packet contents is either a `SequenceChunk` of protocol-specific chunks, or in the case of emulation, a `BytesChunk`.

All protocol-specific chunks support serialization and deserialization to and from raw data, allowing packet contents to be automatically converted from one format to the other. For example, when a packet is peeked for raw data but the packet is represented with protocol-specific headers, serialization is called automatically. Similarly when a packet is peeked for a protocol-specific header but the packet is represented with a `BytesChunk`, deserialization is called automatically. Serialization and deserialization of protocol-specific headers are provided as separate C++ classes, which must be registered.

Integrated serialization has several important benefits. First, it allows simulated protocol models to talk to other implementations or real protocols transparently without relying on a specific representation. Second, it also allows injecting bit errors into arbitrary packets, potentially leading to misinterpretation at the receiver side.

When deserializing a `FieldsChunk`, it may be marked as incomplete if there is not enough binary data available for a complete representation. The resulting `FieldsChunk` may also be improperly represented if the binary data cannot be mapped to the fields of the chunk. For example, when one of the fields would contain an invalid value. Finally, a chunk may also be marked to be incorrect, without actually changing it, for example, when a packet gets corrupted in the physical layer. These states are represented with flags in the `FieldsChunk` class.

When a chunk is added to a packet, it automatically becomes immutable (frozen), and cannot be changed anymore. The advantage of freezing chunks is that immutable chunks can be shared among multiple packets using C++ shared

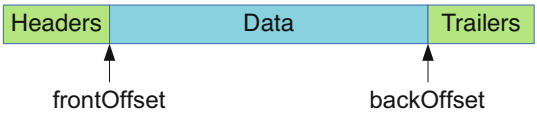



**Fig. 2.10** Popping chunks from a packet

pointers, which saves both memory and CPU time compared to duplication. Chunks can also be added to and shared between in-order send and receive queues using `ChunkQueue`, and out-of-order receive buffers using `ReassemblyBuffer` or `ReorderBuffer`.

In order to facilitate processing packets at the receiver side, packets maintain two offset values that split its contents into three parts, as indicated by Fig. 2.10. The front and back parts denote the "already processed" parts, and the middle one is the "unprocessed" part (see Fig. 2.10).

During processing in the receiver node, the packet is passed through the protocol layers, and meanwhile, headers and trailers are popped off from its beginning and its end. The pop operation adjusts the corresponding (front or back) offset, effectively reducing the remaining unprocessed part, but leaving intact the actual data stored in the packet. This approach makes processing more efficient (compared to mutating the packet) and allows inspecting the whole packet in higher layers, too.

It is entirely possible (on purpose) that a protocol misunderstands a packet. This can happen when popping a different protocol-specific header than the one that is meant to be in the binary representation (`BytesChunk`) of a packet. This normally happens in the case of corrupt packets which are actually modified, or when emulation is used. By default, INET protects users from programming errors which would cause converting one protocol-specific header to another using serialization and immediate subsequent deserialization.

Communication protocols sometimes use a Cyclic Redundancy Check (CRC) or Frame Check Sequence (FCS) field to carry some form of a checksum of some packet part. In INET, such checksums are often not modeled explicitly to save CPU cycles, and instead, just a flag is stored indicating the checksum's state (*declared correct* or *declared incorrect*). Alternatively, the checksums can also be totally disabled, or can be properly computed if explicit checksums are more important than performance. All protocol models that contain a CRC or FCS field allow the checksum computing mode to be set via parameters.

An essential part of communication network simulation is the understanding of protocol behavior in the presence of errors. INET provides several alternatives for representing corrupt packets. The alternatives range from simple but computationally cheap, to accurate but computationally expensive solutions. For example, the whole packet can be marked erroneous (inaccurate but fast), or individual chunks can be marked erroneous (more accurate and a good compromise), or bytes or bits in raw chunks can be altered (accurate but slow). Communication protocols detect errors by checking the error bits on packets and chunks, and by verifying checksums.

Several user interface and other features require understanding what data is inside a `Packet`. This task is not as simple as it seems. Simply examining the

chunks present in the packet may be insufficient, because parts or the whole packet may be represented, for instance, with `BytesChunk`. INET contains numerous protocol-specific packet dissectors which can deeply analyze the packet contents approximately according to the logic of the involved protocols.

It is also often required to filter packets based on their contents. With the help of the aforementioned packet dissectors, INET implements arbitrary packet filters. Such packet filters can be specified, for example, to visualizers or PCAP recorders to limit the scope of packets to be visualized or recorded.

### *2.13.4 Tagging Packets and Cross-Layer Communication*

When protocols communicate with each other inside a network node, they often need to pass meta-information along with packets. For example, when a transport layer protocol, for instance TCP, wants to send a packet over IP, it needs to supply the IP layer with the requested destination address in addition to the packet itself. Sometimes metadata needs to travel several layers, for example, when a sensor node application wants send a packet with a specific transmission power instead of the default one. The latter case, when nonadjacent protocols exchange metadata, is commonly called cross-layer communication.

Such metadata in INET is represented with *tags*. Tags are plain C++ data classes usually generated by the OMNeT++ message compiler. A packet can carry an arbitrary number of tags of various types. Tags can be attached either to the packet as a whole (*packet tags*) or to a specific region of the packet's contents (*region tags*).

**Packet Tags** The primary purpose of packet tags is communication between protocol layers within a network node. Packet tags can pass through protocol layers, and they can travel multiple layers from the originator protocol in both the downward and upward direction (see Fig. 2.11). Most packet tags fall into one of the following two categories:

- *Requests* carry information from a higher layer towards lower layers.
- *Indications* carry information from a lower layer towards higher layers.

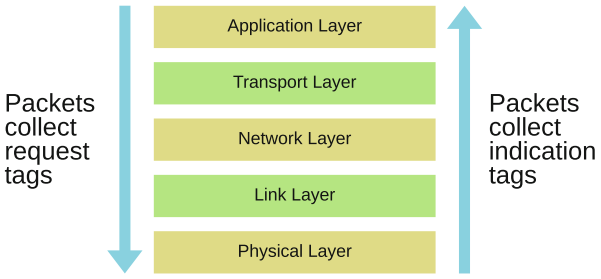



**Fig. 2.11** Using packet tags for cross-layer communication

The most notable exception (i.e., a tag which is neither a request nor an indication) is `PacketProtocolTag`, which indicates the uttermost protocol of the packet. This is necessary because the protocol cannot be unambiguously identified just by looking at the raw data.

**Request Tags** Request tags can be seen as parameters accompanying the packet. For example, a protocol above a LAN interface can attach a `MacAddressReq` to the packet to request a specific destination and/or source MAC address to be used in the layer-2 frame the packet will be encapsulated in.

Because tags can travel multiple layers, an application can control which interface a specific UDP packet is transmitted on by attaching an `InterfaceReq` tag to the packet. The forwarding protocol, e.g., IPv4, will take the request into account if possible. Similarly, an application in a wireless ad hoc network can request a specific transmission power to be used for a packet, by attaching a `SignalPowerReq` to it.

Most request tags are considered as hints. That is, the corresponding protocol implementation may chose to ignore the request, e.g., if it is configured that way.

In theory, request tags could be removed from the packet as soon as they have been processed (taken into account or willfully ignored). However, the solution chosen in INET is to remove them in the last (lowest) protocol module, that is, just before sending out the packet to the network.

The following code fragment shows how a packet is tagged by the `Ipv4` protocol module before sending it to the network interface.

**Listing 2.21** Packet tagging example

```
1  void Ipv4::sendDown(Packet *packet, Ipv4Address nextHopAddr, int interfaceId)
2  {
3     auto macAddressReq = packet->addTag<MacAddressReq>(); // add new tag for MAC
4     macAddressReq->setSrcAddress(selfMacAddress); // source is our MAC address
5     auto nextHopMacAddress = resolveMacAddress(nextHopAddr); // simplified ARP
6     macAddressReq->setDestAddress(nextHopMacAddress); // destination is next hop
7     auto interfaceReq = packet->addTag<InterfaceReq>(); // add tag for dispatch
8     interfaceReq->setInterfaceId(interfaceId); // set designated interface
9     auto packetProtocolTag = packet->addTagIfAbsent<PacketProtocolTag>();
10    packetProtocolTag->setProtocol(&Protocol::ipv4); // set protocol of packet
11    send(packet, "out"); // send to MAC protocol module of designated interface
12 }
```

**Indication Tags** Indication tags carry information obtained in lower layers during the processing of an incoming packet towards higher layers. For example, `InterfaceInd` is attached to a packet to indicate which interface the packet was received on. Another tag, `SignalPowerInd`, is attached to a packet in wireless networks by the radio's receiver module to indicate the received signal strength. Such information can be used for arbitrary purposes by any higher layer protocol, e.g., by a routing protocol.

Normally, indication tags are deleted together with the packet when the packet has been processed and is discarded in a higher layer such as an application, a routing protocol, or a stream-based transport protocol like TCP. In other cases, when

the packet is sent down again to lower layers, possibly after modification, it is the responsibility of that protocol to remove the indication tags from the packet. For example, `Ipv4` is obliged to remove the indication tags when the packet is being forwarded and sent down to an outgoing interface.

**Region Tags** As opposed to packet tags, region tags do travel across network links, as they have been designed to solve a completely different set of problems. A representative problem is how to measure in the application layer the End-to-End (E2E) delay of data sent over a TCP connection. The issue is that in a TCP stream, the data can be arbitrarily split, reordered, and merged in the underlying network. However, when the sender attaches the current time in a region tag to a byte offset (or byte region) of the data it sends, the region tag will survive the transmission over TCP, and can be read by the receiver application.

The packet data representation maintains the attached region tags as if they were individually attached to bits, and ensures that the tags move together with the data.

### *2.13.5 Using Sockets*

INET sockets are C++ classes which wrap the standard OMNeT++ message passing interface of the corresponding communication protocol into a more natural C++ interface. INET provides several socket classes for the most commonly used protocols, for example `UdpSocket`, `TcpSocket`, `Ipv4Socket`, and `EthernetSocket`.

Sockets are most often used by applications and routing protocols to access the underlying protocol services. Applications simply call the socket class member functions (e.g., `bind()`, `connect()`, `send()`, and `close()`) to create and configure sockets, and to send packets. They can also implement the socket-specific callback interface to receive packets. Applications often use several sockets of the same kind or of different kinds simultaneously.

**Using TCP Sockets** As an example, the following gives a quick glance at how TCP sockets are used in INET.

Since TCP is a connection oriented protocol, a connection must be established before applications can start exchanging data. For example, a server application can start listening at a local address and port for incoming TCP connections as follows:

**Listing 2.22** Listening for incoming TCP connections

```
1 socket.bind(Ipv4Address("192.168.1.1"), 12000); // local address/port
2 socket.listen(); // start listening for incoming connections
```

A client application can initiate a new connection by calling `connect()` with a remote address and a port:

**Listing 2.23** Initiating a new TCP connection

```
1 socket.connect(Ipv4Address("192.168.1.1"), 12000); // remote address/port
```

The `Tcp` module automatically notifies the server application about an incoming connection using the `TcpSocket:ICallback` interface, which is implemented by the application. The TCP connection is established if the server application accepts the incoming connection. After the connection is established, both applications can send data to the remote peer via a simple method call:

**Listing 2.24** Sending data through a TCP socket

```
1 socket.send(packet);
```

Packet data is enqueued by the local `Tcp` module, and it gets transmitted over time according to the TCP protocol logic. The remote `Tcp` module notifies the application about the received data using the corresponding method of the callback interface.

When the TCP connection is no longer needed, either application can close it via a method call, and the other application is notified via the socket callback interface.

Other sockets have similar C++ interfaces and they also have their own specific callback interfaces.

## 2.14 Experimenting with New Protocols and Algorithms

INET allows for experimentation by providing several options for extensions and customizations at different abstraction levels: networks, nodes, interfaces, protocols, and applications. The following three subsections demonstrate how such experiments can be carried out with INET.

### *2.14.1 TCP Congestion Algorithm Experiments*

INET's `Tcp` module, as mentioned in Sect. 2.3, supports several TCP congestion control algorithms, and users can also extend it with new ones. The TCP flavor can be selected via the `tcpAlgorithmClass` parameter of the `Tcp` module, like this:

```
1 **.tcp.tcpAlgorithmClass = "TcpWestwood"
```

The string `"TcpWestwood"` is actually the name of the C++ class that implements the TCP Westwood algorithm. Thus, adding support for a new congestion control algorithm is technically very simple: a C++ class that encapsulates the desired behavior needs to be written and registered with OMNeT++'s

`Register_Class()` macro. After that, the class can be used by setting the `tcpAlgorithmClass` parameter to contain the name of the class. Listing 2.25 shows the implementation skeleton of the new class.

**Listing 2.25** C++ skeleton for a custom TCP congestion control algorithm

```
// header file:
class INET_API TcpCustom : public TcpBaseAlg
{
  public:
    TcpCustom();
    virtual void receivedDataAck(uint32 firstSeqAcked) override;
    virtual void receivedDuplicateAck() override;
    virtual void dataSent(uint32 fromseq) override;
    virtual void segmentRetransmitted(uint32 fromseq, uint32 toseq) override;
};

// this should go into the implementation (.cc) file:
Register_Class(TcpCustom);
```

At runtime, the `Tcp` module creates a `TcpConnection` C++ object for each connection. Each `TcpConnection` creates for itself an instance of the configured TCP algorithm class. The class must be derived from `TcpAlgorithm`, a class with a number of pure virtual methods that define how `TcpConnection` will interact with it. `TcpBaseAlg`, which is used as base class in the above listing, is derived from `TcpAlgorithm`, and provides a convenient default implementation for many of `TcpAlgorithm`'s methods.

Methods of `TcpAlgorithm` are called back from `TcpConnection` on events that might be interesting for the congestion control algorithm to learn about. Such events are: connection established; connection closed; timer expired; *send* command invoked; out-of-order-segment received; receive sequence changed; data acknowledged; duplicate ACK received (this often indicates data loss); ACK received for data not yet sent (this should not happen); ACK sent; data sent; segment retransmitted; retransmit timer restarted; etc. If the reader is interested in implementing a new congestion control algorithm, the reading of the documentation of the `TcpAlgorithm` class and studying the existing implementations can provide further guidance.

### *2.14.2 Mobile Ad Hoc Network Routing*

This subsection illustrates how one can implement a new reactive Mobile Ad Hoc NETwork (MANET) routing protocol for INET. The new protocol is implemented as an OMNeT++ simple module. The NED module definition must implement the `IManetRouting` module interface to allow it to be used in `ManetRouter` network nodes. Listing 2.26 presents the NED definition.

**Listing 2.26** NED definition of the new routing protocol

```
simple NewProtocol like IManetRouting
{
   parameters:
      string ipv4Module = default(absPath("^.ipv4.ip"));
}
```

The new routing protocol is a reactive one, which means that it is going to start a route discovery when a route is needed for a packet. Therefore, the C++ module class is going to implement the `INetfilter::IHook` interface (or rather a subclass from `NetfilterBase::HookBase` which is a default implementation). *Netfilter* is a widely used programming API provided by the Linux kernel. It allows various networking-related operations to be implemented in the form of customized event handlers. For example, Netfilter interface allows implementing on-demand routing protocols, packet filtering, network address and port translation, and many other interesting functions. INET itself offers a very similar programming API for its network layer protocols (e.g., IPv4 and IPv6): the `INetfilter` interface. Modules can register hooks into a network protocol module by implementing the `INetfilter::IHook` interface. When multiple hooks are registered, their priority determines their call sequence. The following hook types are enabled by the Netfilter interface:

- `PREROUTING` hooks are called before routing a lower layer datagram.
- `POSTROUTING` hooks are called before sending a datagram to the lower layer.
- `FORWARD` hooks are called after routing a datagram.
- `LOCALIN` hooks are called before sending a datagram to the upper layer.
- `LOCALOUT` hooks are called before routing an upper layer datagram.

Hooks can examine and/or modify the packet data as well as the attached packet tags as they see fit. For instance, they may change specific fields in the protocol headers such as IPv4 addresses and/or TCP ports, or attach request tags such as `InterfaceReq` or `NextHopAddressReq` to influence the routing decision. Moreover, hooks return a result which affects the processing of the datagram with respect to the remaining hooks. The possible hook return results are as follows:

- `ACCEPT` allows the datagram to pass to the next hook.
- `DROP` does not allow the datagram to pass to the next hook, it is instead deleted.
- `QUEUE` queues the datagram for later re-injection (e.g., when the route discovery is complete).
- `STOLEN` does not allow a datagram to pass to next hook, but does not delete it.

The code snippet in Listing 2.27 shows which Netfilter hooks must be overridden.

**Listing 2.27** Routing Netfilter hook interface

```
class NewRouting : public cSimpleModule, public NetfilterBase::HookBase
{
   Result datagramPreRoutingHook(Packet *datagram) override
      { return ensureRouteForDatagram(datagram); }
   Result datagramPostRoutingHook(Packet *datagram) override { return ACCEPT; }
   Result datagramForwardHook(Packet *datagram) override { return ACCEPT; }
   Result datagramLocalInHook(Packet *datagram) override { return ACCEPT; }
   Result datagramLocalOutHook(Packet *datagram) override
      { return ensureRouteForDatagram(datagram); }
}
```

The Netfilter hook must be registered at the `Ipv4` module, which can be found by using a module path parameter, as follows.

**Listing 2.28** Netfilter hook registration

```
auto np = getModuleFromPar<INetfilter>(par("ipv4Module"), this);
np->registerHook(0, this); // register with 0 priority
```

The reactive routing protocol works as follows. If there is an existing route for the destination, then the datagram is accepted and the normal IP processing resumes. Otherwise, the datagram is inserted into a queue, and if there is no ongoing route discovery for the given destination address, then the protocol starts a new one. When the route discovery completes, all queued datagrams are re-injected for forwarding. The implementation of this process is shown in Listing 2.29.

**Listing 2.29** Implementation of the custom routing protocol

```
INetfilter::IHook::Result NewRouting::ensureRouteForDatagram(Packet *datagram)
{
   auto ipv4Header = datagram->popAtFront<Ipv4Header>();
   Ipv4Address destination = ipv4Header->getDestAddress();
   Ipv4Route *route = routingTable->findBestMatchingRoute(destination);
   if (route != nullptr) // if a route already exists for destination
      return ACCEPT;
   else {
      delayDatagram(datagram); // store datagram for later re-injection
      if (!hasOngoingRouteDiscovery(destination)) // check for discovery
         startRouteDiscovery(destination); // start a new one
      return QUEUE;
   }
}

void NewRouting::completeRouteDiscovery(Ipv4Address& destination)
{
   auto lt = delayedPackets.lower_bound(destination);
   auto ut = delayedPackets.upper_bound(destination);
   for (auto it = lt; it != ut; it++) // for all delayed datagrams
      networkProtocol->reinjectQueuedDatagram(it->second);
}
```

Typically, a routing protocol would communicate with its peers using UDP as part of the route discovery process. As the UDP communication part is similar to an application, this part is omitted from the example and the descriptions here.

### 2.14.3 IEEE 802.11 Rate Control

The IEEE 802.11 standard [3] includes a mechanism that is able to adapt the data frame bit rate to various channel conditions and to dynamically choose the optimal bit rate for the current conditions. For example, when too many packets are lost (their ACKs do not arrive), the bit rate is lowered; when several packets are sent without loss, the bit rate is increased. This mechanism is called the *rate control*. INET implements several variants (e.g., ARF, AARF, and ONOE); in this section, we illustrate how one can implement a new rate control algorithm.

The rate control mechanism is encapsulated into a submodule in the compound module `Ieee80211Mac`. The submodule has parametric type, making it very easy to implement and install a new rate control mechanism without changing other aspects of the simulation. (There are several such replaceable modules in `Ieee80211Mac`.)

The INI configuration file fragment in Listing 2.30 shows how the rate control mechanism is replaced with the new one.

**Listing 2.30** Rate control configuration example

```
[Config RateControlExperimentation]
*.host[0].wlan[*].mac.dcf.rateControl.typename = "NewRateControl"
```

The `NewRateControl` module type must be declared in NED (Listing 2.31) and implemented in C++ (Listing 2.32) before it can be used. The C++ class must implement the `IRateControl` C++ interface, because other modules will use it via C++ method calls.

**Listing 2.31** NED definition of `NewRateControl` module

```
simple NewRateControl like IRateControl {
    //...
}
```

The C++ class is informed about important events by other parts of the MAC by calling its `frameTransmitted()` and `frameReceived()` methods, and the MAC obtains the bit rate suggested by the algorithm by calling its `getRate()` method. The class may use additional information available from other modules by subscribing to signals or accessing the required modules via C++ directly.

**Listing 2.32** Implementation of the new `NewRateControl` class

```
class NewRateControl : public IRateControl {
    IIeee80211Mode *getRate() { ... }
    void frameTransmitted(Packet *f, int retryCount, bool isSuccessful,
                          bool isGivenUp) { ... }
    void frameReceived(Packet *frame) { ... }
}
```

## 2.15 Conclusion

The introduced INET Framework is a versatile library of simulation models for all layers of the communication stack. Users base their simulative experiments on the existing models, extend them, or derive their own simulation models and integrate them into the INET cosmos. Being developed and maintained by a number of OMNeT++ developers also ensures that INET model implementations often use new OMNeT++ features. New OMNeT++ and INET users are therefore advised to go through the code base of INET and use the extensive examples library and the showcases and tutorials provided by the OMNeT++ developers and the community.

## References

1. Beuran, R.: Introduction to Network Emulation. Pan Stanford Publishing, Singapore (2012)
2. Förster, A.: Code contribution: implementation of the B-MAC protocol for WSN in MiXiM. In: Proceedings of the 4th International Workshop on OMNeT++ (2011). https://summit.omnetpp.org/archive/2011/uploads/slides/OMNeT_WS2011_S5_C1_Foerster.pdf
3. IEEE Standards Association: IEEE Standard for Information Technology – Telecommunications and Information Exchange Between Systems Local and Metropolitan Area Networks– Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE Std 802.11-2016 - Revision of IEEE Std 802.11-2012, Institute of Electrical and Electronics Engineers, Piscataway (2016). https://doi.org/10.1109/IEEESTD.2016.7786995
4. IEEE Standards Association: IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. Tech. rep. (2018). https://doi.org/10.1109/IEEESTD.2018.8277153
5. Oller, J., Demirkol, I., Casademont, J., Paradells, J., Gamm, G.U., Reindl, L.: Has time come to switch from duty-cycled MAC protocols to wake-up radio for wireless sensor networks? IEEE/ACM Trans. Netw. **24**(2), 674–687 (2016). https://doi.org/10.1109/TNET.2014.2387314
6. Rüngeler, I., Tüxen, M., Rathgeb, E.P.: Integration of sctp in the omnet++ simulation environment. In: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops, Simutools'08, pp. 78:1–78:8. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels (2008). http://dl.acm.org/citation.cfm?id=1416222.1416310
7. van Hoesel, L., Havinga, P.: A lightweight medium access protocol (LMAC) for wireless sensor networks: reducing preamble transmissions and transceiver state switches. In: Proceedings of the 1st International Workshop on Networked Sensing Systems (INSS), pp. 205–208. Society of Instrument and Control Engineers (SICE), Tokyo (2004)

# Chapter 3
# INETMANET Framework


**Alfonso Ariza and Vincenzo Inzillo**

## 3.1 Introduction

Mobile Ad Hoc NETworks (MANETs) is an area of networking research that has been the focus of a special attention in the latest years. With the emergence of the Internet of Things (IoT) concept, the protocols that had been developed for MANETs have returned to be an object of interest. A characteristic of this type of networks is the auto-configuration capability. The nodes in a MANET can discover the topology of the network, adapting autonomously to possible topology changes, and creating a mesh network with the ability to send data using intermediate nodes (e.g., routers), thus being able to reach nodes that are outside their coverage area.

Several simulation tools have offered solutions to support the modeling and simulation of MANETs. However, all have several aspects in common, namely support for MANET routing protocols, the inclusion of wireless technologies that could work in ad-hoc mode and in the Industrial, Scientific and Medical (ISM) bands, wireless propagation models, and mobility models that allow to simulate the movement of the nodes. Some of these simulators also include energy models, which allow simulating the battery consumption of the nodes, as well as obstacles models, which allow studying the performance of a network in the presence of obstacles that attenuate the radio signal strength.

A. Ariza (✉)
E.T.S.I. Telecomunicación, Universidad de Málaga, Málaga, Spain
e-mail: aarizaq@uma.es

V. Inzillo
DIMES, University of Calabria, Arcavacata, Italy
e-mail: v.inzillo@dimes.unical.it



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_3

The first version of the INET Framework had serious limitations in the simulation of MANET networks. It included only a basic support of some of these modules, offering some mobility models and the support of the standard IEEE 802.11b-1999.

Keeping this in mind, INETMANET emerged initially as a fork of the INET Framework focused on the simulation of wireless MANETs with the inclusion of several MANET routing protocols. It offered a modified INET Framework including an adaptation of the protocol Ad Hoc On-Demand Distance Vector (AODV) [13] while overcoming some of the INET Framework limitations. A significant limitation of the first INET versions (before the release of the current INET Framework version 4.0, see Chap. 2) regarding the simulation of realistic MANET networks was the link layer protocol support and the lack of routing protocols.

INETMANET overcame this limitation by introducing advanced IEEE 802 protocols; in this case, it included 802.11a/g/e and 802.15.4 (now both included in INET as well), to enable the simulation of realistic scenarios and complex networks.

Later, INETMANET included the propagation models developed for MIXIM,[1] allowing a comparison with the simulation results obtained with Network Simulator 2 (ns-2) (even if the interference model of ns-2 is very basic).

Another limitation that the INET Framework had in its initial versions (up until version 3.0 where energy consumer and producer models were integrated) was the lack of an energy model. With the objective of solving this problem, INETMANET adapted the code of MiXiM, thus facilitating the simulation of the energy consumption in wireless networks.

With the objective of using some of the tools developed for ns-2, the module `Ns2MotionFile` was included in INETMANET, thus allowing the usage of several tools that can generate mobility patterns for ns-2.

Another aspect to be solved was the necessity of a source of traffic with the capacity of generating complex patterns. To solve this, INETMANET includes the model `UDPBasicBurst` that allows generating different patterns of traffic with the objective to test the performance of MANETs.

Several simulation models and support code that was originally developed for INETMANET have been included in INET Framework over time. However, INETMANET continues to have differences with INET, focusing on the following aspects:

- routing protocols,
- mobility models,
- applications model,
- routing and forwarding in the link layer,
- antenna models,
- and miscellaneous tools.

[1]The MiXiM project has been discontinued, and its contents have been merged into the INET Framework. New projects should be based on a recent version of INET instead of MiXiM.

Despite these differences, INETMANET is fully compatible with INET. Any code and model developed for INET works without modification in INETMANET.

## 3.2 Routing Protocols

The initial fork of INETMANET had the objective to include several MANET routing protocols, which at that moment were not included in INET. Later, the INET Framework incorporated several routing protocols, but INETMANET continues to support link-layer routing and protocol implementations that are not present in INET. The INETMANET specific routing protocol implementations that were ported from external projects are depicted in Fig. 3.1 and listed as follows:

- `AODV-UU`[2]—Ad Hoc On-Demand Distance Vector [13],
- `DYMO-UM`[3]—DYnamic MANET On-demand [14],
- `DYMO-FAU`[4]—DYnamic MANET On-demand [14],
- `OLSR-UM`[5]—Optimized Link State Routing [7],
- `DSR-UU`[6]—Dynamic Source Routing [10],
- `BATMAN`[7]— Better Approach To Mobile Adhoc Networking [12],
- `SAORS`[8]—Socially-Aware Opportunistic Routing System,
- `PASER`[9]—Position-Aware Secure and Efficient Mesh Routing Protocol [16, 17],
- `DSDV`—Destination-Sequenced Distance Vector [15].

Most of these routing protocols are defined in their respective Request for Comments (RFC) as programs that run in the user space and use the User Datagram Protocol (UDP) transport protocol [10, 13–15]. However, INETMANET has a peculiar implementation as the MANET routing protocols are directly connected to the Internet Protocol (IP) network layer module. Nevertheless, the MANET protocols encapsulate packets into UDP datagrams to ensure realistic overhead simulation. The main difference is that the identifier of the protocol type in the IP header is set to 254.

[2] https://sourceforge.net/projects/aodvuu/.

[3] https://sourceforge.net/projects/dymoum/.

[4] http://www.ccs-labs.org/~sommer/projects/dymofau/.

[5] https://sourceforge.net/projects/um-olsr/.

[6] https://sourceforge.net/projects/dsruu/.

[7] https://downloads.open-mesh.org/batman/releases/batman-0.3.2/batman-0.3.2.tar.gz.

[8] https://sourceforge.net/projects/saors/.

[9] http://www.paser.info/index.php/downloads.html.

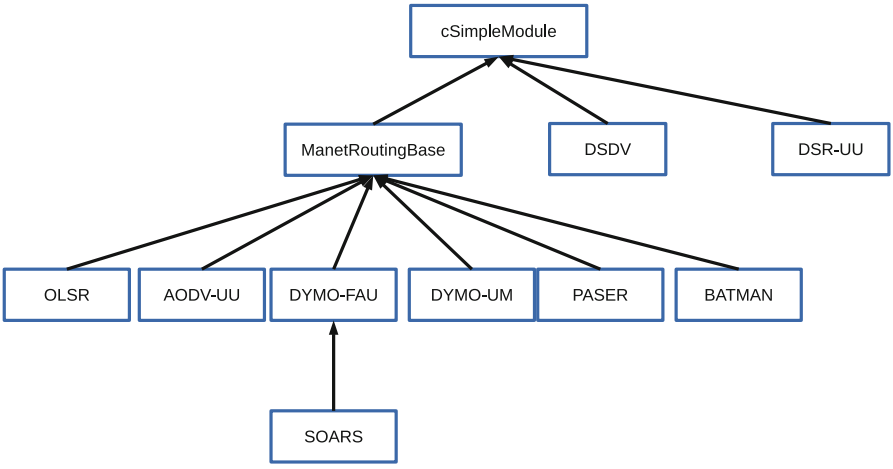



**Fig. 3.1** Inheritance of the MANET routing protocols implemented in INETMANET

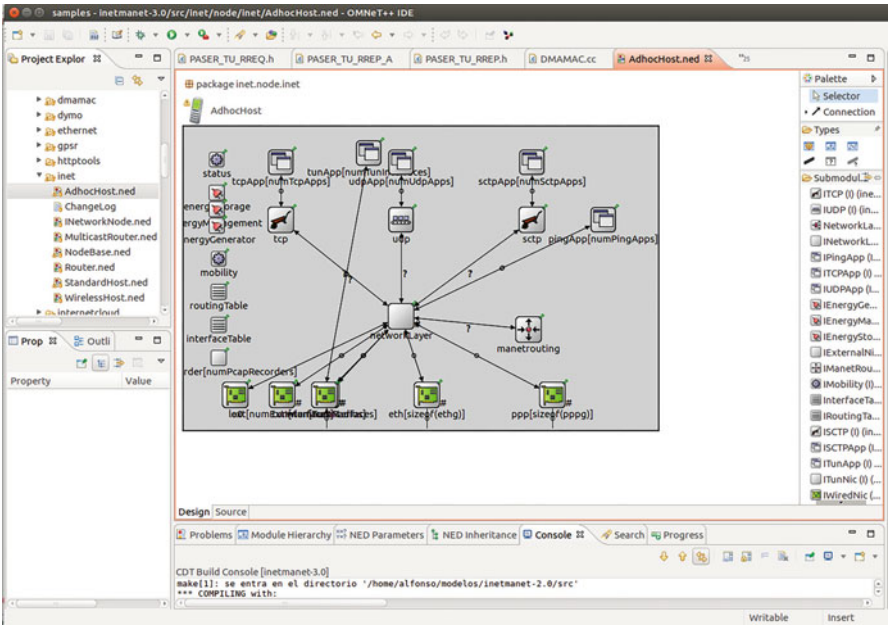



**Fig. 3.2** MANET routing node with the MANET routing protocol connected directly to the `networkLayer` module

The MANET routing protocol is directly connected to the network layer module (cf. Fig. 3.2). To activate the routing protocol it is enough to set the option `**.routingProtocol` in the configuration file. For example, to select the Dynamic Source Routing (DSR) protocol [10] one only has to include the option shown in Listing 3.1 in the configuration file (by default *omnetpp.ini*).

**Listing 3.1** Exemplary selection of the DSR protocol

```
1 **.routingProtocol = "DSR"
```

**Table 3.1** List of MANET routing protocols and the option that has been selected in the configuration file to enable the corresponding routing protocol

| Option | Routing protocol |
|---|---|
| DSR | DSR-UU |
| AODVUU | AODV-UU |
| DSDV_2 | DSDV |
| OLSR | OLSR |
| OLSR_ETX | OLSR with ETX implementation |
| DYMOUM | DYMO-UM |
| DYMO | DYMO-FAU |
| BATMAN | BATMAN |
| SaorsManager | SAORS |
| PASER | PASER |

The list of available protocols for this option is presented in Table 3.1.

Besides the configuration parameters specific to each protocol, some common configuration parameters are available (cf. Listing 3.2), the most important one being the `interfaces` parameter, which controls the interfaces that the routing protocol will use to disseminate the routing packets.

**Listing 3.2** Common configuration options of the routing protocols

```
1 string excludedInterfaces = default(" "); // list of interfaces excluded by the
      MANET routing protocol
2 string interfaces = default("prefix(wlan)"); // list of routing interfaces were
      the routing protocol is active, the prefix(namei) indicate all the
      interfaces with "namei" in his names
3 bool useManetLabelRouting = default(true); // the routing en-tries are marked
      IPv4Route::MANET, this label force an exact search with independent of
      the mask
4 bool useICMP = default(true); // By default the module use the ICMP but it's
      necessary to activate with the method set-SendToICMP(true)
5 bool setICMPSourceAddress = default(false); // Set the datagram source address
      if the address is undefined to the address of node. This parameter allow
      that the ICMP messages could arrive to the source applications
6 bool manetPurgeRoutingTables = default(true); // The code deleted the entries
      in the IP routing table at the simulation be-ginning.
7 bool autoassignAddress = default(false); // assign IP addresses automatically
      to the interfaces
8 string autoassignAddressBase = default("10.0.0.0");
```

### 3.2.1 `ManetRoutingBase` Class

As depicted in Fig. 3.1, most of the routing protocols are derived from the `ManetRoutingBase` class. This class offers an abstraction interface to other modules in the INET Framework (e.g., access to the information of the routing or interfaces tables) that facilitates the adaptation of code. This class simplifies porting code from other simulators like ns-2 or its successor, Network Simulator 3 (ns-3).[10] Another advantage of this class is that it facilitates the adaptation of routing protocols to work in different layers. The routing protocols can work in the network layer, but also in the link layer, facilitating the routing and forwarding within the link layer as proposed in the standard IEEE 802.11s [8].

The basic services that `ManetRoutingBase` offers to derived classes are:

- signal processing,
- timer triggering functions,
- the position of the node in the simulation area (similar to a Global Positioning System (GPS) based device),
- access to the Internet Control Message Protocol (ICMP),
- access to the IP routing table,
- access to the interface table (it is also possible to access the node interfaces),
- discovery of the wireless interfaces,
- UDP encapsulation,
- and a transparent interface that allows the same code to be executed using IP, or directly in the link layer without the network layer.

It is mandatory that protocols derived from the `ManetRoutingBase` class invoke the method `registerRoutingModule()` in the `initialize()` function in the stage `INITSTAGE_ROUTING_PROTOCOLS` as shown in Listing 3.3.

**Listing 3.3** Example for the protocol registration in the `initialization()` function

```
1  class MyClass : public ManetRoutingBase
2  {
3  public:
4     MyClass();
5     virtual ~MyClass();
6     virtual void initialize(int stages) override{
7        if (stages == INITSTAGE_ROUTING_PROTOCOLS) {
8           // Initialize common manet routing protocol structures, mandatory
9           registerRoutingModule();
10       }
11    }
12 }
```

[10] ns-3 project website: https://www.nsnam.org/.

#### 3.2.1.1 Signals

Table 3.2 shows the signals that a routing protocol can receive from the lower layers, and the method that will be triggered by these signals. These methods simplify the implementation of the link-layer feedback mechanisms. For example, the method `processLinkBreak()` (see Listing 3.4—line 20) will be executed automatically every time the node cannot confirm the correct reception of a frame, which allows implementing the link-layer feedback functionalities present in the AODV, DSR, and DYnamic MANET On-demand (DYMO) protocols. The method `processFullPromiscuous()` (see Listing 3.4—line 29) is used by the DSR protocol to implement the overhearing mechanism recommended in the standard.

**Table 3.2** List of the signals received by the routing modules

| Signal | Method | Description |
|---|---|---|
| NF_LINK_PROMISCUOUS | `processPromiscuous()` | Allow to receive any packet that is destined to this node in the routing module. |
| NF_LINK_FULL_PROMISCUOUS | `processFullPromiscuous()` | Send all packets received by the node to the routing modules (even the packets that are destined to other nodes). |
| NF_LINK_BREAK | `processLinkBreak()` | Notify the routing module that the transmission of a packet has failed. |

**Listing 3.4** Example of signals usage

```
1  class MyClass : public ManetRoutingBase
2  {
3  public:
4  MyClass();
5  virtual ~MyClass();
6  virtual void initialize(int stages) override
7  {
8     if (stages == INITSTAGE_ROUTING_PROTOCOLS) {
9        // Initialize common manet routing protocol structures, mandatory
10       registerRoutingModule();
11       /// Activate the LLF break (subscribe to NF_LINK_BREAK)
12       linkLayerFeeback();
13       /// activate the promiscuous option (subscribe to NF_LINK_PROMISCUOUS)
14       linkPromiscuous();
15       /// Activate the full promiscuous option (subscribe to
               NF_LINK_FULL_PROMISCUOUS)
16       linkFullPromiscuous();
17    }
18 }
19
20 virtual void processLinkBreak(const cObject *details) override
21 {
22    if (dynamic_cast<IPv4Datagram *>(const_cast<cObject*> (details))) {
23       dgram = const_cast<IPv4Datagram *>(check_and_cast<const IPv4Datagram *>(
               details));
24       packetFailed(dgram);
25       return;
```

```
26    }
27 }
28
29 void processFullPromiscuous(const cObject *details) override
30 {
31    if (dynamic_cast<Ieee80211TwoAddressFrame *>(const_cast<cObject*> (details))
          ) {
32       Ieee80211TwoAddressFrame *frame = dynamic_cast<Ieee80211TwoAddressFrame
             *>(const_cast<cObject*>(details));
33       L3Address sender(frame->getTransmitterAddress());
34    }
35 }
36
37 void processPromiscuous(const cObject *details) override
38 {
39    if (dynamic_cast<Ieee80211DataFrame *>(const_cast<cObject*> (details))) {
40       Ieee80211DataFrame *frame = dynamic_cast<Ieee80211DataFrame *>(const_cast
             <cObject*>(details));
41       cPacket * pktAux = frame->getEncapsulatedPacket();
42       if (dynamic_cast<IPv4Datagram *>(pktAux)) {
43          ....
44       }
45    }
46 }
47 };
```

#### 3.2.1.2 Timer Handling

Timers can be implemented using the `ManetRoutingBase` class. It is necessary to derive the new timer from the class `ManetTimer` and re-implement the method `expire()`. This method will be automatically executed when the timer expires.

The method `scheduleEvent()` programs the next event. It should be called at the end of `handleMessage()` to guarantee that events are correctly executed at the programmed simulation time. At the beginning of `handleMessage()` the method `checkTimer()` should be called (cf. Listing 3.5).

**Listing 3.5** The `checkTimer` method

```
1 if (checkTimer(msg)) {
2    // Timer events
3    scheduleEvent(); // before return schedule events
4    return;
5 }
```

In Listing 3.6, it is possible to observe an example of how to use the `ManetTimer` class. As shown there, it is necessary to call the method `createTimerQueue()` in the `initialize()` method.

**Listing 3.6** Example of how to use `ManetTimer` class

```
1 class MyTimer : public ManetTimer
2 {
3 public:
4    virtual void expire() {
5       EV << "expire " << endl;
6    }
```

```
7     myTimer(ManetRoutingBase* agent) : ManetTimer(agent) {};
8  };
9
10 class MyClass : public ManetRoutingBase
11 {
12 public:
13    MyClass();
14    virtual ~MyClass();
15    virtual void initialize(int stages) override{
16       if (stages == INITSTAGE_ROUTING_PROTOCOLS) {
17          // Initialize timer queue, it is necessary for to use ManetTimer class
18          createTimerQueue();
19          // Initialize common manet routing protocol structures, mandatory
20          registerRoutingModule();
21       }
22    }
23 }
24
25 virtual void handleMessage(cMessage *msg) override {
26    if (checkTimer(msg)) {
27       // Timer events
28       scheduleEvent(); // before return schedule events
29       return;
30    }
31    MyTimer * myTimer = new MyTimer();
32    myTimer->resched(simTime()+10); // schedule the event 10 seconds in the
          future
33    scheduleEvent(); // before return schedule events
34 }
```

#### 3.2.1.3 Position Access

`ManetRoutingBase` provides methods that allow accessing the position, speed, and direction of the node (see Listing 3.7).

**Listing 3.7** Methods to access position, speed, and direction

```
1 virtual const Coord& getPosition(); .
2 virtual double getSpeed(); // provides the actual node position in the
      landscape
3 virtual const Coord& getDirection();
```

- `getPosition()`: provides the current position of the node in the playground.
- `getSpeed()`: provides the speed of the node in meters per second.
- `getDirection()`: provides a unit vector with the correct movement direction of the node.

#### 3.2.1.4 Encapsulation

The method `sendToIp()` allows encapsulating the messages over UDP before sending them to the IP layer. In case that the routing protocol is executed directly in the link layer, the UDP encapsulation is not performed.

**Listing 3.8** Encapsulation methods

```
1 virtual void sendToIp(cPacket *pk, int srcPort, const L3Address& destAddr, int
      destPort, int ttl, double delay, const L3Address& ifaceAddr);
2 virtual void sendToIp(cPacket *pk, int srcPort, const L3Address& destAddr, int
      destPort, int ttl, double delay, int ifaceIndex = -1);
```

#### 3.2.1.5 Access to the Routing Table

In order to simplify porting the code of routing protocols to INETMANET, the `ManetRoutingBase` class provides methods that ease the access and modification of the information in the routing tables (cf. Listing 3.9).

**Listing 3.9** Access to the routing table

```
1 void omnet_chg_rte(const L3Address &dst, const L3Address &gtwy, const L3Address
      &netm, short int hops, bool del_entry, const L3Address &iface =
     L3Address());
2 L3Address omnet_exist_rte(const L3Address&);
3 virtual void omnet_clean_rte();
```

- `omnet_chg_rte()`: allows writing or deleting a new routing entry in the Internet Protocol Version 4 (IPv4) module. It can be easily extended to work with an Internet Protocol Version 6 (IPv6) routing table, allowing the routing protocols to work with IPv4 and IPv6 without any significant modification.
- `omnet_exist_rte()`: extracts the next-hop address to a particular destination in the IP routing tables. It returns the address of the next hop if the entry exists in the routing table, otherwise it returns an unspecified `L3Address`.
- `omnet_clean_rte()`: deletes all the entries in the IPv4 routing table.

### *3.2.2 Simulating MANET Routing Protocols*

In this section, we cover typical mistakes that can occur when simulating MANET routing protocols.

#### 3.2.2.1 Simulating Time

One of the most common issues is the use of a short simulation time. In order to obtain sound results, it is necessary to choose a simulation time that allows generating several hundred thousands of the type of events that we want to study; for instance, in case of routing protocols it is necessary to put attention to the actualization mechanism. If the routing protocol is proactive and the actualization of the routing tables is every 60 s, a simulation time of 300 s, a value quite commonly used in the simulation of MANET networks, is inadequate, as it only forces five actualizations of the routes in the node.

If the repetition of the experiment with different seeds presents high variations, this can be a symptom of a too short simulation time.

#### 3.2.2.2 Landscape Area and Coverage Area

Sometimes the simulation area and the radio coverage area can be of comparable size. In this case, this will not be adequate to simulate a multi-hop scenario, as nodes will all be within each other's communication range. On the other hand, if the area is too big and the nodes are deployed randomly in the floor plan, a partitioned network might be possible, resulting in a lack of communication among nodes. This problem can be solved either by adding more nodes, by reducing the simulation area, or by placing the nodes adequately within the simulation area.

#### 3.2.2.3 Mobility Models

The used mobility model has an enormous impact on the behavior of routing protocols. It must be chosen carefully. Many of the used models do not reflect the behavior of real nodes, but this does not imply that they are not useful. If the objective is to test the performance of a routing protocol under stress, *random way-point* or *constant speed* models can be useful, even if these mobility models are not very realistic.

The most common mistakes when using mobility models are:

- *Inadequate model*: As an example, if the network under analysis is composed of laptops, the mobility model should represent the behavior of nodes that are static for the whole simulation period. If the network is a Vehicular Ad Hoc Network (VANET), a random way-point model is not meaningful, as vehicles usually move along predefined paths (roads). Moreover, some models tend to concentrate the nodes in the center of the simulation area. In this case, it is possible that all the nodes could be neighbors.
- *Inadequate parameters*: If the objective is to simulate a network of pedestrians using smartphones, the speed must be consistent with the speed of a pedestrian, not an Olympic runner. Another common issue with the parameters is related to the random way-point model: if the minimum allowed speed is near to 0, the nodes will stop their movement together with the time (if the speed is very close to 0, the node will need a time bigger than the simulation time to move a meter).

#### 3.2.2.4 Statistic Errors

There are two common statistic errors in the simulation. The first one is related to the trend of the plots, as displayed in Fig. 3.3. The trend shown for the Packet Delivery Ratio (PDR) as a function of the mobility of nodes is quite unlikely. This

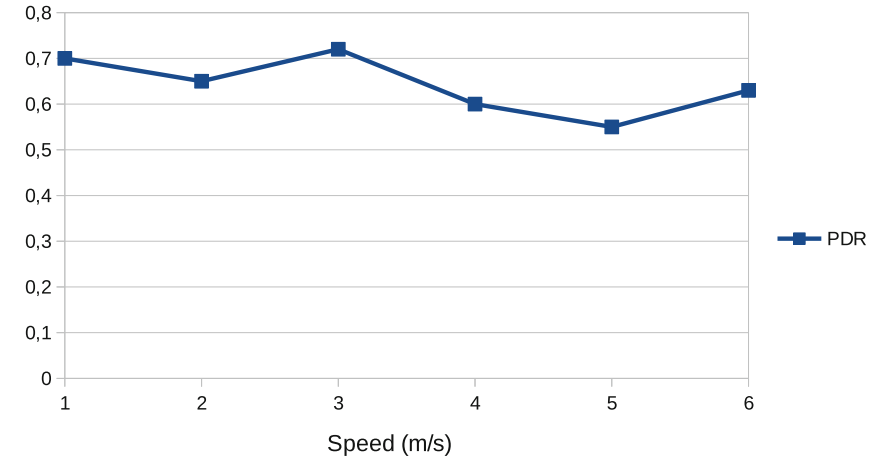



**Fig. 3.3** Exemplary tendency graph with possible simulation errors

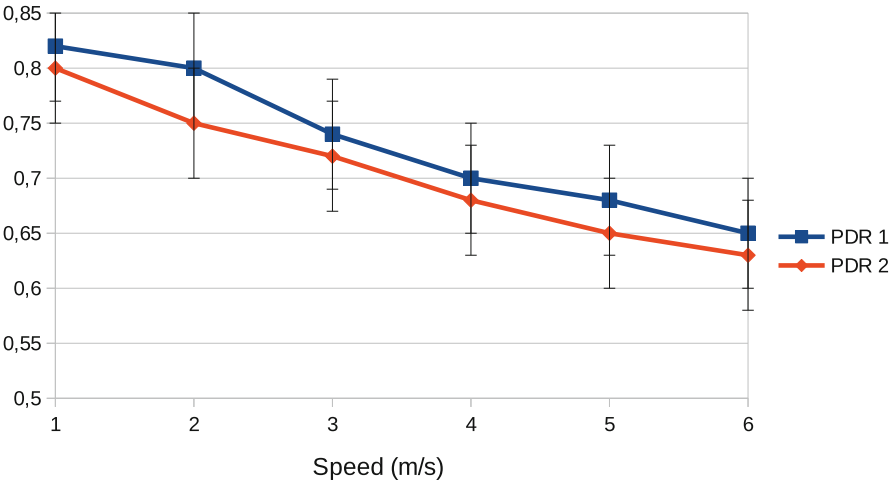



**Fig. 3.4** Example of two functions with overlapping confidence interval

behavior could be due to several reasons: a too short simulation time, errors in the configuration parameters, or errors in the data used to create the plot. There can also be multiple possible causes. Further investigations are therefore necessary when a figure with a strange trend is obtained.

Another common error is to draw conclusion without considering result variances. We can use Fig. 3.4 to undermine this. If we consider only the functions without the errors, then function PDR 2 is better compared to function PDR 1. But if we consider the error interval computed using the *T-Student* confidence interval, we can observe that both functions are overlapping. In this case, it is practically impossible to state that function PDR 2 certainly performs better than function PDR

1. Only when the confidence intervals are non-overlapping it is possible to state, from the point of view of the data obtained in the simulation, that a solution is better than the other.

In order to obtain the confidence intervals, one should repeat the simulation multiple times using different seeds. There is not an ideal number of replications, it depends on the variance of the results. Replicating the simulation at least 5 times can be a starting point. Considering that an inverse square root relationship exists between confidence intervals and sample sizes, if the variance of the results is high, it will be necessary to increase the number of experiments in order to reduce the confidence interval. An excellent book to delve into this topic is "Design and Analysis of Experiments" [11].

#### 3.2.2.5 Traffic Sources

The selection of an appropriate traffic source is fundamental for the simulation. This source must be chosen depending on the parameters under study. For example, if the goal is to evaluate the performance of a routing protocol in terms of PDR or End-to-End (E2E) delay, the traffic source must be of type UDP. It is very common to choose the destination for every packet randomly. This type of selecting destinations presents disadvantages for reactive protocols against proactive, but the real traffic has a bursty nature. Usually, the traffic from a source to a destination is composed of several tens to hundreds of consecutive packets.

## 3.3 Physical Layer Models

The INET Framework provides several modules that allow the simulation of the most common node features and operations related to the physical layer, including channel propagation, modulation, energy-consumption, and channel-error models.

Figure 3.5 illustrates the hierarchical block diagram of the main modules involved in the physical-layer operations. The radio model describes the physical device and implements transmission/reception functionalities. The `Radio` module serves an analogical model that is responsible for supporting the wireless channel propagation model for example. The `Radio` module contains the following main parameters:

**Listing 3.10** Main parameters from the *Radio.ned* file

```
1 module Radio like IRadio {
2 parameters:
3 string antennaType; // the antenna model
4 string transmitterType; // transmitter model
5 string receiverType; //receiver model
6 string energyConsumerType = default(""); // energy consumer model
7 string radioMediumModule = default("radioMedium"); // path of the medium module
8 string energySourceModel = default(""); // path of the energy source module
```

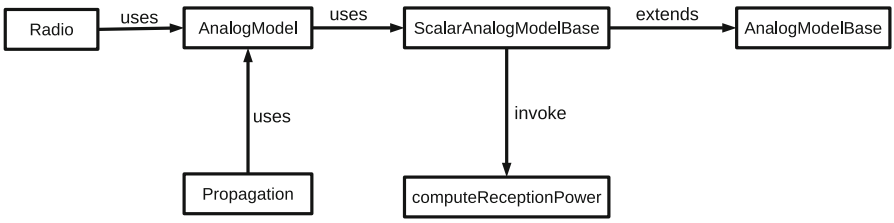



**Fig. 3.5** Physical layer logical block diagram

The `antennaType` parameter points to the *.ned* of the used antenna module; the `transmitterType` and `receiverType` indicate, respectively, the kind of transmitter and receiver module that is equipped in the mobile node. Please note that the `antennaType`, the `transmitterType`, and the `receiverType` parameters are tightly related to the kind of `radioMediumModule` set in the *.ini* configuration file. More specifically, the INET Framework provides several `radioMediumModule`; the most significant are:

- `IdealRadioMedium`: provides a simple channel-propagation model, including a path-loss free-space model and a constant-speed propagation model; in this regard, the transmitter and the receiver nodes are equipped with isotropic antennas.

**Listing 3.11** Contents of the *IdealRadioMedium.ned* file

```
1 module IdealRadioMedium extends RadioMedium {
2 parameters:
3 propagationType = default("ConstantSpeedPropagation");
4 pathLossType = default ("FreeSpacePathLoss");
5 analogModelType = default("IdealAnalogModel");
```

- `ScalarRadioMedium`: this radio-medium model uses scalar transmission power in the analog representation; it is possible to choose between the different propagation models offered by INET and to equip the nodes with one of the different antenna models provided by INET.

**Listing 3.12** Contents of the *Ieee80211ScalarRadioMedium.ned* file

```
1 module Ieee80211ScalarRadioMedium extends Ieee80211RadioMedium {
2     parameters:
3         analogModelType = default("IdealAnalogModel");
4         backgroundNoiseType = default("IdealAnalogModel");
```

Listing 3.12 shows the features of the `Ieee80211ScalarRadioMedium`, one of the most relevant `ScalarRadioMedium` modules. It should be noted that the analog model provided by this class is also scalar, whereas the type of noise is almost "ideal"; this module is usually used in cooperation with a `ScalarTransmitter` and a `ScalarReceiver` that perform their operations depending on the value of power that is set in the configuration

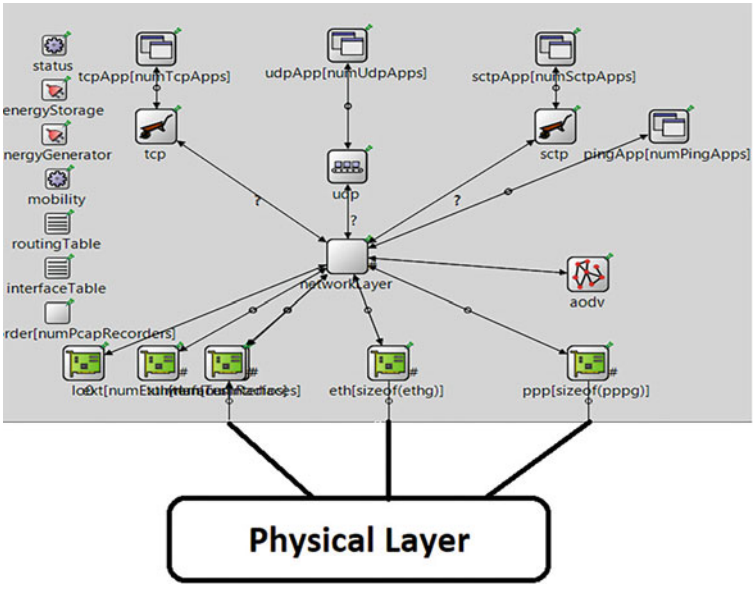



**Fig. 3.6** The `StandardHost` module

file. The `IdealTransmitter/Receiver` instead operates according to the `maxCommunicationRange` parameter, which determines the coverage area of a certain mobile node.

The physical layer features related to nodes are defined in the `StandardHost` module as functions of the type of the `RadioMedium` model set in the simulation. Figure 3.6 shows the internal structure of an exemplary `StandardHost`. It can be noted that the overall structure is designed according to the Transmission Control Protocol (TCP)/IP stack principle. The layered architecture in Fig. 3.2 is related to a mobile node, in fact, the third level (from bottom to top) that is the network layer executes the AODV protocol, a routing protocol for MANETs; however, the analysis of the routing-protocol modules is beyond the scope of this section. The physical layer communicates with the data link layer, which in this case consists of a set of Network Interface Card (NIC) interfaces either in wired or in wireless mode. It is important to highlight that all the physical-layer functionalities and modules are not mapped in the default `StandardHost` architecture, consequently the communication process with the data link layer is indirect.

### 3.3.1 Antenna Models

The package `inet.physicalLayer.antenna` contains all the antenna modules of INET. The default available modules are the following:

- `ConstantGainAntenna`: a simple antenna having a unique basic parameter, the gain. As suggested by the name, the gain set in the configuration file remains constant while a simulation is executed. This module is useful when information such as the orientation and the direction of the signal do not need to be considered.
- `CosineAntenna`: is the cosine pattern antenna designed in [6]. This model is very usefully for Wideband Code Division Multiple Access (W-CDMA) systems where the handover issue could become a critical task. Basically, this antenna is a combination of a directional high capacity antenna and a sectorized antenna. This combination allows to achieve some intra-cell benefits such as the isolation of interferences and user signals.
- `DipoleAntenna`: the well-known dipole antenna consisting of two identical conductive elements, which are bilaterally symmetrical; it is possible to set the length of the dipole (in meters) in the configuration file.
- `InterpolatingAntenna`: this antenna model computes the gain in a function of the direction of the signal, using linear interpolation. More specifically the antenna gain is computed based on the direction of the signal using linear interpolation extracting direction information expressed in Euler angles. Results from this antenna module are useful when a fast-scanning and approximated antenna pattern is required if particular constraints of time-consuming applications need to be satisfied.
- `IsotropicAntenna`: is the classical omnidirectional/isotropic antenna. It provides a unity gain by radiating the signal at the same way towards all directions.
- `ParabolicAntenna`: this model is based on a parabolic approximation of the main lobe radiation pattern. The gain is the function of the `maxGain` and the `minGain` together with the 3 dB beam-width. This module is very useful when employed in environments where path loss issues have to be addressed. In particular, through the functionalities of this antenna it is possible to avoid problems such as inter-cell interference that usually occurs when an antenna with poor orientation is used. The resulted main beam in this case is very directive.

Observe that, originally, the only antenna modules provided for INETMANET were the `IsotropicAntenna` and the `DipoleAntenna`; other modules have been developed later and then added to INETMANET. In order to understand the logical structure of a generic antenna module, the following two listings illustrate the source code of the most simple antenna module offered by INETMANET, the `IsotropicAntenna` module.

**Listing 3.13** *IsotropicAntenna.h* main functions

```
1 class INET_API IsotropicAntenna : public AntennaBase
2 {
3    public:
4       IsotropicAntenna();
5       virtual std::ostream& printToStream(std::ostream& stream, int level)
              const override;
6       virtual double getMaxGain() const override { return 1; }
7       virtual double computeGain(const EulerAngles direction) const override {
              return 1; }
8 };
```

**Listing 3.14** *AntennaBase.h* main functions

```
1 class INET_API AntennaBase : public IAntenna, public cModule
2 {
3    protected:
4       IMobility *mobility;
5       int numAntennas;
6       virtual void initialize(int stage) override;
7
8    public:
9       AntennaBase();
10      virtual std::ostream& printToStream(std::ostream& stream, int level)
              const override;
11      virtual IMobility *getMobility() const override { return mobility; }
12      virtual int getNumAntennas() const override { return numAntennas; }
13 };
```

Listing 3.13 highlights the main functions of the *IsotropicAntenna.h* file, in particular `computeGain()` and `getMaxGain()`. The first one implements the gain computation expression and returns the gain value. The gain expression depends on the kind of the used antenna. With regards to the isotropic antenna, for example, the gain is unitary and consequently the `getMaxGain()` function returns 1. It is important to note that the `IsotropicAntenna` class inherits all functions and variables of the `AntennaBase` class whose features are exposed in Listing 3.14. Observe that in the `AntennaBase` class, it is possible to set the number of antennas used by the node to create an antenna array. This parameter is an integer and is denoted by `numAntennas`. The `mobility` interface points to the mobility pattern *.ned* file associated with the node; the utility function `getNumAntennas()` simply returns the number of antennas related to a certain node.

Because the standard INET Framework does not support directional antennas and asymmetrical communications between nodes, we extended the INETMANET default features by adding a new antenna module which operates according to the Smart Antenna System (SAS) technology. Hence, future versions of INETMANET will provide a `PhasedArray` antenna module [9]. This module allows to emulate the most simple SAS strategy that is the switched-beam technology [4]. Note that the array elements are placed in the space according to the Uniform Linear Array (ULA) pattern. Listing 3.15 displays the main features of the `PhasedArray` module and the main parameters of a phased array antenna technology.

**Listing 3.15** *PhasedArray.h* main functions and parameters

```
class INET_API PhasedArray : public AntennaBase, IEnergyConsumer, protected
      cListener
{
   public:
      static simsignal_t phaseArrayConfigureChange;

   protected:
      m length;
      double freq;
      double distance;
      mutable double phiz;

      virtual double getMaxGain() const override {
         int numel = getNumAntennas();
         double maxGain = numel * log10(numel);
         return maxGain;
      }
      virtual double computeGain(EulerAngles direction) const override;
      virtual double getPhizero() {return phiz; }
}
```

The most important parameters include the `distance` which represents the fraction of the horizontal spacing between elements in the array expressed in wavelength and the `phiz` that denotes the steering angle. The gain computation is implemented in the `PhasedArray.cc` file, given in Listing 3.16.

**Listing 3.16** `computeGain` function from the `PhasedArray` module

```
double PhasedArray::computeGain(EulerAngles direction) const
{
   IRadio *radio= check_and_cast<IRadio *>(getParentModule());
   IRadioMedium *ra = const_cast< IRadioMedium *> (radio->getMedium());
   RadioMedium *rm=dynamic_cast< RadioMedium *>(ra);

   if (phiz == -1) {
      return 1; // omni-directional
   }

   double c = 300000 * 10 * 10 * 10; // speed of light
   double lambda = c / freq;
   double d = distance * lambda;
   double k = (2 * M_PI) / lambda; // wave number
   double phizero = phiz * (M_PI / 180);
   double psi = k * d * (cos(direction.beta) - cos(phizero));
   double num = sin((numel * psi) / 2);
   double den = sin(psi / 2);
   double rap = num / den;
   double mod = fabs(rap);
   double ef = cos(direction.beta);
   double efa = fabs(ef);
   double gain = 10 * log10(mod * efa);
   return gain;
}
```

The gain is implemented according to the following three equations:

$$G(\theta, \phi)_{TOT} = G(\theta, \phi)_{EF} * G(\theta, \phi)_{AF} \tag{3.1}$$

$$AF = \left| \frac{\sin\left(\frac{N\psi}{2}\right)}{N \sin\left(\frac{\psi}{2}\right)} \right| \tag{3.2}$$

$$\psi = kd\,(\cos\phi - \cos\phi_0) \tag{3.3}$$

In Eq. (3.1), the total gain $G(\theta, \phi)_{TOT}$ is expressed as a function of the element factor gain $G(\theta, \phi)_{EF}$ and the array factor gain $G(\theta, \phi)_{AF}$, where $AF$ is the array factor while $\psi$ is a term that depends on the steering angle. Please note that Eq. (3.2) is normalized with respect to the number of antenna elements $N$.

## 3.4 Mobility Models

INETMANET includes, in addition to the mobility models provided by the INET Framework, two additional models: `TRACI` and `LaptopModelManager`.

### *3.4.1 The Traffic Control Interface Model*

The Traffic Control Interface (TraCI) model is an adaptation of the TraCI implementation of the Veins framework (see Chap. 6). It allows to connect the simulation with the Simulation of Urban MObility (SUMO) tool.[11] This tool simulates realistic VANET movements. The main parameters used by the TraCI module are:

- `updateInterval`: time interval between updates of the SUMO simulation,
- `moduleType`: which OMNeT++ module (in this case the module, defined in the Network Topology Description (NED) language, that is going to be used in the simulation) to instantiate for each driving vehicle,
- `port`: which TCP port the client must use to connect to SUMO,
- `launchConfig`: configuration file sent to SUMO,
- `roiRects`: to limit the simulation to vehicles currently driving within a Region of Interest (ROI),
- `host`: address of the node where SUMO is installed.

[11] SUMO website: http://www.sumo.dlr.de/userdoc/Tools/Main.html.

If you want to install and execute SUMO, you can use the standard Veins tutorial.[12] There is an example in the directory *examples/traci* with the necessary files to launch and configure SUMO.

This model has two limitations. The first one is that the nodes are created but they are not shown in the OMNeT++ Graphical User Interface (GUI). The second one is that nodes are dynamically created during simulation time. Modules that configure themselves at the start of the simulation (e.g., `IPv4NetworkConfigurator`) do therefore not work. It is possible to use the module `HostAutoConfigurator2` inside the node model. This module can configure the node during runtime, as soon as a new node is created.

### 3.4.2 The `LaptopModelManager` Module

This module allows one to create and delete nodes at runtime. This facilitates model implementations similar to the presence of a set of laptop computers in an area. The computers dynamically appear and disappear, changing the number of nodes over time. The model configuration parameters are given in Listing 3.17.

**Listing 3.17** The `LaptopModelManager` configuration options

```
int NumNodes = default(0); // Maximum number of nodes that can be active at the
      same time in the simulation
volatile double startLife @unit(s); // Distribution of how new nodes will be
     added at the simulation
volatile double endLife @unit(s); // Lifetime
string nodeType; // NED description of the node that will be created
string nodeName; // Base name of the nodes that will be created
```

It is possible to have multiple `LaptopModelManager` simultaneously with different types of nodes, the only restriction is that the `nodeName` parameter must be different. An example in the directory *examples/manetrouting/dynamicNodeCreation* shows how the model works. In Listing 3.18, the configuration for two `LaptopModelManager` modules is shown. In this case, the first `LaptopModelManager` creates 10 nodes in the time interval between 30 s and 140 s with a uniform distribution and a node lifetime from 500 s to 600 s. The names of the nodes created by this `LaptopModelManager` are `node1[*]`. The second `LaptopModelManager` creates 20 nodes in the time interval between 60 s and 140 s with a uniform distribution and a node lifetime of 300 s to 1000 s. In this case, the nodes have the name `node2[*]`.

**Listing 3.18** Exemplary configuration options with two `LaptopModelManager` modules

```
**.dinamicWirelessNodeManager1.NumNodes = 10
**.dinamicWirelessNodeManager1.startLife = uniform(30s,140s)
**.dinamicWirelessNodeManager1.endLife = uniform(500s,600s)
```

[12] Veins tutorial website: http://veins.car2x.org/tutorial/.

```
**.dinamicWirelessNodeManager1.nodeType = "inet.node.inet.AdhocHost"
**.dinamicWirelessNodeManager1.nodeName = "node1"

**.dinamicWirelessNodeManager2.NumNodes = 20
**.dinamicWirelessNodeManager2.startLife = uniform(60s,200s)
**.dinamicWirelessNodeManager2.endLife = uniform(300s,1000s)
**.dinamicWirelessNodeManager2.nodeType = "inet.node.inet.AdhocHost"
**.dinamicWirelessNodeManager2.nodeName = "node2"
```

Listing 3.19 shows the NED file of a network with two `LaptopModelManager`. It is possible to observe that the configuration `IPv4NetworkConfigurator` has been removed as the nodes use either the `HostAutoConfigurator` or the `HostAutoConfigurator2`. Using `HostAutoConfigurator` is required to dynamically create modules during simulation. `IPv4NetworkConfigurator` can only configure the nodes at the start of the simulation.

**Listing 3.19** A NED file with two `LaptopModelManager` modules

```
import inet.node.inet.AdhocHost;
import inet.mobility.single.LaptopModelManager;
import inet.physicallayer.ieee80211.packetlevel.Ieee80211ScalarRadioMedium;
import inet.networklayer.configurator.ipv4.IPv4NetworkConfigurator;
import inet.physicallayer.ieee80211.packetlevel.Ieee80211ScalarRadioMedium;

network TestNetwork
{
  submodules:
    radioMedium: Ieee80211ScalarRadioMedium {
       @display("p=60,50;i=misc/sun");
    }

    laptopModelManager1: LaptopModelManager {
       @display("p=68,28;is=s");
    }

    laptopModelManager2: LaptopModelManager {
      @display("p=78,28;is=s");
    }
}
```

## 3.5 Application Models

INETMANET includes two additional useful models for UDP use cases:

- `UDPBasicBurstNotification` and
- `UDPVideoStreamSvr2` /`UDPVideoStreamCli2`.

The first one solves the problem of nodes getting dynamically created at runtime, where the possible destinations can change dynamically. The second model supports the *Video-Trace Library* of the University of Arizona [1].

### 3.5.1 UDPBasicBurstNotification

When nodes are created or erased dynamically in the simulation scenario, usually the position or the status of a certain receiver node related to a communication process also changes dynamically. For example, it is possible that a source node attempts to send a packet towards a receiver node but that packet could not reach the destination because the receiver changed its position in the network or, alternatively, the receiver no longer exists within the network. A source node might also be unable to successfully send packets to nodes that are created after the traffic application at the source is initialized.

To solve this problem, INETMANET has a special implementation of the source `UDPBasicBurst`. This source emits a signal every time that a source of this type is created or destroyed, forcing the rest of sources of the type `UDPBasicBurstNotification` to re-evaluate the possible destinations. The class `AddressModule` is the base module that facilitates this behavior. It defines two signals (cf. Listing 3.20) that are registered in the system module. This way, it does no matter what `AddressModule` emits the signal, the module will receive it.

Listing 3.20 shows that the signal `changeAddressSignalInit` is emitted when a new module is created. Likewise, when the module is destroyed because the module has been deleted, the signal `changeAddressSignalDelete` is emitted. Thus, it is possible to rebuild the list of possible destinations every time a node is created or deleted.

**Listing 3.20** `AddressModule` signal definition

```
simsignal_t AddressModule::changeAddrSignalInit;
simsignal_t AddressModule::changeAddrSignalDelete;

void AddressModule::initModule(bool mode) {
  ...
  getSimulation()->getSystemModule()->subscribe(changeAddrSignalInit,this);
  getSimulation()->getSystemModule()->subscribe(changeAddrSignalDelete,this);
  if (simTime() > 0)
    owner->emit(changeAddressSignalInit, this);
  ...
}

AddressModule::~AddressModule() {
  ...
  cSimpleModule * owner = check_and_cast<cSimpleModule*> (getOwner());
  owner->emit(changeAddressSignalDelete,this);
  getSimulation()->getSystemModule()->unsubscribe(changeAddrSignalDelete,this);
  getSimulation()->getSystemModule()->unsubscribe(changeAddrSignalInit,this);
  ...
}
```

A function that `UDPBasicBurstNotification` and `UDPBasicBurst` implement, which is not available in INET, is the possibility of generating broadcast packets. Listing 3.21 depicts an example of how to configure the source to emit broadcast packets. The interface name is mandatory. It is possible to set this parameter to `ALL`. In this case, the source will emit broadcast packets in all interfaces.

**Listing 3.21** Configuration parameters for broadcast packet transmission

```
**.destAddresses = "Broadcast" # destination address broadcast
**.outputInterfaceMulticastBroadcast = "wlan0" # interface in which
                # the broadcast packets will be sent, it can be "ALL"
**.setBroadcast = true # configure the socket for receive broadcast packets
```

### 3.5.2 The `UDPVideoStreamSvr2` and `UDPVideoStreamCli2` Models

The source models `UDPVideoStreamSvr2` and `UDPVideoStreamCli2` allow one to use realistic video traffic. They use the *Video-Trace Library* of the Arizona State University [1]. This library offers trace files with the codification in Moving Picture Experts Group Layer-4 Video (MPEG-4) of different videos. A description of the format and sample use of H.264 traces are available in [18].

The trace file has seven fields per row (cf. Listing 3.22): the first column is the sequence number, the second is the time of the frame creation, the third is the type of frame, the fourth is the size (expressed in bits) of the frame, and the last three fields contain information about the signal/noise of the codified frame. From the point of view of the source, only the first four fields have useful information.

**Listing 3.22** The trace file format

```
0  0.0000 I 8776  60.0000  60.0000  60.0000
2  0.0667 P 456   60.0000  60.0000  60.0000
1  0.0333 B 64    60.0000  60.0000  60.0000
4  0.1333 P 488   84.3880  60.0000  60.0000
3  0.1000 B 80    79.9947  60.0000  60.0000
6  0.2000 P 608   79.8017  60.0000  60.0000
```

The `UDPVideoStreamSvr2` generates the traffic using the information from the file, the time at which the frame must be sent, and its size. The client can compute the total number of frames for each type that it has received. Using this information it is possible to compute the distortion introduced by the network.

In Listing 3.23, it is possible to see the configuration parameters of the module `UDPVideoStreamSvr2`. This source includes some additional parameters such as `traceFileName`, which contains the route and name of the trace file; `macroPackets`, if true, the source, instead of sending MPEG-4 frames encapsulated in UDP packets, can store several frames in a single UDP packet; `maxSizeMacro` is the size (expressed in bits) of the macro packets; in this case, the source creates aggregated frames up to this size. It is also possible to use this source to broadcast video frames if the parameter `videoBroadcast` is true.

**Listing 3.23** Configuration parameters of the `UDPVideoStreamSvr2` module

```
string interfaceTableModule; // The path to the InterfaceTable module
int localPort; // port to listen on
```

```
3  volatile double sendInterval @unit(s); // interval between sending video stream
       packets
4  volatile int packetLen @unit(B); // length of a video packet in bytes
5  volatile int videoSize @unit(B); // length of full a video stream in bytes
6  volatile double stopTime @unit(s) = default(0);
7  string traceFileName = default(""); // University of Arizona video trace format
8  bool macroPackets = default(false); // it allows that several video frames can
       be grouped in a packet
9  int maxSizeMacro @unit(B) = default(512B); // maximum size of a grouped packet
10 bool videoBroadcast = default(false); // the server will send broadcast video
       frames
11 double startBroadcast @unit(s) = default(0s); // start time of the video
       broadcast
12 string broadcastInterface = default("wlan0"); // interface used to broadcast
       the video frames
13 volatile double restartBroadcast @unit(s) = -1s; // after finish the broadcast
       sequence the server will restart the broadcast after an interval
```

The configuration of the client from Listing 3.24 also includes some differences with respect to the basic `UDPVideoStreamCli` module. The most interesting parameters are `reintent` and `multipleRequest`. The first allows that the client starts a new connection if the first one has failed; the second allows a client to start another connection as soon as the current one finishes.

**Listing 3.24** Configuration parameters of the `UDPVideoStreamCli2` module

```
1  int localPort = default(-1); // local port (-1: use ephemeral port)
2  string serverAddress; // server address
3  int serverPort; // server port
4  double startTime @unit(s) = default(1s); // start time of the client
5  volatile double reintent @unit(s) = default(60s); // the client will request a
       new sequence if no video frame has arrived after this period, if a video
       frame has been received, the client can request other sequences only if
       the parameter multipleRequest is true
6  bool multipleRequest = default(false); // if true the client request other
       sequence after finish the current
7  double timeOut @unit(s) = default(10s); // maximum time without receive a frame
        that the client use to determine a fail of request or end of sequence.
       If it is an end of sequence and multipleRequest = false the client does
       not request another sequence, if the client has never received a video
       frame, it will request another sequence
8  double limitDelay @unit(s) = default(0.5s); // maximum delay in a video frame
       to discard it
```

## 3.6 Link Layer Models

INETMANET includes several link layer models that are not present in the INET Framework. The most interesting ones are a basic implementation of link layer routing and forwarding similar to the ones available in the 802.11 standard, and the implementation of the Very High Throughput (VHT) extension, included in the latest revision of the standard [8].

### 3.6.1 Routing and Forwarding in the Link Layer

The current implementation only covers the forwarding and routing aspects, including the possibility to use several routing protocols. The routing protocols that the implementation can use are DYMO [14], Hybrid Wireless Mesh Protocol (HWMP) [8], AODV [13], and Optimized Link State Routing (OLSR) [7]. The module that implements the forwarding mechanism is `Ieee80211Mesh`. It has additional feature that is not included in the basic `Ieee80211MgmtAdhoc` module incorporated in the INET Framework. These extended functionalities are:

1. start the routing protocols in the link layer,
2. forward the IEEE 802.11 four address frames using the information obtained by the routing protocols,
3. flood the broadcast IEEE 802.11 frames in the network,
4. encapsulate IEEE 802.11 frames over Ethernet [3].

The `Ieee80211Mesh` configuration is similar to `Ieee80211MgmtAdhoc` but it includes additional parameters. These parameters are shown in the Listing 3.25.

**Listing 3.25** Configuration parameters of the `Ieee80211Mesh` module

```
string meshReactiveRoutingProtocol = default("inet.routing.extras.DYMOUM");
bool useHwmp = default(false); // If active automatically deactivate
    useProactive and useReactive and activate ETXEstimate
bool useProactive = default(false);
bool useReactive = default(true);

bool useGreenie = default(false);
bool greenieCompleteNode = default(true);

bool FixNode = default(false); // used by routing protocol to indicate no
    mobility node
bool UseLwMpls = default(false);
double maxDelay = default(0.1);
int maxTTL = default(32); // the same that IP, sets the maximum number of hops
    that a frame can propagate in the network.
bool ETXEstimate = default(false);
bool IsGateWay = default(false);
double GateWayAnnounceInterval @unit("s") = default(100s);
```

The four parameters `useGreenie`, `useProactive`, `useReactive`, and `useHwmp` are used to select the routing protocol. The protocol used in each case is:

- `useHwmp`: HWMP.
- `useReactive`: OLSR.
- `useProactive`: the protocol that is being selected in the configuration field `meshReactiveRoutingProtocol`, by default `DYMO-UM`.
- `useGreenie`: Greenie, a hybrid protocol designed for wireless mesh networks, which operates on the link layer [2].

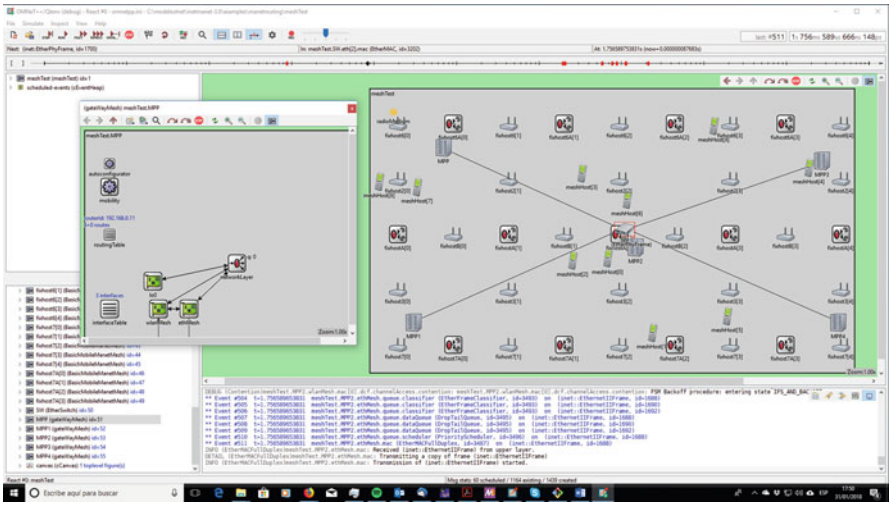


**Fig. 3.7** Wireless network with MPP nodes, and detail of an MPP node with a direct connection between the wireless MAC and the Ethernet MAC

The parameter `FixNode` is used by the Greenie routing protocol. This permits the protocol to handle static nodes and mobile nodes differently, allowing Greenie to search more stable routes, with lower probability of breaking.

The parameter `IsGateWay` permits the module to use a functionality that is not in the standards. With it, it is possible to extend the wireless network using wired Ethernet segments. In this case, the gateway nodes can encapsulate the 802.11 frames over an Ethernet frame and send the frame to other Mobile Peer-to-Peer Protocol (MPP) gateways that re-inject the frame in the network. In Fig. 3.7, a network with MPP nodes is shown. It is possible to observe the connection between the 802.11 Medium Access Control (MAC) and the Ethernet MAC.

In order to integrate 802.11 nodes in infrastructure mode into this type of network, a special module called `LocatorModule` has been developed and a new type of node has been included in INETMANET.

The new type of node is called `ApRelayNode` (cf. Fig. 3.8). It connects an interface that works in mesh mode with an interface that works in access point mode. This node allows the frames to arrive at the client nodes connected to the access point and to the frames sent by the clients to arrive at the mesh network. By using this, frames can arrive at the destination.

### *3.6.2 VHT and the IEEE 802.11ac Standard*

INETMANET has implemented the standard IEEE 802.11ac that allows to use bit rates up to 6933.3 Mbit/s. This implementation has already been ported to the new

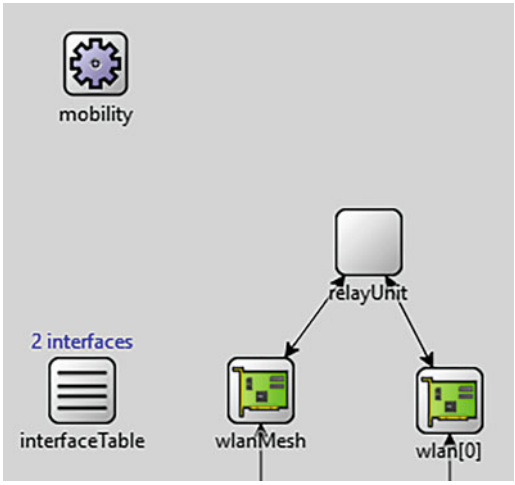



**Fig. 3.8** `ApRelayNode` implementation in OMNeT++ with a relay unit that connects an IEEE 802.11 interface that operates in mesh mode and an 802.11 interface that operates in access point mode

INET Framework version 4.x. It adds the new classes `Ieee80211VHTMode` and `Ieee80211VHTCode` and modifies the `Ieee80211NistErrorModel` and the `Ieee80211YansErrorModel` class.

The class `Ieee80211VHTMode` implements the different modulation modes and computes the bit rates for the different modes. It also computes the different time parameters used in the 802.11ac implementation, like the physical header duration or the *DIFS* or *SIFS* parameters used in the MAC sublayer.

The `Ieee80211NistErrorModel` and `Ieee80211YansErrorModel` classes have been modified to include the errors models for the different modulations used in the VHT standard. The class `Ieee80211Band` has also been modified to include the new bands used in the VHT standard that allow to compute the interferences between the different channels.

Listing 3.26 displays the selection of the 802.11ac mode and the selection of the bandwidth to compute the central frequency of the channel and the interference among channels. The available bit rates are calculated with a 400 ns guard interval, except for the mandatory bit rates that are computed using the 800 ns guard interval. If multiple entries with the same bit rate exist, the model selects the entry with lower bandwidth and/or lower modulation.

**Listing 3.26** Configuration parameters to select IEEE 802.11ac

```
1 **.bandName = "5 GHz&20 MHz" #,"5 GHz&40 MHz","5 GHz&80 MHz","5 GHz&160 MHz"
2 **.opMode = "ac"
3 **.bitrate = 693.3Mbps
4 **.wlan[*].radio.antenna.numAntennas = 8 #max. nr. of 802.11ac streams is 8
```

## 3.7 Miscellaneous Tools

INETMANET includes some modules that help in the simulation and analysis of wireless networks. These modules are `WirelessNumHops`, `WirelessGetNeig`, `DijkstraKshortest`, and `GlobalWirelessLinkInspector`.

### 3.7.1 The `WirelessNumHops` Class

This class can be used to find a path between any pair of nodes (if it exists), assuming that the maximum distance between two nodes that can communicate is a known parameter. Listing 3.27 shows an example that assumes that the maximum separation between two neighbor nodes is 120 m.

**Listing 3.27** Example use of the `WirelessNumHops` class

```
#include "inet/common/WirelessNumHops.h"

IPv4Address myAddress;
IPv4Address destAddress;
...
WirelessNumHops *routing = new WirelessNumHops();
routing->setRoot(myAddress);
std::deque<IPv4Address> pathNode;
bool found = routing->findRoute(120.0, destAddress,pathNode);
if (found) { ... }
```

This class can be used to find out if a possible route exists, even if the routing protocol cannot find it.

### 3.7.2 The `WirelessGetNeig` Class

This class is similar to `WirelessNumHops`, but in this case, it only provides the list of nodes having a separation lower than a predetermined distance. Listing 3.28 depicts a usage example.

**Listing 3.28** Example use of the `WirelessGetNeig` class

```
#include "inet/common/WirelessGetNeig.h"

IPv4Address node;
...
WirelessGetNeig *neig = new WirelessGetNeig();
std::vector<IPv4Address> neigList;
neig->getNeighbours(nodee, neigList, 120.0);
...
```

### 3.7.3 The `DijkstraKshortest` Module

The module `DijkstraKshortest` provides a solution for the *K Shortest* problem [19]. This solution follows the algorithm presented by Chong et al. [5], and allows finding the K shortest paths between a source and a destination, if they exist. In the case that the number of possible routes is lower than the parameter K, this implementation finds all possible routes.

This solution can be used to resolve the problem of multiple constrained shortest path [20]. It is possible to set multiple cost parameters to each link and to find the minimum path cost, with respect to a parameter. At the same time, the cost of the rest of the parameters in the path must not exceed a predefined limit.

In Listing 3.29, we show an example that extracts the topology using the class `cTopology`. In this case, the example tries to find a maximum of 5 routes per destination. The routes are ordered from lower to higher costs, with `route[0]` being the route of the lowest cost.

**Listing 3.29** Example of use `DijkstraKshortest` module

```
cTopology topo("topo");
topo.extractByProperty("networkNode");
cModule *mod = getContainingNode(this);
DijkstraKshortest djk;

djk.setFromTopo(&topo, L3Address::IPv6);
L3Address id = L3AddressResolver().addressOf(mod, L3AddressResolver::ADDR_IPv6)
    ;
djk.setRoot(id);
djk.setKLimit(5); // 5 paths
djk.run();

for (int i = 0; i < topo.getNumNodes(); i++) {
  cTopology::Node *destNode = topo.getNode(i);
  if (mod == destNode->getModule())
    continue;
  L3Address nodeId = L3AddressResolver().addressOf(destNode->getModule(),
      L3AddressResolver::ADDR_IPv6);
  for (int i = 0; i < djk.getNumRoutes(nodeId); i++) {
    std::vector<L3Address> pathNode;
    EV<<"Total paths "<<djk.getNumRoutes(nodeId)<<" to node "<<nodeId<<"\n";
    if (djk.getRoute(nodeId, pathNode,i)) {
     EV << "Path :"<< i << "Nodes ;
     for (auto elem : pathNode) {
       EV << elem;
       if (elem != pathNode.back())
         EV << "-";
       else
         EV << "\n";
      }
    }
  }
}
```

Listing 3.30 gives an example of how to define the network from Fig. 3.9 in an object of type `inetmanet-DijkstraKshortest`. For simplicity, all the links in this example have the same cost (cost = 1, delay = 0.2 units, available bandwidth = 100 units, and quality = 10 units).

**Listing 3.30** Defining the connections using the class `DijkstraKshortest` with constrained cost, applied to the network from Fig. 3.9

```
32 DijkstraKshortest djk;
33                                        // cost, delay, bandwidth, quality
34 djk.addEdge(L3Address(IPv4Address(1)),L3Address(IPv4Address(2)),1,0.1,100,10);
35 djk.addEdge(L3Address(IPv4Address(2)),L3Address(IPv4Address(1)),1,0.1,100,10);
36 djk.addEdge(L3Address(IPv4Address(1)),L3Address(IPv4Address(3)),1,0.1,100,10);
37 djk.addEdge(L3Address(IPv4Address(3)),L3Address(IPv4Address(1)),1,0.1,100,10);
38 djk.addEdge(L3Address(IPv4Address(2)),L3Address(IPv4Address(3)),1,0.1,100,10);
39 djk.addEdge(L3Address(IPv4Address(3)),L3Address(IPv4Address(2)),1,0.1,100,10);
40 djk.addEdge(L3Address(IPv4Address(2)),L3Address(IPv4Address(4)),1,0.1,100,10);
41 djk.addEdge(L3Address(IPv4Address(4)),L3Address(IPv4Address(2)),1,0.1,100,10);
42 djk.addEdge(L3Address(IPv4Address(3)),L3Address(IPv4Address(5)),1,0.1,100,10);
43 djk.addEdge(L3Address(IPv4Address(5)),L3Address(IPv4Address(3)),1,0.1,100,10);
44 djk.addEdge(L3Address(IPv4Address(2)),L3Address(IPv4Address(5)),1,0.1,100,10);
45 djk.addEdge(L3Address(IPv4Address(5)),L3Address(IPv4Address(2)),1,0.1,100,10);
46 djk.addEdge(L3Address(IPv4Address(3)),L3Address(IPv4Address(4)),1,0.1,100,10);
47 djk.addEdge(L3Address(IPv4Address(4)),L3Address(IPv4Address(3)),1,0.1,100,10);
48 djk.addEdge(L3Address(IPv4Address(4)),L3Address(IPv4Address(6)),1,0.1,100,10);
49 djk.addEdge(L3Address(IPv4Address(6)),L3Address(IPv4Address(4)),1,0.1,100,10);
50 djk.addEdge(L3Address(IPv4Address(5)),L3Address(IPv4Address(6)),1,0.1,100,10);
51 djk.addEdge(L3Address(IPv4Address(6)),L3Address(IPv4Address(5)),1,0.1,100,10);
```

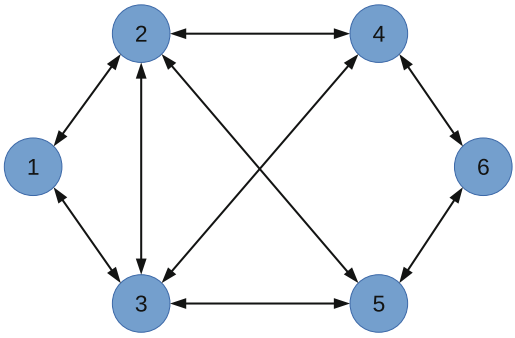



**Fig. 3.9** Topology of the network used in Listing 3.30

Listing 3.31 shows how to execute an exemplary search of a multiple-constrained path between nodes 1 and 6. The limits have been set to 10, 1, 10, and 1000. Please note that the bandwidth metric is concave, whereas the other metrics are additives.

**Listing 3.31** Running a multiple-constrained path search using `DijkstraKshortest`

```
 1 DijkstraKshortest djk;
 2 ...
 3 std::vector<double> limits;
 4 limits.resize(4);
 5 limits[0] = 10; // cost limit, additive
 6 limits[1] = 1; // delay limit, additive
 7 limits[2] = 10; // bandwith limit, concave
 8 limits[3] = 1000; // qos limit, additive
 9
10 djk.setLimits(limits);
11 djk.setRoot(L3Address(IPv4Address(1)));
12 djk.setKLimit(5); // 5 paths
13 djk.runUntil(L3Address(IPv4Address(6)));
14 if (djk.getRoute(L3Address(IPv4Address(6)), pathNode)) {
```

```
15     EV << "Path :"<< i << "Nodes ;
16     for (auto elem : pathNode) {
17        EV << elem;
18        if (elem != pathNode.back())
19           EV << "-";
20        else
21           EV << "\n";
22     }
23  }
```

### 3.7.4 *The `GlobalWirelessLinkInspector` Module*

The objective of the module `GlobalWirelessLinkInspector` is to aid in debugging routing protocols. This module can record the changes in the routing tables that are derived from `ManetRoutingBase` class (cf. Sect. 3.2.1). The module must be set in the landscape, it helps finding the source-destination path.

### 3.7.5 *NETA Framework Integration*

INETMANET has integrated the NETwork Attacks Framework for OMNeT++ (NETA) framework[13] as well. NETA allows to simulate three types of attacks in an IPv4 network, in particular *sink-hole attacks*, *delay attacks*, and *IP dropping attacks*. By default, this module is disabled in the basic configuration of INETMANET. It can be activated by selecting the box *Net attack* in the project features list.

## References

1. Arizona State University: Video Trace Library. http://trace.eas.asu.edu/
2. Barekatain, B., Maarof, M.A., Quintana, A.A., Cabrera, A.T.: Greenie: a novel hybrid routing protocol for efficient video streaming over wireless mesh networks. EURASIP J. Wirel. Commun. Netw. **2013**(1), 168 (2013). https://doi.org/10.1186/1687-1499-2013-168
3. Barekatain, B., Raahemifar, K., Ariza, A., Triviño, A.: Promoting wired links in wireless mesh networks: an efficient engineering solution. PLoS One **10**(3), e0119679 (2015). https://doi.org/10.1371/journal.pone.0119679
4. Bellofiore, S., Balanis, C.A., Foutz, J., Spanias, A.S.: Smart-antenna systems for mobile communication networks. Part 1. Overview and antenna design. IEEE Antennas Propag. Mag. **44**(3), 145–154 (2002)
5. Chong, E., Maddila, S., Morley, S.: On finding single-source single-destination K shortest paths. In: Proceedings of the International Conference on Computing and Information (ICC'95), pp. 40–47 (1995)

[13] NETA website: https://nesg.ugr.es/index.php/en/2015-06-10-08-14-34/neta-2.

6. Chunjian, L.: Efficient antenna patterns for three-sector WCDMA systems. M.Sc. thesis, Chalmers University of Technology, Göteborg (2003)
7. Clausen, T., Jacquet, P.: Optimized Link State Routing Protocol (OLSR). RFC 3626. IETF, Fremont (2003)
8. IEEE Standards Association: IEEE Standard for Information Technology – Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks–Specific Requirements—Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE Std 802.11-2016—Revision of IEEE Std 802.11-2012, Institute of Electrical and Electronics Engineers, Piscataway (2016). https://doi.org/10.1109/IEEESTD.2016.7786995
9. Inzillo, V., De Rango, F., Quintana, A., Santamaria, A.: A new switched beam smart antenna model for supporting asymmetrical communications extending inet OMNeT++ framework. In: International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS). IEEE, Piscataway (2017)
10. Johnson, D., Hu, Y., Maltz, D.: The Dynamic Source Routing Protocol (DSR) for Mobile Ad hoc Networks for IPv4. RFC 4728. IETF, Fremont (2007)
11. Montgomery, D.: Design and Analysis of Experiments, 8th edn. Wiley, London (2012). https://books.google.es/books?id=XQAcAAAAQBAJ
12. Neumann, A., Aichele, C., Lindner, M., Wunderlich, S.: Better Approach to Mobile Ad-hoc Networking (BATMAN). Draft. IETF, Fremont (2008)
13. Perkin, C., Belding-Royer, E., Das, S.: Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561. IETF, Fremont (2003)
14. Perkin, C., Ratliff, S., Dowdell, J.: Dynamic MANET On-demand (AODVv2) Routing. Draft. IETF, Fremont (2010)
15. Perkins, C., Bhagwat, P.: Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In: ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications, pp. 234–244. ACM, New York (1994)
16. Sbeiti, M., Wietfeld, C.: PASER: Position Aware Secure and Efficient Mesh Routing Protocol. Draft. IETF, Fremont (2012)
17. Sbeiti, M., Goddemeier, N., Behnke, D., Wietfeld, C.: PASER: secure and efficient routing approach for airborne mesh networks. IEEE Trans. Wirel. Commun. **15**(3), 1950–1964 (2016). https://doi.org/10.1109/TWC.2015.2497257
18. Seeling, P., Reisslein, M.: Video transport evaluation with H.264 video traces. IEEE Commun. Surv. Tutorials **14**, 1142–1165 (2012)
19. Yen, J.Y.: Finding the K shortest loopless paths in a network. Manag. Sci. **17**(11) (1971). http://www.jstor.org/stable/2629312
20. Ziegelmann, M.: Constrained shortest paths and related problems. Ph.D. thesis, Universität des Saarlandes (2001). https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/23809

# Chapter 4
# RINASim


**Vladimír Veselý, Marcel Marek, and Kamil Jeřábek**

## 4.1 Introduction

Recursive InterNetwork Architecture (RINA) is the clean-slate architecture aimed to change the whole Internet unlike just temporary fixes for current status quo. The RINA concept is based on John Day's thoughts, lectures, and book [2] regarding International Organization for Standardization (ISO)/Open Systems Interconnection (OSI) initiative failure, Transmission Control Protocol (TCP)/Internet Protocol (IP) development, commercial adoption of the Internet, and other technical/political events in Internet history.

The proposed RINA architecture is fundamentally different from the current TCP/IP networking. The RINA approach focuses on a few principles instead of a broad and complex ecosystem of the modern Internet. The idea of the recursive composition of layers arises naturally from the structure of repeating computer networking patterns. Instead of strictly separating network functions into a predefined set of layers, RINA enables to compose a stack of layers that may offer a nearly the same set of functions. All RINA layers employ the same protocols which contrast to the TCP/IP model, where each layer defines its set of protocols. RINA was designed to provide a simpler and efficient alternative to the current Internet architecture.

Section 4.2 provides brief information how to install RINASim and start working with it. Section 4.3 starts with a description of high-level RINA network nodes. Section 4.4 goes deeper and outlines various components and policies. The whole

V. Veselý (✉) · M. Marek · K. Jeřábek
Brno University of Technology, Brno, Czech Republic
e-mail: veselyv@fit.vutbr.cz; imarek@fit.vutbr.cz; ijerabek@fit.vutbr.cz



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_4

content of Sect. 4.5 is dedicated to a thorough description of simulations that illustrate basic scenarios of RINA network operation. This chapter is summarized in Sect. 4.6.

## 4.2 Installation

RINASim [22] is a stand-alone framework for the OMNeT++ Discrete Event-based Simulation (DES) environment. RINASim is coded from scratch and independent from other (model) libraries. The main purpose is to offer the community a reliable and the most up-to-date tool (in the sense of RINA specification compliance) for simulating RINA-based computer networks. RINASim, in its current state, represents a working implementation of the simulation environment for RINA. The simulator contains all mechanisms of RINA according to the current specifications.

The RINASim installation is a straightforward process with two key phases: (1) obtain/download the source code; (2) compile the RINASIM project, which creates one static library (`librinasimcore`, containing the simulation core) and one dynamic library (`librinasim`, binding together the core and the implementation of various policies).

RINASim is developed for OMNeT++ v5.2.1. RINASim August 2016's release is the last one that is compatible with OMNeT++ v4.6. The current trend is to make RINASim compatible with any OMNeT++ version that supports the C++ 11 language standard and the **GCC** 4.9.2 compiler. All source codes (including master and other thematic branches) are publicly available on the project's GitHub repository [23]. The manual installation and building RINASim from the source code is pretty simple. Just clone or download the main branch, import the project into OMNeT++ (it is named `rina`), compile it, and start simulating with RINASim. The user should be prepared for a rather long initial compilation time.

## 4.3 High-Level Design

The purpose of this section is to provide future RINASim users with a short introduction (or more accurately an executive summary) to RINA concepts. These concepts and ideas formulate the design and development of the whole RINASim framework.

### *4.3.1 Overview*

We now introduce the theoretical background. However, an explanation of the complete Recursive InterNetwork Architecture is beyond the scope of this chapter.

Hence, only parts relevant to the current RINASim functionality are captured. The synthesis of RINA information provided below is from the following sources [8, 9, 11, 12, 14].

#### 4.3.1.1 Nature of Applications and Application Protocols

The set of Internet applications was rather simplistic before the world wide web—one application with a single instance using only one protocol. Hence, there is nearly no distinction between an application and its networking part. However, the web completely changed this situation—one application protocol may be used by more than one application (e.g., Hypertext Transfer Protocol (HTTP) is being over-employed as a communication protocol), and also one application may have many application protocols (e.g., web browsers, mail clients).

The following terms are recognized in the context of RINA, their relationship is depicted in Fig. 4.1:

- *Application Process (AP)*: Program instantiation to accomplish some purpose,
- *Application Entity (AE)*: AE is the part of AP, which represents application protocol and application aspects concerned with communication.

There may be multiple instances of the AP in the same system. AP may have multiple AEs; each one may process a different application protocol. There also may be more than one instance of each AE type within a single AP.

All application protocols are stateless; the state is and should be maintained in the application. Thus, all *application protocols* modify shared state external to the protocol itself on various objects (e.g., data, file, hardware peripherals). Because of that, there is only one application protocol that contains trivial operations (e.g., read/write, start/stop). *Data transfer protocols* modify state internal to the protocol; the only external effect is the delivery of Service Data Units (SDUs).

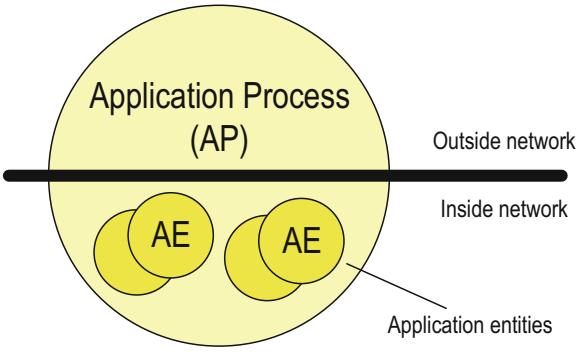



**Fig. 4.1** Application Process (AP) and Application Entity (AE) relationship

#### 4.3.1.2 Core Terms

The data transport and inter-networking tasks together (generally known as networking) constitute *Inter-Process Communication (IPC)*. IPC between two APs on the same operating system needs to locate processes, evaluate permission, pass data, schedule tasks, and manage memory. IPC between two APs on different systems works similarly plus adding functionality to overcome the lack of shared memory.

In traditional networking stacks, the layer provides a service to the layer immediately above it. The recursion (and repeating of patterns) is the main feature of the whole architecture. Layer recursion became more popular even in TCP/IP with technologies like Virtual Private Networks (VPNs) or overlay networks (e.g., Overlay Transport Virtualization (OTV), TOR). Recursion is a natural thing whenever we need to affect the scope of communicating parties. However, so far it was just recursion of repeating functions in existing layers.

In ISO/OSI or TCP/IP, there is a set of layers each with completely different functions. RINA, on the other hand, yields the idea of the single generic layer with fixed mechanisms but configurable policies. In RINA, this layer is called *Distributed Inter-Process Communication Facility (DIF)*—a set of cooperating APs providing IPC. There is not a fixed number of DIFs in RINA; we can stack them according to the application or network needs. From the DIF point of view the actual stack depth is irrelevant, DIF must know only $(N + 1)$-layer above and $(N - 1)$-layer below. DIF stacking partitions networks into smaller, thus, more manageable parts.

The concept of the RINA layer could be further generalized to a *Distributed Application Facility (DAF)*—a set of cooperating APs in one or more computing systems, which exchange information using IPC and maintain a shared state. A DIF is a DAF that does only IPC. *Distributed Application Process (DAP)* is a member of a DAF. The *Inter-Process Communication Process (IPCP)* is a special AP within a DIF delivering inter-process communication. The IPCP is an instantiation of a DIF membership; computing system can perform IPC with other DIF members via its IPC process within this DIF. An IPCP is a specialized DAP. The relationship between all newly defined terms is depicted in Fig. 4.2.

DIF limits and encloses cooperating processes in the one scope. However, its functionality is more general and versatile apart from rigid TCP/IP layers with dedicated functionality (i.e., a data link layer for adjacent node communication, a transport layer for reliable data transfer between applications). DIF provides IPC to either another DIF or to DAF. Therefore, DIF uses a single application protocol with generic primitive operations to support inter-DIF communication.

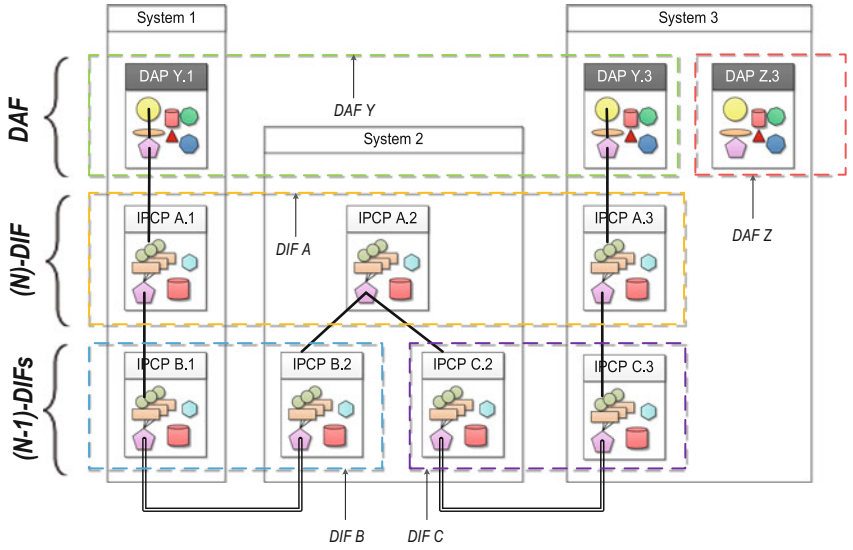



**Fig. 4.2** DIF, DAF, DAP, and IPCP illustration

#### 4.3.1.3 Connection-Oriented vs. Connectionless

The clash between connection-oriented and connectionless approaches (that also corrupted ISO/OSI tendencies) is from the RINA perspective quite easy to settle. Connection-oriented and connectionless communication are both just functions of the layer that should not be visible to applications. Both approaches are equal, and it depends on application requirements which one to use. On the one hand, connectionless is characterized by the maximal dissemination of the state information and dynamic resource allocation. On the other hand, connection-oriented limits the dissemination and tends toward static resource allocation. The first one is good for low volume stochastic traffic. The second one is especially useful for scenarios with deterministic traffic flows.

If the applications request the allocation of communication resources, then the layer determines what mechanisms and policies to use. Allocation is accompanied with access rights and description of Quality of Service (QoS) (e.g., what minimum bandwidth or delay is needed for correct operation of application). QoS demands are then translated into the appropriate QoS class called *QoS-cube*.

#### 4.3.1.4 Delta-t Synchronization

All properly designed data transfer protocols are soft-state. There is no need for explicit state synchronization (hard-state) or tools like *SYN* and *FIN* flags.

Initial synchronization of communicating parties is done with the help of the *Delta-t protocol* with the main variable denoted as $\Delta t$ (cf. [15, 27]). Delta-t was developed by Richard Watson, who proposed a time-based synchronization technique. He proved that conditions for distributed synchronization were met if the following three timers are realized: (a) Maximum Packet Lifetime (*MPL*); (b) maximum time to attempt retransmission a.k.a. maximum period during sender is holding a Protocol Data Unit (PDU) for retransmission while waiting for a positive acknowledgment (a.k.a. $R$); (c) maximum time before an Acknowledgment (ACK) (a.k.a. $A$).

Delta-t assumes that all connections exist constantly. Synchronization state is maintained only during the activity, which is defined as $2 \cdot \Delta t$ from the receiver side and $3 \cdot \Delta t$ from the sender side, where $\Delta t = R + MPL + A$. After 2–3 $\Delta t$ periods without any traffic the state may be discarded, effectively resetting the connection. Because of that, there are no hard-state protocols (with explicit synchronization), only soft-state ones. Delta-t postulates that port allocation and synchronization are distinct.

#### 4.3.1.5 Separation of Mechanism and Policy

We understand the term mechanism as the fixed part and policy as the flexible part of IPC. If we clearly separate them, we discover that there are two types of mechanisms:

- *tightly-bound* that must be associated with every PDU, which handles fundamental aspects of data transfers,
- *loosely-bound* that may be associated with some data transfer PDUs, which provide additional features (in particular reliability and flow control).

Both groups are coupled through a state vector maintained separately per flow; every active flow has its state-vector holding state information. For instance, the behavior of retransmission and flow control can be heavily influenced by chosen policies, and they can be used independently of each other.

This implies that only a single generic data transfer protocol based on Delta-t is needed, which may be governed by different transfer control policies. This data transfer protocol modifies the state internally, whereas the application protocol (carried inside) modifies the state externally.

#### 4.3.1.6 Naming and Addressing

The AP communicates in order to share states. We mentioned that an AP consists of AEs. We need to differentiate between different APs and also different AEs within the same AP. RINA is therefore using the following set of identifiers to achieve the desired naming properties:

- *Distributed Application Name (DAN)* is the name that identifies a distributed application. It is globally unambiguous. One DAF might have assigned more than one DAN with different access control properties.
- *Application Process Name (APN)* is a globally unambiguous synonym for an AP of the DAF.
- *Application Process Instance Identifier (API-id)* is an identifier bound to an AP instance to distinguish multiple AP instances. It is unambiguous within the AP.
- *Application Entity Name (AEN)* is unambiguous within the scope of the AP.
- *Application Entity Instance Identifier (AEI-id)* is an identifier that is also unambiguous within a single AP. It facilitates the identification of different Application Entity Instances (AEIs).
- *Application Naming Information (ANI)* references a complete set of identifiers to name particular application. It consists of a four-tuple: APN, API-id, AEN, and AEI-id. The only required part of ANI is APN, others are optional.

An IPC process has an APN to identify it among other DIF members. A RINA *address* is a synonym for the IPCP's APN with a scope limited to the layer and structured to facilitate forwarding. The APN is useful for management purposes but not for forwarding. Address structure may be topologically dependent (indicating the nearness of IPCPs). APN and address are simply two different means to locate an object in different context. There are two local identifiers important for IPCP functionality—port-id and connection-endpoint-id. *Port-id* binds this $(N)$-IPCP and $(N+1)$-IPCP/AP; both of them use the same port-id when passing messages. Port-id is returned as a handle to the communication allocator. It is unambiguous within a computing system. *Connection Endpoint Identifier (CEP-id)* identifies a shared state of one communication endpoint. Since there may be more than one flow between the same IPCP pair, it is necessary to distinguish them. For this purpose, connection-id is formed by combining source and destination CEP-ids with QoS requirements descriptor. CEP-id is unambiguous within IPCP and connection-id is unambiguous between a given pair of IPCPs. Figure 4.3 depicts all relevant identifiers between two IPCPs.

Watson's Delta-t implies port-id and CEP-id in order to help separate port allocation and synchronization. RINA's *connection* is a shared state between ends identified by CEP-ids. RINA's flow is when connection ends are bound to ports identified by port-ids. The lifetimes of flow and its connection(s) are independent of each other.

The relationship between node and Point of Attachment (PoA) is relative—node address is $(N)$-address, and its PoA is $(N-1)$-address. Routes are sequences of $(N)$-addresses, where $(N)$-layer routes based on this addresses (not according to $(N-1)$-addresses). Hence, the layer itself should assign addresses because it understands address structure.

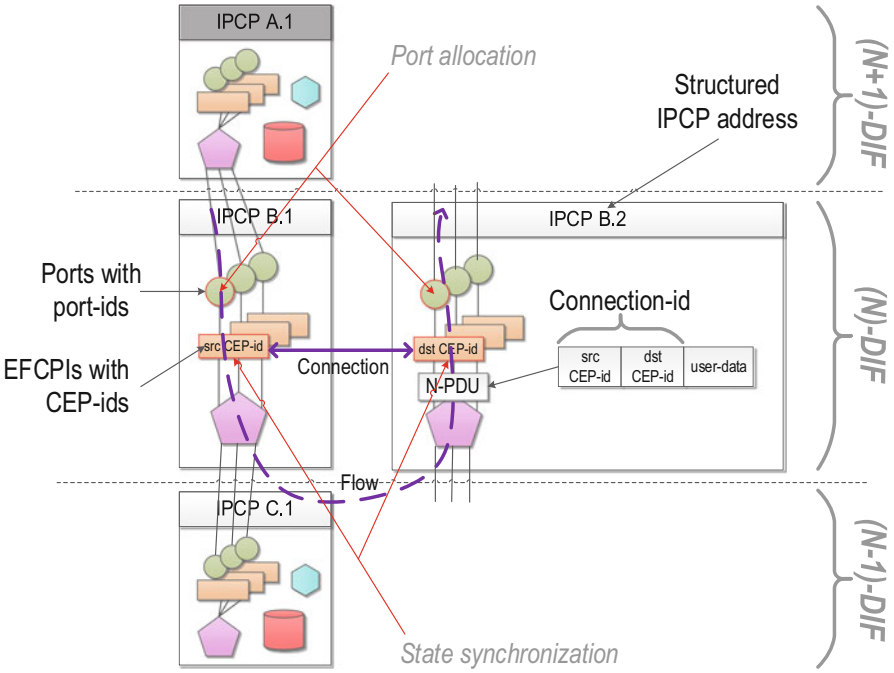



**Fig. 4.3** Overview of IPCP local identifiers

### *4.3.2 Nodes*

There are only three basic kinds of nodes in a RINA network (illustrated in Fig. 4.4). Each kind represents a computing system running RINA:

- *Hosts*: end-devices for IPC containing AEs in the top layer; they employ two or more DIF levels;
- *Interior routers*: interim devices, which are interconnecting $(N)$-DIF neighbors via multiple $(N-1)$-DIFs; they employ two or more DIF levels;
- *Border routers*: interim devices, which are interconnecting $(N)$-DIF neighbors via $(N-1)$-DIFs, where some of $(N-1)$-DIFs are reachable only through $(N-2)$-DIFs; they employ three or more DIF levels.

As depicted in Fig. 4.4, the main difference between node kinds is in an overall number of DIF levels present in a computing system. Due to the limited number of Network Interface Cards (NICs), hosts usually have a single 0-DIF (connected to the physical medium) and a few 1-DIFs leveraging on this lowest level DIF. Interior routers have potentially a lot of 0-DIFs (for each interface) but only a few relaying 1-DIFs. Border routers also perform relaying but serve as gateways between those $(N-1)$-IPCs, which are not connected directly. Thus, an $(N-2)$-DIF is needed to reach the physical medium.

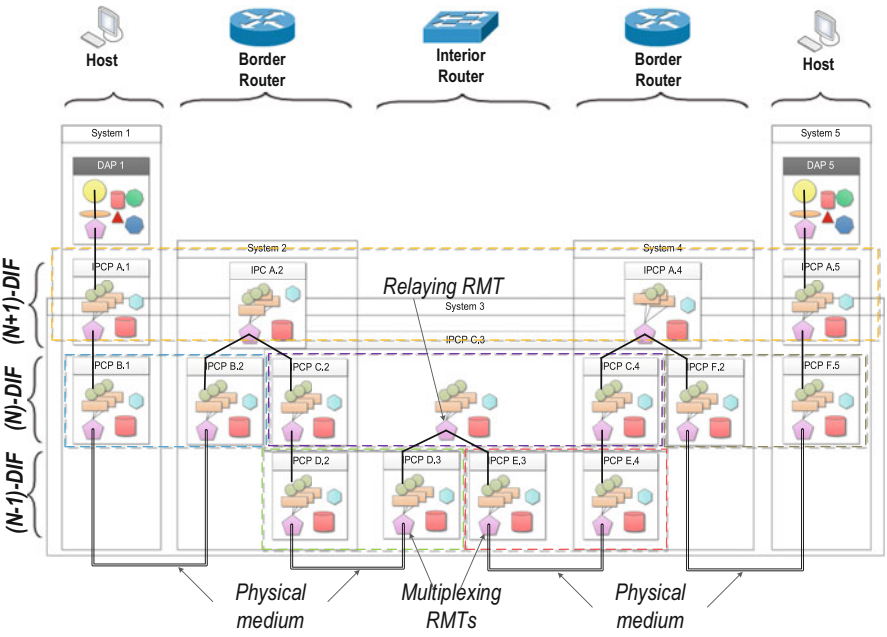



**Fig. 4.4** Example of a RINA network with three levels of DIFs and different nodes

## *4.3.3 DAF Design*

### 4.3.3.1 DIF Allocator Interface

The primary task of the *Distributed Inter-Process Communication Facility Allocator (DA)* is to return a list of DIFs where the destination application may be found based on the ANI reference and access control information. An additional and more complex DA description is available in [25]. The DA contains and works with multiple mapping tables to provide its services:

- *Naming information table*: provides an association between the APN and its synonyms;
- *Search table*: provides a mapping between the requested APN and the list of DAs where to search for it next;
- *Neighbor table*: maintains a list of adjacent peers when trying to reach other DAs;
- *Directory*: contains records mapping the APNs with access rights to the list of supporting DIFs including the DIF's name, access control information, and the provided QoS.

#### 4.3.3.2 IPC Resource Manager

*Inter-Process Communication Resource Manager (IRM)* (cf. the specification [6]), as its name suggests, manages DAF resources. This involves multiple different tasks:

- IRM processes allocate calls by delegating them to appropriate local IPCPs in relevant DIFs;
- IRM manages DA queries and acts upon their responses. When the DA response contains more than one DIF, the IRM chooses which DIF to use;
- IRM administers the use of flows between AEs and DIFs. IRM may choose to multiplex a single or multiple AE flows into single/multiple flows to a set of DIFs;
- IRM initiates joining or creating a DAF and/or a DIF. The IRM acts upon the DAF, or a DIF lost (e.g., sending notifications or perform subsequent actions).

#### 4.3.3.3 Application Process

The *Application Process (AP)* is a program intended to accomplish some purpose, which can be instantiated on a processing system. An AP contains one or more AEs, which are introduced in the following section. The AP manages some of the system resources, for example the processor, the storage, and the IPC. An AP must have at least one AE. Otherwise, the AP would have no input/output and would lack the state-sharing purpose.

#### 4.3.3.4 Application Entity

An *Application Entity (AE)* is a component of an AP (task). An AP needs to communicate with other APs for multiple purposes, potentially at the same time. The goal of AEs is to create and manage these application connections. The AE implements an application protocol that provides a shared understanding of the purpose of communication, protocol, set of objects, and their meaning that the two AEs exchange. There is only one application protocol called *Common Distributed Application Protocol (CDAP)* used for communication.

AEs could be implemented by subroutine libraries which should be hidden to application programmers—they would control it via Application Programming Interface (API) calls (e.g., similar to Berkeley Software Distribution (BSD) sockets). The example should provide a better understanding of the purpose of the AEs. Imagine an application that contains two different AEs. One AE should be responsible for serving requests for web pages, while another AE might be involved in communicating with network database server. Each of these AEs has its defined purpose, set of objects they communicate, and communication protocol.

#### 4.3.3.5 Instances of Application Processes and Application Entities

The *Application Process Instance* (AP Instance) and the *Application Entity Instance* (AE Instance) are instantiations of the AP and AE tasks. One processing system may contain multiple instances of the same AP. It is also possible to have many instances of the same AE in one AP. It is fruther possible to create an application connection specifically to one of the instances of the AP and the AE. Each instance can manipulate with unique data, and it can have unique parameters. A video conference is a good use-case of how the AP and the AE instances might be distinguished. Imagine that one AP instance is a video call with more than one participant. An AE implements a video-streaming protocol and an AEI represents the feed from one camera of a single participant.

#### 4.3.3.6 Common Distributed Application Protocol

The *Common Distributed Application Protocol (CDAP)* is the only required application protocol in RINA. It provides a platform for building all distributed applications. CDAP allows distributed applications to deal with communication at the object level without the need to do an explicit serialization and other input/output operations. The CDAP unifies the approach of sharing data over the network, so we do not need to create any additional specialized protocols.

From the application perspective, the only operations that can be performed on objects are create/delete, read/write, and start/stop (execute/suspend). These operations are primarily supported by the CDAP, which consists of three subparts: (1) *Common Application Connection Establishment (CACE)*, which is involved in connection initialization; (2) *Authentication*, which is responsible for authentication of communication parties; and (3) the *CDAP*, which is processing all other messages.

#### 4.3.3.7 Enrollment

*Enrollment* is the phase of communication that follows right after the CDAP connection is established with a member of a DIF/DAF. Two types of enrollment exist in RINA: within a DIF and within a DAF. They differ mainly from the perspective of the information that is exchanged during this phase. An AP must always be enrolled to become a full-featured member of a DAF.

The enrollment may perform the following operations:

- determine the current state of the member AP (if AP is joining the DAF for the first time, or if it is a returning member);
- assign capabilities and synonyms to the new member relevant within the DAF;
- initialize static aspects, policies, and synchronize the Resource Information Bases (RIBs);
- create additional connections.

#### 4.3.3.8 Resource Information Base

The *Resource Information Base (RIB)* is the logical representation of the local repository of objects in the DAF. Each member of a DAF has its portion of the information stored in the local RIB. All objects are accessible via the RIB Daemon, which is responsible for managing and maintaining them. The RIB is a storage (from the operating system perspective) and there are no restrictions on how to implement it. In the DAF, it should most likely be implemented as some (key-value/relational/temporal) database of application objects. In current TCP/IP based networks, the RIB can be compared to the Management Information Base (MIB) of the Simple Network Management Protocol (SNMP), which is also used for storing objects.

#### 4.3.3.9 Objects

The *object* is the designation for a structured data that the CDAP is dealing with. Objects are the primary building blocks of the RIB. Two communicating AEs create a shared object space, and they provide access to the portion of the application's RIB. All objects enforce some access rights. There are two types of RIB objects: passive (that contain static data) and active (which trigger various control events).

#### 4.3.3.10 RIB Daemon

The *Resource Information Base Daemon (RIBd)* is one of the key components of the AP. It is responsible for managing and maintaining objects in the RIB within the DAF. The RIBd monitors all events occurring within the DAF and notifies the subscribers. The AP of a DAF may have several tasks (threads), which share a state via the RIB with the help of the RIBd. If any task has requirements for information from another participant in a distributed application, it uses the RIBd to get the information.

The RIBd accepts subscriptions from tasks. The subscription for an object is a mechanism to manage (read or write) data objects in the RIB. The subscription requests should be time-driven, event-driven, or direct. The RIBd should process the subscriptions as efficiently as possible. The main functions of the RIBd within a DAF are:

- notify sets of members about the current value of selected objects;
- monitor events occurring within the DAF;
- provide the RIBd API that can be used by APs sub-tasks;
- respond to requests for information from other members of the DAF;
- maintain a mandatory log of received events.

### 4.3.4 DIF Design

#### 4.3.4.1 Delimiting

The SDU in RINA is a contiguous chunk of data. IPC might fragment the SDU (when passing it down to the $(N-1)$-DIF) or combine user-data (when passing it up to the $(N+1)$-DIF). Hence, the operation performed by the delimiting module (for specification cf. [3, 10]) is to delimit the SDU into/from the PDU's user-data preserving its identity. Employed mechanism indicates the beginning and/or the end of the SDUs. Either internal (special pattern) or external (SDU length) delimiting could be used. The delimiting module can perform both fragmentation and concatenation.

Encapsulation/decapsulation of data messages happens in the RINA components lying in the data path. Figure 4.5 depicts this process in the DIF/DAF together with the message nomenclature.

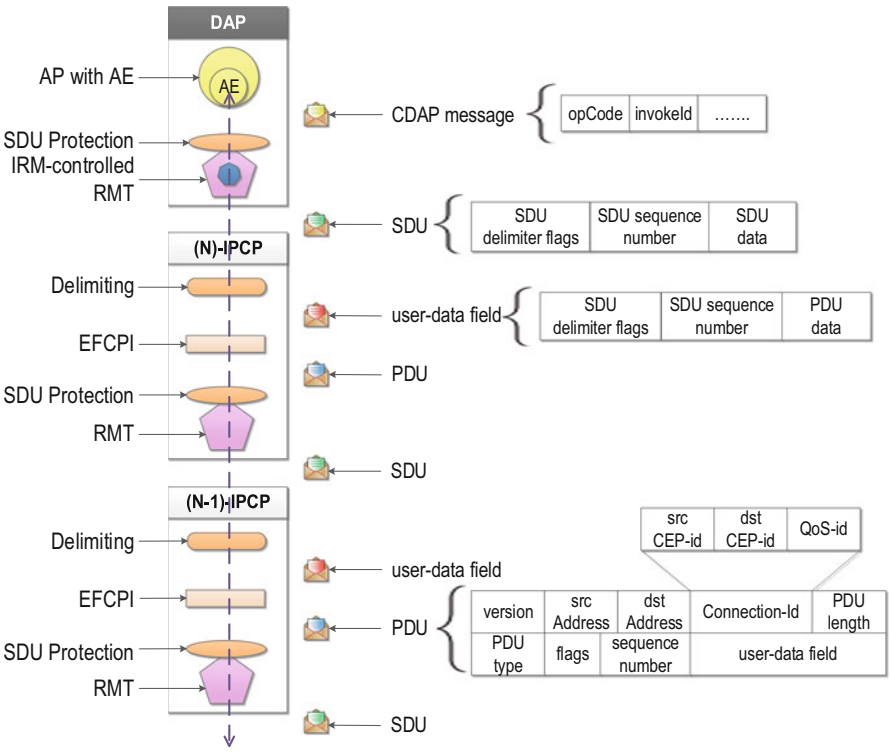



**Fig. 4.5** Message passing between RINA components

#### 4.3.4.2 Data Transfer with Error and Flow Control

The *Error and Flow Control Protocol (EFCP)* is split in two independent protocol machines coupled and coordinated through a state vector. EFCP guarantees data transfer and data control. The full EFCP functionality is described in [13]. The *Data Transfer Protocol (DTP)* implements mechanisms tightly coupled with the transported SDUs, for instance reassembly and sequencing. The DTP protocol machine operates on data PDU's fields requiring minimal processing: source/destination addresses, QoS requirements, connection-id, and (optionally) sequence number or checksum. DTP carries the user-data in the Data Transfer Protocol Data Unit (DT-PDU).

The *Data Transfer Control Protocol (DTCP)* implements mechanisms that are loosely coupled with the transported SDUs, for instance (re)transmission control using various acknowledgment schemes and flow control with data-rate limiting. DTCP functionality is based on Delta-t and DTCP processes control PDUs (ControlPDU). DTCP provides error and flow control over user-data.

There is an *Error and Flow Control Protocol Instance (EFCPI)* module per every active flow. EFCPI consists of DTP and DTCP submodules. The QoS demands drive DTCP policies. The DTCP submodule is unnecessary for flows that do not need it, i.e., flows without any requirements for reliability or flow control. The relationship between DTP and DTCP is illustrated in Fig. 4.6. Depicted are also data transfer and data control transfer paths. The control traffic stays out of the main data transfer.

#### 4.3.4.3 Relaying and Multiplexing Task

The *Relaying and Multiplexing Task (RMT)* modules have two main responsibilities: relaying and multiplexing as characterized in [7]. The goal of multiplexing is to pass PDUs from the EFCPIs and the RIBd to appropriate $(N-1)$-flows and reverse of that. Relaying handles incoming PDUs from $(N-1)$-ports that are not directed to its

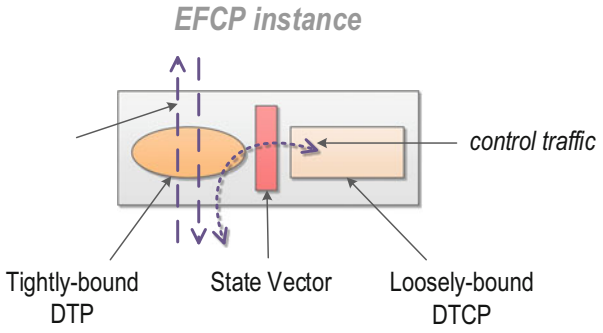



**Fig. 4.6** A EFCP instance divided into DTP and DTCP part

IPCP and forwards them to other $(N-1)$-ports using the information provided by its forwarding policy. RMT instances in hosts and bottom layers of routers usually perform just the multiplexing task, while RMTs in the top layers of interior/border routers do both multiplexing and relaying. In addition to that, RMTs in top layers of border routers perform flow aggregation.

Each $(N-1)$-port handled by the RMT has its set of input and output buffers. The number of buffers, their monitoring, the scheduling discipline, and the classification of traffic into distinct buffers are all policy matter. The RMT is a straightforward high-speed component. As such, most of its management (state configuration, forwarding policy input, buffer allocation, data rate regulation) is handled by the resource allocator, who makes the decisions based on the observed IPC process performance.

#### 4.3.4.4 SDU Protection

The *SDU Protection* is the last part of the IPCP data path, before an SDU is handed over to an underlying DIF. It is responsible for protecting SDUs from untrusted $(N-1)$-DIFs by providing mechanisms for lifetime limiting, error checking, data integrity protection, and data encryption. The SDU protection also provides mechanisms for data compression or other two-way manipulations that depend on the $(N-1)$-flow.

Due to different levels of trust, the SDU protection handles each $(N-1)$-flow on its own. This gives us the ability to skip some SDU protection mechanisms in favor of performance for trusted networks while still being protected from untrusted networks. This is controlled by using different policies that may protect SDU content with the help of integrity checks or encryption.

#### 4.3.4.5 Flow Allocator

The *Flow Allocator (FA)*, as specified in [5], processes allocate/deallocate IPC API calls and further management of all IPCP flows. The FA instantiates a *Flow Allocator Instance (FAI)* to manage each flow; FA is a controller/container for all flow allocator instances. An FAI is created upon allocate request call. It manages a given flow for its whole lifetime. The FAI handles creating/deleting EFCPIs while managing a single flow's connection. FAI returns port-id to the allocation requestor upon successful allocation as a referencing handle. The FAI participates only on port allocation, not on synchronization, which is the responsibility of the EFCPI. The FAI maintains a mapping between the flow's local port-id and the connection's local CEP-id.

An FA contains a *Namespace Management (NSM)* interface for assigning and resolving names (including synonyms) within a DIF. This activity involves maintaining the table with entries that map a requested ANI to the IPCP's address.

The *Flow object* contains all information necessary to manage any given flow between communicating parties. It is carried inside create/delete flow request/response messages controlling the FA and FAI operation. A flow object contains: a source and a destination ANI, source and destination port-ids, a connection-id, a source and a destination address, the QoS requirements, a set of policies, access control information, hop-count, and current and maximal retries of *create flow requests*.

#### 4.3.4.6 Resource Allocator

If a DIF has to support different qualities of service, then different flows will have to be allocated to different policies and traffic for them will be treated differently. The *Resource Allocator (RAl)*, delineated in [4], is a component accomplishing this goal by handling the management of various IPCP resources, in particular:

- controls the creating/deleting and the enlarging/shrinking of RMT queues;
- modifies EFCPI's DTCP policy parameters;
- controls the creating/deleting of $(N-1)$-flows and their assignment to appropriate RMT queue(s);
- manage QoS classes and their assignment to RMT queue(s);
- manage routing information affecting RMT's relaying, initiate congestion control.

The RAl maintains a catalog of meters and dials by monitoring various management resources. Each catalog item can be manipulated and shared with other IPC processes within the DIF.

## 4.4 Components and Policies

This section provides a general overview of the components design, which includes high-level abstract models of computing systems (like hosts and routers) and also their low-level submodules (like the IPCP). In general, a structure of RINASim models follows the structure proposed in the RINA specification. This intentional correspondence enables anyone understanding the RINA specifications to orient easily in RINASim as well. Though this structure does not always stand for the most natural representation of the RINA concepts in simulation models, it provides a framework for evaluating properties of the architecture and to identify missing or inaccurate information in the original specification.

The RINA specifications present the proposed network architecture as a generic framework, where mechanisms are intended to perform basic common functionality and policies are defined to select the most appropriate implementation of variable functionality. Rather than providing an exhaustive implementation of policies for

each parameterized function, RINASim provides interfaces that are used by the core implementation to call functions defined by the selected policies.

The RINASim policy framework is based on the OMNeT++ Network Topology Description (NED) module interface [20], which helps to minimize the need for modifying existing C++/NED source code. Instead of placing a simple module with a policy implementation inside the simulation network graph, a placeholder interface module is used. This design allows the potentially unlimited amount of user policy implementations to be defined and easily switchable via the configuration files (by setting a parameter of the encompassing module). Each policy consists of a NED module interface and a base C++ class.

### *4.4.1 Nodes*

RINASim offers a variety of high-level models simulating the behavior of independent computing system (examples of all three types are provided in Fig. 4.7). These models can be employed to quickly set up simulation experiments. Through parameterization and extension, it is possible to test different deployments and

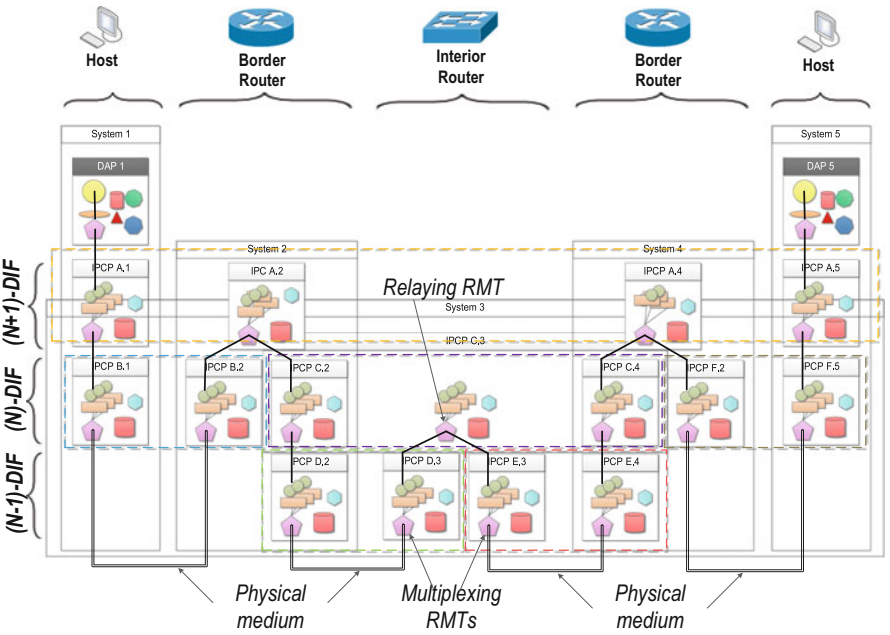



**Fig. 4.7** Examples of RINASim node modules of different types

settings. Based on the RINA specification, we can distinguish the following node types:

- *Host nodes*, which represent devices or systems that run distributed applications. These nodes implement the full RINA stack and they contain an application process(es) as well. AP instances are configured to communicate with each other to simulate the behavior of an arbitrary RINA application. Currently, there are several predefined host nodes depending on a count of APs and AEs.
- *Routers* (intermediate nodes), which can be either interior or border routers. A router is a device that interconnects different underlying DIFs and often does not run user applications. Just as in the RINA specification, there are either interior or border routers depending on the DIF stack depth (influenced partially also by a count of interfaces).

### *4.4.2 DAF Modules*

DAF components can be divided into submodules: (a) representing IPC endpoints; (b) interconnecting APs and available IPCPs; and (c) discovering APNs. The internal structure of these components and their relationship are depicted in Fig. 4.8 and described below.

#### 4.4.2.1 DIF Allocator

The `difAllocator` module handles locating a destination application based on its name. The DA is a component of the DAP's IPC management that takes the ANI and access control information and returns a list of DIF-names through which the requested application is available. Moreover, the `difAllocator` module provides statically configured knowledge about the simulation network graph. RINASim's DA overloads any NSM calls. The DA comprises the following submodules:

- `da`: delivers the core functionality of the DA;
- `namingInformation`: provides the mapping between APN synonyms;
- `directory`: provides a mapping between APN and DIF-names;
- `searchTable`: provides the mapping between APN and peer DA for authoritative search;
- `neighborTable`: provides the mapping between peer DA and neighboring DA instances.

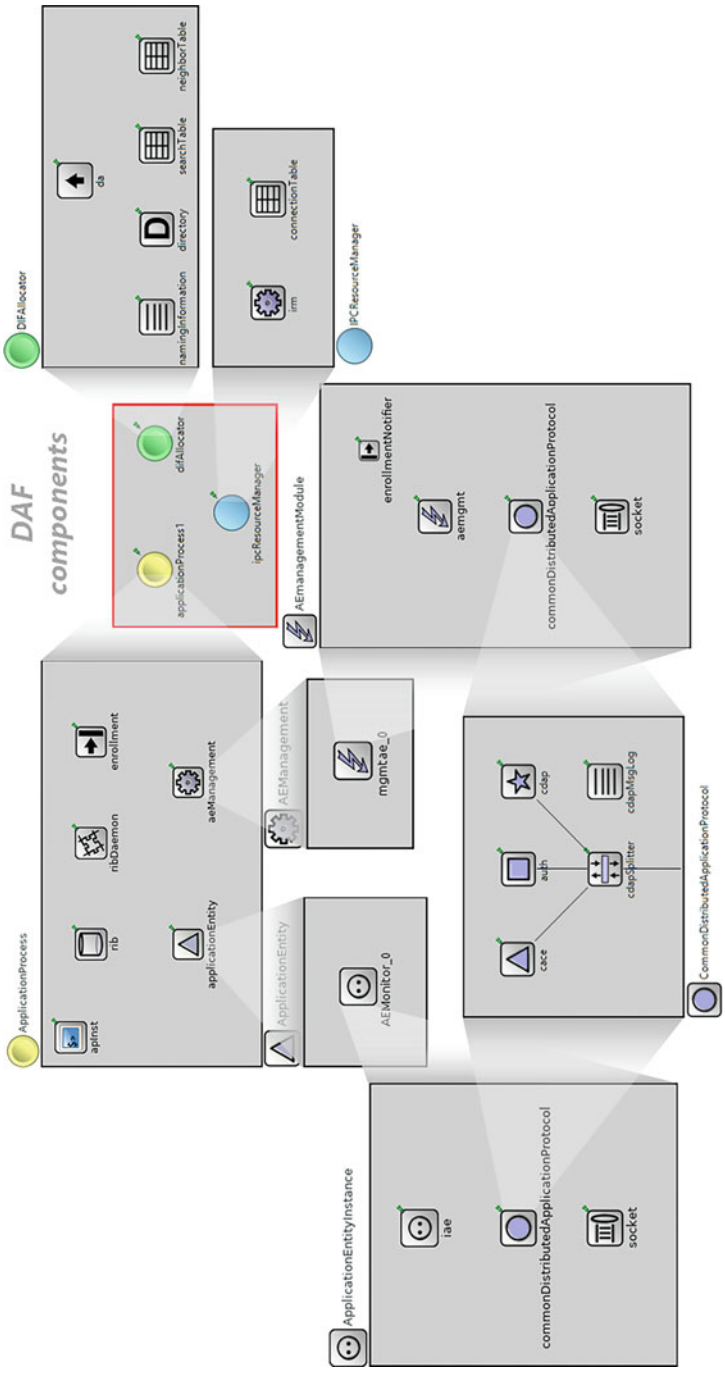



**Fig. 4.8** Overview of the DAF modules

#### 4.4.2.2 IPC Resource Manager

The `ipcResourceManager` module currently queries the DA module to find suitable IPCP and relays communication between the AE and the IPCP. It comprises the following submodules:

- `irm`: the broker between APs and IPCs, handles AP flow (de)allocation calls;
- `connectionTable`: maintains the state of the $N - 1$ flows.

#### 4.4.2.3 Enrollment

The `enrollment` module manages all communication within the CACE phase and during the enrollment phase. It is responsible for managing the states of communication and exchanging initial application objects. The `enrollment` module contains the following submodules:

- `enrollment`: delivers the core functionality of Enrollment;
- `enrollmentStateTable`: provides state information about connections.

#### 4.4.2.4 Application Process

The `applicationProcess` module currently handles all application communication from initialization of the first connection to deallocation of all resources. The module acts as an independent unit within the DAF. The `applicationProcess` module also provides statically configured information about its name within the DAF. The following submodules comprise the `applicationProcess` module:

- `apInst`: spawns a running application;
- `rib`: provides an interface for object management in the database;
- `ribDaemon`: handles subscription for objects and manages them within DAF members;
- `enrollment`: handles initial phases of communication;
- `applicationEntity`: wrapper for standard AEIs;
- `aeManagement`: wrapper for AE management instances.

#### 4.4.2.5 AE Monitor Instance

The `AEMonitor` module is an instance of a specialized AE that mimics a ping-like application. It contains the following submodules:

- `iae`: delivers the core functionality of an AEI;
- `commonDistributedApplicationProtocol`: handles CDAP messages;

- `socket`: application buffer for read/write operations cooperating with $(N-1)$-EFCP.

#### 4.4.2.6 AE Management Instance

The `mgmtae` module is used for handling the management communication. It is mainly used for enrollment and for maintaining RIB updates. It comprises the following submodules:

- `aemgmt`: delivers the core functionality;
- `commonDistributedApplicationProtocol`: handles CDAP messages;
- `enrollmentNotifier`: serves as the mediator in communication between `enrollment` and `aemgmt`;
- `socket`: application buffer for read/write operations cooperating with $(N-1)$-EFCP.

#### 4.4.2.7 RIB Daemon

The `ribDaemon` module manages objects in the local RIB repository. It provides an interface for manipulating objects (e.g., read/write/delete) in the RIB.

#### 4.4.2.8 RIB

The `rib` module acts as a database of application objects. It has an interface for searching, writing, and deleting objects. The RIB is primarily accessed by the RIBd that manages these objects within the AP.

#### 4.4.2.9 Common Distributed Application Protocol

The `commonDistributedApplicationProtocol` module accepts and sends all CDAP messages. The following submodules help to differ between types and purposes of CDAP messages and allows their logging.

- `cace`: handles all messages that belong to CACE;
- `auth`: handles messages that belong to authentication phase;
- `cdap`: accepts and sends all other messages;
- `cdapSplitter`: forwards messages to appropriate module;
- `cdapMsgLog`: provides logging of all messages.

### 4.4.3 DIF Modules

All currently implemented DIF components are enclosed to the `IPCProcess` container module (IPCP instance). The overall structure of the `IPCProcess` is depicted in Fig. 4.9. The following subsections describe the included (sub)modules.

#### 4.4.3.1 Delimiting

The delimiting module handles SDUs in the form of `SDUData` from the $N + 1$ DIF and produces the `UserDataField` for the EFCP instance module. In the opposite direction, it accepts the `UserDataField` and produces the `SDUData` to the $N + 1$ DIF. The module is capable of fragmenting and concatenating incoming SDUs. Fragmentation is based on `maxPDUsize`. Concatenation takes the incoming SDUs and puts them in a single `PDUData` until `maxPDUSize` is met or until the delimiting timer expires. The delimiting module does not contain any submodules. No policies are currently associated with this module.

#### 4.4.3.2 EFCP Compound Module

The `efcp` compound module handles the data transfer and the associated state vectors. It takes the SDUs from the $(N + 1)$-IPCP or the CDAP message from the RIBd and creates PDUs. This module dynamically spawns EFCP instances. Dynamic modules consist of one delimiting module (`delimiting_<portId>`) and (possibly) multiple EFCPI modules (`efcp_<cepId>`) per one flow. The EFCPI module itself is also a compound module and contains the static modules `DTP` and `DTPState`. If the flow (QoS demands) requires control, then there are `DTCP` and `DTCPState` modules. It also includes the `efcpTable` to store bindings between instances of delimiting and EFCP. Moreover, there is a `mockEFCPI` module containing a simplified EFCPI with only DTP-like capabilities for management flows.

There are three static submodules:

- `efcp`: creates and deletes EFCP instances and delimiting modules;
- `efcpTable`: contains bindings between delimiting and EFCPI (DTP and DTCP);
- `mockEFCPI`: it is a simplified EFCPI with only DTP-like capabilities;

Furthermore, the `efcp` compound module may contain dynamically created pairs of delimiting and EFCPI modules:

- `delimiting_<portId>`: handles fragmentation/concatenation;
- `efcpi_<cepId>`: handles data transfer and control loop functions.

Policies related to the EFCP are specified in the DTP and DTCP subsections.

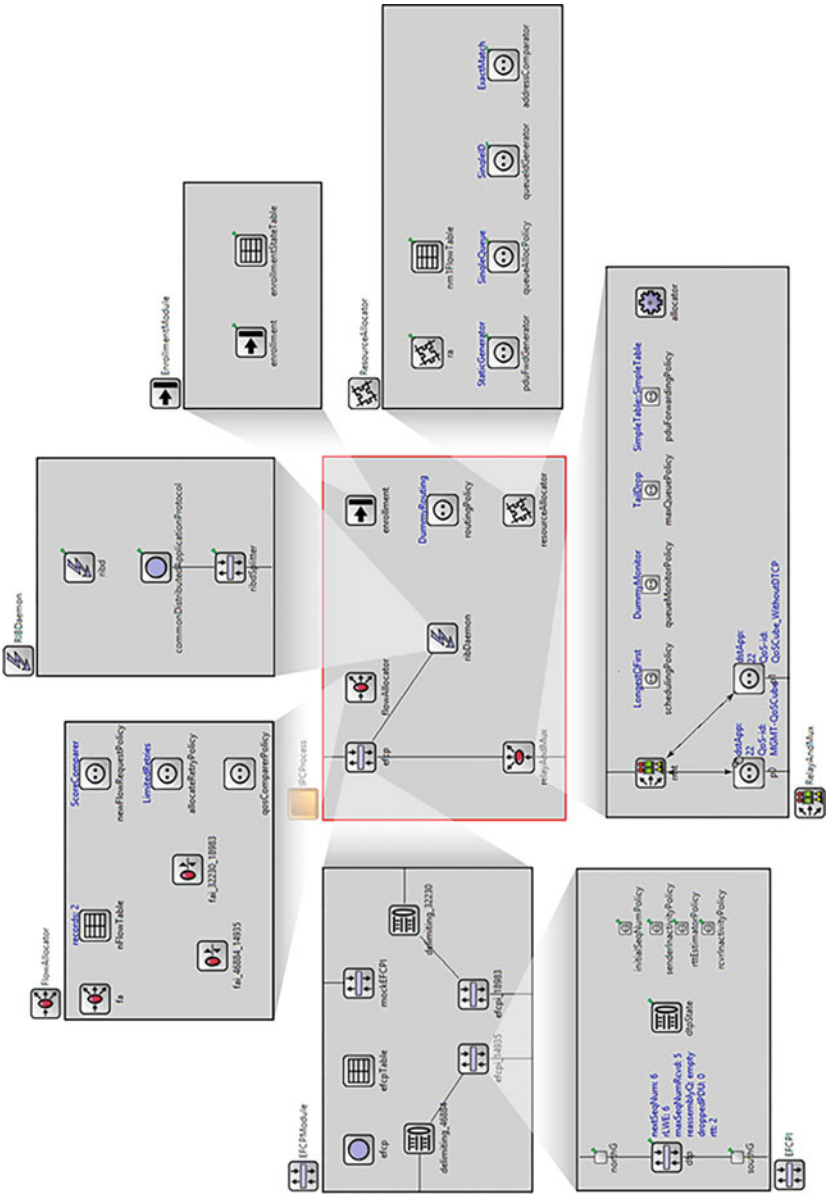


**Fig. 4.9** IPCP's DIF modules

#### 4.4.3.3 EFCPinstance

An EFCP instance locally manages the established flow. The `efcpi_<cepId>` module contains the `DTP` and `DTPState` submodules. Any necessary policy submodules associated with the flow are part of this compound module as well. The incorporated submodules are:

- `dtp`: this module implements the DTP. The `dtp` module accepts user data content from the delimiting module, generates PDUs, and passes them to `relayAndMux`. If necessary, it asks `dtcp` for reliable data transfer. DTP policies are configurable via the *config.xml* file by specifying an EFCP policy set in the QoS-cube;
- `dtpState`: this module holds properties related to the actual data transfer. In RINASim, `dtpState` module stores all necessary variables and queues;
- `dtcp`: this module implements the DTCP. The `dtcp` handles retransmission and flow control related tasks. From the perspective of RINASim, `dtcp` executes policies to update the `dtcpState`. Policies react to situations when error recovery and/or flow control are expected. The current implementation supports retransmission, window-based flow control, allowed gap, and A-Time;
- `dtcpState`: It maintains DTCP-related variables;
- `northG, southG`: these pass-through modules enable better link visualization.

The `dtp` module is associated with the following DTP policy types:

- `InitialSeqNumPolicy`: it allows some discretion in selecting the initial sequence number when a Data Run Flag (DRF) is going to be sent. (default: sets the new sequential number to 1);
- `RcvrInactivityPolicy`: if no PDUs arrive in the watched period, the receiver should expect a DRF in the next transfer PDU. This policy represents a timer that should be set to $2 \cdot (MPL + R + A)$. (default: resets all receiver-side variables and queues);
- `SenderInactivityPolicy`: this policy represents a timer, which detects long periods of no traffic. It indicates that a DRF should be sent. $\Delta t$ should be set to $3 \cdot (MPL + R + A)$. (default: resets all sender-side variables and queues);
- `RTTEstimatorPolicy`: this policy is executed by the sender to estimate the duration of the retransmission timer. This policy is usually based on an estimate of Round-Trip Time (RTT) and received/lost ACKs. (default: computes RTT as an average from the current RTT and the last computed estimate).

The associated DTCP policies are:

- `ECNPolicy`: handles the Explicit Congestion Notification (ECN) bit in incoming DT-PDUs. (default: sets inner variable based on bit in DT-PDU header);
- `ECNSlowDownPolicy`: it is executed after IPCP's RAI receives a congestion notification. (default: no action);

- `LostControlPDUPolicy`: determines what action to take when the protocol machine detects that a control PDU (ACK or flow control) may have been lost. (default: sends `ControlAck` and empty DT-PDU);
- `NoOverridePeakPolicy`: allows the rate-based flow control to exceed its nominal rate for a presumably short period of time. (default: puts the DT-PDU on `ClosedWindowQ`);
- `NoRateSlowDownPolicy`: is used to lower the send rate momentarily below the allowed rate. (default: no action);
- `RateReductionPolicy`: is executed in case of rate-based flow control. When local shortage of buffers occur or when buffers are less full than a given threshold, this policy increases the rate agreed during the connection establishment. (default: slows down by 10% if buffers are getting clogged);
- `RcvFCOverrunPolicy`: it determines what action to take if the receiver receives PDUs, but the credit or rate has been exceeded. (default: drops the PDU and sends a control PDU as response);
- `RcvrAckPolicy`: the policy is executed by the receiver of the DT-PDU, as it provides some discretion in the action taken. (default: either acknowledges immediately or starts the A-Timer and acknowledges the `RcvLeftWindowEdge` when it expires);
- `RcvrControlAckPolicy`: executes when receiving the `ControlAckPDU`. (default: it checks the received values and, if needed, returns a control PDU with updated information);
- `RcvrFCPolicy`: this policy is invoked when a DT-PDU is received to give the receiving protocol machine an opportunity to update the flow control allocations. (default: increments receiver's right window edge);
- `ReceivingFCPolicy`: it is invoked by the receiver of a DT-PDU in case there is a flow control present, but no retransmission control. (default: sends `FlowControlPDU`).
- `ReconcileFCPolicy`: it is invoked when both credit and rate-based flow control are in use, and they disagree on whether the protocol machine can send or receive data. If this is the case, then the protocol machine can send or receive; otherwise, it cannot;
- `RxTimerExpiryPolicy`: it is executed by the sender when a retransmission timer expires. This policy must be run in less than the maximum time to ACK. (default: retransmits DT-PDU with `seqNum` equal to the one in `RXTimer`);
- `SenderAckPolicy`: this policy is executed by the sender when PDUs might be deleted from the retransmission queue. It is useful for multicast-like use-cases, when it is feasible to delay the removal of PDUs from the retransmission queue. (default: removes DT-PDU from retransmission queue up to the acknowledged sequence number);
- `SenderAckListPolicy`: is executed by the sender when PDUs might be deleted from the retransmission queue. The policy is used in conjunction with the selective ACK aspects. It is useful for multicast transfers just like the previous policy. (default: removes specified `seqNum` ranges from retransmission queue);

- `SendingAckPolicy`: is executed upon A-Timer expiration in case there is a DTCP present. (default: updates receiver's left window edge and sends ACK/FlowControlPDU);
- `SndFCOverrunPolicy`: it determines what action to take if the sender has a DT-PDU ready for dispatch but values of `SndRightWindowEdge` or `SndRate` are blocking them. (default: put DT-PDU in `ClosedWindowQueue`);
- `TxControlPolicy`: it is used when there are conditions that warrant sending fewer PDUs than allowed by the sliding window flow control. (default: puts as many DT-PDUs from `generatedPDUQ` to `postablePDUQ` as possible).

#### 4.4.3.4 RMT

The *Relaying and Multiplexing Task* represents a stateless function that takes incoming PDUs and relays them within the current IPC or passes them to the outgoing port. In particular, the RMT takes PDUs from $(N-1)$-port Identifiers (IDs), consults their address fields, and performs one of the subsequent actions:

- If the address is not an address (or a synonym) for this IPC process (which is determined by RAI's `AddressComparator` policy), it consults the PDU forwarding policy and posts it to the appropriate $(N-1)$-port(s).
- If the address is one assigned to this IPC process, the PDU is delivered either to the appropriate EFCP flow or the RIBd (via a mock EFCP instance).
- Outgoing PDUs from EFCP instances or the RIBd are posted to the appropriate $(N-1)$-port-id(s).

In RINASim, all functionality of the RMT, including a policy architecture, is encompassed in a single compound module named `relayAndMux`, which is present in every IPC process. The included RMT submodules are:

- `relayAndMux`: consists of multiple simple modules of various types, some of which are instantiated dynamically at runtime;
- Static submodules:
  - `rmt`: implements fundamental RMT logic that decides what should be done with messages passing through the module;
  - `allocator`: a manager unit for dynamic modules that provides an API for adding, deleting, and reconfigurating RMT ports and queues;
- Dynamic submodules:
  - `RMTPort` (encompassed in `RMTPortWrapper`): a single endpoint of an $(N-1)$-flow;
  - `RMTQueue`: a representation of either an input or an output queue. The number of `RMTQueues` per $(N-1)$-port is a matter of RAI policies;
  - `sdup`: it performs SDU protection.

The RMT provides the following policies:

- `schedulingPolicy`: it is invoked each time that some $(N-1)$-port has data to send. The policy employs the algorithm to make a decision about which of the port's queues should be handled first;
- `queueMonitorPolicy`: it is a stateful policy that manages variables used by other RMT policies. It is invoked by various events occurring inside RMT and its ports and queues;
- `maxQueuePolicy`: this is a policy used for deciding what to do when queue lengths are overflowing their threshold lengths. It is invoked whenever the size of items in a queue reaches a threshold;
- `pduForwardingPolicy`: it is a policy deciding where to forward a PDU. It accepts the PDU as an argument, does a lookup in its internal structures (usually a forwarding table populated by the Protocol Data Unit Forwarding Generator (PDUFG) policy), and returns a set of $(N-1)$-ports.

#### 4.4.3.5 Routing

The *Routing* module is a policy that serves by exchanging information with other IPCPs in the DIF in order to generate a set of routing information. It indirectly provides input for populating the RMT `PDUForwardingPolicy`. Routing policies are used to propagate information about routing in the DIF. They depend on the PDUFG. The policies consist of a single replaceable policy that conducts routing within the DIF. There are examples of policies (e.g., `DummyRouting`, `DomainRouting`, `SimpleRouting`) leveraging distance-vector, link-state, or native RINA approaches for routing PDUs.

#### 4.4.3.6 Flow Allocator

The `flowAllocator` module handles (de-)allocation request and response calls from the `IRM`, `RIBDaemon`, `DAFEnrollment`, or the `AE`. The FA itself is separated in the flow table and the flow management modules. Included submodules are:

- `fa`: provides the core functionality involving the instantiation of FAIs;
- `nFlowTable`: it maintains mappings between $(N)$-flow and bound FAI;
- `fai_<portId>_<CEPid>`: it manages the whole flow lifecycle.

The supported policies of the `flowAllocator` module are:

- `allocateRetryPolicy`: occurs whenever initiating an FAI receives a negative create flow response. It allows the FAI to reformulate the request and/or to recover properly from failure;
- `qosComparePolicy`: checks if existing $(N-1)$-flows can be used to support a $(N)$-flow;

- `newFlowRequestPolicy`: is invoked after the FAI's instantiation. Policy subtasks involve both (1) evaluation of access control rights and (2) translation of QoS requirements specified in allocate request to appropriate RAI's QoS-cubes.

#### 4.4.3.7 Resource Allocator

The `resourceAllocator` is one of the most important components of an IPC process. It monitors the operation of the IPC process and makes adjustments to its operation to keep it within the specified operational range. The `resourceAllocator` submodules are:

- `ra`: provides core functionality by managing the RMT and the connections to other IPCPs via $(N-1)$-flows;
- `nm1FlowTable`: maintains a table containing information about the active $(N-1)$-flows;

Supported/included policies are:

- `pduFwdGenerator`: a policy, to manage the RMT's PDU forwarding policy;
- `queueAllocPolicy`: a policy handling RMT queue allocation strategy;
- `queueIdGenerator`: a policy generating queue IDs from flow information and PDUs;
- `addressComparator`: a policy matching PDU address and IPCP address.

## 4.5 Demonstrations

This section outlines some of the scenarios where RINA is employed as the native network stack. General instructions (how to set up and run the examples) are provided for the reader. Furthermore, a detailed description tries to reveal advantages of adopting RINA for certain Internet use-cases. Demonstration source codes are located in the `/examples/` folder, each one includes the following files (which may be reused as templates when creating other RINASim scenarios):

- *<name>.ned*: the OMNeT++ simulation network, which contains definitions of nodes and their interconnections;
- *omnetpp.ini*: the scheduled setup including model configurations (e.g., node addresses, ANI for AEs, references to the XML configuration) applied during the initialization of scenario;
- *config.xml*: the file contains more complex/structured configurations (e.g., DA's mappings, RAI's QoS-cubes, pre-allocation and pre-enrollment settings) in the form of XML data, which are mostly applied during initialization;
- **.anf*: the file is describing which statistics should be collected and evaluated during a simulation run;
- *./results/**: the scalar/vector results of various simulation runs.

### 4.5.1 Demonstration Network

Simulation source code files relevant for this scenario are located in the example folder */examples/Demos/UseCase5*. The motivation behind the demo simulation is to show a ping-like application within the simple network consisting of all different node types. The topology contains two host nodes (called `HostA` and `HostB`), two border routers (called `BorderRouterA` and `BorderRouterB`), and one interior router (identified as `InteriorRouter`) interconnected together.

There are a total of six *named* DIFs of three different ranks. The network is shown in Fig. 4.10. Please note the addressing scheme where the same node may use the same (*italicized*) address on a different DIF as long as this address is unambiguous within the layer's scope. The RINA address length and syntax is policy-dependent (compared to IP or Medium Access Control (MAC) addresses). The demonstration uses a flat address space with simple strings as addresses. The six DIFs are:

- The *TopLayer* DIF is common to `HostA` (with address *hA*), `BorderRouterA` (address *rA* and self-enrolled), `BorderRouterB` (address *rB*), and `HostB` (*hB*). Self-enrolled APs/IPCPs in RINASim are automatically members of some DAF/DIF and they are skipping the enrollment phase during any communication.
- The three middle DIFs are *MediumLayerA*, *MediumLayerAB*, and *MediumLayerB*. *MediumLayerA* is common to `HostA` (*ha*) and `BorderRouterA` (address *ra* and self-enrolled). *MediumLayerAB* is common to `BorderRouterA` (*rA*), `InteriorRouter` (address *rC* and self-enrolled), and `BorderRouterB` (*rB*). *MediumLayerB* is common to `BorderRouterB` (address *rb* and self-enrolled) and `HostB` (*hb*).
- The two bottom most DIFs are *BottomLayerA* and *BottomLayerB*. *BottomLayerA* is common to `BorderRouterA` (*ra*) and `InteriorRouter` (address *rc* and self-enrolled). *BottomLayerB* is common to `InteriorRouter` (address *rc* and self-enrolled) and `BorderRouterB` (*rb*).

By default, every RAl contains an implicit QoS-cube (with QoS-id *MGMT-QoSCube*) that defines QoS parameters (e.g., reliability and minimum bandwidth) for management traffic and that guarantees successful mapping of management SDUs onto the appropriate (*N*)-flow. Apart from this default QoS-cube, each RAl loads the QoS-cube set according to the simulation configuration. For demonstration, there are two more QoS-cubes available for each RAl called *QoSCube-RELIABLE* and *QoSCube-UNRELIABLE* (with the same QoS parameters differing only in data transfer reliability).

The DA implementation currently allows only the static change of its settings (namely different kinds of mappings). Hence, the necessary configuration step is to initialize the DA properly in order to provide services to FA, RAl, and other components depending on naming information. In particular, two DA's tables are important for overall functionality: `Directory` (helps to search target IPCP for a given APN) and `NeighborTable` (used by the FA to find a neighbor IPCP for a given IPCP).

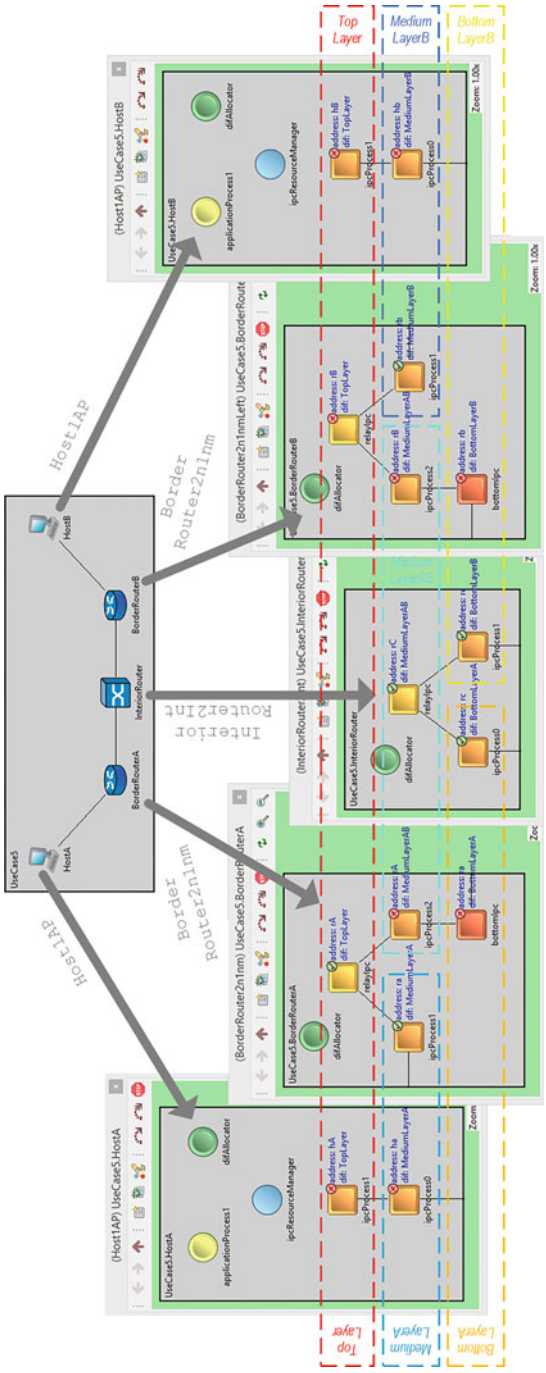



**Fig. 4.10** Demonstration network diagram

The simulation description is divided into two phases:

1. *Enrollment Phase*: if another IPCP wants to communicate within a given DIF, then it needs to be enrolled by a DIF member. Self-enrolled IPCPs are members of certain DIFs from the beginning of the simulation. They also help other IPCPs to join a DIF. In order to allow communication between any node, the simulation is scheduled to commence enrollment of: `BorderRouterA` into *BottomLayerA* at $t = 1$ s; `BorderRouterA` into *MediumLayerAB* at $t = 1.5$ s; `BorderRouterB` into *TopLayer* at $t = 2$ s; and `HostB` into *TopLayer* at $t = 5$ s. The enrollment usually involves recursive calls of enrollment procedures in lower ranked DIFs.
2. *Data Transfer Phase*: the IPC comprises of flow allocation, data transfer, and optional flow deallocation. `HostA` and `HostB` are configured to exchange messages via a ping-like protocol (measuring one-way and round-trip delays). In this case, flow allocation is initiated at $t = 10$ s, the first ping is sent at $t = 15$ s and flow deallocation occurs at $t = 20$ s.

All steps related to the enrollment phase are described in the example of an application connection in Sect. 4.5.2. For now, let us skip the enrollment phase by stating that except `HostA` all other DIF/DAF members were successfully enrolled. All flows created during the enrollment phase carry only CACE messages (for connection establishment), and they are intended for direct RIBd-to-RIBd communication employing various management messages. These flows are called *management flows*.

The main outcome of this scenario is a simulation of data transfer events between ping-like applications (`APPing`) that run on `HostA` and `HostB`. This ping-like protocol sends probe request (M_READ) from `HostA` to `HostB`, where `HostB` replies with the response (M_READ_R). One-way and round-trip time delays are measured according to the timestamp differences of these messages.

The data flow allocation starts at $t = 10$ s. `HostA's applicationProcess1` requests flow for communication with `HostB's applicationProcess1`. The event goes through the following set of steps:

1. An *Allocate request* is delivered to the IRM. Over there, the DA is asked to resolve the destination ANI into an IPC address within certain DIF available to `HostA`. The following result is returned yielding that the destination is reachable via IPCP *hB* in the *TopLayer*;
2. `HostA` can access the *TopLayer* leveraging `ipcProcess1`. Hence, IRM delegates a *allocate request call* to `ipcProcess1's` FA. The FA instantiates the EFCPI and verifies whether the IPCP is enrolled into the DIF before any attempt for sending the *create request flow*. A couple of enrollment events repeat recursively: (a) enrollment of `HostA's ipcProcess0` into *MediumLayerA* by `BorderRouterA`; (b) creation of management flow between IPCP *ha* and IPCP *ra* within *MediumLayerA*; (c) enrollment of `HostA's ipcProcess1` into *TopLayer* by `BorderRouterA`;

3. After a successful enrollment of `ipcProcess1`, the FA may continue with the flow allocation. FA exchanges the *create request/respond flow* with `HostB`. This includes the creation of an $(N-1)$-flow between *ha* and *ra* in *MediumLayerA* and the creation of the $(N)$-flow between *hA* and *hB* in the *TopLayer*. It gets more complex in the *TopLayer* because M_CREATE and M_CREATE_R messages must be relayed by the border routers to reach `HostB`, which causes additional recursive flow allocations between interim devices (i.e., *BorderRouterA*, *InteriorRouter*, *BorderRouterB*). All interim devices are already enrolled into their DIFs, thus the established flows serve as carriers for `HostA` and `HostB` data transfer;
4. M_CREATE from `HostA` to `HostB` is received by `BorderRouterA's relayIpc`. `BorderRouterA` inspects the *create request flow* and determines `BorderRouterB` with the help of the DA as the next-hop. Because border routers are not directly connected, they can communicate via `InteriorRouter` as a proxy. Therefore, `BorderRouterA` establishes a flow between *ra* and *rc* of *BottomLayerA* and sends a *create request flow* in *MediumLayerAB*;
5. M_CREATE from `BorderRouterA` to `BorderRouterB` is received by `InteriorRouter's relayIpc`. The message then needs to be relayed to `BorderRouterB`. Hence, the flow is created between *rc* and *rb* in *BottomLayerB*. Then, the *create request flow* is forwarded within this DIF;
6. M_CREATE from `BorderRouterA` to `BorderRouterB` within *MediumLayerAB* is received by `BorderRouterB's ipcProcess2`. `BorderRouterB` accepts the flow and sends a *create response flow* back to `BorderRouterA`. Because the flow connecting both border routers (*rA* and *rB* within *MediumLayerAB*) is established, the flow allocation from step 4 may continue;
7. M_CREATE from `HostA` to `HostB` is received by `BorderRouterB's relayIpc` after passing through flows created during steps 5 and 6. *BorderRouterB* inspects the *create request flow* and determines that `HostB` is reachable via its *MediumLayerB*. In order to successfully relay M_CREATE to its final destination, `BorderRouterB` allocates the flow between *rb* and *hb* in *MediumLayerB*. Subsequently, M_CREATE is forwarded to `HostB`;
8. M_CREATE is received by `HostB's ipcProcess1`. The FA then notifies the `applicationProcess1` about the current flow allocation. The flow is accepted by `applicationProcess1` for data transfer between APs. The decision is returned to `ipcProcess1's` FA. The IRM is asked to create bindings between the AP and the IPCP. The FA instantiates the EFCPI, updates the flow object, and replies back to the requestor with M_CREATE_R;
9. M_CREATE_R is relayed via all flows formed during steps 4–7 to `HostA` until `ipcProcess1's` FA receives this message. The FA updates the flow object and notifies `applicationProcess1` about the successful flow allocation. Then the IRM adds the missing bindings, and the whole data path between `HostA` and `HostB` is ready. The $(N)$-flow in *TopLayer* can carry data traffic between AEs with the help of all underlying flows.

The next event is a transfer of data traffic between AEs. HostA sends five ping-like probing objects inside the M_READ message starting at $t = 15$ s. Upon the reception of these messages, HostB replies with a probe response (in the form of a dedicated M_READ_R message). Data path and flows are depicted in Fig. 4.11 with different colors accompanied by the following description. The event consists of five repetitions of these two steps:

1. HostA's applicationProcess1 sends an M_READ message, which is passed through the IRM into ipcProcess1 to a flow prepared during the previous event and descends to ipcProcess0. The message travels through the medium and flow connecting HostA with BorderRouterA in *MediumLayerA*, where it is received by ipcProcess1. It is relayed by BorderRouterA's relayIpc to ipcProcess2 and flow interconnecting BorderRouterA and BorderRouterB in *MediumLayerAB*. Because border routers are not directly connected, the message is passed to a lower bottomIpc into flow interconnecting BorderRouterA with the neighboring InteriorRouter in *BottomLayerA*. The message traverses through the medium and it reaches InteriorRouter's ipcProcess0. Over there, the message ascends to relayIpc, where it is relayed within *MediumLayerAB*. Then it descends to ipcProcess1 into the flow interconnecting the InteriorRouter and the BorderRouterB in *BottomLayerB*. The message travels through the medium to BorderRouterB's bottomIpc. It ascends to ipcProcess2 and is relayed by relayIpc to ipcProcess1. Finally, the message reaches HostB's ipcProcess0 through the medium inside the flow within *MediumLayerB*. It ascends to the flow in ipcProcess1 (member of *TopLayerB*) and through the IRM to HostB's applicationProcess1 as the recipient;
2. HostB's applicationProcess1 responds with an M_READ_R message that returns to HostA traveling in opposite direction through the same data path (marked with a thick line) as in step 1. Referring to Fig. 4.11, the message is either encapsulated or decapsulated depending on the direction.

After the APs exchanged pings, HostA's AE closes the connection and sends a *deallocate submit* to HostB at $t = 20$ s. Deallocation affects only the flow present in the *TopLayer*. The current RINASim implementation leaves the underlying ($N - 1/2$)-flows (i.e., those not directly connected with APs) intact because they may be reused later by other applications. This event is accompanied by the following steps:

1. HostA's applicationProcess1 tells the IRM to deliver a *deallocate submit*. The IRM unbinds the port from its side. Then, the IRM delegates the *flow deallocation* to ipcProcess1's FA;
2. This FA generates an M_DELETE message with an updated Flow object state inside and sends the object towards HostB through the flow in the *TopLayer*. The message follows the data path leveraging existing management flows created during the enrollment phase;

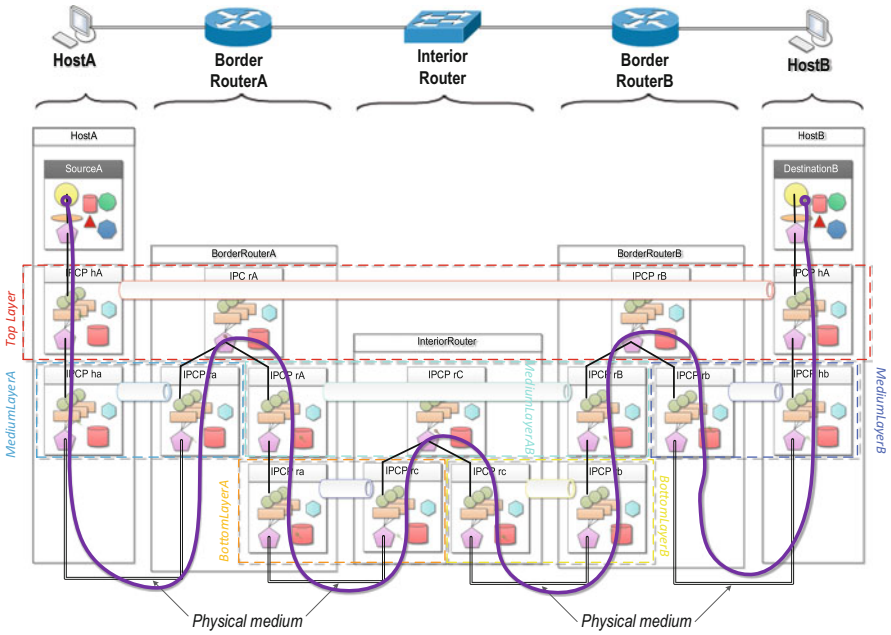



**Fig. 4.11** Data transfer path illustration for the demonstration network

3. `HostB's ipcProcess1` receives the M_DELETE. The FA updates its version of the `Flow` object. The FA then delivers a *deallocation submit* to `HostB's applicationProcess1`, which tells the IRM to remove the bindings;
4. `ipcProcess1's` FA on `HostB` then replies with an M_DELETE_R acknowledging the successful flow deallocation. This message is carried back to `HostA`;
5. `HostA's ipcProcess1` receives the M_DELETE_R. The FA marks the flow as deallocated and disconnects remaining bindings between IPCP and IRM.

The result of the flow (de)allocation and the flow's state is maintained in `ipcProcess1's NFlowTable` of `HostA` and `HostB`.

### *4.5.2 Simple Application*

The simulation source code that is relevant for this scenario is located in the folder */examples/Demos/UseCase2*. There is only one *named* DIF `Layer0` connecting `hostA` and `hostB`. The demonstration uses a flat address space with simple strings as addresses. The IPCP at `hostA` has the address `1` and the IPCP at `hostB` has the address `2`. The application on `hostA` has the APN `SourceA` and the destination application on `hostB` has the APN `DestinationB`.

Every application starts with the creation of an application connection to an AP that is part of a DAF. This process contains several events such as AEI creation, flow allocation, management connection establishment, and enrollment that need to proceed before the application connection is created. The application goes through the following steps to enable a communication within the DAF:

1. In `HostA's applicationProcess1`, the connection request to `HostB's applicationProcess1` is induced via an API call. Because `HostA's applicationProcess1` is not a DAF member yet, it must create a management connection to the destination AP first. The `applicationProcess1` of `HostA` spawns a management AEI inside `AEManagement` that is used to handle the management communication between APs in a DAF. The flow to the destination AP is allocated. `HostB's applicationProcess1` spawns the corresponding management AEI when it accepts flow allocation. The connection is successfully allocated, and the CACE between the APs may proceed;
2. `HostA's` management AEI sends an M_CONNECT request carrying the authentication information. The `HostB's` management AEI receives the M_CONNECT, validates the authentication data, and sends the M_CONNECT_R back. The application connection is successfully established upon reception of the positive response (the CACE phase proceeded successfully);
3. The next communication phase is the enrollment. The `application Process1` of `HostA` sends an M_START message containing object(s) related to the enrollment. The `HostB applicationProcess1` replies with an M_START_R. Subsequent exchange of M_CREATE / M_CREATE_R initiated by `HostB's applicationProcess1` (that is a member of the DAF) may proceed. These messages should synchronize essential objects in RIBs, which would allow `HostA's applicationProcess1` to operate as a full-fledged member of the DAF. RINASim has placeholders to leverage this synchronization property of the enrollment;
4. When the necessary objects are created, an M_STOP message is generated by the `HostB` AP. `HostA's applicationProcess1` sends an M_STOP_R indicating that no more messages need to be exchanged. Otherwise, the `HostA applicationProcess1` might send several M_READs to obtain information from the member AP (i.e., `HostB`). Then `HostA's applicationProcess1` is a new member of the DAF.

The mentioned enrollment phase is the enrollment applied on a DAF membership. Enrollment within a DIF follows the same message order and states except for the data in the exchanged objects. In case of a DIF, RIBd processes above-mentioned message directly instead of AEs (because AEs are not part of any DIF).

Once the management connection is established and the enrollment phase is finished, the AP starts to operate as a full-fledged member of a DAF. The simple ping-like application follows these steps:

1. HostA's applicalicationProcess1 spawns a AEMonitor instance (which is a specialized AE offering ping-like behavior). The AEMonitor instance requests the underlying IPCP (i.e., *ipcProcess1*) for the flow with a specified application QoS. The flow is then allocated (refer to Sect. 4.5.1 for additional details);
2. HostB's applicationProcess1 instantiates the AEMonitor in response to the positive allocation of the requested flow. The applicationProcess1 of HostA generates an M_CONNECT carrying the authentication information and HostB's applicationProcess1 sends an M_CONNECT_R with a positive response back. This establishes the application connection (the CACE phase is completed), and the data objects may be transferred over this connection;
3. Generally, AEs support two types of communication: either via the RIB or direct messaging omitting the RIB completely. Our ping-like application uses the direct messaging to keep the demonstration as simple as possible. The HostA applicationProcess1 sends an M_READ message request with a specific object. The request is received by HostB's applicationProcess1 (by an instance of AEMonitor), which replies with an M_READ_R;
4. The HostA applicationProcess1 receives the M_READ_R. The application then displays a result to the user (e.g., a console text). Another round of M_READ/M_READ_R exchange follows later. The object inside these messages contains just an integer value that is incremented by each host.

This simple ping-like application shows a direct communication between two APs. The communication in a real ping scenario should be the same.

### *4.5.3 Reliable Data Transfer*

The simulation source code is the same as in the case of Sect. 4.5.2. It is located in the folder */examples/Demos/UseCase2*. In this scenario, we demonstrate reliable data transfer with a detailed description of EFCP events. We use a minimal topology with just two hosts (see Fig. 4.12), each one with a single IPCP for the sake of simplicity. We omit the description of events prior to the EFCP instance creation like the enrollment phase or the flow allocation.

We demonstrate the reliable data transfer using a ping-like application between hostA and hostB (see Fig. 4.12). The transfer over the link between hostA and hostB is delayed by 0.4 s in both directions. As described before, the ping-like application exchanges ping request and response (i.e., M_READ and M_READ_R messages containing appropriate object referencing time). Both messages are sent inside the DataTransferPDU and are acknowledged by the ControlPDU type. The EFCP connection is configured with the default policy set. Both flow control (i.e., speed administration) and retransmission control (i.e., acknowledgments and retransmits guarantee that nothing gets lost) are active.

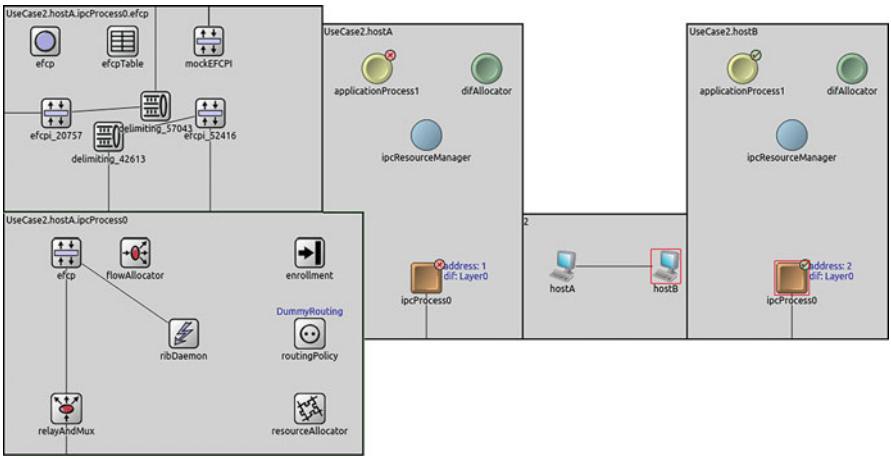



**Fig. 4.12** Reliable data transfer illustration of two directly connected hosts

At time $t = 5\,\text{s}$, the `applicationProcess1` on `hostA` initiates the communication. The data transfer starts after the successful enrollment phase and the flow allocation at $t = 14.8\,\text{s}$. The reliable data transfer includes the following steps:

1. The module `delimiting_57043` encapsulates the incoming SDU (i.e., *ping request* in form of an M_READ) into a `PDUData`, fragments it if necessary, and forwards it to the EFCP instance `efcpi_20757` as a `UserDataField`. In the EFCP instance it is handled by the `dtp` module. The `dtp` module initializes `dtpState` upon the reception of `UserDataField`.

   The `UserDataField` is encapsulated in the `DataTransferPDU` with a DT-PDU header. The `SequenceNumber` is set to the value produced by `initialSeqNumPolicy` which generates a random number by default. For simplicity, we use 1 as the initial sequence number in our example. The first PDU is labeled with the DRF indicating the start of a new connection.

   After the complete PDU has been generated, the control is handed over to the `DTCP` which tries to send it. With the window-based flow control active (with the maximum windows size of 10 PDUs), `dtcp` checks if the Sender's Right Window Edge (SRWE) is bigger than the sequence number of the `DT-PDU` it is trying to send. The initial value of the SRWE is set to the sum of `nextSeqNum` and `intialSenderCredit`.

   All DT-PDUs that satisfy the flow control check are put in the `postablePDUs` queue. The DTP then iterates over the set of `postablePDUs` and sends them to the `RMT`. Simultaneously, the DTP also puts their copies into the `retransmissionQ` queue and starts the `RetransmissionTimer`. The timer is initiated with a default RTT value of the first PDU in the connection. The RTT value is adjusted during the connection lifetime using `ControlPDUs`

and the RTTEstimatorPolicy. After passing the PDU to the RMT, the SenderInactivityTimer is started.

2. At time $t = 15.2$ s, the DataTransferPDU is delivered to the EFCP instance in the ipcProcess0 on hostB. Since the PDU has the DRF set, it indicates the (re)start of a (new) connection. In the case of a restart, DTP dequeues as many PDUs from reassemblyQ to the delimiting module as possible and deletes the rest. Subsequently, DTP sets the Receiver's Left Window Edge (RLWE) to the seqNum of the incoming PDU (in our example to 1) and adds the PDU to the reassemblyQ. Simultaneously, DTP updates the state vector. The SVUpdate initiates RcvrAckPolicy which sends an AckFlowPDU to confirm the DT-PDU (the one with seqNum = 1). The ACK received by hostA updates the SRWE. Then, the UserDataField is passed on hostB to the delimiting module and the rcvrInactivityTimer is started.
3. The AP on hostB sends back a *ping response* (i.e., an M_READ_R) message. The same set of steps (see step 1) related to the first DT-PDU applies to hostB's EFCPI as in case of hostA's corresponding EFCPI: EFCP generates the initial sequence number, sets the DRF, and starts the senderInactivityTimer.
4. At time $t = 15.6$ s, the EFCP on hostA receives the AckFlowPDU. This triggers both the RTTEstimatorPolicy (subsequently updating the RTT value) and the SenderAckPolicy. The SenderAckPolicy removes a subset of the DT-PDUs from the retransmissionQ with the sequence numbers up to the value in the AckFlowPDU.

   At the same time, hostA also processes the DT-PDU with the ping response message. The same steps as in step 2 are repeated on hostA.
5. At time $t = 15.6$ s, hostA sends a new *ping request*. The EFCP fills the header with sequenceNumber= 2, puts the PDU in retransmissioQ with associated RetransmissionTimer, and restarts the SenderInactiveTimer.
6. At time $t = 16$ s, hostB receives an ACK of the last *ping response*. As a result, the DT-PDU with sequenceNumber= 1 is removed from retransmissionQ.
7. Simultaneously to step 6 at the time $t = 16$ s, the ipcProcess0 on hostB receives a DT-PDU with the second *ping request*, and sends back an ACK. The process repeats itself afterwards.

The summary of hostA's state vector is in Table 4.1a and hostB's state vector is in Table 4.1b. Both tables include Receiver's Right Window Edge (RRWE) and Sender's Left Window Edge (SLWE), which are counterparts of RLWE and SRWE. The values indicate the state after the corresponding event.

Both state vectors continue to evolve in a similar fashion throughout the rest of the communication. After the last DataTransferPDU is sent, the inactivity timers are restarted one last time. After they expire, the state-vectors are deallocated.

The events are mirrored on both hosts, because the application on top is sending data in both directions. The EFCP instance on both sides uses its receiver and sender state vector and finite state machine.

**Table 4.1** Values of the hostA and the hostB EFCP

(a)

| Variable | Value | | | |
|---|---|---|---|---|
| Event | #1 | #4 | #5 | #6 |
| Time | 14.8 s | 15.6 s | 15.6 s | 16 s |
| Next SeqNum | 2 | 2 | 2 | 3 |
| RLWE | 0 | 0 | 1 | 1 |
| RRWE | $\infty$ | $\infty$ | 11 | 11 |
| SLWE | 0 | 2 | 2 | 2 |
| SRWE | 10 | 11 | 11 | 11 |
| RetransmissionQ | 1 | $\varnothing$ | $\varnothing$ | 2 |

(b)

| Variable | Value | | | |
|---|---|---|---|---|
| Event | #2 | #3 | #7 | #8 |
| Time | 15.2 s | 15.2 s | 16 s | 16 s |
| Next SeqNum | 1 | 2 | 2 | 2 |
| RLWE | 1 | 1 | 1 | 2 |
| RRWE | 11 | 11 | 11 | 12 |
| SLWE | 0 | 2 | 2 | 2 |
| SRWE | 10 | 10 | 11 | 11 |
| RetransmissionQ | $\varnothing$ | 1 | $\varnothing$ | $\varnothing$ |

## 4.6 Conclusion

In this chapter, we described the core RINA principles. We summarized the RINA theory in the text that lacks any forward references. We know how hard the "mental shift" from TCP/IP concepts towards RINA is, thus we urge a reader to follow the citations in order to learn more. We described different kinds of high-level RINA nodes including hosts, interior routers, and border routers. Subsequently, we dived into the low-level RINA components that are being used by the DIF and DAF.

RINASim, in its current state, represents an entirely working implementation of the simulation environment for RINA. The simulator contains all mechanisms of RINA according to the current specification. The RINASim philosophy benefits from the clever OMNeT++ module interfacing, which enables flexible changes of the employed policies. Furthermore, the chapter also contains a detailed illustration of the RINA principles using three RINASim scenarios. The demonstration descriptions show the impact of recursion and help others to understand the enrollment and the flow (de)allocation procedures in practice. The demonstration setup may be employed as a template when creating new scenarios.

The main motivation behind RINASim's development is that it should allow:

- researchers to prototype and test new policies and mechanisms in a native and fully-compliant RINA environment—*scientific goal*;
- others to visualize and understand the RINA principles—*educational goal*.

RINASim (first published in [26]) started as an FP7 EU PRISTINE deliverable and continued beyond the end of the project. However, RINASim is just one independent implementation of the RINA concepts (cf. Appendix). RINASim is an open environment that can be extended with experimental features. The simulator helps to evaluate new features and to compare them with existing methods.

Our future work involves the following improvements for RINASim:

- Real-life layer-2 simulation modules: RINASim's core lacks any real-life 0-DIF medium implementations (e.g., Ethernet, Long Term Evolution (LTE), serial). This severely impacts some simulation results and collected statistics because

the medium's main properties (delay and bandwidth) are fixed and do not respect the usual processing.
- Topology generator: preparing a scenario and the accompanied configuration is a cumbersome process even with all the help of OMNeT++'s Integrated Development Environment (IDE). We want to create a dedicated web-application that would allow to generate even complex RINASim topologies using a few mouse clicks and drag-and-drop operations.
- Hardware-in-the-Loop (HIL) simulation: by the time that this book is publicized, RINA will already pass the ISO standardization process. Since multiple projects obey the RINA specification, we would like to try connect our simulation modules with real operating system implementations.

We encourage anyone interested in RINA to step in and contribute to the project!

**Acknowledgements** This work was supported by the Brno University of Technology organization and by the research grant FIT-S-17-3964.

RINASim is a joint international project, which would not be possible without the huge support and contribution of the following individuals:

- Tomáš Hykel—Brno University of Technnology (Czech Republic);
- Sergio Leon Gaixas—Universitat Politècnica de Catalunya (Spain);
- Ehsan Elahi—TSSG in the Waterford Institute of Technology (Ireland);
- Kewin Rausch—Fondazione Bruno Kessler (Italy);
- Peyman Teymoori—University of Oslo (Norway);
- Kleber Leal—Universidade Federal de Pernambuco (Brazil).

## Appendix

### *CDAP Messages*

Table 4.2 collects all CDAP messages for further reference.

### *RINA Adoption*

Pouzin Society [21] is the formal body in charge of maintaining the RINA specifications. Any individual or organization can become a member and participate in related research and development. RINA is successfully targeted as an alternative to the traditional TCP/IP stack in the frame of multiple EU projects. Here is a list of projects and their main interests concerning RINA:

- IRATI [18]: IRATI advances the state of the art of RINA towards an architecture reference model and specifications that are closer to enable implementations deployable in production scenarios. The design and implementation of the IRATI

**Table 4.2** CDAP messages

| Message opcode | Purpose |
|---|---|
| M_CONNECT | Initiate a connection from a source to a destination application |
| M_CONNECT_R | Response to M_CONNECT, carries connection information or an error indication |
| M_RELEASE | Orderly close of a connection |
| M_RELEASE_R | Response to M_RELEASE, carries final resolution of the close operation |
| M_CREATE | Create an application object |
| M_CREATE_R | Response to M_CREATE, carries result of the create request, including identification of the created object |
| M_DELETE | Delete a specified application object |
| M_DELETE_R | Response to M_DELETE, carries result of the deletion attempt |
| M_READ | Read the value of a specified application object |
| M_READ_R | Response to M_READ, carries parts or all of the object values, or an error indication |
| M_CANCELREAD | Cancel a prior read issued using M_READ for which a value has not been completely returned |
| M_CANCELREAD_R | Response to M_CANCELREAD, indicates outcome of the cancellation |
| M_WRITE | Write a specified value to a specified application object |
| M_WRITE_R | Response to M_WRITE, carries the result of the write operation |
| M_START | Start the operation of a specified application object, used when the object has operational and non-operational states |
| M_START_R | Response to M_START, indicates the result of the operation |
| M_STOP | Stop the operation of a specified application object, used when the object has operational and non-operational states |
| M_STOP_R | Response to M_STOP, indicates the result of the operation |

prototype on top of Ethernet permits further evaluation and deployment of RINA in real computer networks;

- IRINA [16]: IRINA aims to compare RINA against TCP/IP in a lab environment using the IRATI prototype. Moreover, it proposes use-cases, where RINA is a better option for big national research and educational networks;
- PRISTINE [24]: PRISTINE investigates the programmability of the RINA architecture, in particular its separation of mechanisms and policies to achieve more flexible behavior of network components;
- OCARINA [19]: OCARINA's objectives are to research and develop new congestion control and routing mechanisms in RINA to show that RINA is a better solution for the Internet than TCP/IP in terms of performance;
- ARCFIRE [1]: ARCFIRE's main goal is to demonstrate the RINA benefits at a large scale by running real life experiments on the FIRE+ experimental facilities, for instance the deployment of virtualized infrastructure, the End-to-

End (E2E) service provisioning, or the applicability of Distributed Denial-of-Service (DDoS) attacks in the RINA stack;
- RINAiSense [17]: The RINAiSense project improves the scalability and security of wireless sensor networks while investigating the applicability of RINA to resource-constrained systems.

Moreover, notable implementations are introduced and facts about RINA readiness and deployment status:

- OpenIRATI: open-source programmable implementation of RINA protocols for Linux. Enables Linux device to behave as a RINA-enabled host or software router;
- rlite: open-source implementation of RINA for Linux. The implementation is divided into user-space and kernel-space parts. Kernel-space parts can be loaded as modules in the unmodified Linux kernel;
- Ouroboros: prototype IPC subsystem for Portable Operating System Interface (POSIX) operating systems, that incorporates a fully decentralized packet switched transport network based on a part of the RINA concepts;
- ProtoRINA: a RINA architecture implementation that serves as a teaching tool and also enables the design and development of new protocols and applications;
- RINASim: the official OMNeT++ framework to simulate the RINA architecture.

## References

1. ARCFIRE Consortium: ARCFIRE project—experimenting with RINA on FIRE+. http://ict-arcfire.eu/ (2018)
2. Day, J.: Patterns in Network Architecture: A Return to Fundamentals. Pearson Education, London (2008). https://books.google.cz/books?id=5FDdAAAACAAJ
3. Day, J.: D-Base-2010-007: Delimiting Module. Tech. rep., Pouzin Society (2009)
4. Day, J.: RINA-RFC-2010-002: Notes on the Resource Allocator. Tech. rep., Pouzin Society (2010)
5. Day, J.: D-Base-2011-015: Flow Allocator Specification. Tech. rep., Pouzin Society (2011)
6. Day, J.: D-Base-2011-017: IPC Resource Manager (IRM) Specification. Tech. rep., Pouzin Society (2012)
7. Day, J.: D-Base-2012-010: Relaying and Multiplexing Task Specification. Tech. rep., Pouzin Society (2012)
8. Day, J.: RINARefModelPart3-1 140102: Part 3—Distributed InterProcess Communication, Chapter 1—Fundamental Structure. Tech. rep., Pouzin Society (2012)
9. Day, J.: RINARefModelPart3-2 140102: Part 3—Distributed InterProcess Communication, Chapter 2—DIF Operations. Tech. rep., Pouzin Society (2012)
10. Day, J.: DelimitingGeneral130904: Delimiting Module. Tech. rep., Pouzin Society (2013)
11. Day, J.: RINARefModelPart1-0 130925: Part 1—Basic Concepts of Distributed Systems. Tech. rep., Pouzin Society (2013)
12. Day, J.: RINARefModelPart2-1 130925: Part 2—Distributed Applications, Chapter 1—Basic Concepts of Distributed Applications. Tech. rep., Pouzin Society (2013)
13. Day, J., Marek, M., Tarzan, M., Bergesio, L.: Error and Flow Control Protocol Specification version 6.6. Tech. rep., Pouzin Society (2015)

14. Day, J., Trouva, E.: RINARefModelPart2-2 140102: Part 2—Distributed Applications, Chapter 2—Introduction to Distributed Management Systems. Tech. rep., Pouzin Society (2014)
15. Fletcher, J.G., Watson, R.W.: Mechanisms for a reliable timer-based protocol. Comput. Netw. **2**(4), 271–290 (1978). http://dx.doi.org/10.1016/0376-5075(78)90006-5
16. GÉANT Project: IRINA. https://geant3plus.archive.geant.net/opencall/Optical/Pages/IRINA.aspx (2018)
17. IMEC-Distrinet: The Recursive Internet Architecture as a solution for optimal resource consumption, security and scalability of sensor networks. https://distrinet.cs.kuleuven.be/research/projects/RINAiSense (2018)
18. IRATI Consortium: Investigating RINA as an Alternative to TCP/IP. http://irati.eu/ (2018)
19. OCARINA: OCARINA: Optimizations to Compel Adoption of RINA. http://www.mn.uio.no/ifi/english/research/projects/ocarina/ (2018)
20. OMNeT++: OMNeT++ Simulation Manual. https://www.omnetpp.org/doc/omnetpp/manual/#sec:ned-ref:module-interfaces (2018)
21. Pouzin Society: Pouzin Society | Building a better network. http://pouzinsociety.org/
22. Pouzin Society: RINASim—Recursive InterNetwork Architecture Simulator. https://rinasim.omnetpp.org/
23. Pouzin Society: kvetak/RINA: RINA Simulator. https://github.com/kvetak/RINA (2018)
24. PRISTINE Consortium: PRISTINE | PRISTINE will take a major step forward in the integration of networking and distributed computing. http://ict-pristine.eu/ (2018)
25. Trouva, E., Grasa, E., Day, J., Bunch, S.: Layer discovery in rina networks. In: 2012 IEEE 17th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), pp. 368–372. IEEE, Piscataway (2012)
26. Vesely, V., Marek, M., Hykel, T., Rysavy, O.: Skip this paper-rinasim: your recursive internetwork architecture simulator. arXiv preprint arXiv:1509.03550 (2015)
27. Watson, R.W.: Delta-t protocol specification. UCID 19293, Lawrence Livermore National Laboratory, Livermore (1983)

# Chapter 5
# Cellular-Networks Simulation Using SimuLTE


**Antonio Virdis, Giovanni Nardini, and Giovanni Stea**

## 5.1 Introduction

With the advent of their fourth-generation deployment, known as Long Term Evolution (LTE), cellular networks have undergone a massive increase in popularity, due to their large bandwidth, ubiquitous coverage, and built-in features. More interestingly, they have progressively shifted from single-service to general-purpose access networks, capable of supporting diverse packet-based services simultaneously. Such a paradigm shift has been accompanied by a parallel one in the related research: research discussing physical-layer issues (i.e., waveforms, signal propagation models, receiver algorithms, coding and modulation, etc.) has lately been complemented by research related to Medium Access Control (MAC) layer issues, like resource allocation, admission control, user-side power saving techniques, performance guarantees, as well as the one dealing with the performance of services offered through cellular networks, from mobile web browsing to smart Internet of Things (IoT). Currently, cellular networks are being considered as a viable alternative to other technologies, such as WiFi for traditional mobile applications, IEEE 802.15.4 and Long Range (LoRa) [5] for sensors and smart things, IEEE 802.11p for vehicular networks, and Asymmetric Digital Subscriber Line (ADSL) for home Internet access. The main advantages of an LTE network are: being infrastructure-based and operator-managed, which relieves service users from the need of deploying and managing a (service-specific) infrastructure of their own, or making do with the lack of one. The fact that it operates on licensed spectrum, guaranteeing absence of external interference, and its built-in features for security, mobility management, and terminal-side power saving. Current trends,

A. Virdis · G. Nardini (✉) · G. Stea
University of Pisa, Pisa, Italy
e-mail: antonio.virdis@unipi.it; g.nardini@ing.unipi.it; giovanni.stea@unipi.it



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_5

such as the progression towards fifth-generation (5G) access and the standardization of Multi-access Edge Computing (MEC), all concur to foresee that the role of cellular networks in next-generation communications will increase, incorporating new key features like Device-to-Device (D2D) communications and a tighter coupling between communication and computation resources.

Evaluating the performance of cellular networks poses several challenges. The fact that LTE includes a whole stack of layered protocols, each one having buffers and timers, which interact with other features (such as power saving at both the base station and the terminal), intrinsically defies analytical modeling. On the other hand, building prototypes to do live measurements incurs non-trivial difficulties: despite the fact that several efforts have been done to realize open-source frameworks for LTE experimentation on Commercial Off-The-Shelf (COTS) hardware (e.g., OpenAirInterface [6]), licensed spectrum makes live experimentation difficult, and prototypes can only scale so much as for number of users and base stations, transmission and computing power, and available bandwidth. This leaves simulation as the ideal performance evaluation technique, trading some accuracy for a large gain in experiment manageability. Several simulators of cellular networks have been developed so far. Some are link-level simulators (e.g., [2, 4]). These do an extensive job of modeling the physical layer of a link, with little or no interest in what is above it. They are good for evaluating signal propagation, spectral efficiency, the impact of transmission techniques such as Multiple Input Multiple Output (MIMO) or beamforming, and interference management, but they are generally unsuitable to understand issues related to resource scheduling, protocol interaction, or application-level performance. A different approach is instead that of system-level simulators, where some modeling details (e.g., signal propagation) are abstracted in favor of a stronger focus on the interplay and communication among several complex submodels (e.g., protocol layers). Among system-level LTE simulators, LTE-EPC Network simulAtor (LENA) [1] is part of the Network Simulator 3 (ns-3) framework, and it focuses on the design and testing of Self-Organizing Network (SON) algorithms and solutions.

This chapter presents SimuLTE, a system-level simulator based on OMNeT++ for 4G LTE and Long Term Evolution Advanced (LTE-A) networks. SimuLTE is especially focused on the LTE data plane. Most of the control-plane functions are abstracted by using an oracle, called the *Binder*, which the various modules can query to obtain information which would otherwise require control-plane protocols and elements. SimuLTE includes all the protocols of the LTE stack. It models the effects of the physical layer by computing the received Signal-to-Interference-plus-Noise-Ratio (SINR) at receivers, taking into account all the simultaneous transmitters. This allows one to use it to test, for instance, interference-coordination schemes. It allows both infrastructure-mode communications, where the endpoints of communications are always one User Equipment (UE) and the evolved Node B (eNB), and network-assisted D2D communications, where both endpoints are UEs, and the eNB is in charge of resource scheduling. For the latter, both one-to-one and one-to-many communications are modeled. Within SimuLTE, the LTE functions are confined to a Network Interface Card (NIC), to facilitate the setup of

mixed scenarios, where LTE coexists with other layer-2 technologies (like WiFi), or to use LTE as a layer-2 technology in application scenario simulations (e.g., communicating vehicles).

The rest of this chapter is organized as follows: Sect. 5.2 describes the cellular-networks background, to introduce the reader to the necessary terminology and acronyms. Section 5.3 describes the modeling of an LTE network done within SimuLTE. Section 5.4 presents two tutorials on hot topics in LTE research: interference-coordination techniques at the MAC-level and D2D communications, respectively.

## 5.2 Background

An LTE network is composed of the Evolved Packet Core (EPC) part and the Radio Access Network (RAN) part, as shown in Fig. 5.1. The EPC is an Internet Protocol (IP)-based network that includes entities performing core functionalities for the network operator, such as Mobile Management Entity (MME), Home Subscriber Server (HSS), and Serving Gateway (SGW). The Packet Data Network Gateway (PGW) provides the EPC with the connectivity to the Internet. The RAN is composed of base stations, called eNBs, which are in charge of resource allocation, and UEs, e.g., smartphones or any device capable of connecting to an LTE network. Downlink (DL) transmission occurs from the eNB to the UEs, and Uplink (UL) ones in the opposite direction. As far as data-plane transmissions are concerned, in the DL an eNB receives IP packets from the EPC, processes them through the LTE protocol stack, which includes fragmentation and reassembly, and sends them on the air.

While LTE occupies the layer 2 of the Open Systems Interconnection (OSI) stack, its functions are split among three protocols, namely (from top to bottom) the Packet Data Convergence Protocol (PDCP), the Radio Link Control (RLC), and

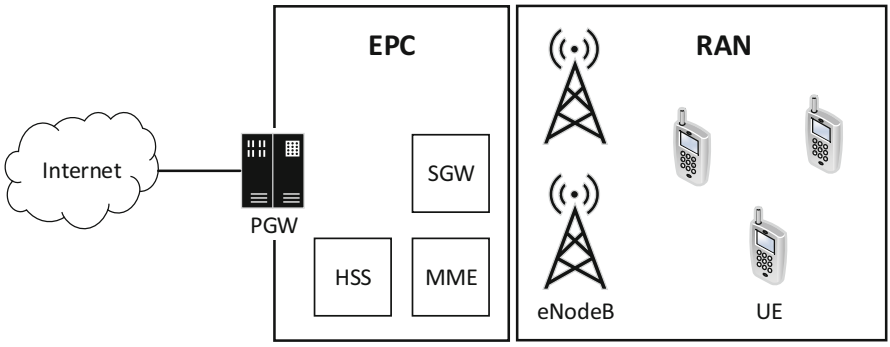



**Fig. 5.1** Architecture of the LTE network

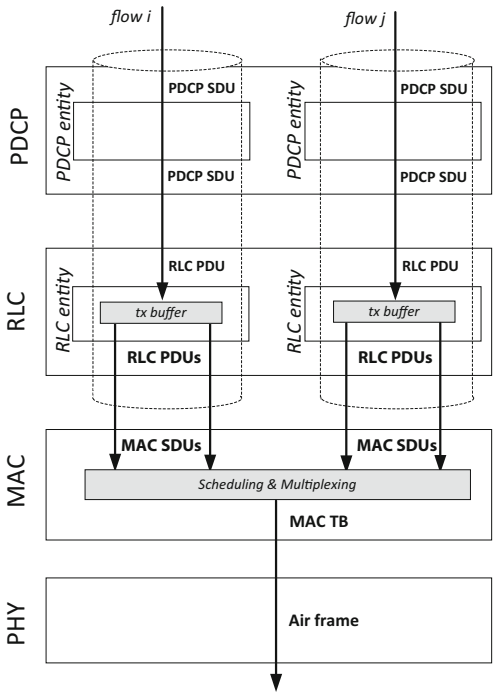



**Fig. 5.2** Top-down traversal of the LTE protocol stack

the MAC. The downstream flow of data within the LTE stack, i.e., the one that an IP packet would undergo on transmission, is shown in Fig. 5.2. At each layer, data comes from the upper layer in the form of Service Data Units (SDUs) and goes to the lower layer as Protocol Data Units (PDUs). A PDCP entity maintains numbering and ciphering information, among others. Each PDCP SDU is assigned a PDCP Sequence Number (SN) and ciphered so that only the peering PDCP entity can decode it. Data from the PDCP is sent to the RLC. It is buffered in the RLC transmission buffer, until sent down in the form of RLC PDUs, upon request from the MAC layer. A flow is allocated to a PDCP and a RLC entity at a node, and its PDCP/RLC entities at the transmitter and receiver are synchronized. The RLC can work in one of three modes: Transparent Mode (TM), Unacknowledged Mode (UM), or Acknowledged Mode (AM). The first one does not perform any operation; hence, a RLC SDU corresponds exactly to a RLC PDU. The second one performs segmentation/concatenation of SDUs on transmission, as well as reassembly, duplicate detection, and reordering of PDUs on reception. The third one adds an Automatic Repeat-reQuest (ARQ) mechanism on top of that to ensure reliable delivery of RLC PDUs.

The MAC layer performs scheduling and multiplexing of RLC PDUs coming from different flows and encapsulates them into MAC Transport Blocks (TBs), which are sent over the physical layer. At every Transmission Time Interval (TTI), which is 1 ms in LTE, the MAC scheduler at the eNB assembles the DL and the

UL subframes. The latter include a fixed number of Resource Blocks (RBs), e.g., 50 in a 10 MHz deployment. In the DL, the MAC scheduler selects which UEs will receive information, how large their TB will be, and which RBs it will occupy. The number of bits carried in each RB depends on the Modulation and Coding Scheme (MCS) used by the eNB for that TB. The MCS is determined by the Channel Quality Indicator (CQI) reported by the UE, which depends on the measured SINR. A higher CQI warrants more bits per RB, hence a higher throughput. The MAC layer includes a Hybrid Automatic Repeat-reQuest (H-ARQ) function for error recovery: a receiving UE acknowledges (via ACKs) (or disacknowledges via NACKs) a MAC TB four TTI after its transmission, and the eNB can schedule a predefined number of retransmissions at any future TTI (asynchronous H-ARQ).

UL scheduling mirrors the DL one: the eNB allocates transmission grants to UEs having backlogged data, specifying which RBs they can use, using which modulation. However, since data is physically stored at the UEs, a signaling method is needed for a UE to signal to the eNB its intention to transmit. UEs can append quantized Buffer Status Reports (BSRs) to their scheduled data, to inform the eNB of their residual backlog. UEs can also signal the presence of backlog using an out-of-band Random Access (RAC) procedure. Simultaneous RAC requests from different UEs may collide, in which case UEs undergo a backoff procedure before attempting another RAC. The eNB responds to a RAC request by scheduling the UE in a future TTI. If unanswered, RAC requests are reiterated. The handshake for UL transmissions is summarized in Fig. 5.3. It consists of up to three parts: a RAC request, usually followed by a (small) grant that the eNB gives to the UE, large enough to contain a BSR, and finally a (larger) grant that the eNB can give to the UE once it knows its backlog.

Unlike the DL ones, H-ARQ processes are synchronous, i.e., retransmissions are scheduled exactly eight TTIs after the previous attempt.

In the latest LTE-A releases, new functions have been introduced to address increasing traffic demand and the requirements of new services (e.g., IoT and vehicular communications). For example, Coordinated MultiPoint (CoMP) transmission and reception and D2D communications are two of the main innovations. CoMP addresses the problem of mitigating the effects of inter-cell interference, since all the eNBs in an LTE network share the same spectrum. In particular, CoMP Coordinated Scheduling (CoMP-CS) and CoMP Joint Transmission (CoMP-JT) techniques aim at improving the performance of cell-edge UEs. This is achieved by allowing neighboring eNBs to exchange coordination information using a common

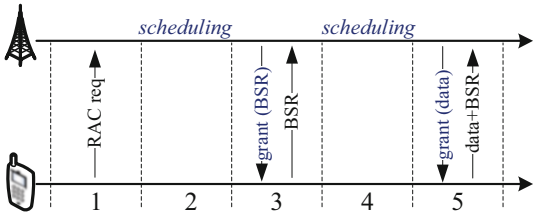



**Fig. 5.3** Handshake for the scheduling of uplink UE traffic

logical interface, called *X2*. On the other hand, D2D allows UEs in proximity to communicate directly, without traversing the conventional infrastructure path through the eNB (i.e., one UL and one DL transmission). This allows the UEs to reduce latency and exploit better link quality. Although the standard focuses on *one-to-many* D2D communication (one D2D transmission can be decoded by all UEs in proximity), *one-to-one* D2D communication between two UEs is also envisaged, especially in the research community. In network-controlled D2D, control functions are still handled by the eNB: considering resource allocation as an example, UEs have to request transmission grants from the eNB, using the same handshake procedure as for standard UL communication. Given the short distance between the endpoints of D2D communications (possibly reducing UEs' transmission power), the eNB can also allocate the same RBs to multiple D2D flows simultaneously if their mutual interference is low, thus enabling *frequency reuse* and serving more traffic with the same spectrum.

## 5.3 Structure of the SimuLTE Simulator

In this section, we describe the general structure of SimuLTE. We first describe the main nodes composing a simulated LTE network in terms of their structure and interconnections. We then move to the core of the architecture, i.e., the NIC card, which implements the actual communication among nodes. Finally, we discuss the available main functionalities, detailing where they are located in the codebase and providing insights on design choices.

Figure 5.4 shows a simplified view of the project's folder structure. The *simulations* and *test* folders contain exemplary simulations together with the respective networks and fingerprint-based tests, which are used for verification purposes. The

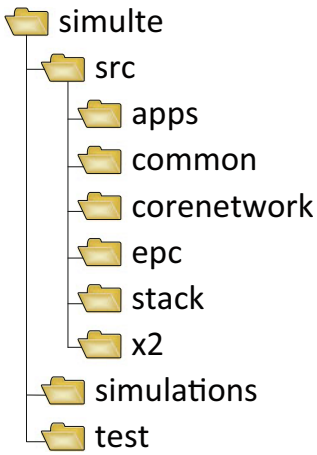



**Fig. 5.4** Simplified representation of the SimuLTE-Project folders structure

*src* folder contains all the source code and is further divided into subfolders, which identify specific portion of the codebase. The *apps* folder contains four traffic models, for Constant Bit Rate (CBR), Voice-over-IP (VoIP), Video on Demand (VoD), and alert traffic. The *common* folder includes utility functions and definitions of LTE-related parameters. The *corenetwork* folder contains the definition of all nodes in the simulator, which are described in detail in the next section. *epc* and *x2* folders contain the definition of the entities involved, respectively, in the EPC and in X2 communications. Finally, the *stack* folder contains the files defining the structure of the LTE NIC card and all its internal layers and modules, as discussed in Sect. 5.3.2.

### 5.3.1 Nodes

As we explained in Sect. 5.2, the two main nodes of a simulated LTE network are eNBs and UEs, which handle all data-plane traffic through the LTE network. SimuLTE adds a third node, namely, the Binder, which acts as the "oracle" of the network, keeping track of communication associations, and which is also used to abstract control-plane operations. Nothing prevents one to implement some of these control procedures through the exchange of control messages in the LTE network, in any case. Finally, one or more nodes can optionally be added to realize EPC-related operations, e.g., for handover management. The general structure of a simulated network is shown in Fig. 5.5.

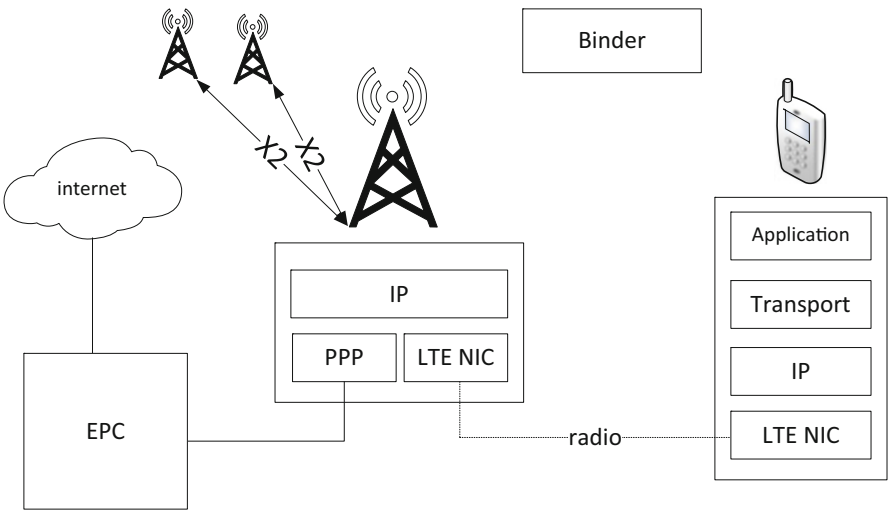



**Fig. 5.5** High-level view of the main simulator nodes

#### 5.3.1.1 Evolved Node B

The eNB is a communication relay for each transmission to/from UEs. It has a Point-to-Point Protocol (PPP) interface towards the EPC, and an LTE NIC card to communicate with UEs. Moreover, there is an array of PPP interfaces that implement the X2 connections with neighboring eNBs. The architecture of the X2 interface will be discussed in more detail in Sect. 5.3.3.2. The eNB does not generate traffic on the data plane, thus does not contain any application-layer module for communication with UEs.

#### 5.3.1.2 User Equipment

The UE node models the behavior of an LTE-based cellphone. Its structure is inspired from INET's `StandardHost`, having multiple Transmission Control Protocol (TCP)/User Datagram Protocol (UDP) applications, UDP and TCP modules implementing the respective transport layer protocols. Each TCP/UDP application is one end of a connection, the other end of which may be located within another UE or anywhere else in the simulated network. SimuLTE provides models of real-life applications (e.g., VoIP and VoD), but it can include any TCP/UDP-based OMNeT++ application. The IP module is taken from the INET package as well. Finally, an LTE NIC card is used for communication with the eNB or with other UEs in case of D2D communications.

#### 5.3.1.3 Binder

The Binder is a simple module that stores information about every LTE-related node within the system. Its main purpose is to cover all the operations that are not modeled as a message exchange, either to simplify the model or to speed up the simulation process. This includes references to nodes, communication associations among eNBs and UEs, peering associations for D2D pairs, etc.

Each module registers to the Binder during its `initialize()` function. Listing 5.1 shows an exemplary registration process performed by the `IP2LTE` submodule (which is explained in Sect. 5.3.2.5) residing in either a UE or an eNB. In both cases, the module hierarchy is navigated to obtain a pointer to the module. The `getBinder()` function is a utility function that returns a pointer to the Binder. The `registerNode()` function is a member function of the Binder class. It stores the OMNeT++ Identifier (ID) of the module and associates it with its node type, its serving eNB specified via the Network Topology Description (NED) language in case of UE registration, and a unique `macNodeId`. The latter value will be used to obtain information on the node during the simulation.

**Listing 5.1** Example of a node registration to the Binder within the `IP2LTE` module

```
if (nodeType_ == UE)
{
    cModule *ue = getParentModule()->getParentModule();
    getBinder()->registerNode(ue, nodeType_, ue->par("masterId"));
}
else if (nodeType_ == ENODEB)
{
    cModule *enodeb = getParentModule()->getParentModule();
    MacNodeId cellId = getBinder()->registerNode(enodeb, nodeType_);
}
```

### 5.3.2 LTE NIC

The LTE NIC module implements the LTE stack within eNBs and UEs. Figure 5.6 shows its internal structure and connections with other modules, namely, one between the UE and the eNB through an air channel and one with an IP module. The LTE NIC module is built by extending INET's `IWirelessNic` interface, to ease its deployment into INET nodes. This is one of the key points that allows the creation of complex scenarios, e.g., where LTE NICs are included in cars (e.g., Chap. 11) or used to build hybrid connectivity scenarios (e.g., Chap. 13), where nodes are equipped with both WiFi and LTE interfaces. With reference to Fig. 5.6, each of the NIC submodules models a corresponding part of the LTE protocol stack. Following the OMNeT++ paradigm, data-plane communications between the different layers of the protocol stack take place only via message exchange, hence ensuring a tight control over module interactions. The NIC structure is the same for both UE and eNB, the only exception being the `FeedbackGenerator`, only implemented in the UE. It is responsible for creating channel feedback that is then managed by the Physical Layer (PHY) module.

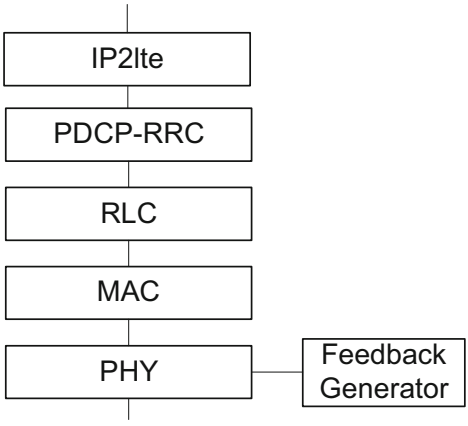



**Fig. 5.6** Internal structure of the LTE NIC module

Each layer is implemented as a base module, specifying interfaces and main parameters. Base modules are then inherited to define specific versions for eNB and UEs, with additional parameters and gates. The same approach is followed for the implementation of the module behavior via C++ classes. In the following subsections, we describe each submodule and its functionalities.

#### 5.3.2.1 PHY

The PHY module resides at the bottom of the LTE protocol stack and implements physical layer related functions, such as channel-feedback reporting and computation, simulation of the air channel, and data transmission and reception. It also stores the node's PHY parameters, like the transmission power and the antenna profile (i.e., whether transmissions are omni-directional or anisotropic). This control over transmission parameters allows one to define so-called heterogeneous scenarios, composed of macro-, micro-, and pico-eNBs, each one having its one radiation profiles.

PHY modules at both the eNB and the UE are associated with a module that models the behavior of the physical channel as perceived by the node itself. Channel modeling is implemented in a hierarchical manner: first, a base module called `LteChannelModel` defines two main functions `getSINR()` and `error()`. The first one, `getSINR()`, should compute the SINR of a given transmission. The second one, `error()`, should check if a packet has been corrupted during transmission. In the following, we describe the implementation of this module, which is available in the current implementation of SimuLTE. However, one can easily obtain its own version of the channel model by implementing the above interface.

Part of the tasks related to physical-layer procedures on the UE side is handled by a module called `FeedbackGenerator`. The latter measures the status of the channel and generates feedback, which is then sent to the eNB in the form of CQIs. The feedback-generation process can be configured to work aperiodically, i.e., on demand from the eNB, or periodically, with a configurable period. The feedback-generation process models channel measurements using the functions provided by the `LteChannelModel` module, in particular the `getSINR()` function. This function returns a vector of SINR values, one for each RBs available in the system bandwidth, which are then processed to generate CQIs. These can be either *wideband*, one value for the whole bandwidth, or *per-band*. In the latter case, the user can configure the number of RBs that compose a band.

As discussed above, the physical LTE channels, such as the Physical Uplink Control Channel (PUCCH), the Physical Downlink Control Channel (PDCCH), and the Physical Random Access Channel (PRACH), are not modeled down to the level of Orthogonal Frequency Division Multiplex (OFDM) symbols. We implemented the functions associated to said channels either via control messages sent between the eNB and UE nodes or via function calls through the Binder. However, any limitation or constraints imposed to the system by those channels (e.g., max. number

of UEs that can be scheduled simultaneously on the PDCCH) can be simulated by imposing constraints on the simulated message flow. Our modeling choice is to limit both memory and processor usage while retaining a good level of simulation accuracy.

As far as data flow is concerned, in the downstream MAC-PDUs received from the MAC layer are encapsulated in packets of the type `LteAirFrame`. Packets are marked with a type (i.e., data or control), a set of transmission parameters (e.g., transmission power and position, number of used RBs, etc.), and are sent to the destination module using the `sendUnicast()` function. In the upstream, instead, a received `LteAirFrame` is checked for its type and processed consequently: control packets are assumed to be correctly received and are forwarded to the upper layer. Data packets are instead checked for correct reception using the `error()` function of `LteChannelModel`, then decapsulated and sent to the upper layer. Note that while decapsulated packets are always forwarded upstream, they are marked with the result of the `error()` function, which will be then evaluated at the upper layers.

SimuLTE provides an implementation of the `LteChannelModel` base module, which is called `LteRealisticChannelModel`. In this implementation, each value of the SINR is computed as:

$$\text{SINR} = P_{RX}^{\text{eNB}} \Big/ \left(\sum_i P_{RX}^i + N\right), \tag{5.1}$$

where $P_{RX}^{\text{eNB}}$ is the power received from the serving eNB, $P_{RX}^i$ is the power received from the $i$th interfering eNB, and $N$ is the Gaussian noise. Furthermore, $P_{RX}$ is computed as $P_{RX} = P_{TX} - P_{loss} - F - S$, where $P_{TX}$ is the transmission power, $P_{loss}$ is the path loss due to the eNB-UE distance, which also depends on the frequency where the considered RB lies, and $F$ and $S$ are the attenuation due to fast and slow fading, respectively [3]. Computing the SINR on each RB allows one to take into account interference on a per-RB basis, e.g., taking into account the transmissions from neighboring eNBs. This can be used to evaluate interference-coordination mechanism, notably the CoMP.

The `error()` function, instead, evaluates the correct `LteAirFrame` reception by analyzing the CQI used at transmission and the current channel status. These two values are used together with a set of realistic Block Error Rate (BLER) curves to obtain an error probability $X \in [0, 1]$. Finally, a uniform random variable $P_{err}$ is sampled and the packet is assumed to be corrupted if $X < P_{err}$.

As stated above, custom models of the channel can be created and used within SimuLTE. Two main approaches are possible in this respect:

1. Create a new module that extends the `LteChannelModel` base module, i.e., redefining the `getSINR()` and `error()` functions.
2. Create a new module that inherits from `LteRealisticChannelModel`, redefining the functions for path loss, fading, and shadowing computation.

#### 5.3.2.2 MAC

Most of the intelligence of LTE nodes is implemented in the MAC module. The main tasks of this module comprise buffering packets coming from lower layers (PHY) and requesting data for transmission from upper ones (RLC), encapsulating MAC-SDUs into MAC-PDUs and vice versa, handling and storing channel feedback, performing scheduling, and Adaptive Modulation and Coding (AMC). Most of the operations related to the flow of packets are the same at the UE and the eNB. Scheduling and channel-feedback management, instead, are performed differently on the two nodes.

Figure 5.7 shows a high-level view of the layer structure. The main functions of the MAC layer are executed periodically at a 1 ms pace. This behavior is implemented by the `LteMacBase` module, which schedules a self-message each millisecond and handles it via the `handleSelfMessage()` function. The latter is redefined to obtain specialized versions for the eNB and UE, respectively, by the `LteMacEnb` and `LteMacUe` classes. Figure 5.8 shows an overview of the operations performed by the `handleSelfMessage()` function in the two cases. They both start with the decapsulation of the successfully decoded PDU from the H-ARQ buffers in reception. In the upstream, MAC-PDUs coming from the PHY are stored into H-ARQ buffers, where they are then checked for correctness, decapsulated, and then forwarded to the RLC in the form of MAC-SDUs.

On the eNB side, UL connections are scheduled for transmission according to a configurable policy. Scheduling decisions are notified to the UEs via grant messages, i.e., the PDCCH is not simulated. Similarly, DL scheduling is performed by selecting which connection to serve, and how much data to send. For each scheduled connection, MAC-SDUs are then requested to the RLC layer and encapsulated into MAC-PDUs. The latter are finally stored in the H-ARQ buffers and forwarded to the PHY for transmission. The structure of H-ARQ buffers will be explained later on.

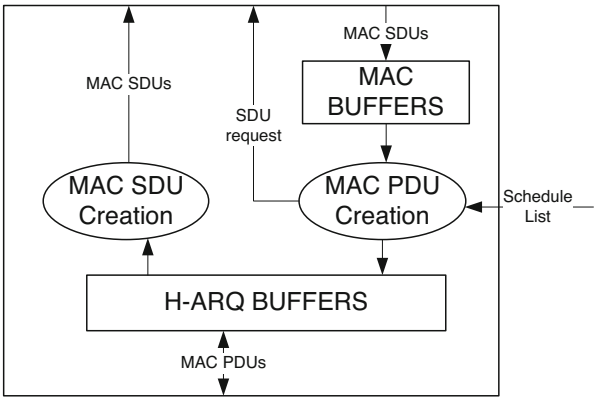



**Fig. 5.7** High-level view of the MAC layer structure

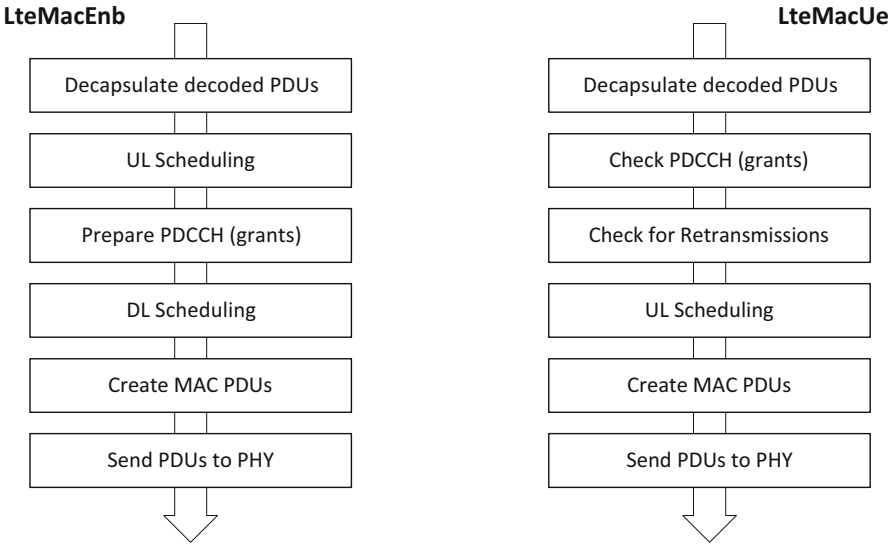



**Fig. 5.8** Main MAC-level operations

On the UE side, instead, each UE checks if any grants have been received and decides which local connection will be able to use the granted resources, if any. If no resources are available, it will perform a resource request via the RAC procedure, which is again implemented through messages generated by the MAC module. Then, the UE will check its H-ARQ buffers to see if any transmission is expected for this TTI[1] and will proceed with the scheduling of new transmissions otherwise.

H-ARQ buffers are used to store MAC-PDUs that are being sent and received. There is one set of such buffers for transmissions and one for receptions, to which we will refer as *TxHbuff* and *RxHbuff*, respectively, and as *Hbuff* to denote both.

A transmitted MAC-PDU is stored in the *TxHbuff* until it is received correctly or the maximum number of retransmissions is reached. Correct reception is notified via H-ARQ feedback messages. A received MAC-PDU is instead stored in the *Rxbuff* until its decoding process has been completed. On the eNB side, H-ARQ buffers store MAC-PDU information for each H-ARQ process, for each connected UE in both downlink and uplink. In Fig. 5.9 we show the general structure of a *Hbuff*. The latter contains $K$ buffers, one for each active connection to another node. An eNB has thus as many buffers as connected UEs in each direction, whereas a UE has up to one per direction, as it communicates directly only with its serving eNB. This architecture is slightly modified in case of D2D communications, where a UE has one additional buffer for each peering UE.

[1] Please note that UL transmissions are synchronous.

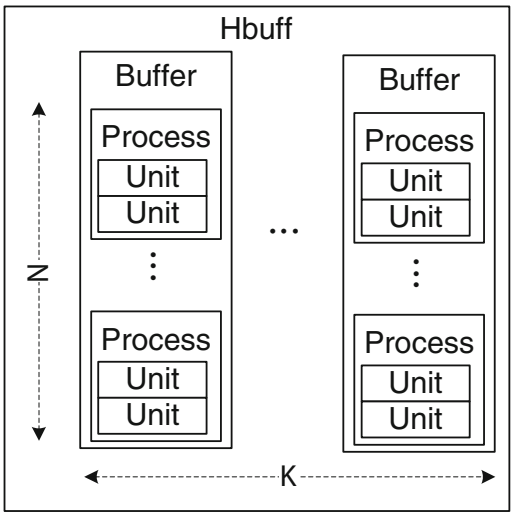



**Fig. 5.9** Internal structure of H-ARQ buffers

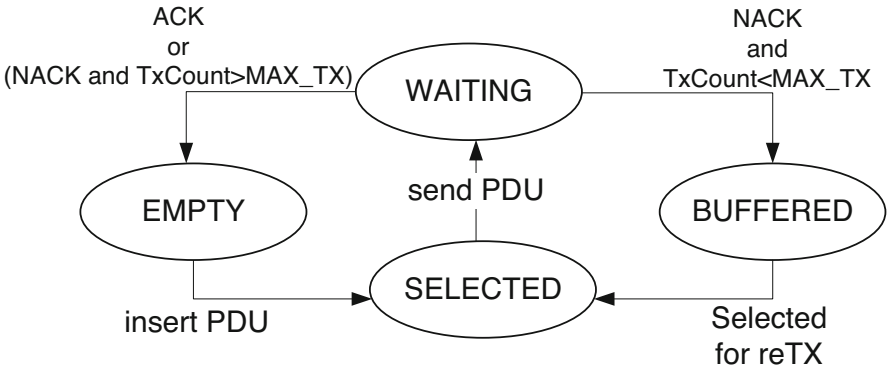



**Fig. 5.10** High-level view of H-ARQ operations

Each buffer is composed of $N$ processes (usually $N = 8$), one per H-ARQ process, and each process contains two *units*, to support single-user MIMO. Units are the actual data structure containing the information related to a transmitting/receiving MAC-PDU. The way unit status is stored depends on the feedback management procedure, resulting in different data structures for *Tx*- and *Rx-Hbuffs*. Figure 5.10 shows the finite-state automaton for transmissions (the one for receptions is similar, mutatis mutandis). Acknowledgment (ACK) and Negative-Acknowledgment (NACK) are the reception of the corresponding H-ARQ feedback. $TxCount$ is the transmission counter, which is increased in the `SELECTED` state and then reset in the `EMPTY` one. $maxTx$ is the maximum number of transmissions before a PDU is discarded.

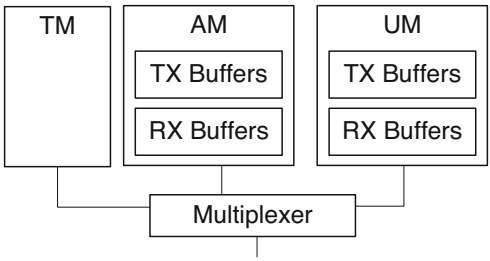



**Fig. 5.11** High-level view of the architecture for the three RLC modes

#### 5.3.2.3 RLC

This module implements RLC operations, which are identical for the eNB and the UE. It performs multiplexing and demultiplexing of MAC-SDUs to/from the MAC layer, implements the three RLC modes TM, UM, and AM as defined in 3GPP-TS 36.322, and forwards packets from/to the PDCP-Radio Resource Control (RRC) to/from the proper RLC mode entity. Figure 5.11 shows the general structure of the RLC module. As we can see, there is one different gate for each RLC mode, which are connected towards the PDCP-RRC one. The TM submodule forwards packets transparently and has no buffer, whereas AM and UM have their own set of transmission and reception buffers, one for each connection associated to the according RLC mode.

#### 5.3.2.4 PDCP-RRC

The PDCP-RRC module models operations of the highest layer of the LTE protocol stack. It receives data from the `IP2LTE` module (in the downstream direction) and from the RLC one (in the upstream). In the first case, the PDCP-RRC performs a RObust Header Compression (ROHC) of the received packet and assigns it a Logical Connection Identifier (LCID). The latter uniquely identifies the connection to which the packet belongs. It is obtained from the 4-tuple composed of <*sourceIPAddr, destIPAddr, sourcePort, destPort*>. The packet is then encapsulated in a PDCP-PDU and forwarded to the proper RLC port, according to the selected RLC mode. In the upstream, a packet coming from the RLC is first decapsulated, then its header is decompressed, and the resulting PDCP SDU is finally sent to the upper layer.

#### 5.3.2.5 IP2LTE

The `IP2LTE` module acts as an interface between the network layer (i.e., IP) and the LTE NIC. In the downstream, it receives layer-3 datagrams and extracts both source/destination IP addresses and port numbers. The latter are written into a `ControlInfo` object that is attached to the message before it is sent to the lower layers, as shown in Listing 5.2. This allows the PDCP-RRC module to obtain

the above 4-tuple without inspecting the packet and to associate a LCID to the flow, as explained in the previous section. In the upstream, `IP2LTE` only forwards the message from PDCP-RRC to the network layer without further processing. Moreover, the `IP2LTE` is responsible for registering the LTE NIC to the Binder and to the interface table of the network layer during the initialization.

**Listing 5.2** Creation of the `ControlInfo` object

```
1 FlowControlInfo *controlInfo = new FlowControlInfo();
2 controlInfo->setSrcAddr(srcAddr.getInt());
3 controlInfo->setDstAddr(destAddr.getInt());
4 controlInfo->setSrcPort(srcPort);
5 controlInfo->setDstPort(dstPort);
6 [...]
7 datagram->setControlInfo(controlInfo);
8 send(datagram,stackGateOut_);
```

### *5.3.3 Main Functions*

We now discuss the scheduling, inter-eNB, and D2D-communication operations.

#### 5.3.3.1 Scheduling

As discussed in Sect. 5.2, resource scheduling is the process of deciding how to allocate RBs to UEs, in both the DL and UL subframes. This process is realized at the MAC layer and performed by the eNBs on each TTI. In SimuLTE, scheduling operations are implemented at the eNB by the `LteSchedulerEnb` class, which defines all the operations that are common to the DL and UL, such as data structure initialization, allocation management via the `Allocator`, and statistics collection. The `Allocator` is a C++ class that keeps the information about the RBs' occupation. The core of this class is the `schedule()` function, whose main operations are shown in Fig. 5.12. These are common to both the DL and UL scheduling. First, the data structures containing the allocation decisions of the previous TTI are cleared, and the per-UE modulation and coding information is updated using the most recent channel information. Then, the actual scheduling is performed, processing RAC requests (UL only), retransmissions, and first transmissions. The way these two last operations are performed defines the scheduling policy, i.e., how contention among UEs is managed. Finally, statistics on the allocation (e.g., the number of allocated RBs) are computed and emitted to the simulation environment.

The scheduling hierarchy is shown in Fig. 5.13 (left part of the figure). The `LteSchedulerEnbUl` and `LteSchedulerEnbDl` classes extend the base class `LteSchedulerEnb` by implementing the `rtxSchedule()` method, which handles RAC requests and manages retransmissions. Scheduling policies are instead implemented by extending the `LteScheduler` class. The latter's

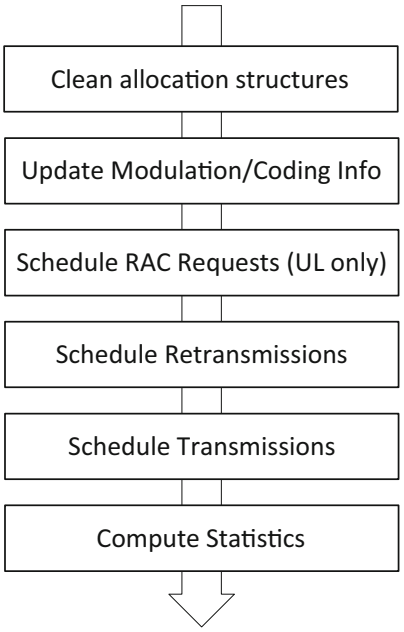



**Fig. 5.12** Depiction of the main scheduling operations

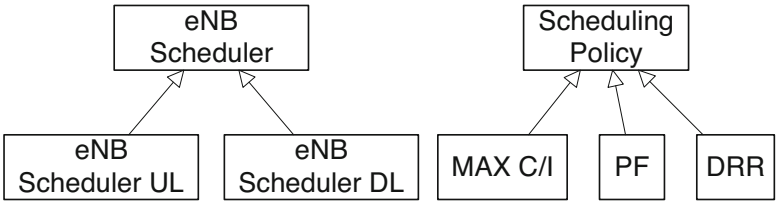



**Fig. 5.13** High-level representation of the scheduling hierarchy

main function, called `prepareSchedule()`, is responsible for applying the scheduling policy on backlogged connections and building a schedule list that associates each connection to the number of allocated RBs. To build the schedule list, the scheduling policy examines backlogged connections one at a time and polls the `Allocator`, via the `requestGrant()` member function, to check whether there are RBs available for the given connection. The latter obtains the amount of bytes that can be transmitted in a single RB for that connection from the AMC module. Finally, the schedule list is passed to the MAC layer, which enforces it by fetching the data from the connections' RLC buffers and constructing the MAC-PDUs.

Three well-known scheduling policies are included in the current release, namely, Maximum Carrier-over-Interference (MaxC/I), Proportional Fair (PF), and Deficit Round Robin (DRR), as depicted in Fig. 5.13 (right part of the figure). The scheduling policy can be modified by changing the `schedulingDisciplineDl` and `schedulingDisciplineUl` parameters of the MAC layer, as follows:

```
1 *.eNodeB.lteNic.mac.schedulingDisciplineDl = "MAXCI"
2 *.eNodeB.lteNic.mac.schedulingDisciplineUl = "MAXCI"
```

As far as RBs are concerned, SimuLTE allows one to group RBs into logical bands, i.e., logical groups of RB that are considered as the minimum scheduling unit by the scheduler. Let us consider the following exemplary configuration:

```
1 **.deployer.numRbDl = 20
2 **.deployer.numRbUl = 20
3 **.deployer.numBands = 20
```

The first two lines define the number of available RBs in the DL and UL subframes, respectively. The third one defines the total number of logical bands among which the RBs are divided into. In this case, we will end up with 20 logical bands with 1 RB each, i.e., one RB for each band. The scheduling policy will work on bands rather than the RBs, different mappings can hence be used to modify the way the scheduler accesses resources.

Moreover, the scheduling policy can ask the `Allocator` to restrict the set of RBs that can possibly be allocated to a connection, by using the `BandLimit` concept. The latter is a data structure stored at the MAC layer of every eNB that specifies, for each UE and RB, the amount of bytes (if any) that can be allocated to that UEs connections in that RB. This concept can be exploited when enforcing interference-coordination mechanisms, for which we provide an example in Sect. 5.4.1.

#### 5.3.3.2 Inter-eNB Communications

Interactions among eNBs are crucial to support functionalities like handover and interference coordination, which are the subject of several research works. eNBs interact via the X2 interface, which is modeled in SimuLTE. Within the eNB, we model the X2 protocol stack depicted in Fig. 5.14, where the `LteX2App` handles the communication with one peering eNB and runs on top of Stream Control Transmission Protocol (SCTP) as the transport protocol. The task of the `LteX2App` is to pass messages originated from the LTE NIC to the peering eNB and vice versa. The two directions are managed by two different inner modules, namely, `X2AppServer` and `X2AppClient`. If the eNB peers with multiple eNBs, one `LteX2App` is needed for each connection. `LteX2App` is transparent to the kind of messages to be sent on behalf of modules within the LTE protocol stack. Within the latter, `X2User` entities are base modules that can be extended in order to implement

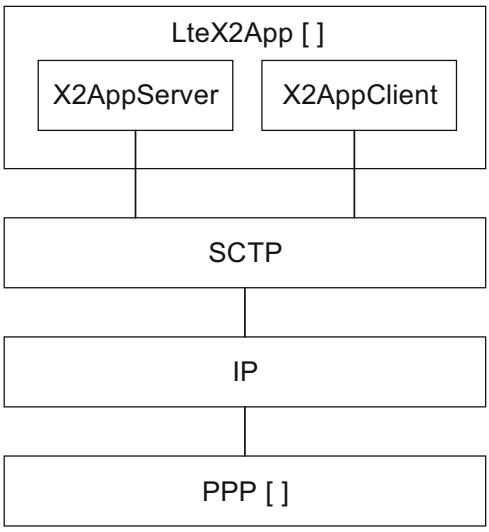



**Fig. 5.14** High-level view of the X2 stack

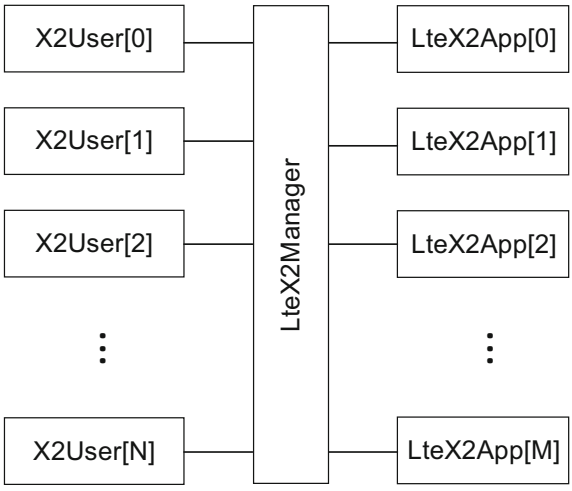



**Fig. 5.15** High-level view of the X2 manager

a functionality that exploits the X2 to perform its task. `LteX2App` and `X2User` modules are transparent to each other and the interface between them is provided by the `LteX2Manager`, as shown in Fig. 5.15. Listing 5.3 shows the code of its `handleMessage()` function, which calls different handlers based on the origin of the message.

**Listing 5.3** The role of the `LteX2Manager`

```
1  void LteX2Manager:handleMessage(cMessage *msg)
2  {
3      cPacket* pkt = check_and_cast<cPacket*>(msg);
4      cGate* incoming = pkt->getArrivalGate();
5      if (strcmp(incoming->getBaseName(), "dataPort") == 0) // from LTE stack
6      {
7          EV << "LteX2Manager:handleMessage - Received message from LTE stack" <<
                   endl;
8          fromStack(pkt); // call handler
9      }
10     else  // from X2
11     {
12         int gateIndex = incoming->getIndex();
13         EV << "LteX2Manager:handleMessage - Received message from X2, gate " <<
                   gateIndex << endl;
14         fromX2(pkt); // call handler
15     }
16 }
```

As an example, consider the CoMP-CS function implemented within SimuLTE: in order to mitigate possible inter-cell interference, neighboring eNBs exchange scheduling information via X2 and avoid allocating the same RBs. This is done by the `LteCompManager`, which extends the `X2User` module. Therefore, the `LteCompManager` can interact with the LTE protocol stack via direct method calls and then sends its messages to the `LteX2Manager`, which in turn passes them to the peering eNBs. According to the architecture explained above, if eNB *i* has to send a CoMP message to neighboring eNB *j* and *k*, the `LteCompManager` only needs to send one message to the `LteX2Manager` including the list of destinations. The `LteX2Manager` then identifies the `LteX2App` modules handling the X2 connections to *j* and *k* and passes one copy of the message to each for the transmission over the X2 interface.

#### 5.3.3.3 D2D Operations

SimuLTE supports both *one-to-one* and *one-to-many* D2D communications. In the one-to-one case, a D2D flow consists of a peering connection between two UEs. The Binder keeps a data structure containing the peering relationships between D2D-capable UEs, i.e., which pairs of UEs can communicate directly. For each pair of D2D endpoints, the Binder also stores whether the flow actually uses the direct path or the conventional infrastructure path through the eNB. The communication mode can be either static or selected dynamically by the `D2DModeSelection` module residing within each eNB. On the other hand, there are no peering relationships for one-to-many communications. Messages transmitted by a UE include a `multicastGroupID` field and only UEs enrolled within the multicast group can try to decode the transmission.

Processing of D2D flows is carried out by specialized, D2D-enabled versions of the LTE NIC (both the UE and the eNB sides), which inherit functionalities from the base ones. To this aim, `LteNicUe` and `LteNicEnb` modules are extended by `LteNicUeD2D` and `LteNicEnbD2D`, respectively. The latter, in turn, defines

specialized versions of each layer, e.g., `LteMacUeD2D` and `LteMacEnbD2D` for the MAC layer.

At the PHY layer, `getSINR_D2D()` and `error_D2D()` functions are provided to compute the SINR and check transmission errors of D2D flows. As far as scheduling is concerned, frequency reuse among D2D flows is accomplished by extending the `Allocator` module so that it can associate more than one UE to each allocated RB. Frequency-reuse-enabled scheduling policies can then be implemented on top of this general structure. Moreover, the eNB also needs to know whether to schedule H-ARQ retransmissions for D2D flows, although (N)ACKs are exchanged between UEs without involving the eNB. Thus, we allow the receiving UE to send a copy of the (N)ACKs to the eNB, too. The MAC layer of the eNB uses such information to update a data structure that *mirrors* the status of each *TxHbuffs*. RLC layer does not need additional D2D-related functionalities, whereas the PDCP layer takes care of associating downstream packets to either D2D or infrastructure mode exploiting the Binder.

## 5.4 Tutorials

We now describe two case studies, namely: (1) an analysis of the problem of inter-eNB interference coordination and (2) the mode selection for D2D. We use a tutorial approach in both cases, starting with the problem statement, then guiding the reader to the simulation-setup process, including the network definition and the parameters configuration, and prompting some possible modifications to the code.

### *5.4.1 Tutorial 1: Interference Coordination*

We consider the downlink of a multicell network, where eNBs serve their UEs while sharing frequency resources. In such a scenario, CoMP-CS can be enforced to mitigate inter-cell interference. In this context, eNBs participating in the coordination send their load information to one eNB, chosen as coordinator, through the X2 interface. In turn, the coordinator periodically runs an algorithm to assign non-overlapping sets of usable RBs to each eNB and communicates them which RBs they can/cannot use in the subsequent scheduling operations. Thus, the objective of this tutorial is to show how to configure X2 connections and change the coordination algorithm implemented by the eNBs.

#### 5.4.1.1 Network Definition

We consider the `MultiCell_X2Mesh` network depicted in Fig. 5.16, which is composed of three eNB connected to each other via the X2 interface. In particular,

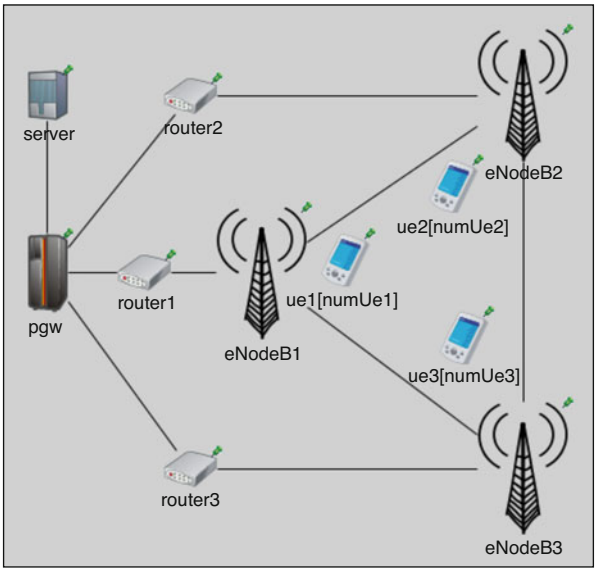



**Fig. 5.16** Network definition for the scenario `MultiCell_X2Mesh`

full-mesh topology is established between eNBs; hence, point-to-point connections are provided between any pair of eNBs. Moreover, each eNB serves a set of UEs, respectively, `ue1[*]`, `ue2[*]`, and `ue3[*]`. Parameters `numUe1`, `numUe2`, and `numUe3` specify the number of UEs served by each eNB.

### 5.4.1.2 Parameters Configuration

With reference to the snippet of the *omnetpp.ini* file in Listing 5.4, we first configure X2-related parameters for this tutorial.

**Listing 5.4** Configuration of X2-related parameters

```
# one x2App per peering eNodeB
*.eNodeB*.numX2Apps = 2

# X2 Server-side ports (x2App[0]=5000, x2App[1]=5001, ...)
*.eNodeB*.x2App[*].server.localPort = 5000 + ancestorIndex(1)

################## Peering configuration ###################
# - eNodeB1%x2ppp0 <--> eNodeB2%x2ppp0
# - eNodeB1%x2ppp1 <--> eNodeB3%x2ppp0
# - eNodeB2%x2ppp1 <--> eNodeB3%x2ppp1
############################################################
*.eNodeB1.x2App[0].client.connectAddress = "eNodeB2%x2ppp0"
*.eNodeB1.x2App[1].client.connectAddress = "eNodeB3%x2ppp0"
*.eNodeB2.x2App[0].client.connectAddress = "eNodeB1%x2ppp0"
*.eNodeB2.x2App[1].client.connectAddress = "eNodeB3%x2ppp1"
*.eNodeB3.x2App[0].client.connectAddress = "eNodeB1%x2ppp1"
*.eNodeB3.x2App[1].client.connectAddress = "eNodeB2%x2ppp1"
```

As explained in Sect. 5.3.3.2, the LteX2App module inside the eNB is responsible for maintaining the connection with one peering eNB. In order to build a full-mesh topology, we need to provide $N - 1$ LteX2App modules within each eNB. To do this, we set the *.numX2Apps parameter to 2 in our example.

Then, we need to configure both the server- and the client-side of each LteX2App. For the server-side, we need to ensure that LteX2Apps within the same eNB are bound to different port numbers; otherwise, local forwarding would not be possible. A simple way to achieve this is to assign incremental port numbers to every LteX2App, as shown at line 5. On the other hand, the client-side of each LteX2App must be configured so that it will connect to the IP address of the peering eNB. This is accomplished by setting the connectAddress parameter using symbolic addresses. Since each eNB has several X2 interfaces, it is necessary to specify the full name of the desired interface using the format <module%interface>. Note that the interface part of the address must specify the gate index, too.

Once the X2 is ready, it can transport every kind of message between eNBs, including CoMP messages. CoMP functionalities are disabled by default; hence, we need to activate them by setting the compEnabled parameter to true. This is exemplified in the *ini* configuration file below.

**Listing 5.5** Configuration of CoMP-related parameters

```
1  ################ CoMP configuration ##################
2  *.eNodeB*.lteNic.compEnabled = true
3
4  # Master configuration
5  *.eNodeB1.lteNic.compManager.compNodeType = "COMP_CLIENT_COORDINATOR"
6  *.eNodeB1.lteNic.compManager.clientList = "2 3"
7
8  # Slaves configuration
9  *.eNodeB*.lteNic.compManager.coordinatorId = 1
10
11 # CoMP algorithm
12 *.eNodeB*.lteNic.compManagerType = "LteCompManagerProportional"
13
14 # Scheduling policy
15 *.eNodeB*.lteNic.mac.schedulingDisciplineDl = "MAXCI_COMP"
```

Since we modeled CoMP-CS algorithms according to the master–slave paradigm, we need to specify the role of every eNB participating in the coordination. Assume that the coordinator's role is co-located with eNodeB1. Then eNodeB1's CoMP manager gets the value COMP_CLIENT_COORDINATOR as its compNodeType. On the other hand, eNodeB2 and eNodeB3 get the COMP_CLIENT default value. Moreover, the coordinator needs to know the IDs of the client eNBs, whereas the latter have to be configured with the ID of the coordinator. Parameter compManagerType allows one to instantiate different user-defined CoMP algorithms. SimuLTE comes with a simple example that we will introduce in the next subsection. Finally, a CoMP-enabled scheduling policy has to be selected for the downlink scheduler.

#### 5.4.1.3 Modifying the Code

Coordinated-scheduling operations are defined within the `LteCompManager` module. According to our model, at every TTI, the eNBs participating in the coordination run a local algorithm that computes the required number of RBs and sends those requests to the coordinator via the X2 interface. Then, at every coordination period, the coordinator runs the coordination algorithm based on the input received by the clients and sends the results back to the clients. `provisionalSchedule()` and `doCoordination()` functions are virtual methods. Thus, one can define its own coordination algorithm, by implementing a new module derived from the `LteCompManagerBase` class, redefining the `provisionalSchedule()` and `doCoordination()` functions.

**Listing 5.6** Modifying the CoMP algorithm

```
1  void LteCompManagerBase:runClientOperations()
2  {
3      EV<<"LteCompManagerBase:runClientOperations - node "<<nodeId_<<endl;
4      provisionalSchedule();
5      X2CompRequestIE* requestIe = buildClientRequest();
6      sendClientRequest(requestIe);
7  }
8
9  void LteCompManagerBase:runCoordinatorOperations()
10 {
11     EV<<"LteCompManagerBase:runCoordinatorOperations - node "<<nodeId_<<endl;
12     doCoordination();
13     // for each client, send the appropriate reply
14     std:vector<X2NodeId>:iterator cit = clientList_.begin();
15     for (; cit != clientList_.end(); ++cit)
16     {
17         X2NodeId clientId = *cit;
18         X2CompReplyIE* replyIe = buildCoordinatorReply(clientId);
19         sendCoordinatorReply(clientId, replyIe);
20     }
21
22     if (nodeType_ == COMP_CLIENT_COORDINATOR)   // local reply
23     {
24         X2CompReplyIE* replyIe = buildCoordinatorReply(nodeId_);
25         sendCoordinatorReply(nodeId_, replyIe);
26     }
27 }
```

SimuLTE provides the exemplary `LteCompManagerProportional` CoMP module, which inherits functionalities from `LteCompManagerBase` and overrides the `doCoordination()` function. According to this policy, the master eNB partitions the number of available RBs among eNBs in a proportional fashion based on their requested RBs. Slave eNBs can then use only the assigned subset of the available RBs when doing their scheduling. Listing 5.7 shows that the set of usable bands in the next TTIs is received within the master's reply. This information is then used to pilot the `BandLimit` data structure mentioned in Sect. 5.3.3.1.

**Listing 5.7** Handling of CoMP master's reply

```
void LteCompManagerProportional:handleCoordinatorReply(X2CompMsg* compMsg)
{
    while (compMsg->hasIe())
    {
        X2InformationElement* ie = compMsg->popIe();

        if (ie->getType() != COMP_REPLY_IE)
            throw cRuntimeError("LteCompManagerProportional:
                  handleCoordinatorReply - Expected COMP_REPLY_IE");

        // parse reply message
        X2CompProportionalReplyIE* replyIe =
            check_and_cast<X2CompProportionalReplyIE*>(ie);
        std:vector<CompRbStatus> allowedBlocksMap =
            replyIe->getAllowedBlocksMap();

        UsableBands usableBands = parseAllowedBlocksMap(allowedBlocksMap);
        setUsableBands(usableBands);

        delete replyIe;
    }
}
```

#### 5.4.1.4 Results

We now discuss the results obtained by simulating the network from Fig. 5.16. We consider 500 m as the inter-eNB distance and randomly deploy a varying number of UEs per eNB. Each UE is the destination of a CBR data flow; hence, it runs one `CbrReceiver` application on top of UDP. Flows originate at the server that has one `CbrSender` application per UE, each of them sending a 40 B packet every 20 ms. `CbrSender` and `CbrReceiver` applications are defined in the *apps* folder. The available number of RBs is set to 25, corresponding to a 5 MHz bandwidth system and MaxC/I is employed as the scheduling policy. We run five independent repetitions for each scenario configuration.

We compare the results obtained with and without interference coordination provided by the CoMP algorithm described in Sect. 5.4.1.3, to show that the latter improves the system fairness in terms of UE throughput. To this aim, we obtain the application-layer throughput by extracting the `cbrReceivedThroughput` statistics from the simulations results and process it to produce the *Lorenz curve* depicted in Fig. 5.17, in a scenario with 30 UEs per eNB. This curve provides a graphical representation of the cumulative portion of the throughput (on the $y$-axis) achieved by the cumulative portion of the UEs (on the $x$-axis). The bisector represents the ideal case, where all the UEs obtain the same throughput; hence, the more a curve is close to the bisector, the more the system is fair. Figure 5.17 shows that the scenario where CoMP is enabled guarantees more fairness among UEs than the scenario with no interference coordination.

This is due to the improvements of the channel quality for UEs close to the cell edge, which are more protected from the interference produced by non-serving eNBs. This argumentation is supported by MAC-level metrics provided by

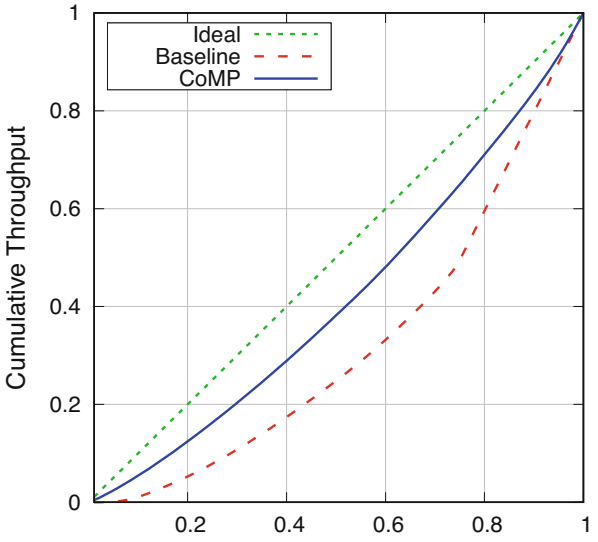



**Fig. 5.17** Lorenz curve, 30 UEs per eNB

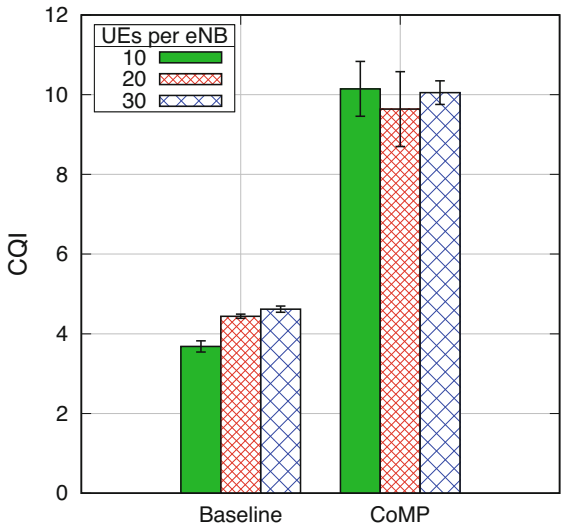



**Fig. 5.18** Average CQI with increasing number of UEs

SimuLTE, such as the `averageCqiDl` one, which is shown in Fig. 5.18. The latter reports the average CQI used by the eNBs for transmitting in the DL subframe, with an increasing number of UEs per eNB. Besides improving throughput, better channel quality also allows the eNBs to reduce resource utilization. Figure 5.19 reports the `avgServedBlocksDl` statistic, i.e., the average number of RBs

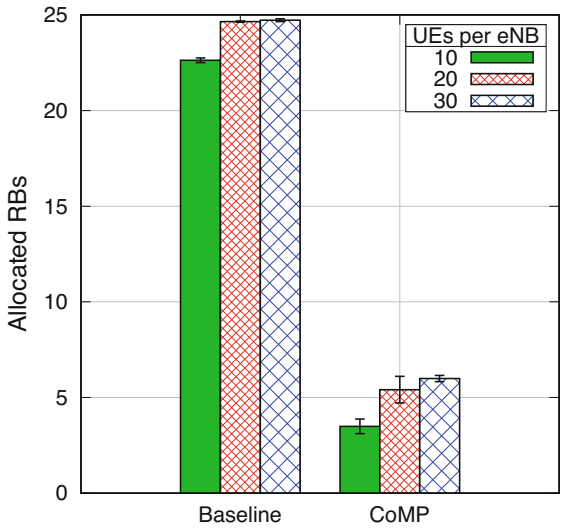



**Fig. 5.19** Average number of allocated RBs with increasing number of UEs

occupied by one eNB on each TTI. While the subframe is basically saturated in the baseline case, only a small number of RBs is used when CoMP is enabled.

## *5.4.2 Tutorial 2: D2D Communication*

This tutorial describes the configuration of a scenario where two UEs are capable of D2D communications and their serving eNB can switch their actual communication mode from direct to infrastructure mode and vice versa. In this scenario, the eNB periodically runs a decision algorithm that selects the communication mode that ensures the best channel quality for the D2D-capable flows. We show which parameters can be tuned in such a scenario and how to implement a new mode-selection policy.

### 5.4.2.1 Network Definition

We consider the `SingleCell_D2D` network, which is composed of one eNB. As Fig. 5.20 shows, UEs are divided into three groups, namely, `ueCell`, `ueD2DTx`, and `ueD2DRx`, which, respectively, represent conventional cellular UEs, transmitting D2D UEs, and receiving D2D UEs. The number of UEs can be configured using the corresponding NED parameters `numUeCell`, `numUeD2DTx`, and `numUeD2DRx`.

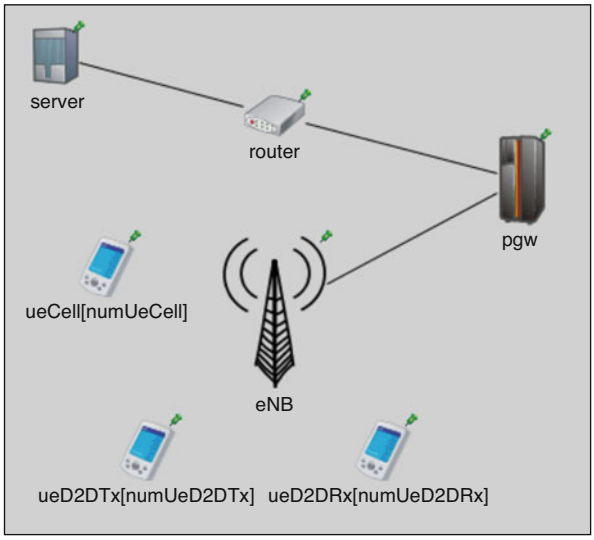



**Fig. 5.20** Network definition for the D2D communication scenario

### 5.4.2.2 Parameters Configuration

With reference to the *omnetpp.ini* file portion shown in Listing 5.8, we first need to enable D2D capabilities on both the eNB and the UEs by setting the D2D-capable version of the LTE NICs, using the parameter `nicType`. Then, we specify the `d2dInitialMode` parameter for `ueD2DTx[0]`, i.e., the communication mode used at the beginning of the simulation. In this example this parameter is set; hence, `ueD2DTx[0]` performs transmissions using the D2D path. Regarding the MAC layer, we need to specify the D2D mode for the AMC module, which extends the default one supporting UL/DL communications only. We also allow D2D flows to transmit using either fixed modulation or the CQI reported periodically. In the former case, we need to set the `usePreconfiguredTxParams` parameter and specify `d2dCqi` in a range between 1 and 15. In the latter case, we set `enableD2DCqiReporting`, whereas `usePreconfiguredTxParams` is disabled and `d2dCqi` is ignored. For the purposes of this tutorial, the second mode is used. Going down to the PHY layer, it is possible to select different transmission power for UL and D2D communications through `ueTxPower` and `d2dTxPower` parameters, expressed in dBm. By default, mode-selection functionality is disabled at the eNB. We can enable it by specifying the name of the module implementing the mode-selection algorithm with `d2dModeSelectionType` parameter. Since mode selection is performed periodically by the eNB, it is possible to specify the period duration through the `modeSelectionPeriod` parameter, expressed in seconds. In the next subsection, we will show how to implement customized algorithms.

**Listing 5.8** Configuration of D2D-related parameters

```
# Enable D2D for the eNB and the UEs involved in direct communications
*.eNB*.nicType = "LteNicEnbD2D"
*.ueD2D*[*].nicType = "LteNicUeD2D"

# Set the initial communication mode
*.ueD2DTx[0].lteNic.d2dInitialMode = true

# Select CQI reporting mode for D2D transmissions
*.eNB.lteNic.mac.amcMode = "D2D"
*.eNB.lteNic.phy.enableD2DCqiReporting = true
**.usePreconfiguredTxParams = false
**.d2dCqi = 7

# Select Tx Power
*.ueD2DTx[0].lteNic.phy.ueTxPower = 26 # in dBm
*.ueD2DTx[0].lteNic.phy.d2dTxPower = 20 # in dBm

# Enable Mode-selection algorithm
*.eNB.lteNic.d2dModeSelectionType = "D2DModeSelectionBestCqi"
*.eNB.lteNic.d2dModeSelection.modeSelectionPeriod = 1s
```

#### 5.4.2.3 Modifying the Code

The module responsible for selecting the transmission mode of D2D-capable flows is `D2DModeSelectionBase`. The latter provides basic functionalities for periodic mode-selection operations and it can be extended to realize the preferred policy. As shown in Listing 5.9, this module periodically schedules a self-message, which serves as a trigger for calling the `doModeSelection()` function. The latter implements the actual mode-selection algorithm. Since it is a virtual function, one can build its own module extending the base one and redefining the behavior of `doModeSelection()`. SimuLTE provides an example module, called `D2DModeSelectionBestCqi`, which selects either UL or D2D for a flow based on the best CQI value reported for the two links. After the execution of `doModeSelection()`, the decisions are notified to the UEs involved in D2D-capable communications.

**Listing 5.9** Modifying the mode-selection algorithm

```
void D2DModeSelectionBase:handleMessage(cMessage *msg)
{
    if (msg->isSelfMessage())
    {
        if (strcmp(msg->getName(),"modeSelectionTick") == 0)
        {
            // run mode selection algorithm
            doModeSelection();
            // send switch notifications to selected flows
            sendModeSwitchNotifications();
            scheduleAt(NOW+modeSelectionPeriod_, msg);
        }
        else
            throw cRuntimeError("D2DModeSelectionBase:handleMessage -
                  Unrecognized self message %s", msg->getName());
    }
```

```
16     else
17         delete msg;
18 }
```

The message including a switch notification traverses the whole LTE stack at the UE side in the upstream direction. This way, each layer is able to perform switching-related tasks. Listing 5.10 refers to a snippet of the `handleMessage()` function in `LteMacUeD2D` module. If the message is recognized as a switch notification, the corresponding handler `macHandleD2DModeSwitch()` is invoked. The MAC-layer handler is responsible for clearing buffers and terminating ongoing H-ARQ processes. However, one can re-implement this handler (as well as higher-layer handlers) to provide advanced switching operations, e.g., to avoid packet loss.

**Listing 5.10** Modifying mode-switching handler at the UEs

```
1  if (incoming == down_[IN])
2  {
3      UserControlInfo *userInfo = check_and_cast<UserControlInfo *>(pkt->
           getControlInfo());
4      if (userInfo->getFrameType() == D2DMODESWITCHPKT)
5      {
6          EV<<"LteMacUeD2D:handleMessage - Received packet "<<
7          pkt->getName()<<" from port "<<pkt->getArrivalGate()->getName()<<endl;
8
9          // message from PHY_to_MAC gate (from lower layer)
10         emit(receivedPacketFromLowerLayer, pkt);
11
12         // call handler
13         macHandleD2DModeSwitch(pkt);
14         return;
15     }
16 }
```

#### 5.4.2.4 Results

In order to demonstrate the effects of mode switching, we consider the network from Fig. 5.20 and simulate one pair of D2D UEs. One UE is the transmitter and one is the receiver of a CBR data flow, sending one packet every 20 ms. We make the packet length vary from 500 B to 1000 B to assess the performance of the D2D flow with different traffic loads. The two UEs are 300 m away from the eNB and they swing back and forth in a straight line at a speed of 3 m/s. Such a path allows the direct link between them to experience different channel quality during the simulation, whereas the channel quality for UL stays constant. This way, we can simulate a scenario where the eNB implements the aforementioned `D2DModeSelectionBestCqi()` policy, making the flow bounce between the direct link and the infrastructure one. We compare this scheme with the scenario where mode selection is disabled and the flow uses always the direct link.

In this dynamic scenario, OMNeT++ vectors are useful to evaluate the behavior of the system during the simulation. The left side of Fig. 5.21 reports the CQI used by the transmitting UE, which experiences large variations due to the change in

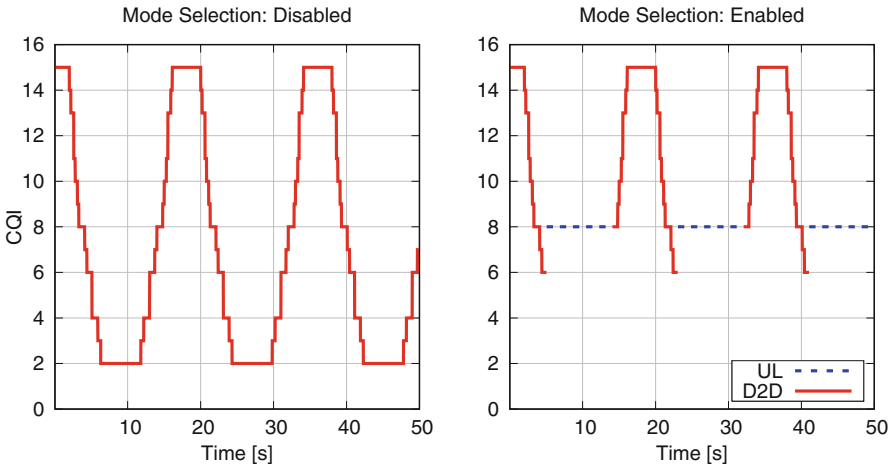



**Fig. 5.21** CQI with mode selection disabled (left) and enabled (right)

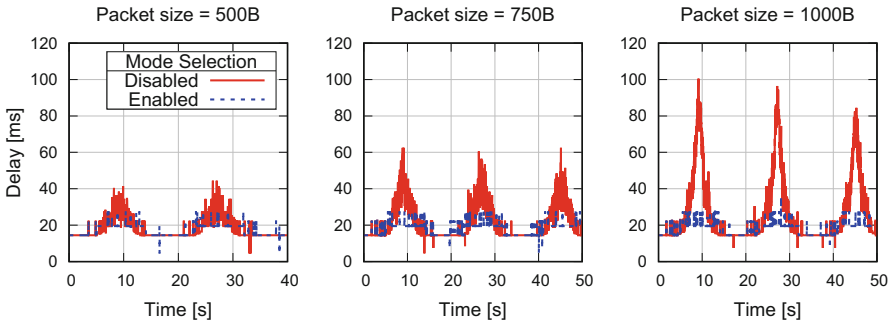



**Fig. 5.22** Latency of CBR packets with increasing packet length

the distance between the UEs. The right part of Fig. 5.21, instead, shows the CQI used when mode selection is employed: the chart is obtained by putting together `averageCqiD2D:vector` and `averageCqiUl:vector` statistics, since the flow periodically switches from the direct link to the infrastructure one and vice versa. As a result, we observe that enabling mode selection allows the flow to use better CQI, hence use better modulation.

This affects the latency of the flow, as shown in Fig. 5.22. The latter reports the `cbrFrameDelay:vector` statistic with different packet lengths. When no mode selection is active, the latency of the flow drastically increases at the points where the CQI is smaller. This behavior is more pronounced when the traffic load increases due to larger packets. On the other hand, the configuration with mode selection allows the flow to keep the latency small.

## References

1. Baldo, N., Miozzo, M., Requena-Esteso, M., Nin-Guerrero, J.: An open source product-oriented LTE network simulator based on ns-3. In: Proceedings of the 14th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, MSWiM '11, pp. 293–298. ACM, New York (2011). https://doi.org/10.1145/2068897.2068948
2. Bouras, C., Diles, G., Kokkinos, V., Kontodimas, K., Papazois, A.: A simulation framework for evaluating interference mitigation techniques in heterogeneous cellular environments. Wirel. Pers. Commun. **77**(2), 1213–1237 (2014). https://doi.org/10.1007/s11277-013-1562-5
3. Jakes, W. (ed.): Microwave Mobile Communications. Wiley, New York (1975)
4. Mehlführer, C., Wrulich, M., Ikuno, J.C., Bosanska, D., Rupp, M.: Simulating the long term evolution physical layer. In: 2009 17th European Signal Processing Conference, pp. 1471–1478 (2009)
5. Sanchez-Iborra, R., Sanchez-Gomez, J., Ballesta-Viñas, J., Cano, M.D., Skarmeta, A.F.: Performance evaluation of LoRa considering scenario conditions. Sensors **18**(3), 772 (2018). https://doi.org/10.3390/s18030772
6. Virdis, A., Iardella, N., Stea, G., Sabella, D.: Performance analysis of openairinterface system emulation. In: 2015 3rd International Conference on Future Internet of Things and Cloud, pp. 662–669 (2015). https://doi.org/10.1109/FiCloud.2015.77

# Chapter 6
# Veins: The Open Source Vehicular Network Simulation Framework


**Christoph Sommer, David Eckhoff, Alexander Brummer, Dominik S. Buse, Florian Hagenauer, Stefan Joerer, and Michele Segata**

## 6.1 Introduction

Veins [56] is a model library for (and a toolbox around) OMNeT++, which supports researchers conducting simulations involving communicating road vehicles; either as the main focus of a study (such as Vehicular Ad Hoc Networks - VANETs) or as a component (such as in Intelligent Transportation Systems - ITS). It is distributed as open-source software; as such, it is free to download, adapt, and use.

The model library includes a full stack of simulation models for investigating communicating vehicles and infrastructure; as of Veins 4.7, predominantly cars and trucks using Wireless Local Area Network (WLAN)-based technologies. For this,

C. Sommer (✉) · D. S. Buse · F. Hagenauer
Heinz Nixdorf Institute and Department of Computer Science, Paderborn University, Paderborn, Germany
e-mail: sommer@ccs-labs.org; buse@ccs-labs.org; hagenauer@ccs-labs.org

D. Eckhoff (✉)
TUMCREATE Ltd, Singapore, Singapore
e-mail: david.eckhoff@tum-create.edu.sg

A. Brummer
Computer Networks and Communication Systems, University of Erlangen-Nürnberg, Erlangen, Germany
e-mail: alexander.brummer@fau.de

S. Joerer
Institute of Computer Science, University of Innsbruck, Innsbruck, Austria
e-mail: joerer@ccs-labs.org

M. Segata
Department of Information Engineering and Computer Science, University of Trento, Trento, Italy
e-mail: msegata@disi.unitn.it



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_6

Veins includes a sophisticated model of IEEE 802.11 MAC layer components [12] used by standards such as IEEE Wireless Access in Vehicular Environments (WAVE) (of which a simple simulation model is included), ETSI ITS-G5 (as provided by, e.g., Artery [43] which is described in Chap. 12), or ARIB T-109 [23]. Because Veins is a modular framework, it can equally well be used as the basis for modeling other mobile nodes such as pedestrians, bikes, trains, and Unmanned Aerial Vehicles (UAVs)—or for other communication technologies like Long Term Evolution (LTE) mobile broadband [21] (cf. Sect. 6.4.1) and Visible Light Communication (VLC) [37].

The history of Veins goes back to early 2006 with the first public release being an extension for the INET Framework version 2006-10-20. Because of limitations in the fidelity of wireless channel modeling at the time, for its 1.0 release Veins was ported to be an extension of MiXiM (an alternative, now discontinued library of OMNeT++ simulation models for wireless channel modeling) instead. Veins was then increasingly augmented with new models, e.g., of IEEE 802.11p, IEEE 1609.4, and WAVE, which would later be re-factored all the way down to the physical layer for the 2.0 release. As more refactoring and rewriting was taking place in the channel models, Veins 3.0 became a proper fork of MiXiM, but was kept compatible with mixed simulations incorporating models from the INET Framework. Up to the current 4.7 release, Veins was then continuously streamlined and augmented with more and more of the aforementioned models specific to communicating road vehicles. This release is compatible with OMNeT++ 5 (up to the current version 5.4.1) and SUMO 0.32.0 (the latest release of SUMO; please refer to Sect. 6.2.1 for details on its role for Veins). A full compatibility list is available online.[1]

Veins has become well-established in the domain of Vehicular Ad Hoc Networks (VANETs) and Intelligent Transportation System (ITS). It is employed by both academia and industry around the globe. It serves as the basis of hundreds of publications and contributed to the standardization process of Inter-Vehicle Communication (IVC). Common fields of application include channel access control [17, 58], safety applications [26, 57], privacy [14] and security [44], platooning [49], communication with traffic lights [16], electric vehicle operation [3], as well as traffic optimization [65]. For some of these uses cases there exist dedicated extensions for Veins such as PREXT for location privacy [19], PLEXE for platooning [48], an extension to incorporate a real world driving simulator [2], or a simulation framework for electric vehicles [3].

In this chapter, we give a brief overview of recent developments regarding the internals of Veins (bi-directional coupling, communication stack, antenna characteristics, unit testing, and timer management; in Sect. 6.2), present two practical use cases (platooning and intersection collision avoidance; in Sect. 6.3), and conclude with a brief discussion of two extensions (Veins LTE and Veins_INET) as well as using Veins as a virtual appliance (cf. Sect. 6.4).

[1] http://veins.car2x.org/.

## 6.2 Internals

In this section, we explain how the bi-directional coupling works (cf. Sect. 6.2.1) and give details on the implementation of the IEEE 802.11p-based communication stack (cf. Sect. 6.2.2). Discussion on Veins internals continues with the modeling of antenna characteristics (cf. Sect. 6.2.3), followed by a section on how unit testing can help in the development of new simulation models (cf. Sect. 6.2.4), and simplified timer management (cf. Sect. 6.2.5).

### *6.2.1 Architecture and Bidirectional Coupling*

Contrary to what might be expected, Veins does not include custom mobility models of road vehicles. Rather, it has simulations establish a connection to a dedicated road traffic simulator which is running as a separate process, as illustrated in Fig. 6.1. This way, Veins can benefit from the years of research and development by domain experts who have created fully featured tools for road traffic simulation. The road traffic simulator that Veins was designed to interoperate with is Simulation of Urban MObility (SUMO)[2] (though, in theory, any simulator supporting the Traffic Control Interface (TraCI) simulator coupling interface can be used).

SUMO can simulate medium to large road networks of cities, urban areas, highways, and freeways. On those, it can simulate the movement of road vehicles

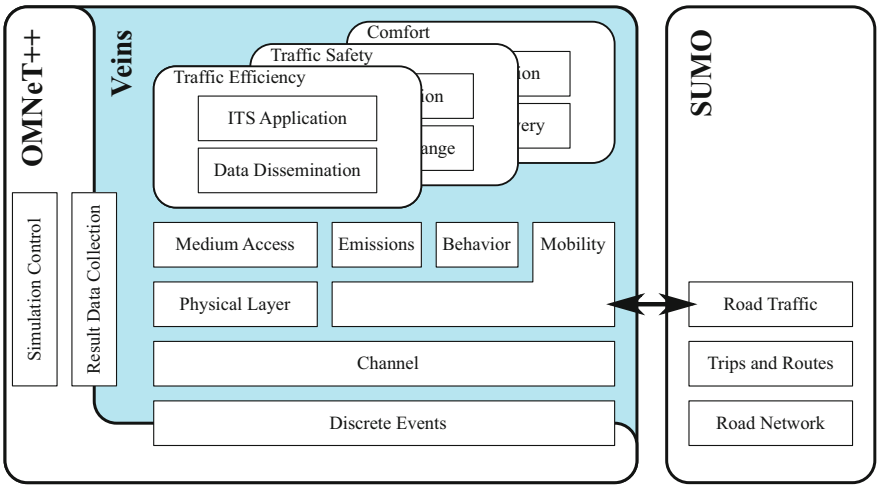



**Fig. 6.1** High-level architecture of Veins

[2] SUMO website: http://sumo.dlr.de/.

like cars and trucks, of scooters and bicycles, of pedestrians, and of trains. SUMO supports a wide range of different mobility models (from idealized, lane-discrete models to sub-lane models of mixed car/scooter traffic), a set of different intersection controllers (from simple right of way to demand-actuated traffic lights), and a wide range of road network input formats (from **OpenStreetMap** and **TIGER** to proprietary, specialist Geographic Information System (GIS) formats).

By default, mobility information is polled from SUMO at fixed intervals of, e.g., 100 ms, though adaptive polling is equally well supported by the interface. Execution of the OMNeT++ simulation pauses while SUMO computes mobility information for the desired point in time. The performance impact of this is, however, minimal as SUMO is designed to simulate at least an order of magnitude more mobile nodes than can be afforded in a highly detailed wireless network simulation. As a consequence, in a reasonably complex wireless network simulation, only fractions of percent of simulation time are spent calculating and communicating mobility information.

Whenever SUMO simulates the departure of a mobile node, Veins creates a dedicated simulation module in OMNeT++. Then, as the mobile node moves in SUMO, Veins keeps the corresponding OMNeT++ module updated wrt its position, heading, and speed (along with the status of turn signal indicators and similar miscellaneous data). Similarly, when SUMO simulates the mobile node arriving at its destination, Veins removes the corresponding OMNeT++ module from the simulation. This way, Veins couples node mobility in OMNeT++ to that in SUMO.

This coupling is bi-directional: in addition to the OMNeT++ simulation evolving as dictated by the SUMO simulation, the OMNeT++ simulation can influence the simulated road traffic in SUMO, for example, to have cars choose a different route to their destination in response to received traffic information—or to have a car perform an emergency brake in response to received warnings. This is done by calling methods of the `TraCICommandInterface` and component class instances associated with each mobile node. They are available by obtaining a pointer to the mobility module via `TraCIMobilityAccess().get(getParentModule())` and calling its `getCommandInterface()` or `getVehicleCommandInterface()` method from any simulation model of a mobile node that contains a mobility model of type `TraCIMobility` (a requirement for mobile nodes managed by Veins). These interfaces offer a wealth of methods—from the simple, like `getRoadId` and `newRoute` (for vehicles) to the complex, like `setProgramDefinition` (for a traffic light). Details on this concept are available in the literature [56].

Programmatically, the coupling is performed by instantiating a simulation module of type `TraCIScenarioManager`, which bi-directionally couples OMNeT++ and SUMO. However, the user needs to manually run one SUMO simulation for every OMNeT++ simulation. As an alternative and to ease the management of two simulators running in parallel, Veins also includes tools to automatically set up and run SUMO simulations. This is done by instantiating a subclass of `TraCIScenarioManager` called `TraCIScenarioManagerLaunchd`. It expects the user to have run a command line utility, **sumo-launchd.py**, which waits

for incoming network connections from an OMNeT++ simulation and launches one instance of SUMO for each simulation and proxies the connection. Alternatively, another subclass called `TraCIScenarioManagerForker` can be employed, which will directly run a local instance of SUMO when needed. All of these coupling variants are included with Veins.

What is not included with Veins are road traffic scenarios to generate the SUMO traffic from. While, these days, road network data and building positions are easy to come by (thanks to open data sources), information about traffic demand (that is, how typical traffic moves through the road network), traffic light timings, or meta-data like bus and train schedules are much harder to obtain. In the early days of VANET simulation, road traffic scenarios were thus often generated synthetically, e.g., modeling an ideal Manhattan Grid of roads. This had the obvious downside of requiring a lot of skill on the part of the researcher generating the road traffic scenario (lest the simulation test the system under study in unrealistic conditions).

A better choice is to pick one of the well-tested road traffic scenarios that have been made available recently. Examples for the SUMO road traffic simulator are:

- The Bologna "Pasubia" and "Acosta" scenarios [6], depicted in Fig. 6.2a, feature 9k trips each on two areas of 2 km × 1 km each.[3] They can be run individually or as one bigger road traffic scenario and feature traffic driving in a small part of the city core of Bologna, though care must be taken as no building positions are included with the scenario.
- The Bologna "Ringway" scenario [4], depicted in Fig. 6.2b, features 22k trips on an area of 4 km × 3 km.[4] It focuses on road traffic on an arterial road running around a city center. Like the Pasubia and Acosta scenarios, no building positions are included with the scenario.
- The Luxembourg "LuST" scenario [10], depicted in Fig. 6.2c, features 288k trips on an area of 14 km × 11 km.[5] It is the largest and most complete scenario to date and includes a full day of mobility data for a complete city, including the positions of buildings and parking lots.
- The Monaco "MoST" scenario [9] in Fig. 6.2d includes 18k trips in an area of 10 km × 7 km.[6] Still under development, it focuses on multi-modal traffic, comprising added information regarding public transport, bicycles, and pedestrians.

[3] http://sourceforge.net/projects/sumo/files/traffic_data/scenarios/Bologna_small.

[4] http://www.cs.unibo.it/projects/bolognaringway/.

[5] https://github.com/lcodeca/LuSTScenario.

[6] https://github.com/lcodeca/MoSTScenario.

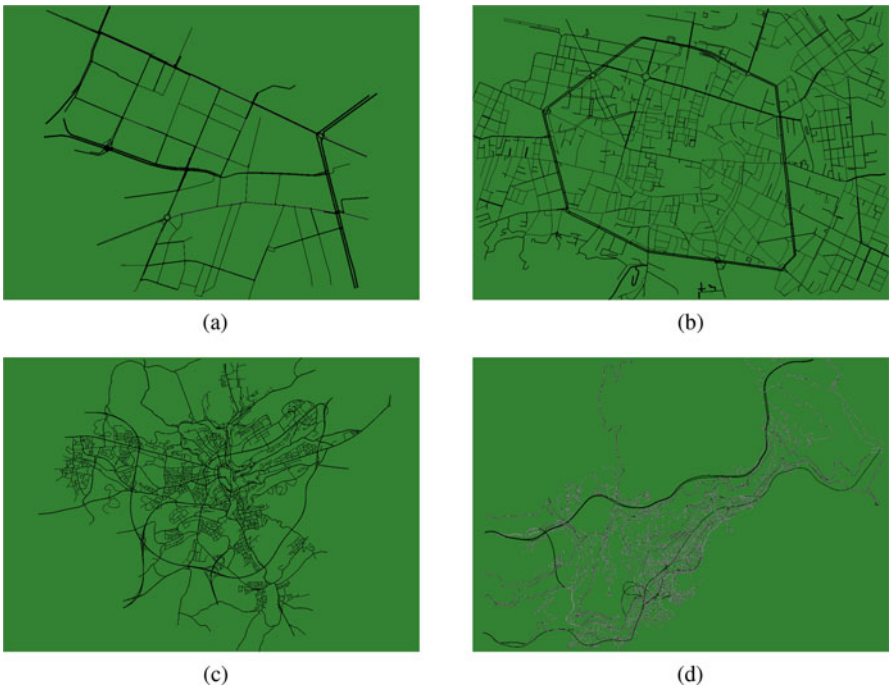



**Fig. 6.2** Selection of existing openly available scenarios for SUMO. (**a**) Bologna: Pasubia and Acosta. (**b**) Bologna: Ringway. (**c**) Luxembourg: LuST. (**d**) Monaco: MoST

### *6.2.2 The MAC and PHY Layer*

One of the core features of Veins is the detailed modeling of the lower layers of Inter-Vehicle Communication (IVC). For the evaluation of most IVC applications and networks, a detailed packet-level simulation using accurate models of the evaluated technology is required [15]. For vehicular networks, the technology in question is often IEEE WAVE (or ETSI ITS-G5 in Europe). The core of this family of standards is the IEEE 1609.4 multi-channel operation using the IEEE 802.11p Medium Access Control (MAC) and Physical Layer (PHY). An overview of the stack is given in Fig. 6.3a. While it is possible to implement and integrate each of these layers and standards, Veins puts a focus on the lower layers as these are decisive for the actual channel access and transmission of packets [17]. Other simulation models (not included with Veins, but publicly available, such as ARIB T-109 [23]) can build on this foundation if additional protocol layers of the various protocol stacks of ITS protocols around the world are to be modeled as well.

Figure 6.3b shows the representation of the stack within Veins. Each node, be it a vehicle, a road-side unit, or even a pedestrian or cyclist making use of wireless communications would need to consist of at least an 802.11p Network Interface

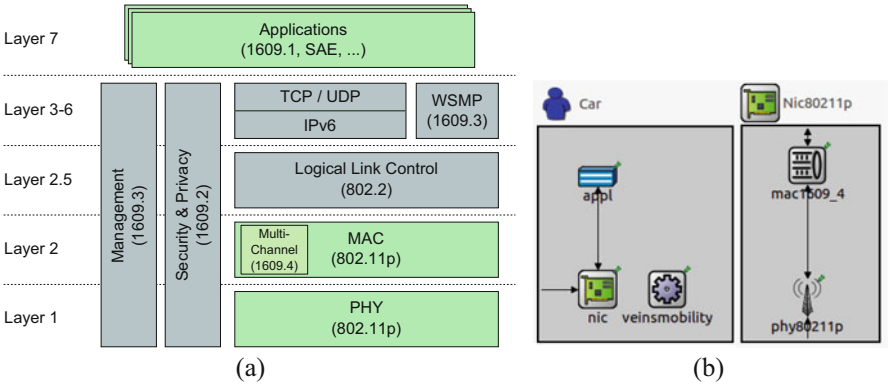



**Fig. 6.3** The IEEE WAVE stack and its representation in Veins. (**a**) The IEEE WAVE family of standards. PHY, MAC, and application layer are represented in Veins. (**b**) Layer representation in OMNeT++

Card (NIC) to be able to communicate with other devices. Higher layers (in some stacks: the application layer) are directly connected to this NIC which itself is a compound model consisting of the MAC and the PHY layer. This results in a simple APP-MAC-PHY architecture for each node in Veins. The `veinsmobility` module is responsible for updating the position of the vehicle (see Sect. 6.2.1). In the case of a road-side unit, the mobility would be a constant `BaseMobility`.

In OMNeT++, each module can exchange messages with other modules if they are connected. These messages can be of any type inheriting from `cMessage*`, that is, just plain messages or (encapsulated) packets of any given message format (e.g., Wave Short Messages (WSMs) or Wave Service Advertisements (WSAs)). Inside a node, messages can either be "normal" messages that might be forwarded to layers above or below or control messages to trigger a certain action in the receiving layer. Depending on the type, a different function will be called in the receiving layer. As can be seen in the figure, the physical layer is connected only to the MAC layer and to the outside world.

In the following subsections, we will discuss how messages are generated, processed, forwarded, and received.

#### 6.2.2.1 Medium Access Control and Upper Layers

The Medium Access Control (MAC) layer in the simulation should represent the simulated system as closely as possible (for example, evaluating an IEEE 802.11p system for IVC using a model for IEEE 802.11b can give misleading or even wrong results) [15, 27]. Veins comes with a detailed IEEE 1609.4 and IEEE 802.11p MAC layer that supports multi-channel operation, channel switching (alternate access), transmission of unicast and broadcast messages, and an IEEE 802.11e Enhanced Distributed Channel Access (EDCA) implementation with four different

access categories [12]. For a detailed description we refer the interested reader to IEEE 802.11e and IEEE 802.11p [12, 15, 53] and the actual standardization documents [25] and [24]. The level of detail in Veins' MAC and PHY layer implementation allows researchers to conduct various simulation studies, for instance comparing wireless network performance [17, 58], studying the applicability of the wireless network for vehicular cooperative safety [26] (cf. Sect. 6.3.2), or the simulative analysis of platoons [48] (cf. Sect. 6.3.1).

The implementation in Veins follows a different paradigm compared to most other OMNeT++ frameworks. The behavior of the MAC layer can be specified in the form of a state machine, which is a useful method to understand as well as implement the system. Transitions between states are triggered after, e.g., a timeout has expired, the backoff counter has reached zero, a packet arrived, and so on. Indeed, an implementation might choose to directly follow the state diagram. However, the MAC layer has several properties which make such an implementation hard to maintain and read: packets can arrive from an upper layer regardless of the state the MAC layer is in, multiple timers can run in parallel (e.g., for each of the EDCA queues as well as the channel switching time), and the multi-channel operation would require two independent state machines. Not only does this lead to plenty of nested `if` statements in each function (to check which state the system is currently in) which makes extending and understanding the code base challenging, it also has an impact on performance as multiple timers have to be managed in parallel, i.e., inserted into and removed from the event queue.

This prompted the design decision to not rely on a state machine implementation but follow a different, more efficient approach: The MAC layer always tracks the time at which it can send the *next* packet, instead of tracking all the different intervals such as interframe spaces or backoff times, separately. When an event occurs that affects this time, e.g., the channel turns busy or a new packet in a higher priority queue arrives from the upper layer, the timer is canceled or rescheduled and the backoff counters for each EDCA queue are updated. When the channel turns idle again, the time is recomputed and the timer is scheduled again. The result of this design is that Veins will only use one single timer (`nextMacEvent`) when using a single channel MAC layer with broadcast messages only, which is a rather common setup for vehicular networks. Multi-channel operation and unicast packets require additional timers.

**Transmitting a Packet** The MAC layer expects a `WaveShortMessage` from higher layers (e.g., the application layer) with attached information on which channel it should be sent and a user priority which will be mapped to an EDCA queue. The packet will be queued accordingly and the `nextMacEvent` timer will be updated if necessary. If the channel is busy and the respective EDCA queue has a backoff counter of 0 with the newly arrived packet at the front of the queue, then a backoff procedure is invoked according to the standard.

The core of the MAC layer is the `startContent` function which models the start of contention for the channel and returns the time the next packet can be sent. It iterates through each of the EDCA queues and computes this time based on the

queue-specific interframe time (AIFSn × slot length + SIFS), the current backoff counter, and the last time the channel went idle. If the channel was idle long enough when a new packet arrives from the upper layer, the packet will be sent at the next slot boundary. When the timer expires, the MAC layer sets the channel to busy and calls the `stopContent` function. In this function, the backoff counters of the remaining EDCA queues are updated and Transmit Opportunitys (TXOPs) for ready-to-transmit queues are generated. Then `initiateTransmit` function is invoked which is responsible for returning the actual packet that is supposed to be sent. In the case of an internal collision, that is, when there are two or more packets ready, the lower priority queues will be sent into backoff. The winning packet will be encapsulated with the corresponding MAC header and `controlInfo` (containing transmit power and data rates), and if there is enough time left in the current control or service channel interval, handed to the PHY layer. The `stopContent` function is also invoked when the channel turns busy due to an external transmission. In this case, no TXOPs are generated and the `nextMacEvent` timer is canceled.

**Receiving a Packet** The role of the MAC layer in the reception of a packet is straightforward. If the PHY layer sends up a `Mac80211Pkt`, the MAC will check whether the destination address is the layer 2 broadcast address or whether it matches its own MAC address. If this is the case, the packet will be decapsulated and the `WaveShortMessage` will be handed to the application layer. When dealing with unicast transmissions, the received packet can be an Acknowledgment (ACK) packet. The reception of an ACK packet marks the successful transmission of a unicast packet, causing the MAC layer to remove it from the respective EDCA queue. If the MAC layer is expecting an ACK packet but has received another packet, then the originally sent packet has to be retransmitted.

The PHY layer also informs the MAC layer of several other events such as successful or unsuccessful reception of a packet, the channel turning busy or idle, erroneous decoding of a packet, and so on. This is achieved by means of control messages. Veins collects various statistics about received packets and failures (split by broadcast and unicast and by cause of loss) as well as about internals of the state machine (e.g., how busy the channel was), giving the researcher methods to evaluate the underlying network in great detail.

#### 6.2.2.2 The Physical Layer and the Wireless Channel

The benefit of packet level simulation is the capability to (more or less) realistically determine for each packet if it can be successfully received. There are several factors affecting the decoding of a packet: the position in space of sender and receiver, the antenna characteristics (see Sect. 6.2.3), whether there is an obstacle blocking the line of sight, and interference from other transmitting nodes. While Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) significantly reduces the chance of two nearby nodes (i.e., they can hear each other) sending at the same

time, it does not offer a solution to the hidden terminal problem [22]. All these effects can be captured by Veins. In this section, we will outline the functionality of the PHY.

The described models of the Physical Layer and the wireless channel in Veins are currently based on a fork of MiXiM [64], a discontinued framework which models radio signals as generic $n$-dimensional objects (power levels expressed in, e.g., time and frequency) and provides a math toolbox to work with them.

It should be noted that, while this is the most flexible way of modeling radio signals, it is also computationally expensive. Thus, Veins has been updated to use a more specific abstraction of radio signals, tailored to the feature set used in common Vehicle-to-Vehicle (V2V) communication (e.g., forcing any radio signal to always have a time and a frequency dimension—never more, never less) and optimized for efficiency. While this has the obvious drawback of not being able to model radio signals in dimensions other than time and space, these adaptations can allow simulations to run faster—in some cases up to two orders of magnitude. While this functionality is not yet available in Veins 4.7, it is available on Github and will be integrated into upcoming releases of Veins.

**Analogue Models** The connection manager of OMNeT++ maintains a connectivity map to be able to hand transmitted messages to the receiving nodes. Every node inside a configurable *interference range* of a transmitting node will be handed a copy of the transmitted packet. Determining whether this packet is successfully received then lies within the responsibility of the node itself. The setting of an *interference range* is purely an optimization: it defines an artificial range beyond which no radio transmission needs to be considered as interfering. Naturally, it should be set much larger than the maximum range of any successful transmission, as also packets that have a too low receive power (or Received Signal Strength (RSS)) to be decoded can still affect the successful reception of other packets.

The connection manager will hand an airframe at least twice to the PHY layer via the `handleMessage` function: when the receiving starts and when it ends. The first thing that has to be computed is the actual receive power of the frame as this determines whether the channel turns busy or remains idle. This is done by applying the antenna gains ($G_t$, $G_r$) and iteratively applying all the configured loss models $L$ in the `filterSignal` function (see Eq. (6.1)).

$$P_r = P_t + G_t + G_r - \sum L \tag{6.1}$$

Common deterministic loss models include the free space path-loss model and the two-ray interference path-loss model [52] that considers the reflected signal from the road that can cause cancellation and amplification of the received signal. A detailed explanation of these models can be found in [13]. To account for fast fading effects, Veins can make use of *Nakagami-m* fading which is a probabilistic method to reflect multi-path propagation in urban environments [59].

The effect of obstacles (e.g., buildings in the scenario description file) is also accounted for by the use of a loss model. Assuming each obstacle is a polygon, then the receive power is reduced based on the number of edges $n$ the signal

is intersecting (e.g., walls) and the distance $m$ covered inside of polygons (e.g., inside a building). These values are weighted using parameters $\beta$ and $\gamma$ which were calibrated using real world measurements (see Eq. (6.2)). They can be changed according to the material of the obstacle, e.g., brick, concrete, etc.

$$L_{\text{build}} = \beta \cdot n + \gamma \cdot m \tag{6.2}$$

Listing 6.1 shows how to configure a simple chain of analogue models (an XML configuration set as the physical layer's `analogueModels` parameter). In this configuration, each received signal is first passed through a free-space path loss model, then through an obstacle shadowing model.

A comparison of the different models as well as their ability to reproduce real-world measurements [55] is given in Fig. 6.4.

**Listing 6.1** Content of the *config.xml* file (part one)

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <AnalogueModels>
    <AnalogueModel type="SimplePathlossModel">
      <parameter name="alpha" type="double" value="2.0"/>
      <parameter name="carrierFrequency" type="double" value="5.890e+9"/>
    </AnalogueModel>
    <AnalogueModel type="SimpleObstacleShadowing">
      <parameter name="carrierFrequency" type="double" value="5.890e+9"/>
    </AnalogueModel>
  </AnalogueModels>
</root>
```

**The Decider** Once all loss models have been applied, the airframe is handed to the `Decider` which is an outsourced class that determines whether packets can be successfully decoded. If the received power is below the configurable Clear Channel

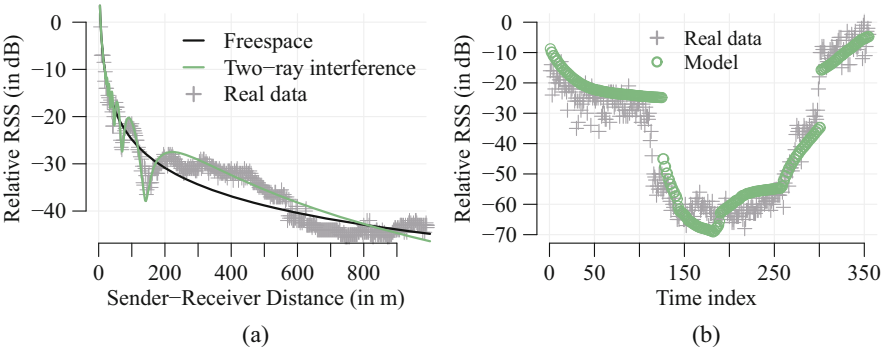



**Fig. 6.4** Analogue models and their effect on the Received Signal Strength (RSS) compared to real-world measurements. (**a**) Real-world measurements compared to the free-space model and the two-ray interference model based on [52]. (**b**) Real-world measurements compared to the obstacle model in Eq. (6.2) based on [55]

Assessment (CCA) sensitivity, this packet is unable to set the channel to busy. The MAC layer will not be notified. If the packet is above the CCA threshold, the decider checks whether the node is already transmitting or receiving another packet. In both cases the packet will fail to decode.

The `processSignalEnd` function in the decider is called when the connection manager hands the airframe to the physical layer the last time. It is the task of the decider to finally determine whether the packet is decodable. To this end, it first has to compute the Signal-to-Interference-plus-Noise-Ratio (SINR) as depicted in Eq. (6.3): the receive power of the packet in question $i$ is divided by the power of all interfering packets $j$ and the background noise $N$.

$$\mathrm{SINR}(i) = \frac{P_i}{N + \sum_{i \neq j} P_j}; \tag{6.3}$$

Once the SINR has been obtained, it can be fed to a bit error model. Depending on the modulation scheme (e.g., Binary Phase Shift Keying (BPSK), Quadrature Phase Shift Keying (QPSK), Quadrature Amplitude Modulation (QAM)), a different equation is applied to calculate the probability of one bit being decoded erroneously. Bit error rates for header and payload are computed separately and then applied to the packet length to derive a packet error rate. Two randomly drawn numbers then decide whether the header and the payload can be decoded successfully.

The packet is handed to the MAC layer or, in the case of an error, a control message is sent. Listing 6.2 shows how to configure the decider model (an XML configuration set as the physical layer's `decider` parameter). Each received signal is then passed through this chain of models.

**Listing 6.2** Content of the *config.xml* file (part two)

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Decider type="Decider80211p">
    <parameter name="centerFrequency" type="double" value="5.890e9"/>
  </Decider>
</root>
```

### 6.2.3 Modeling Antenna Patterns

Antennas are an integral part of wireless communications as they constitute the interface between the radio device and the transmission medium. Yet, despite the multitude of detailed models for the PHY and MAC layers described before, the impact of antenna patterns has not been taken into account in VANET simulation for a long time—even though the gain (or loss) of an antenna can critically influence the receive power and thus the decodability of a sent message. This dependence is

already indicated by Eq. (6.1) (on page 224) with the terms $G_t$ and $G_r$ being related to the sender's and receiver's antenna gain, respectively.

Early work on the impact of antenna patterns on V2V communication [18] demonstrated that the vast majority of messages were received either from the front or from the rear direction of the vehicle. It also demonstrated that, as a consequence, the overall number of received beacons in a typical cooperative awareness simulation was decreased by up to 20% compared to simulations neglecting antenna influence (i.e., assuming isotropic radiators).

A highly configurable model for the consideration of antenna patterns has been added to the Veins framework as of version 4.5.

The power of the sent or received signal depends on several aspects, first of all the type of antenna in use. In the case of an ideal, isotropic antenna, the transmit power is radiated in all directions equally. Omnidirectional antennas, e.g., monopoles, emit the signal power equally in a certain plane. Another category are highly directional antennas, which concentrate the power in one or a few selected directions. These differences in power are usually stated as a dBi value, that is, on a logarithmic scale with respect to an ideal, isotropic radiator.

In the context of vehicular networks, radiation characteristics are further influenced by the vehicles themselves. An important factor is the mounting location, which might be on the roof, at the front, at the rear, on the side mirrors, or even under the car. Example patterns as measured in [31, 34] are depicted in Fig. 6.5. For example, the radiation pattern of a vehicle with patch antennas on the side mirrors (see Fig. 6.5c) exhibits a substantial prevalence towards the front of the car. Moreover, material properties of parts surrounding the antenna can influence the power of a transmitted or received signal. A distinct example is shown in Fig. 6.5a. This radiation pattern is the result of a study by Kwoczek et al. [34] who investigated the consequences of an antenna being mounted next to a panorama glass roof. As can be seen, this leads to a substantial attenuation of up to 20 dBi towards the front of the vehicle as the signal tends to get reflected within the glass roof.

It is quite obvious that such an influence on the signal power can make all the difference in simulation when deciding on the decodability of a packet, which is why the support for antenna patterns has been added to the Veins framework. For this purpose, an object of the newly introduced `Antenna` class is assigned to every vehicle (or more generally: to every module containing a radio). For this, an `Antenna` member is added to the `BasePhyLayer` class, which itself is present in every module capable of wireless communication (see Fig. 6.6). The `Antenna` class can be seen as the superclass for all kinds of specialized antenna implementations and simply returns a factor of 1.0 (representing an isotropic pattern).

This approach facilitates the implementation of various antenna subclasses that differ in the way of computing the specific gain. The subclass capturing one of the most common use cases is `SampledAntenna1D`, which deals with two-dimensional antenna patterns, i.e., only the horizontal plane is considered. In this case the resulting gain depends on one variable, namely the signal's horizontal angle of incidence. The user needs to pick a representative antenna from the included

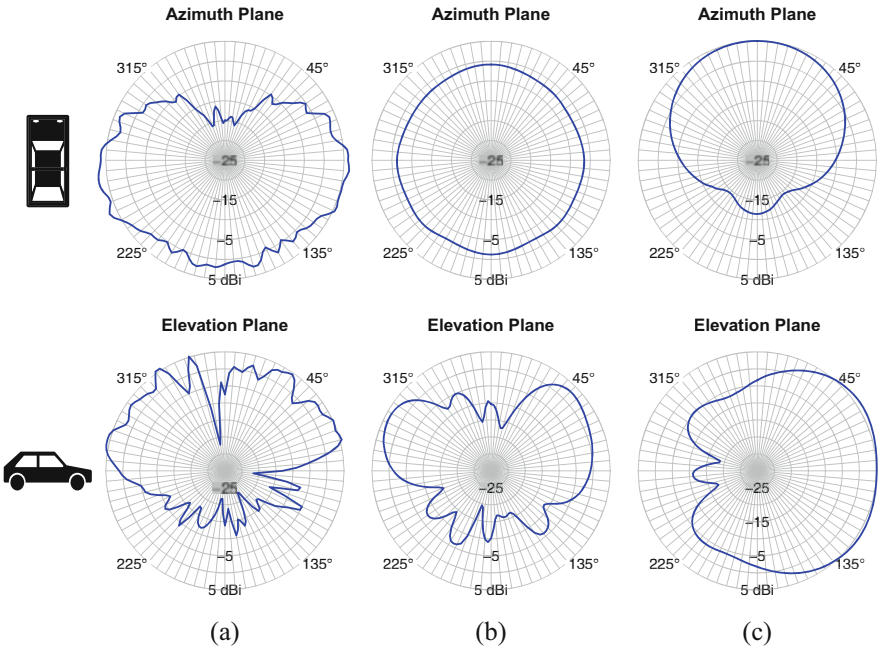



**Fig. 6.5** Azimuth and elevation planes of example vehicular antenna patterns (gain in dBi). (**a**) Monopole antenna on glass roof (based on [34]). (**b**) Monopole antenna (based on [31]). (**c**) Patch antenna (based on [31])

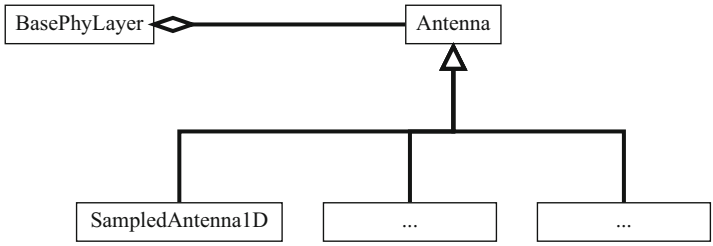



**Fig. 6.6** Overview of the newly added antenna classes

database (or provide samples of the radiation pattern at equidistant angles between 0° and 360°).

For the actual gain calculation, the signal direction has to be determined first. As illustrated in Fig. 6.7, this angle of incidence $\phi$ (also called azimuth angle) depends on the sender's and receiver's position as well as on the orientation of the antenna in question. As all of these parameters are known to the simulation, the azimuth angle $\phi$ can be determined with the help of the scalar product. Next, the stored antenna gain samples are queried at the determined angle. If the angle of the required gain

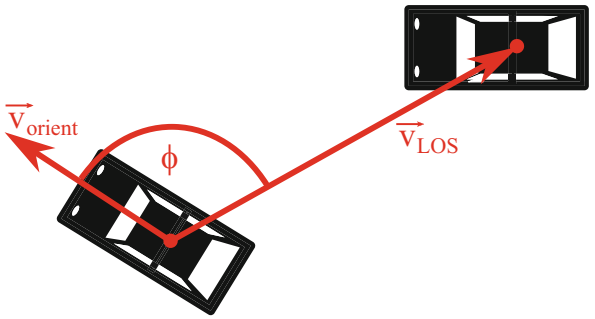



**Fig. 6.7** Dependence of the azimuth angle based on Line of Sight (LOS) and orientation vector

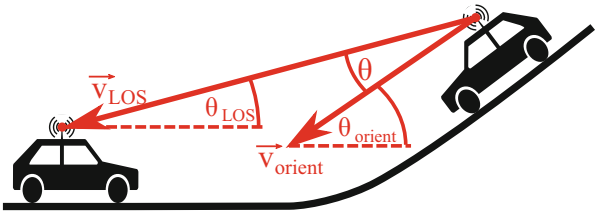



**Fig. 6.8** Dependence of the elevation angle based on Line of Sight (LOS) and orientation vector

value is located between two samples, linear interpolation is applied. Finally, the signal power is multiplied by the determined antenna gain factor.

Recent work also examined the influence of 3D antenna patterns in a three-dimensional environment [8]. To this end, another antenna class has been implemented, namely `SampledAntenna2D`. As the name implies, the antenna gain is now dependent on two parameters. Besides the already introduced horizontal (azimuth) angle $\phi$, the vertical (elevation) angle $\theta$ needs to be determined as well. It can be computed based on the (now three-dimensional) antenna positions and orientation of the ego vehicle (see Fig. 6.8). Only if both angles are known is it possible to specify the signal direction in the three-dimensional space.

Obviously, the user has to provide a 3D antenna pattern in the first place. As a full representation is rarely available and would imply a large number of samples, only the two principal planes are required for our model. The azimuth plane pattern has already been used for the 2D antenna implementation. In addition, the elevation plane pattern becomes necessary. Again, equidistant samples of both patterns of the antenna type to simulate need to be provided by the user. In order to estimate the antenna gain in an arbitrary direction, the 3D antenna pattern interpolation method described by Leonor et al. [36] is applied. It is based on determining the four closest gain values on the principal planes and summing them up weighted proportionally to their contribution. This way, the three-dimensional antenna gain in the required direction can be estimated.

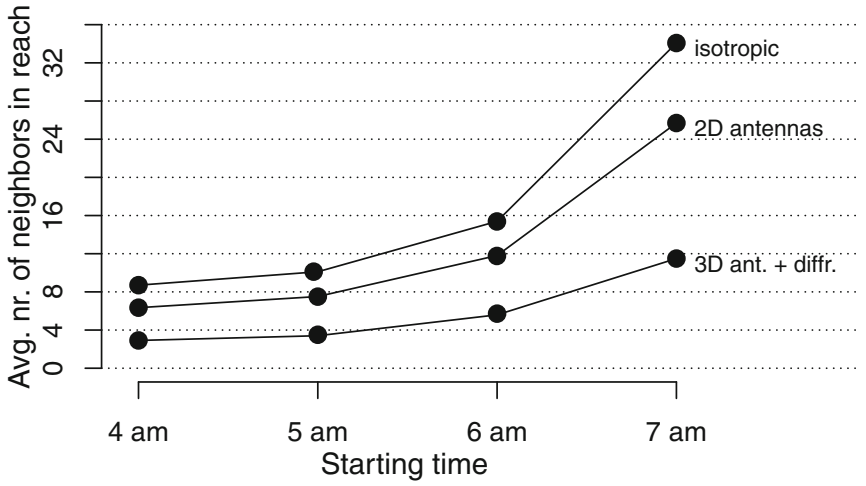



**Fig. 6.9** Average number of neighbors in reach when simulating the LuST scenario with and without 2D antenna patterns as well as 3D antenna patterns (including diffraction effects) [8]

As a matter of course, the assignment of 3D antenna patterns only makes sense if the whole environment of the scenario under investigation itself is modeled in a three-dimensional way. This means that the underlying road network has to include $z$-coordinates and that this additional 3D data also has to be exchanged between SUMO and OMNeT++, where it can be used for the three-dimensional antenna model.

Note, however, that 3D antenna patterns are not the only aspect that needs to be considered for a sufficient three-dimensional simulation of VANET scenarios: Brummer et al. [8] demonstrate that diffraction effects caused by surrounding terrain and other vehicles in the LOS should not be neglected either. Figure 6.9 demonstrates the impact that considering both 3D antenna patterns and terrain has on a simulation measuring the average number of neighbors in reach of a car. It shows substantially differing numbers for the three setups independent of all simulated starting times (and thus traffic densities).

In conclusion, it can be said that antenna characteristics (as well as diffraction effects in the 3D case) should be taken into account for more realistic and reliable results. The means to achieve that are readily available in the Veins framework.

### *6.2.4 Unit Testing in Veins*

Automated testing has become a central element of modern software development. In a world of rapidly changing requirements and short development cycles, a quick and repeatable assurance of code correctness is essential. Automated testing can

provide such assurance by running suites of programmed tests. Each test calls a portion of the original code and compares the results (and in some cases side-effects) to *reference values* embedded in the test. If all tests pass and the test suites cover all (or a large enough portion) of the original code, one can be assured that the code behaves as expected. If some tests fail, one can gain hints about which part of the code is not behaving as expected by observing which tests fail and which portion of the code they call.

The whole process of running a test and evaluating its results can nowadays be integrated into software version control and development workflows to support continuous integration.

Veins users usually implement models of algorithms or protocols to conduct research. While this is not the same as releasing a software product to end users, asserting correctness of the software is just as important in this domain. Before being able to rely on data generated from a simulation (or, indeed, publishing findings based on it), the author has to be confident that all models of the simulation behave as expected. Typically, this is done by comparing measurements recorded from the simulated model to reference data obtained from analytical models or from conducted real-world measurements.

This approach treats the model as a single large black box. Only behavior that is observable from the outside is compared to reference data. While this approach is useful to verify the overall correctness of the model, it is hard to cover the model's complete behavior. For example, there may not be enough reference data for all use cases or there may be mechanisms inside the model that are hard to verify from the outside. Finally, manually comparing the model with reference data is a cumbersome process that takes time and may be prone to errors due to its repetitive nature.

Aside from manual result comparison, Veins and OMNeT++ have provided three more automated testing mechanisms for a while now.

The first one is a simple regression testing approach, already described in Chap. 1. After a simulation finishes, OMNeT++ can output its *fingerprint*, a hash value of its defining characteristics (such as its event trace). Later fingerprints can then be used to verify that changes in the code did not affect how the simulation behaves.

The second testing mechanism is to simply run a simulation model that itself contains calls into model code and assertions to check the results. Veins uses this in its `TraCITestApp` to check if some basic interactions with the SUMO traffic simulation lead to expected results. This approach shares the pros and cons of the general usage of assertions within model code: preconditions and postconditions within model code can easily be checked and are straightforward to write. However, the approach is unstructured and not well suited for testing an entire simulation model. Many checks have to be integrated into a single simulation scenario and may depend upon each other. The whole simulation kernel has to be loaded which eventually increases runtime and complexity. Finally, there is no support for established testing tools, e.g., for automation, coverage reporting, or debugging.

The third mechanism is the OMNeT++ **opp_test** tool.[7] It can run tests in a managed environment similar to the execution environment of the OMNeT++ simulation kernel. Message creation and sending as well as result recording and other OMNeT++ utilities are available just like during simulation execution. Such an environment is hard to set up manually when using generic testing facilities. Tests are completely encapsulated into single test files and run by the **opp_test** tool. Correctness can be ensured by observing the successful termination of the simulation and by validating simulation output of file streams, such as result files and standard output or standard error. All of this makes **opp_test** the prime tool for testing OMNeT++, its modules, and code that is tightly integrated with (or relying on) OMNeT++ mechanisms, such as messages and channels. For everything that is not touching the OMNeT++ simulation kernel or library, however, **opp_test** is not the most straightforward tool to use. The test file format introduces unnecessary overhead—and having to find all values to check against in file streams is cumbersome.

Thus, for testing plain C++ code, more generic unit testing frameworks provide a better solution. A very popular example of such frameworks is Catch2.

Catch2 is a powerful C++ unit testing framework that facilitates writing, running, and evaluating unit tests for C++ code.[8] Tests are written in plain C++ (with some macros), compiled into an executable, and run by a built-in runner application. This runner allows control of the way tests are run, e.g., output verbosity and format (e.g., for continuous integration services). It can also limit a run to subsets of the test suite via tags to speed up execution time or automatically spawn a debugger on failing tests. As it is easy to learn, powerful in its capabilities, and available under a nonrestrictive license (i.e., the Boost Software License Version 1.0), it is an ideal framework to test Veins code that does not touch the OMNeT++ kernel or library.

Limiting tests to plain C++ code may appear to be a restriction, but it is actually an opportunity for improving the code design. When implementing models of algorithms and protocols using Veins, ideally only a small fraction of the code has to be written specifically for OMNeT++. Algorithms can easily be expressed as pure C++ functions or classes—and even protocol implementations can be written more cleanly if they do not rely on the concrete messaging model employed by the OMNeT++ simulation kernel. Code written in such a manner contains fewer external dependencies and moving parts. Especially the ownership model of OMNeT++ messages (which heavily relies on passing raw pointers) is contrary to C++ best practices of high-layer application development and a common source of errors. Integration can then happen in a thin layer of code that implements OMNeT++ modules, channels, or messages and, e.g., could be tested with the **opp_test** tool. This approach results in code that is much easier to test and also much easier to debug (as models can be executed without a full simulation environment) and to port to other simulators or platforms at the same time.

[7] https://www.omnetpp.org/doc/omnetpp/manual/.

[8] https://github.com/catchorg/Catch2.

In Catch2, tests are implemented in C++ files (see Listing 6.3) which only have to include a single header file. As discussed, these files are compiled individually and linked together, which also includes a special runner program that is generated by Catch2. The result is a plain binary that can be executed to run the contained tests.

Since Veins 4.7, this process has been automated in the Veins_Catch subproject. The subproject contains a *Makefile* that automatically builds the test binary from C++ files found in its source directory. It dynamically links the file to the shared library compiled from the original Veins code, so that both can be built individually. This also means that the original Veins code is fully independent of the test code. The test code, on the other hand, only needs to include header files from Veins code as if it were a part of Veins itself.

In addition, all the components (including the test runner) are compiled individually and only have to be recompiled if changed. As a result, build times stay short, which benefits frequent testing and development styles like Test-Driven Development (TDD).

In order to run the tests, one only has to execute the **veins_catch** binary produced by the *Makefile* (given that it and Veins itself have been successfully compiled). The binary provides a number of command line switches to control how and which tests are run. For example, `-s` provides detailed output even for successful tests and `-b` spawns a debugger in case of an error or failed test. The names or tags of tests to be run can be given as command line arguments. See the Catch2 documentation or run it with the `-v` switch for more information.

New tests can be added to existing or new C++ files in the *src* directory within the Veins_Catch subproject. Ideally, every unit (e.g., class or set of functions) should get its own file, mirroring the structure of the original Veins code to some degree. Each such test file first has to include the Catch2 header file (*catch/catch.hpp*) and then any headers of Veins components it wants to test against. Include paths are already configured in a way such that test code can include headers from Veins code as if it was a part of Veins itself. However, if new libraries or dependencies are introduced to Veins (in a way that affects header files), the configuration of Veins_Catch has to be adapted in the same way.

Tests can be written in two styles: normal and Behavior-Driven Development (BDD) style. The former is faster to type, the latter is more expressive in terms of debug output and test case structure. In any case, each individual test case (either stated as a `TEST_CASE` or a `SCENARIO`) gets a description text. Within such a test case, one can write arbitrary C++ code to set up the test. Assertions are then added via the `REQUIRE` macro (there is in fact a whole family of macros to cover a wide range of use cases). It is important to always add at least one assertion to each test case, otherwise it might not be run. Sample test cases and invocation of the unit tests are shown in Listings 6.3 and 6.4.

**Listing 6.3** Sample test case written in Catch2 in the Veins_Catch subproject

```
#include "catch/catch.hpp"
#include "veins/modules/mobility/traci/TraCICoordinateTransformation.h"

using Veins::TraCICoordinateTransformation;
using Veins::TraCICoord;
using OmnetCoord = TraCICoordinateTransformation::OmnetCoord;

SCENARIO( "coordinates can be transformed", "[netbound]") {
   auto o1 = OmnetCoord(2414.90142, 1578.44161, 0.0);
   auto t1 = TraCICoord(646854.991, 5493242.54);

   GIVEN( "The boundaries from a scenario" ) {
      TraCICoordinateTransformation nb{ {644465.09, 5491786.25},
            {647071.55,5494795.98}, 25 };

      THEN( "omnet coords correctly translate to traci coords" ) {
         auto t2 = nb.omnet2traci(o1);
         REQUIRE( t2.x == Approx(t1.x) );
         REQUIRE( t2.y == Approx(t1.y) );
      }
      THEN( "traci coords correctly translate to omnet coords" ) {
         auto o2 = nb.traci2omnet(t1);
         REQUIRE( o2.x == Approx(o1.x) );
         REQUIRE( o2.y == Approx(o1.y) );
      }
   }
}
```

**Listing 6.4** Sample invocation of test cases in the Veins_Catch subproject

```
veins/subprojects/veins\_catch% ./configure
Creating Makefile in veins/subprojects/veins_catch/src...
veins/subprojects/veins\_catch% make
Creating binary: src/veins_catch
veins/subprojects/veins\_catch% ./src/veins_catch
All tests passed (4 assertions in 1 test case)
```

### *6.2.5 Simple Timer Management*

A common use case in many protocols is the handling of timers, that is, doing something and—after a certain time interval elapsed—doing something else, possibly repeatedly. OMNeT++ offers the concept of *Self-Messages* to support this use case: any simulation module may schedule an event to be delivered to itself (using the `scheduleAt` method), annotating its event handler with code to treat this special event as expiration of a timer. Commonly, users create such events in a module's `initialize` method, schedule them in some user-defined method, and handle them in a module's `handleMessage` method.

At the scale (regarding number of timers) needed for many advanced protocols, however, this way of modeling timers has a number of drawbacks for code complexity. Because creation, scheduling, and handling of timeouts is split across multiple methods, data must be communicated in the events themselves (commonly found patterns subclass from `cMessage` to achieve this). Further, boilerplate code

needs to be included with every handler to free memory or re-schedule repeated events, depending on whether the timer is a one-shot or a repeating one.

Starting with Veins 4.7, the model library includes a utility class *TimerManager* to ease writing timers. It supports users needing to write timers in two respects:

- it takes care of all memory management associated with OMNeT++ events; and
- it enforces robust, modern coding standards by relying on C++11 lambda constructs (or, indeed, any `std::function`) for passing data to callback handlers.

To use this functionality, all an OMNeT++ module needs to do is create a private instance of the `TimerManager` class and pass received events to it (by introducing a small chunk of code in its `handleMessage` method). Timers can then be introduced by calling the `create` method of this private instance, passing it an object containing a lambda to execute when the callback fires. This lambda can, of course, bind any local variable (or a reference thereto) that needs to be available in the callback.

Listings 6.5 and 6.6 illustrate the use of the `TimerManager` class by way of a simple example. Though (for the single timer taking a single integer value demonstrated in this example) the overhead in terms of code that needs to be written is identical to a solution using raw OMNeT++ events, it is easy to see that this overhead is now simply a constant—independent of how many timers need to be managed by a protocol implementation.

**Listing 6.5** Content of the *TimerExample.h* file

```
#include "veins/modules/utility/TimerManager.h"

class TimerExample : public cSimpleModule {
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    // create a private instance of the TimerManager
    Veins::TimerManager timerManager{this};
};
```

**Listing 6.6** Content of the *TimerExample.cc* file

```
Define_Module(TimerExample);

void TimerExample::initialize() {
    int n = intuniform(0, 255);
    // example: remind ourselves about the value of n in 500ms from now
    auto callback = Veins::TimerSpecification([this, n](){
        EV << "value of n was " << n << std::endl;
    });
    timerManager.create(callback.oneshotIn(SimTime(500, SIMTIME_MS)));
}

void TimerExample::handleMessage(cMessage *msg) {
    // allow TimerManager to handle any timer events
    if (timerManager.handleMessage(msg)) return;
    // regular handleMessage follows...
}
```

Naturally, the `TimerManager` instance offers not just a method to create, but also to cancel timers—and timers can be both one-shot and repeating (either in a given time interval or for a given number of repetitions).

## 6.3 Use Cases

In this section, we present two practical use cases for Veins. We give insights on the simulation of platoons (cf. Sect. 6.3.1) and intersection scenarios (cf. Sect. 6.3.2).

### *6.3.1 Simulation of Platoons*

Cooperative driving and automated car following (or platooning, illustrated in Fig. 6.10), although not a new idea, is now an active research topic due to the ever-increasing demand for highly safe and sustainable transportation. In brief, the idea of platooning is to form road trains of vehicles—where one vehicle leads the group and others autonomously follow it. The *follow distance* should be small, much shorter than the *safety distance* maintained by human drivers. A close following gap improves infrastructure utilization, as it reduces the portion of road wasted for the safety distance. With an improved utilization comes a reduction of traffic congestion, resulting in a more sustainable transportation infrastructure. In addition, a distance in the order of a few meters reduces the air drag, lowering fuel consumption and thus emissions. Finally, autonomously driven vehicles can improve safety: more than 90% of road accidents are due to human errors [11].

Platooning is becoming technologically feasible, as witnessed by the projects working on this topic and realizing successful Field Operational Tests (FOTs), such as SARTRE, PATH, KONVOI, COMPANION, and PROMOTE-CHAUFFEUR [7, 30, 33, 35, 51]. Before the actual market introduction, however, platooning should be tested in large scale settings to understand to which extent platooning technologies and solutions would provide their expected benefits. In this setting (i.e., with tens or hundreds of vehicles) FOTs are simply infeasible. The solution is thus to resort to realistic simulations and this is what PLEXE has been designed for [47, 48].

PLEXE is a Veins extension designed for the analysis of platooning systems from different perspectives. From a low-level perspective, it enables the analysis

**Fig. 6.10** Screenshot of a platoon simulated in Veins

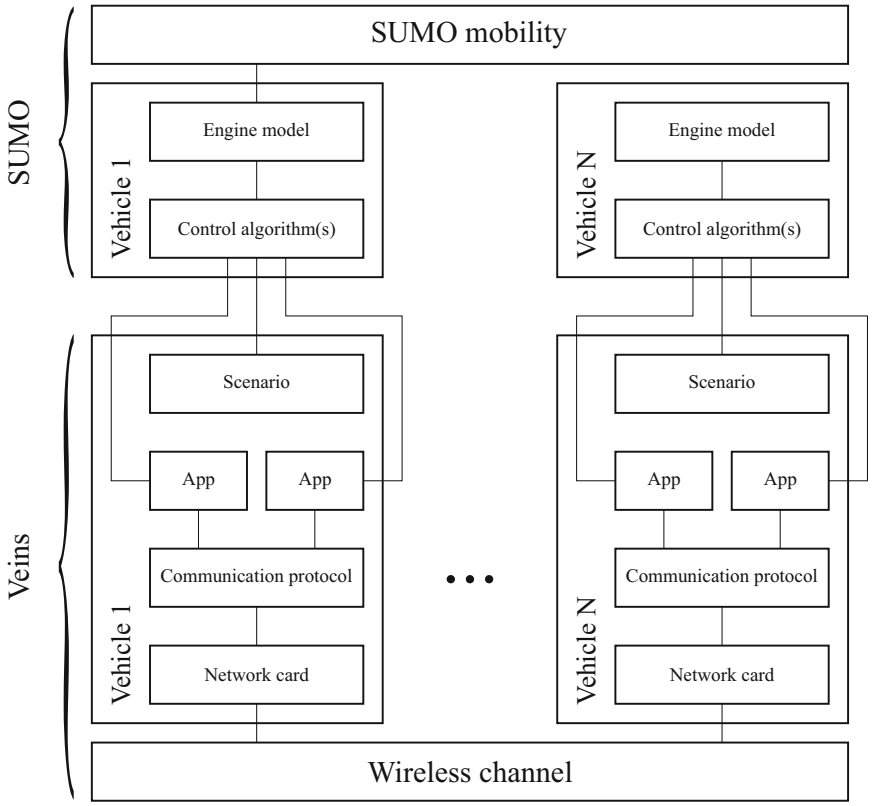



**Fig. 6.11** High-level architecture of PLEXE components

of cooperative control systems under realistic vehicle dynamics and network conditions. This is especially useful to understand the impact of network impairments on the performance of the control system, including heterogeneous vehicles in the analysis. From a high-level perspective, PLEXE permits to design, implement, and test platooning maneuvers, as well as to analyze the impact of different strategies on traffic efficiency.

Figure 6.11 shows the high-level architecture of PLEXE. It does not only extend Veins, but also SUMO:

- On the SUMO side, autonomous control algorithms and vehicle dynamics are implemented.
- On the Veins side, users can develop protocols and applications which take high-level decisions on vehicles' behavior.

On the SUMO side of PLEXE, the difference between a "standard" SUMO simulation and a simulation of a *cooperative* driving system is mobility modeling. SUMO is designed for the simulation of transportation systems with a special

focus on human traffic. Vehicles behave as dictated by *car-following models* which decide, for each time-step, what a vehicle should do depending on its surrounding environment, including other vehicles, intersections, traffic lights, etc. Standard SUMO models such as the Intelligent Driver Model (IDM) [60] or the Krauss model [32] reproduce mobility patterns which are typical of human driving. In cooperative driving, instead, decisions are taken by an automated system, which clearly behaves in a completely different manner. In this regard, PLEXE implements a new car-following model in SUMO which embeds different control systems—and that can thus behave like a cooperative autonomous vehicle.

More formally, PLEXE gives access to a set of systems called *cruise controllers*. The Cruise Control (CC) controller, as the name suggests, automatically maintains a desired speed set by the driver: this way there is no need to keep the foot on the throttle. This system is only a comfort feature, as the driver is required to manually disengage it when approaching a slower vehicle. The next step in automation, automatic braking, is provided by the Adaptive Cruise Control (ACC), which exploits a radar mounted in the front bumper to maintain a safety gap to the front vehicle, if required.

Although the ACC provides the required functionality, it does not implement platooning in the strict sense. The reason is that, due to the delays introduced by the engine driveline and the radar sensor, it cannot perform close following [41]. The safety distance maintained by an ACC is comparable to safety distances typical of human driving, and it would thus fail in providing the required benefits.

The solution to this problem comes from cooperation, i.e., by sharing information through a wireless link to implement a Cooperative Adaptive Cruise Control (CACC) [1, 20, 38, 40, 42, 45] (how this information is shared via the wireless link is modeled in the Veins side of PLEXE, described later in this section). A CACC can have a huge performance improvement with respect to an ACC as communication overcomes the limitations of sensor-based systems. As an example, exploiting a wireless link, the leader can communicate with all its members simultaneously, while a front-mounted radar is only capable of providing information about the preceding vehicle. In addition, any vehicle can share intended actions which will be executed in the near future: a radar can only sense an event after its occurrence.

In essence, the PLEXE car-following model in SUMO makes it easy for users to implement cruise control algorithms. PLEXE already provides some sample implementations, i.e., the ACC defined in [41] and the CACCs designed in [40, 42, 45]. The software is in continuous development and newly developed control systems are announced on the official website.[9]

In addition to the control algorithms, PLEXE models engine characteristics and vehicle dynamics. The control system computes a desired acceleration which needs to be realized by the vehicle. This process, however, requires a certain amount of time, the *actuation lag*, due to the engine driveline or to the braking system. This can be properly taken into account, increasing the realism of the analysis and

[9] PLEXE website: http://plexe.car2x.org.

the trustworthiness of the results. PLEXE provides two sample implementations: a simple but widely assumed first order lag (i.e., a first order low-pass filter) as well as a realistic engine model which takes into account engine torque curve, gear ratios, vehicle mass, aerodynamic characteristics, etc. Describing these models is out of the scope of this chapter. The interested reader can find a detailed mathematical description of the models (as well as of the control algorithms) in [46].

We now turn to the Veins side of PLEXE. On the Veins side of the simulation, each vehicle has a corresponding network node implementing communication protocols, applications, and scenarios (see Fig. 6.11). Each module can influence the behavior of its corresponding vehicle (or retrieve data about it) using the extension of the TraCI Application Programming Interface (API) provided by PLEXE.

The scenario module implements the high-level behavior of the vehicle. Two basic examples included in the online tutorial are the sinusoidal and the braking scenarios. In the first, the scenario continuously changes the leader speed to analyze the behavior of the control system under disturbance. In the second, the leader instead performs an emergency braking, coming to a complete stop.

Applications influence the behavior of vehicles as scenarios do, but they do so based on the information they receive through wireless communication. The most simple example is feeding the CACC using the data of a member of the same platoon. In this case, depending on whether the information is correctly received or not, the behavior of the vehicle changes (as the CACC computes different control actions). Another use case is the implementation of a maneuver and its corresponding protocol. In the case of a *join* maneuver, for instance, a vehicle might get instructions for joining from the leader of a platoon.

Below the application level we find communication components. In particular, we have communication protocols that implement beaconing strategies. This way it is possible to understand what happens to the control system depending on the employed data dissemination mechanism [49, 50]. As an example, the user can analyze the difference between a static beaconing approach vs. a coordinated one. Even further down the stack, we find the network card and the wireless channel models that are included in the standard Veins release. They provide the necessary level of realism for IEEE 802.11p-based V2V communication.

PLEXE's structure on the Veins side is meant for defining the base concepts and to ease the development process. It also enables users to define their own application/communication structure, providing them with the maximum possible flexibility.

### *6.3.2 Communication on Intersections*

In May 2018, the European Commission announced that it wants to reduce the number of fatalities per year on European roads by 2050 to nearly zero. Beside passive safety measures (e.g., advanced seat belts, improved safety glass) the commission proposed different kinds of active safety measures (often called Advanced Driver

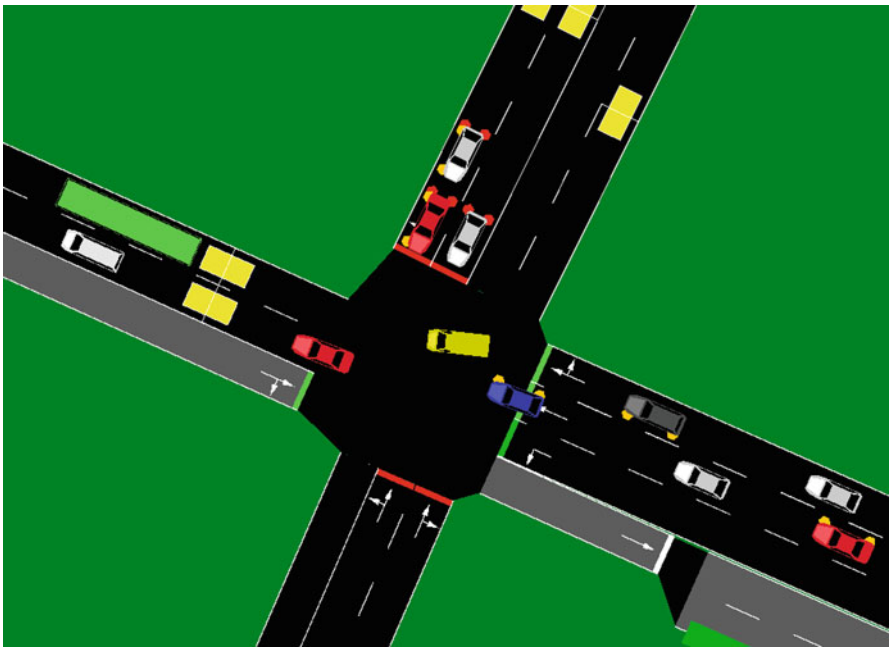


**Fig. 6.12** Screenshot of an intersection simulated in Veins

Assistance Systems (ADAS)), which aim to support drivers and prevent accidents. The envisioned safety features of future vehicles include advanced emergency braking, intelligent speed assistance, and lane keeping assistance.

Some of the safety-relevant ADAS do already exist and use various sensor technologies to assess the situation. However, the current systems are limited by their sensors to visual range. Using inter-vehicle communication, sensor data can be distributed among vehicles outside one another's field of view. One prominent example is Intersection Assistance Systems (IAS) which rely on location and movement information. Veins is a natural fit for simulating communication while vehicles are approaching an intersection (illustrated in Fig. 6.12) in a potentially dangerous situation. SUMO, on the other hand, is designed to simulate collision-free traffic, which makes it a less natural fit. Its car-following models are designed to be collision-free, i.e., vehicles approaching an intersection will never have a crash nor get into a potentially dangerous situation. However, starting with SUMO version 0.20.0, it is possible to turn off different safety checks of the car-following models. Hence, simple crash situations can be simulated by letting two vehicles start at the same time and distance to an intersection. In addition, the time when safety checks are disabled can be varied and hence a wider variety of crash situations simulated. This is possible for all implemented car following models.

Several measurements on how drivers approach intersections can be found in the literature [5]. A comparison of existing car-following models (e.g., the Krauss

model [32] or the IDM [60]) quickly reveals that the IDM better reflects human behavior when approaching an intersection [28]. Note also that default simulation time steps (in the magnitude of seconds) for data exchange between SUMO and OMNeT++ will not allow to sufficiently model such complex situations. Depending on the vehicular safety application under investigation, simulation time steps between 1 and 100 ms will be reasonable. For a detailed analysis of the simulation time step, we refer the reader to the literature [26].

In the following, insights on how Veins can be used to research safety metrics and situation-aware communication for IAS are shared.

Other than typical metrics of network behavior (like latency or load) and typical metrics of road traffic behavior (like emissions or travel time [54]), the primary metrics for traffic intersection must assess criticality. Metrics that assess the criticality of driving situations are called *safety metrics*. The criticality of a situation can only be estimated when driving information of all involved vehicles is available. The information to estimate the risk depends heavily on the situation, but might include: the exact geographical position, the driving direction, the speed, the acceleration, the planned route, or even typical driving behavior of the current driver. Please note that most of these parameters can be accessed in Veins directly or by an extension of the data exchange interface (TraCI) between SUMO and OMNeT++.

Finally, it is of course important to detect crash situations at intersections. This feature is implemented in Veins, which also enables the result recording of interesting simulation data directly in OMNeT++.

In the following, a closer look on intersection scenarios is presented, i.e., a possibility to estimate the likelihood of a crash at an intersection is explained. For a detailed description, we refer the reader to the literature [29]. The considered information for two vehicles $A$ and $B$, which are approaching an intersection, is as follows:

- distances $d_A$ and $d_B$ reflecting the distance to the intersection of trajectories,
- speeds $v_A$ and $v_B$, and
- the maximum acceleration $a_{\max}$ and the maximum deceleration (negative) $a_{\min}$.

The values of $a_{\min}$ and $a_{\max}$ would, of course, be different for each vehicle, but the vehicle-dependent indices are omitted for simplicity.

The intersection collision probability can be estimated by considering all possible driver behaviors (called trajectories) of approaching vehicles. A trajectory is a feasible function of time that satisfies the constraints:

$$\mathcal{T}_A(t_0) = d_A, \quad \dot{\mathcal{T}}_A(t_0) = v_A, \quad a_{\min} \leq \ddot{\mathcal{T}}_A(t) \leq a_{\max}. \tag{6.4}$$

All possible future trajectories are denoted as $\mathbb{T}_A$ and defined by $\mathbb{T}_A = \bigcup \mathcal{T}_A$. Of course, this set depends on the current distance $d_A$ and speed $v_A$ as each trajectory does. In addition, it is limited by the two trajectories applying constant maximum acceleration $a_{\max}$ and constant maximum deceleration $a_{\min}$.

A crash between vehicles $A$ and $B$ happens if the bounding boxes (defined by the length and width of the vehicles) overlap. This is used to define a function

coll $(\mathcal{T}_A, \mathcal{T}_B)$, which returns 0 if no crash happens and 1 if a crash happens for the given trajectories.

The intersection collision probability $\mathcal{P}_C$ depends on the probability that two trajectories are chosen which lead to a crash. This probability function is denoted as $p(\mathcal{T}_A, \mathcal{T}_B)$. Therefore, the intersection collision probability $\mathcal{P}_C$ can be calculated by integrating over all possible trajectories and summing up the probabilities as follows:

$$\mathcal{P}_C = \int_{\mathbb{T}_B} \int_{\mathbb{T}_A} p(\mathcal{T}_A, \mathcal{T}_B) \operatorname{coll}(\mathcal{T}_A, \mathcal{T}_B) \, \mathrm{d}\mathcal{T}_A \, \mathrm{d}\mathcal{T}_B. \tag{6.5}$$

Aside from serving as an output metric of simulations, this metric can also be used to improve communication on intersection scenarios, as we describe subsequently.

Figure 6.3a on page 221 shows that Veins already provides all necessary lower layers for evaluating communication strategies. Therefore, one can directly start designing the application layer, e.g., a message dissemination algorithm, which determines parameters like the content of messages or the interval of message generation. The content of the message may include position, speed, acceleration, and heading, but also neighbor information (last received message sequence number or time) might be helpful for advanced communication strategies.

The message generation interval was subject to extensive research during the past decade (e.g., [58]). Several *congestion control mechanisms* have been proposed to keep the channel load in a reasonable and efficient range. To improve communication reliability in dangerous situations, safety metrics can be used to alter the message dissemination interval alongside congestion control mechanisms.

The intersection collision probability can be used to realize situation-aware communication for intersections. Basically, each vehicle can calculate its intersection collision probability when receiving a message from another vehicle. If the probability exceeds a certain threshold, the vehicle will temporarily lower its message dissemination interval accordingly. Hence, vehicles in a dangerous situation are trying to communicate more frequently, whereas others will automatically adapt their message intervals (which, in turn, helps to keep the channel load balanced).

Finally, proposed communication strategies (such as situation-aware communication) need to be evaluated. Basically, a detailed analysis of message arrival times (which can be recorded in OMNeT++) is sufficient. The following three metrics represent a basis for evaluating communication strategies of safety applications (refer to [26] for details):

- *Last Before Unavoidable*: the last message received and the point in time before a crash becomes unavoidable is of particular concern.
- *Worst-Case Update Lag*: the update lag measures the time between two consecutive messages. Obviously, the most critical update lag is the longest during a certain time interval before a crash happens (called worst-case update lag).
- *Unsafe Time*: when a certain update lag is required by an application, it can help to sum up all times where the update lag was not maintained.

## 6.4 Extensions

In this section, we discuss how to use Veins in simulations involving LTE networks (see Sect. 6.4.1) and in simulations involving regular Internet-centric protocols, that is, with models included in the INET Framework (see Sect. 6.4.2). A discussion of Instant Veins for classroom use and quick deployment in general (see Sect. 6.4.3) concludes the chapter.

### *6.4.1 Using LTE Models in Veins (Veins LTE)*

Combining multiple networking technologies for ITS is called *Heterogeneous Vehicular Networking*. In the context of vehicular networks these networks are often LTE- and IEEE 802.11p-based, i.e., a cellular and an short range communication network. LTE is already widely deployed, mainly for use in mobile phones—but new cars regularly come equipped with a Subscriber Identity Module (SIM). In the EU, as of April 2018, all new cars need to be equipped with the *eCall* system used to automatically call emergency services if an accident occurs. Due to the centralized nature of cellular networks, scheduling can be used to handle situations with an overloaded channel. Nevertheless, there are disadvantages: vehicles are not considered in currently deployed cellular networks, especially when it comes to periodic beaconing. Since 2017, LTE-Vehicle-to-Everything (V2X) is standardized in a first version in LTE Release 14 as an extension to LTE Device-to-Device (D2D). The standard added two additional D2D modes which specifically focus on vehicular networking, one of them requiring an evolved Node B (eNB) (Mode 3) while the other one works in a distributed manner (Mode 4). Nonetheless, there are various points of discussion and an update to LTE-V2X is included in LTE Release 15, which is scheduled for release in 2018. If the infrastructure cannot cope with the additional load of cars using the cellular network (mode 4 is only used when there is no eNB in transmission range), there is the question who pays for the necessary upgrades. Not only the infrastructure improvements need to be paid for, someone needs to finance the usage of the cellular networks. Currently this is mostly included in the price of the cars, but if more cars come equipped with cellular technology this might change. Overall cellular networks are an alternative to WLAN-based networks when it comes to vehicular networking. Nevertheless, they have their own disadvantages, so research has been conducted to use both technologies. Additionally, research is conducted on other alternatives such as VLCs and Bluetooth.

The basic idea of heterogeneous vehicular networking is to use the strengths of one networking technology to overcome the weakness of the other when used in a certain application scenario. Take long-range communication in vehicular networks as an example. Transmitting data to a distant node in an IEEE 802.11p-based network needs a connected network from the source car to the destination. If the

network is not fully connected, the data might get lost or a large delay due to the use of store-carry-forward is induced. When using cellular networks, this is not the case as long as the cars are in range of a base station, i.e., an eNB, which can handle the transmission via the backbone. Similarly, IEEE 802.11p-based networks allow to have a simpler (and potentially faster) communication between cars close to each other compared to using a cellular network where every message potentially needs to traverse the backbone. Furthermore, heterogeneous technologies can be a *fall back* mechanism if one of them is not available. This might prove useful in the initial deployment phases of connected vehicles where IEEE 802.11p will not be used widely while LTE infrastructure exists already.

When using Veins alone, cellular networks can only be simulated in a very rudimentary way by using direct communication with a small amount of delay. Therefore, various solutions exist which support and provide more complete heterogeneous simulations:

- *SimuLTE*: a framework for OMNeT++ which recently introduced experimental support to combine it with Veins [61–63].
- *Veins LTE*: a framework integrating Veins and SimuLTE resulting in a toolbox to develop algorithms exploiting both IEEE 802.11p and LTE channels [21].
- *Artery*: a framework for simulation of ETSI ITS-G5 protocols which among various others includes Veins and SimuLTE [43].

As an example of a framework providing support for heterogeneous vehicular networks, we briefly introduce the first one with support for vehicles, i.e., *Veins LTE* and its features [21] (a discussion of SimuLTE and Artery can be found in Chaps. 5 and 12, respectively). Veins LTE combines short-range communication (Veins providing IEEE 802.11p) with cellular communication (SimuLTE providing LTE).

SimuLTE is, as the name gives away, a simulation model library for LTE. It currently provides support for the major parts of LTE including base stations (eNodeBs), mobile nodes (UEs), a (nearly) complete data plane, multiple example applications, an extensive MAC layer implementation, the backbone in the form of the X2 interface [39], and various basic scheduling algorithms. The downsides of SimuLTE are that it focuses on the user plane and only covers a rudimentary control plane as well as only basic handover between base stations. Both simulation model libraries, SimuLTE and Veins, are based on OMNeT++, which allows to integrate them with each other. The focus of the integration was to include cars as nodes into the cellular network. While instantiating models from both libraries at the same time is easy due to them both using OMNeT++, there are certain issues with their network models, which have proven to be incompatible. This is especially true for the treatment of mobility. On the one hand, SimuLTE did expect a fully set up network (including all moving nodes) and did not allow nodes to enter or leave at runtime. On the other hand, Veins relies on nodes dynamically entering and leaving the simulation to simulate realistic traffic conditions. To successfully integrate Veins with SimuLTE, the whole LTE stack on User Equipment (UE) and eNB side was

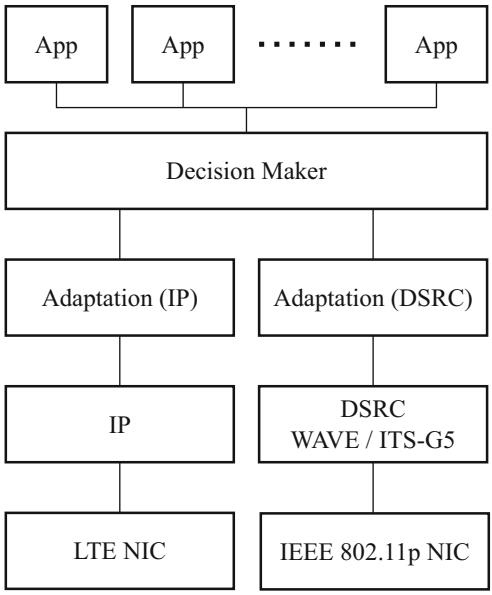



**Fig. 6.13** The heterogeneous networking stack introduced in Veins LTE [21]

modified to accommodate the addition of new vehicles and the correct removal of them during runtime.

The overall architecture of Veins LTE can be seen in Fig. 6.13. To make the development of new heterogeneous algorithms easier, a new layer was introduced—the *Decision Maker*. Residing between the application layer and the two network stacks, it adds the possibility to provide a scheduler spanning both the IEEE 802.11p and the LTE network stack. If the application has set a specific network technology, the corresponding stack is used by this module, even if the chosen network is currently not available. If no such network is set by the application, this layer allows a developer to add a decider module, which decides the network layer on which to send the packet. Such a scheduler can, for instance, choose the target network stack based on the channel load or make this decision based on the distance between sender and receiver. Furthermore, this is useful to test an algorithm with barely any configuration overhead both in an IEEE 802.11p and in an LTE setting. Below this layer is the adaption layer, which adds the necessary parameters to the heterogeneous message in order to make it possible to send it via the chosen stack. An application only needs to set the most basic parameters (e.g., destination, payload) and the rest is added or adapted by the decision maker layer. After applying the necessary attributes to the messages, they are sent via the selected networking stack.

These features, especially the integration of two networking technologies and the decision layer, allow a user of Veins LTE to focus on the development of the algorithm rather than on the underlying network.

### 6.4.2 Using INET Framework Models in Veins (Veins_INET)

Often, Veins simulations need to be combined with simulations of common Internet protocols. Conversely, systems employing Internet protocols like those of cloud services, backbone networks, or Mobile Ad Hoc NETworks (MANETs) often need to be simulated with nodes carried in road traffic. One is the domain of Veins, the other is the domain of the INET Framework, the prime OMNeT++ model library for Internet protocol simulation (see Chap. 2).

Veins hence includes an extension, `Veins_INET`, which allows models of the INET Framework to use Veins as a mobility model. Because many other simulation model libraries, in turn, rely on INET for modeling node movement, this extension also allows any of these simulation model libraries to model nodes in road traffic.

The extension is included as a *subproject*, that is, as a separate simulation project but contained in the source tree of Veins.

All that is needed is to have the target simulation project use the model libraries of all of Veins, the INET Framework, and Veins_INET. In the OMNeT++ Integrated Development Environment (IDE), this is achieved by importing all three projects into the workspace and changing the target simulation's project settings to use all three as *referenced* projects. On the command line, this is achieved by supplying the corresponding `-I`, `-L`, and `-l` switches to **opp_makemake**—as well as the corresponding `-l` and `-n` switches to **opp_run**.

In such simulations, instantiating a module `VeinsInetManager` in the network will take care of connecting to a SUMO road traffic simulation, instantiating one simulation module per road traffic participant in the SUMO simulation, and updating the modules' position information as the simulation executes (as detailed in Sect. 6.2.1). Care must only be taken that modules intended to represent road traffic participants contain `VeinsInetMobility` as their INET Framework mobility module (e.g., by configuring this in the *omnetpp.ini* file).

Figure 6.14 illustrates such a combined simulation. Note the presence of a `VeinsInetManager` module in the network (named "manager") and a mobility module of type `VeinsInetMobility` (named "mobility") in the module representing a car.

### 6.4.3 Instant Veins

Many moving parts comprise a typical Veins simulation: (1) the road traffic simulation tool SUMO; (2) the OMNeT++ simulation kernel, (3) the simulation model under study, and (4) all model libraries it is based on. For example, a simulation model of vehicles communicating with a cloud service reachable via LTE will typically rely not just on Veins, but also on INET (for Internet protocols, see Chap. 2), SimuLTE (for LTE simulation models, see Chap. 5), as well as Veins_INET (for linking them together, see Sect. 6.4.2).

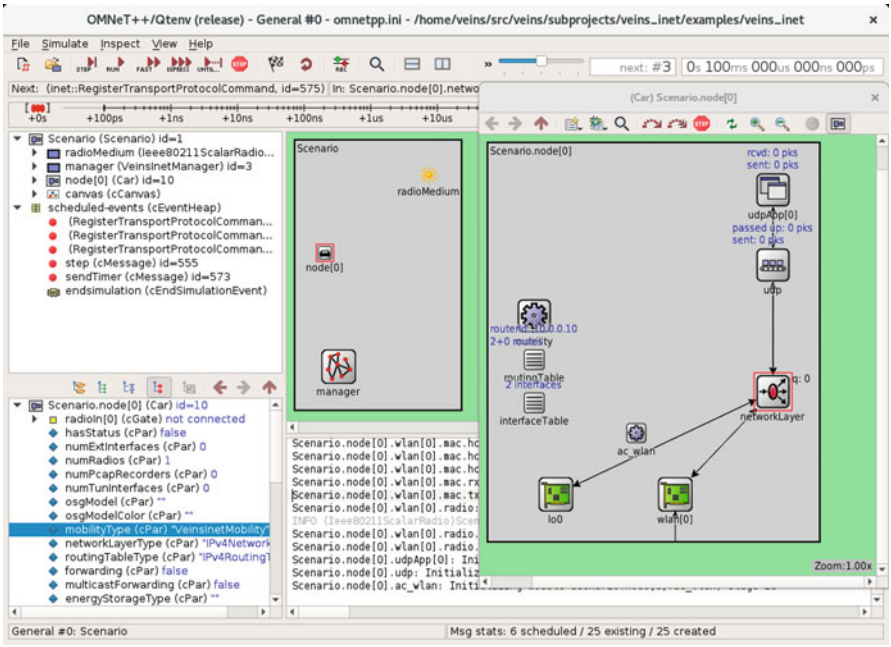



**Fig. 6.14** Screenshot of the sample simulation of Veins_INET running in the OMNeT++ GUI

Interested users have to download all of these components, compile them, and configure them for linking into a mixed simulation. In addition, care must be taken that the software versions of these tools are closely aligned, so that they remain interoperable.

This is a common source of error or delay for newcomers who want to quickly try out a novel tool—and a source of frustration for the teacher who needs to oversee the installation and deployment on hundreds of student machines every course.

Veins is therefore also made available as a virtual appliance, *Instant Veins*, which can be installed with a single click—and run independent of the operating system of the target machine. Its only prerequisite is pre-installed virtualization software, such as the open-source tool **Oracle VM VirtualBox** or any other tool that can read the Open Virtual Appliance (*.ova*) file format, such as the popular **VMware Workstation Player**. Instant Veins already contains compatible versions of all of Veins, the INET Framework, and Veins_INET to link the two (and, as a special download, also of SimuLTE)—along with OMNeT++ and SUMO.

On most machines, all that is needed is to double-click the downloaded *.ova* virtual appliance file to import it into the user's virtualization tool, from where it can then be launched directly—though some machines might have a slightly more involved installation procedure for *.ova* files, e.g., requiring the user to confirm opening the file first. After booting the virtual appliance and logging in, all needed tools can be started from the graphical shell (by clicking their respective launch

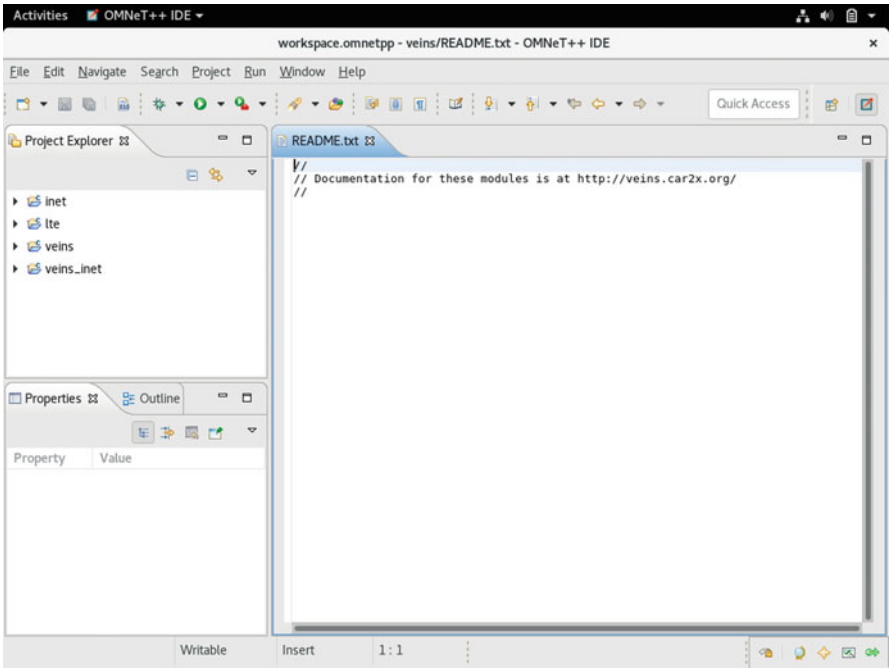



**Fig. 6.15** Screenshot of the Instant Veins virtual appliance, showing the OMNeT++ IDE after clicking on the OMNeT++ launch icon

icon). For example, after clicking on the OMNeT++ launch icon, a user is soon presented with the usual OMNeT++ IDE, which already has a workspace open that includes all four mentioned simulation libraries—ready to run (Fig. 6.15).

Instant Veins is built on fully open-source tools, most importantly Debian GNU/Linux as its base (taking care to only include re-distributable software with the base installation). This makes Instant Veins particularly useful in the classroom: Aside from getting students up and running within as little as a minute, the virtual appliance file can be freely shared with and among students.

**Acknowledgements** The authors are grateful to the community surrounding Veins, the many people who keep contributing their time and smarts to its continuous improvement. We particularly acknowledge the research labs at Univ. Paderborn, Univ. Erlangen-Nuremberg, Univ. Trento, TUMCREATE Singapore, Univ. Sydney, UCLA, Univ. Innsbruck, Univ. Luxembourg, TH Ingolstadt, Fraunhofer, TU Berlin, Carnegie Mellon University, and the German Aerospace Center.

The author D. Eckhoff was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

## References


1. Ali, A., Garcia, G., Martinet, P.: The flatbed platoon towing model for safe and dense platooning on highways. IEEE Intell. Transp. Syst. Mag. **7**(1), 58–68 (2015). https://doi.org/10.1109/MITS.2014.2328670
2. Aramrattana, M., Larsson, T., Jansson, J., Nåbo, A.: A simulation framework for cooperative intelligent transport systems testing and evaluation. Transport. Res. F: Traffic Psychol. Behav. (2017). https://doi.org/10.1016/j.trf.2017.08.004
3. Bedogni, L., Bononi, L., Di Felice, M., D'Elia, A., Mock, R., Morandi, F., Rondelli, S., Salmon Cinotti, T., Vergari, F.: An integrated simulation framework to model electric vehicles operations and services. IEEE Trans. Veh. Technol. **65**(8) (2015). https://doi.org/10.1109/TVT.2015.2453125
4. Bedogni, L., Gramaglia, M., Vesco, A., Fiore, M., Härri, J., Ferrero, F.: The Bologna ringway dataset: improving road network conversion in SUMO and validating urban mobility via navigation services. IEEE Trans. Veh. Technol. **64**(12), 5464–5476 (2015). https://doi.org/10.1109/TVT.2015.2475608
5. Berndt, H., Wender, S., Dietmayer, K.: Driver braking behavior during intersection approaches and implications for warning strategies for driver assistant systems. In: IEEE Intelligent Vehicles Symposium (IV'07), pp. 245–251. IEEE, Istanbul (2007). https://doi.org/10.1109/IVS.2007.4290122
6. Bieker, L., Krajzewicz, D., Morra, A.P., Michelacci, C., Cartolano, F.: Traffic simulation for all: a real world traffic scenario from the city of Bologna. In: SUMO User Conference 2014, pp. 19–26. Deutsches Zentrum für Luft - und Raumfahrt e.V., Berlin (2014). https://doi.org/10.1007/978-3-319-15024-6_4
7. Bonnet, C., Fritz, H.: Fuel consumption reduction in a platoon: experimental results with two electronically coupled trucks at close spacing. In: Future Transportation Technology Conference. SAE, Costa Mesa (2001)
8. Brummer, A., German, R., Djanatliev, A.: On the necessity of three-dimensional considerations in vehicular network simulation. In: 14th IEEE/IFIP Conference on Wireless on demand Network Systems and Services (WONS 2018), Isola 2000, pp. 75–82. IEEE, Isola (2018). https://doi.org/10.23919/WONS.2018.8311665
9. Codecá, L., Härri, J.: Towards multimodal mobility simulation of C-ITS: the monaco SUMO traffic scenario. In: 9th IEEE Vehicular Networking Conference (VNC 2017), pp. 97–100. IEEE, Torino (2017). https://doi.org/10.1109/VNC.2017.8275627
10. Codeca, L., Frank, R., Engel, T.: Luxembourg SUMO traffic (LuST) scenario: 24 hours of mobility for vehicular networking research. In: 7th IEEE Vehicular Networking Conference (VNC 2015). IEEE, Kyoto (2015). https://doi.org/10.1109/VNC.2015.7385539
11. Dávila, A., Nombela, M.: Sartre - safe road trains for the environment reducing fuel consumption through lower aerodynamic drag coefficient. In: 25th SAE Brasil International Congress and Display. SAE Brasil, São Paulo (2011)
12. Eckhoff, D., Sommer, C.: A multi-channel IEEE 1609.4 and 802.11p EDCA model for the Veins framework. In: 5th ACM/ICST International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2012): 5th ACM/ICST International Workshop on OMNeT++ (OMNeT++ 2012), Poster Session. ACM, Desenzano (2012)
13. Eckhoff, D., Sommer, C.: Simulative performance evaluation of vehicular networks. In: Chen, W. (ed.) Vehicular Communications and Networks: Architectures, Protocols, Operation and Deployment, pp. 255–274. Woodhead Publishing, Sawston (2015). https://doi.org/10.1016/B978-1-78242-211-2.00012-X
14. Eckhoff, D., Sommer, C.: Readjusting the privacy goals in vehicular ad-hoc networks: a safety-preserving solution using non-overlapping time-slotted pseudonym pools. Elsevier Comput. Commun. **122**, 118–128 (2018). https://doi.org/10.1016/j.comcom.2018.03.006

15. Eckhoff, D., Sommer, C., Dressler, F.: On the necessity of accurate IEEE 802.11p models for IVC protocol simulation. In: 75th IEEE Vehicular Technology Conference (VTC2012-Spring), pp. 1–5. IEEE, Yokohama (2012). https://doi.org/10.1109/VETECS.2012.6240064
16. Eckhoff, D., Halmos, B., German, R.: Potentials and limitations of green light optimal speed advisory systems. In: 5th IEEE Vehicular Networking Conference (VNC 2013), pp. 103–110. IEEE, Boston (2013). https://doi.org/10.1109/VNC.2013.6737596
17. Eckhoff, D., Sofra, N., German, R.: A performance study of cooperative awareness in ETSI ITS G5 and IEEE WAVE. In: 10th IEEE/IFIP Conference on Wireless on demand Network Systems and Services (WONS 2013), pp. 196–200. IEEE, Banff (2013). https://doi.org/10.1109/WONS.2013.6578347
18. Eckhoff, D., Brummer, A., Sommer, C.: On the impact of antenna patterns on VANET simulation. In: 8th IEEE Vehicular Networking Conference (VNC 2016), pp. 17–20. IEEE, Columbus (2016). https://doi.org/10.1109/VNC.2016.7835925
19. Emara, K.: Poster: PREXT: privacy extension for veins VANET simulator. In: 8th IEEE Vehicular Networking Conference (VNC 2016), Poster Session. IEEE, Columbus (2016). https://doi.org/10.1109/VNC.2016.7835979
20. Giordano, G., Segata, M., Blanchini, F., Lo Cigno, R.: A joint network/control design for cooperative automatic driving. In: 9th IEEE Vehicular Networking Conference (VNC 2017), pp. 167–174. IEEE, Torino (2017)
21. Hagenauer, F., Dressler, F., Sommer, C.: A simulator for heterogeneous vehicular networks. In: 6th IEEE Vehicular Networking Conference (VNC 2014), Poster Session, pp. 185–186. IEEE, Paderborn (2014). https://doi.org/10.1109/VNC.2014.7013339
22. Hassan, M.I., Vu, H.L., Sakurai, T.: Performance analysis of the IEEE 802.11 MAC protocol for DSRC safety applications. IEEE Trans. Veh. Technol. **60**(8), 3882–3896 (2011). https://doi.org/10.1109/TVT.2011.2162755
23. Heinovski, J., Klingler, F., Dressler, F., Sommer, C.: A simulative analysis of the performance of IEEE 802.11p and ARIB STD-T109. Elsevier Comput. Commun. **122**, 84–92 (2018). https://doi.org/10.1016/j.comcom.2018.03.016
24. IEEE: IEEE standard for Wireless Access in Vehicular Environments (WAVE) - multi-channel operation. Std 1609.4-2016. IEEE, Piscataway (2016)
25. IEEE: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Std 802.11-2016. IEEE, Piscataway (2016)
26. Joerer, S.: Improving intersection safety with inter-vehicle communication. Phd thesis (dissertation), University of Innsbruck (2016)
27. Joerer, S., Dressler, F., Sommer, C.: Comparing apples and oranges? Trends in IVC simulations. In: 9th ACM International Workshop on Vehicular Internetworking (VANET 2012), pp. 27–32. ACM, Low Wood Bay (2012). https://doi.org/10.1145/2307888.2307895
28. Joerer, S., Segata, M., Bloessl, B., Lo Cigno, R., Sommer, C., Dressler, F.: To crash or not to crash: estimating its likelihood and potentials of Beacon-based IVC systems. In: 4th IEEE Vehicular Networking Conference (VNC 2012), pp. 25–32. IEEE, Seoul (2012). https://doi.org/10.1109/VNC.2012.6407441
29. Joerer, S., Segata, M., Bloessl, B., Lo Cigno, R., Sommer, C., Dressler, F.: A vehicular networking perspective on estimating vehicle collision probability at intersections. IEEE Trans. Veh. Technol. **63**(4), 1802–1812 (2014). https://doi.org/10.1109/TVT.2013.2287343
30. Jootel, P.S.: SAfe Road TRains for the Environment. Final project report, SARTRE project (2012)
31. Kornek, D., Schack, M., Slottke, E., Klemp, O., Rolfes, I., Kürner, T.: Effects of antenna characteristics and placements on a vehicle-to-vehicle channel scenario. In: IEEE International Conference on Communications (ICC 2010), Workshops. IEEE, Capetown (2010). https://doi.org/10.1109/ICCW.2010.5503935
32. Krauß, S., Wagner, P., Gawron, C.: Metastable states in a microscopic model of traffic flow. Phys. Rev. E **55**(5), 5597–5602 (1997). https://doi.org/10.1103/PhysRevE.55.5597

33. Kunze, R., Ramakers, R., Henning, K., Jeschke, S.: Organization and operation of electronically coupled truck platoons on German motorways. In: Automation, Communication and Cybernetics in Science and Engineering 2009/2010, pp. 427–439. Springer, Berlin (2011)
34. Kwoczek, A., Raida, Z., Láčík, J., Pokorný, M., Puskely, J., Vágner, P.: Influence of car panorama glass roofs on Car2car communication. In: 3rd IEEE Vehicular Networking Conference (VNC 2011), Poster Session, pp. 246–251. IEEE, Amsterdam (2011). https://doi.org/10.1109/VNC.2011.6117107
35. Larson, J., Liang, K.Y., Johansson, K.H.: A distributed framework for coordinated heavy-duty vehicle platooning. IEEE Trans. Intell. Transp. Syst. **16**(1), 419–429 (2015). https://doi.org/10.1109/TITS.2014.2320133
36. Leonor, N.R., Caldeirinha, R.F.S., Sánchez, M.G., Fernandes, T.R.: A three-dimensional directive antenna pattern interpolation method. IEEE Antennas Wirel. Propag. Lett. **15**, 881–884 (2016). https://doi.org/10.1109/LAWP.2015.2478962
37. Memedi, A., Tsai, H.M., Dressler, F.: Impact of realistic light radiation pattern on vehicular visible light communication. In: IEEE Global Telecommunications Conference (GLOBECOM 2017). IEEE, Singapore (2017). https://doi.org/10.1109/GLOCOM.2017.8253979
38. Milanés, V., Shladover, S.E., Spring, J., Nowakowski, C., Kawazoe, H., Nakamura, M.: Cooperative adaptive cruise control in real traffic situations. IEEE Trans. Intell. Transp. Syst. **15**(1), 296–305 (2014). https://doi.org/10.1109/TITS.2013.2278494
39. Nardini, G., Virdis, A., Stea, G.: Modeling X2 backhauling for LTE-advanced and assessing its effect on CoMP coordinated scheduling. In: 1st International Workshop on Link- and System Level Simulations (IWSLS 2016). IEEE, Vienna (2016). https://doi.org/10.1109/IWSLS.2016.7801582
40. Ploeg, J., Scheepers, B., van Nunen, E., van de Wouw, N., Nijmeijer, H.: Design and experimental evaluation of cooperative adaptive cruise control. In: IEEE International Conference on Intelligent Transportation Systems (ITSC 2011), pp. 260–265. IEEE, Washington (2011). https://doi.org/10.1109/ITSC.2011.6082981
41. Rajamani, R.: Vehicle Dynamics and Control, 2nd edn. Springer, Cham (2012)
42. Rajamani, R., Tan, H.S., Law, B.K., Zhang, W.B.: Demonstration of integrated longitudinal and lateral control for the operation of automated vehicles in platoons. IEEE Trans. Control Syst. Technol. **8**(4), 695–708 (2000). https://doi.org/10.1109/87.852914
43. Riebl, R., Günther, H.J., Facchi, C., Wolf, L.: Artery - extending veins for VANET applications. In: 4th International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS 2015). IEEE, Budapest (2015). https://doi.org/10.1109/MTITS.2015.7223293
44. Riebl, R., Monz, M., Varga, S., Maglaras, L., Janicke, H., Al-Bayatti, A.H., Facchi, C.: Improved security performance for VANET simulations. In: 4th IFAC Symposium on Telematics Applications (TA 2016), vol. 49, pp. 233–238. Elsevier, Porto Alwegre (2016). https://doi.org/10.1016/j.ifacol.2016.11.173
45. Santini, S., Salvi, A., Valente, A.S., Pescapè, A., Segata, M., Lo Cigno, R.: A consensus-based approach for platooning with inter-vehicular communications and its validation in realistic scenarios. IEEE Trans. Veh. Technol. **66**(3), 1985–1999 (2017). https://doi.org/10.1109/TVT.2016.2585018
46. Segata, M.: Safe and efficient communication protocols for platooning control. Ph.D. thesis (dissertation), University of Innsbruck (2016)
47. Segata, M.: Platooning in SUMO: an open source implementation. In: SUMO User Conference 2017, pp. 51–62. DLR, Berlin (2017)
48. Segata, M., Joerer, S., Bloessl, B., Sommer, C., Dressler, F., Lo Cigno, R.: PLEXE: a platooning extension for Veins. In: 6th IEEE Vehicular Networking Conference (VNC 2014), pp. 53–60. IEEE, Paderborn (2014). https://doi.org/10.1109/VNC.2014.7013309
49. Segata, M., Bloessl, B., Joerer, S., Sommer, C., Gerla, M., Lo Cigno, R., Dressler, F.: Towards communication strategies for platooning: simulative and experimental evaluation. IEEE Trans. Veh. Technol. **64**(12), 5411–5423 (2015). https://doi.org/10.1109/TVT.2015.2489459

50. Segata, M., Dressler, F., Lo Cigno, R.: Jerk beaconing: a dynamic approach to platooning. In: 7th IEEE Vehicular Networking Conference (VNC 2015), pp. 135–142. IEEE, Kyoto (2015). https://doi.org/10.1109/VNC.2015.7385560
51. Shladover, S.: PATH at 20 – history and major milestones. In: IEEE Intelligent Transportation Systems Conference (ITSC 2006), pp. 22–29. Toronto (2006). https://doi.org/10.1109/ITSC.2006.1706710
52. Sommer, C., Dressler, F.: Using the right two-ray model? A measurement based evaluation of PHY models in VANETs. In: 17th ACM International Conference on Mobile Computing and Networking (MobiCom 2011), Poster Session. ACM, Las Vegas (2011)
53. Sommer, C., Dressler, F.: Vehicular Networking. Cambridge University Press, Cambridge (2014). https://doi.org/10.1017/CBO9781107110649
54. Sommer, C., Krul, R., German, R., Dressler, F.: Emissions vs. travel time: simulative evaluation of the environmental impact of ITS. In: 71st IEEE Vehicular Technology Conference (VTC2010-Spring), pp. 1–5. IEEE, Taipei (2010). https://doi.org/10.1109/VETECS.2010.5493943
55. Sommer, C., Eckhoff, D., German, R., Dressler, F.: A computationally inexpensive empirical model of IEEE 802.11p radio shadowing in urban environments. In: 8th IEEE/IFIP Conference on Wireless on Demand Network Systems and Services (WONS 2011), pp. 84–90. IEEE, Bardonecchia (2011). https://doi.org/10.1109/WONS.2011.5720204
56. Sommer, C., German, R., Dressler, F.: Bidirectionally coupled network and road traffic simulation for improved IVC analysis. IEEE Trans. Mob. Comput. **10**(1), 3–15 (2011). https://doi.org/10.1109/TMC.2010.133
57. Sommer, C., Eckhoff, D., Dressler, F.: IVC in cities: signal attenuation by buildings and how parked cars can improve the situation. IEEE Trans. Mob. Comput. **13**(8), 1733–1745 (2014). https://doi.org/10.1109/TMC.2013.80
58. Sommer, C., Joerer, S., Segata, M., Tonguz, O.K., Lo Cigno, R., Dressler, F.: How shadowing hurts vehicular communications and how dynamic beaconing can help. IEEE Trans. Mob. Comput. **14**(7), 1411–1421 (2015). https://doi.org/10.1109/TMC.2014.2362752
59. Torrent-Moreno, M., Schmidt-Eisenlohr, F., Füßler, H., Hartenstein, H.: Effects of a realistic channel model on packet forwarding in vehicular ad hoc networks. In: IEEE Wireless Communications and Networking Conference (WCNC 2006), pp. 385–391. IEEE, Las Vegas (2006). https://doi.org/10.1109/WCNC.2006.1683495
60. Treiber, M., Hennecke, A., Helbing, D.: Congested traffic states in empirical observations and microscopic simulations. Phys. Rev. E **62**(2), 1805–1824 (2000)
61. Virdis, A., Stea, G., Nardini, G.: SimuLTE - a modular system-level simulator for LTE/LTE-A networks based on OMNeT++. In: 4th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2014). Vienna (2014)
62. Virdis, A., Nardini, G., Stea, G.: Modeling unicast device-to-device communications with SimuLTE. In: 2016 1st International Workshop on Link- and System Level Simulations (IWSLS), pp. 1–8. IEEE, Vienna (2016)
63. Virdis, A., Stea, G., Nardini, G.: Simulating LTE/LTE-advanced networks with SimuLTE. In: Obaidat, S.M., Ören, T., Kacprzyk, J., Filipe, J. (eds.) Simulation and Modeling Methodologies, No. 402. Advances in Intelligent Systems and Computing, pp. 83–105. Springer, Cham (2016)
64. Wessel, K., Swigulski, M., Köpke, A., Willkomm, D.: MiXiM – the physical layer: an architecture overview. In: 2nd ACM/ICST International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2009): 2nd ACM/ICST International Workshop on OMNeT++ (OMNeT++ 2009). ACM, Rome (2009)
65. Zardosht, B., Beauchemin, S.S., Bauer, M.A.: A predictive accident-duration based decision-making module for rerouting in environments with V2V communication. Elsevier J. Traffic and Transp. Eng. (2017). https://doi.org/10.1016/j.jtte.2017.07.007

# Chapter 7
# SEA++: A Framework for Evaluating the Impact of Security Attacks in OMNeT++/INET


**Marco Tiloca, Gianluca Dini, Francesco Racciatti, and Alexandra Stagkopoulou**

## 7.1 Introduction

Computer networks are a fundamental component for a wide range of systems and applications, including sensor networks, smart environments, and critical infrastructures. These networked (systems of) systems are also exposed to several cyber and physical security attacks against their infrastructure and the communication between their components. Conducting a risk assessment process is therefore vital to identify potential threats and risks against the system, and to provide indications on how to mitigate them to an acceptable level already at design time.

However, it is not viable to fully address all possible attacks to the maximum possible extent, i.e., achieving "perfect" security, due to technical, performance, and economical reasons. Therefore, it is especially important to clearly identify the attacks with the most severe effects on the system and network infrastructure. In particular, it is important to have a clear understanding on the expected effects of successful attacks, in order to rank them based on their severity and then accordingly prioritize security countermeasures to adopt.

One approach to achieve this goal is the use of software simulation tools such as OMNeT++/INET. This is particularly convenient as it avoids the impractical alternative of conducting security experiments on real (large scale) systems, possibly already operating. In addition, simulative analysis does not require the deployment

M. Tiloca (✉) · A. Stagkopoulou
Security Lab – RISE SICS, Kista, Sweden
e-mail: marco.tiloca@ri.se

G. Dini · F. Racciatti
Department of Information Engineering, University of Pisa, Pisa, Italy
e-mail: gianluca.dini@unipi.it



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_7

of a real networked system at all, thus enabling a thorough assessment of attack effects already at the design phase. Furthermore, simulation is much more feasible to use for studying large-scale and real systems, with respect to alternative approaches based on analytical models or testbeds.

But when it comes to evaluating security attacks in networked systems, available simulation tools are often limited and inflexible. That is, the user is typically provided with a limited set of pre-defined attacks to mount against the network scenario, or instead one has to actually *implement* the desired attack execution and the adversary behavior as additional/extended modules in the simulation tool. This evidently undermines usability as well as flexibility and requires to rebuild the simulation tool in order to evaluate different attacks or even just different attack configurations.

This chapter presents SEA++, a simulation framework based on OMNeT++/INET that quantitatively evaluates the effects of security attacks against communication network scenarios in a user-friendly and flexible way. To this end, SEA++ reproduces the final actual effects of successful attacks, regardless of how they have been specifically carried out. That is, the specific way such attacks have been mounted is out of the scope of SEA++, as instead related to a separate earlier assessment of attack feasibility and likelihood.

A considerable advantage of SEA++ is that the user describes the security attacks to evaluate by using a high-level specification language. Thus, the user is not required to implement the actual adversary behavior or the actual attack execution, nor to extend or add software modules in OMNeT++/INET. Instead, the high-level attack description is interpreted and translated into an XML attack configuration file, which is finally provided as input to the SEA++ Attack Simulation Engine (ASE). The latter reproduces the effects of the attacks under evaluation, by seamlessly injecting events at simulation runtime according to the user's high-level description.

SEA++ is particularly convenient to be adopted at the last stage of the risk assessment process, in order to quantitatively evaluate the effects of successful security attacks in networked systems, provided that they have been properly modeled as simulation scenarios. This provides valuable insights on current attack activities, i.e., a tangible and understandable impact on performance and outcome of the target network, expressed through the same metric-based approach used in OMNeT++/INET. This information helps to assess the overall effects of security attacks on networks and applications. In particular, a quantitative evaluation of the effects of security attacks makes it possible to rank them according to their severity, and thus helps in selecting and tuning the most appropriate security controls to address them.

The rest of this chapter is organized as follows. Section 7.2 discusses the security risk assessment process, and briefly overviews existing supporting tools adopting a simulative approach. Section 7.3 presents our SEA++ framework in terms of its overall approach as well as its core components. Section 7.4 provides a walk-through example showing how to set up SEA++ and how to use it in a simple illustrative network scenario. Finally, Sect. 7.5 draws the conclusive remarks for this chapter and discusses potential future research directions for this topic.

## 7.2 Evaluation of Attack Impact Through Simulation

There are a lot of possible security attacks against networked computing systems. They can have different objectives, be performed at different communication layers or directly against physical nodes, and result in different effects. However, providing a security countermeasure for every possible attack is prohibitive in terms of impact on performance and cost. It is therefore vital to properly evaluate attacks to decide how to address them, and whether to mitigate, eliminate, transfer, or accept them.

This evaluation activity is referred to as *risk assessment*. It comprises the identification and evaluation of risks and risk impacts, and the recommendation of risk-reducing measures [14]. Risk assessment defines and separates the notions of threat, vulnerability, and attack. A *vulnerability* is a flaw or weakness in system security that could result in a security breach, while a *threat* represents the potential for a *threat-source* (i.e., an attacker) to exercise a specific vulnerability. An *attack* is defined as the entire process allowing a threat-source to realize a threat. Therefore, the risk assessment process practically consists of identifying and evaluating the *risks* to system security. In particular, a risk is defined as a function of (1) the *likelihood* of a given threat-source's exercising a particular potential vulnerability; and (2) the resulting *impact* of that adverse event. Of course, in order to perform attack impact analysis, a thorough *threat model* must have been previously defined. As an example, let us consider a set of attacks $S = \{A_1, A_2, \ldots, A_n\}$. Also, let us assume that a given attack $A_i$ is successfully performed with probability $P_i$ and results in an impact $E_i$. Then, the risk $R_i$ related to that attack can be computed as $R_i = P_i \times E_i$.

Analysis of the impact of an attack can be conducted either *qualitatively* or *quantitatively*. The qualitative approach makes it possible to easily prioritize the considered risks, as well as to identify what areas deserve more attention in addressing vulnerabilities. On the other hand, such a method fails to provide specific quantifiable measurements of impact magnitude. This, in turn, complicates the cost–benefit analysis of any possible security countermeasure. Conversely, the quantitative approach provides an actual measurement of the impact magnitude, and such an information can be effectively used during the cost–benefit analysis of security solutions. However, such a method might make the actual meaning of impact analysis unclear, which would force to interpret results in a qualitative way. In any case, evaluating the impact magnitude should take additional factors into account, such as attack frequency, cost associated to a single attack occurrence, as well as a subjective analysis of specific attack impact.

The presented simulation framework SEA++ supports a quantitative analysis of the impact of cyber-physical security attacks. That is, for any given attack $A_i$, it focuses on $E_i$ but does not take into account the feasibility likelihood $P_i$, which is therefore out of the scope of this chapter. In particular, SEA++ considers a worst-case scenario and it assumes that all the $n$ attacks under evaluation have been successfully performed, i.e., $P_i = 1 \; \forall \; i = \{1, 2, \ldots, n\}$. Therefore, a distinctive

feature of SEA++ is that it simulates the *effects* of security attacks by reproducing the events that such attacks generate.

SEA++ is not the only tool that strives to evaluate the impact of security attacks, and a number of different approaches have been presented so far. For instance, several alternative tools [1, 7, 19] strive to define analytical models aimed at detecting and contrasting attacks, and only afterwards rely on simulation to validate their own correctness and efficiency. Note that producing an effective analytical model of complex systems can be a considerable challenge in the first place, unless one recurs to practical simplifications that hence risk to misrepresent the network scenario and are additionally hard to be reused. Genge et al. presented *AMICI* [6], an assessment/analysis platform for multiple interdependent critical infrastructures. In particular, AMICI relies on simulation for the physical system components and an emulation testbed based on Emulab to recreate cyber components [13].

Wang and Bagrodia proposed *SenSec* [18], a framework that simulates the occurrence of security attacks in Wireless Sensor Networks (WSNs) by injecting events into real application simulators. The framework *NETA* [12] is based on OMNeT++/INET and relies on implementing *attacker nodes*, which can strike attacks when triggered at runtime through dedicated control messages. Queiroz et al. presented *SCADASim* [11], a simulation tool to test the effect of attacks in Supervisory Control and Data Acquisition (SCADA) systems.

Although it displays similarities with SenSec, NETA, and SCADASim, SEA++ is easier as well as more flexible to use and it displays a number of distinctive features. First, it is based on an off-the-shelf network simulator that was extended, but not modified, by integrating components for the processing of attack events. Good simulator tools are always the result of a large effort, and therefore any modification is preferably avoided. Furthermore, as also mentioned above, SEA++ actually focuses on the effects of security attacks rather than on their practical execution. Finally, SEA++ does not require the user to implement or customize any component of the simulation platform. As discussed in detail in Sect. 7.3, this is particularly important because it allows us to simply use simulator components, and it does not require to modify them or create new ones.

## 7.3 The SEA++ Framework

This section describes the SEA++ attack simulation framework. It provides a functional overview of its core idea as well as of its main components. In particular, Sect. 7.3.1 discusses the goals of SEA++ and the approach it adopts to achieve them. Sections 7.3.2–7.3.4 describe the three framework components. SEA++ is freely available[1] as open-source software together with a user documentation [17].

[1] SEA++ Github repository: https://github.com/seapp/seapp_stable.

### 7.3.1 Goals and Benefits

SEA++ is an attack simulation framework built on OMNeT++/INET. It enables a flexible and user-friendly quantitative evaluation of cyber and physical security attacks, as to their effects against networks and applications in terms of typical and customizable network performance indicators, such as network delay and throughput. SEA++ supports the design of secure network scenarios, as it facilitates the ranking of security attacks according to the severity of their effects, hence helping to wisely select proper security countermeasures to adopt. The framework is especially intended to expert security and network architects as end users.

The SEA++ framework focuses on the *effects* of security attacks. That is, it does not consider the likelihood of successfully mounted attacks, i.e., the likelihood of a given threat to be exploited, which is expected to be covered as an early attack feasibility assessment in the broader risk assessment process. As a consequence, SEA++ assumes that attacks to be evaluated are successfully carried out, and thus abstracts away from the specific way they are mounted. Consistently, the user is required only to model security attacks in terms of their final and practical effects against the network and application scenario.

As an example, let us consider a deception attack such as the injection of bogus network messages. Then, it is not relevant for SEA++ how an adversary becomes able to inject fake messages in the system, nor the way the actual message injection occurs. SEA++ instead focuses on assessing and quantitatively evaluating what the final effects of these messages are on the network and application after they have been successfully injected, e.g., in terms of delay and throughput variations, or unfulfilled deadlines and unreliable information gathering on the applications.

In order to achieve its goal, SEA++ relies on three fundamental steps which are also summarized in the flowchart depicted in Fig. 7.1. As a first step, the user produces a high-level description of the security attacks to evaluate. To this end, SEA++ provides an Attack Specification Language (ASL), featuring a set of statements and primitive functions to model security attacks (see Sect. 7.3.2). The

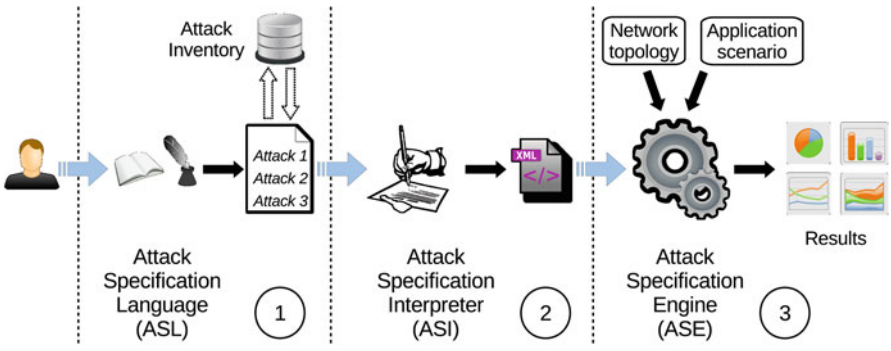



**Fig. 7.1** Overview of the SEA++ framework flowchart

attack description results in a single *adl* file, which can be possibly stored in a separate attack database for future retrieval and reuse. As a second step, the user provides the *adl* file to the Attack Specification Interpreter (ASI), i.e., a Python script that converts the high-level ASL attack description into a structured XML configuration file (see Sect. 7.3.3). In the final step, the user runs a simulation of the network scenario of interest, which has been previously modeled as an attack-free simulation scenario for OMNeT++/INET. The simulation results in a number of (raw) metric indicators, defined as expected in OMNeT++/INET and to be separately processed with analysis tools.

This third simulation step is driven by the ASE that SEA++ integrates in the INET Framework (see Sect. 7.3.4). Intuitively, the ASE parses the XML configuration file received as input, and accordingly injects additional simulation events at runtime. Such additional events reproduce the effects of the security attacks specified by the user in the original attack description. The ASE achieves this by means of two fundamental modules. On the one hand, a Local Event Processor (LEP) module is instantiated on each network node. Every LEP module is able to: (1) intercept network messages flowing through the communication stack of the related node, possibly altering their content; (2) inject new network messages or duplicate existing ones; and (3) influence the physical behavior of the network node. On the other hand, a single Global Event Processor (GEP) module is instantiated in order to interconnect the different LEP modules and enable the evaluation of complex distributed attacks.

Note that the user can indeed use vanilla versions of OMNeT++ and the INET Framework as an alternative in order to quantitatively assess the impact of security attacks. However, this would be much less flexible and convenient. In fact, the user would have to implement ad-hoc the actual attack performance and effects. Moreover, this would require to rebuild the whole framework before running a simulation. Instead, SEA++ allows the user to conduct the very same task, while more flexibly and easily describing events through the high-level ASL. Consequently, the user is not required to write source code or to rebuild the simulation framework. At the same time, the conducted simulations still produce metric-based results, i.e., just like if vanilla OMNeT++/INET was used, the impact of security attacks is measurable and processable with well-established means, and hence usable for attack ranking based on severity.

To summarize, the SEA++ framework displays a number of benefits. First, the user is not required to *implement* security attacks and their actual execution to any extent, but rather he/she simply describes their final effects by using the high-level ASL. Second, the user is not required to modify or extend the SEA++ framework either, with particular reference to the ASE module, nor the application modules and the communication modules composing the whole simulation framework. Third, the user can simply take preexisting network and application scenarios as a starting point, considering them as the attack-free baseline for the attack assessment process. Fourth, attack descriptions in the ASL are portable and combinable, as well as storable and retrievable for future reuse. Finally, the overall approach is in principle

portable to any simulation environment based on discrete events, at the cost of porting the ASE to the specific target platform.

This approach for simulative attack assessment was initially introduced for an alternative attack simulator built on *Castalia* [2] as described in [3, 4]. It was later extended for OMNeT++/INET towards SEA++ as described in [15, 16].

As a final note, it is worth clarifying that SEA++ is not an omni-comprehensive tool. That is, it should not be intended to fully cover all steps and aspects of the risk assessment process. Instead, it provides a non-exhaustive set of indications and highlights to *support* the final stages of the risk assessment. Such indications come as metric-based results produced and collected through basic means available in OMNeT++/INET, although, as discussed before, achievable in a more flexible and convenient way. Consequently, SEA++ should not be intended to provide a full connection with the overall full spectrum of security consequences on the networked system, or with final measures of the handled damage. To this end, either specific extensions to OMNeT++/INET or alternative tools are required, providing modules that implement, among other things, physical and control processes, billing and accounting processes, as well as specialized control and monitoring protocols. At the time of writing, this kind of support and extensions are out of the scope of SEA++, and the risk assessment process as a whole clearly requires to be supported by additional complementary tools.

### *7.3.2 Attack Specification Language (ASL)*

The high-level ASL allows the user to describe attacks to be evaluated in terms of their final effects. That is, the user assumes that attacks are successfully performed, regardless of how an adversary can specifically mount and execute them.

Consistently with the approach adopted by SEA++ and discussed in Sect. 7.3.1, the user describes the successful attacks as a sequence of *events* that atomically take place during the network simulation. In order to specify such events, the ASL provides a collection of primitives organized into two different sets, namely *node primitives* and *message primitives*, which account physical attacks and cyber attacks, respectively. Node primitives are described in Sect. 7.3.2.1, while message primitives are described in Sect. 7.3.2.2.

Besides, the ASL provides statements to specify the occurrence of a *list* of events described by means of message primitives. Specifically, *conditional attacks* and *unconditional attacks* define the execution of a list of message primitives, based on either the evaluation of a condition or a periodic repetition of events, respectively. Conditional attacks are described in Sect. 7.3.2.3, while unconditional attacks are described in Sect. 7.3.2.4. Finally, Sect. 7.3.2.5 discusses syntax contentions relevant for the attack description and the correct reproduction of their effects.

#### 7.3.2.1 Node Primitives

This set includes primitives used to describe physical attacks performed against network nodes. The three primitives in this category are:

- `move` (`nodeID`, `t`, `x`, `y`, `z`)
  Changes the position of node `nodeID` to the new position (`x`,`y`,`z`) at time `t`.
- `destroy` (`nodeID`, `t`)
  Disconnects the node `nodeID` at time `t`. After that, the node discards all incoming and outgoing network packets, and thus stops taking part in the network communication. Unlike when using the `disable()` primitive, the node remains part of the network scenario, i.e., it continues to run its application(s).
- `disable` (`nodeID`, `t`)
  Removes the node `nodeID` from the simulation scenario at time `t`. After that, the node cannot take part in the network communication any longer and becomes totally inoperative, i.e., it stops running its application(s).

#### 7.3.2.2 Message Primitives

This set includes primitives used to describe cyber attacks on network packets, including eavesdropping, content altering, data injection, and packet dropping. The seven primitives in this category are:

- `create` (`packet`, `field`, `content` ...)
  Creates a new packet `packet` and fills its field `field` with content `content`. A single invocation makes it possible to specify the content of multiple fields.
- `change` (`packet`, `field`, `newContent`)
  Changes the content of field `field` of packet `packet` with the `newContent`. `change()` primitive always follows the `create()` action in order to initialize the fields of the newly created packet.
- `clone` (`srcPacket`, `dstPacket`)
  Clones the packet `srcPacket` into packet `dstPacket`.
- `retrieve` (`packet`, `field`, `variable`)
  Retrieves the value of field `field` of packet `packet` and assigns it to the variable `variable`.
- `drop` (`packet`)
  Discards the packet `packet`.
- `send` (`packet`, `forwardingDelay`)
  Schedules the transmission of a selected packet `packet` produced by `clone()` or `create()` to the bottom layer after a delay of `forwardingDelay`.
- `put` (`packet`, `recipientNodes`, `direction`, `updateStats`, `forwardingDelay`)
  Puts the packet `packet` based on the `direction` argument either in the transmission (`TX`) or the reception (`RX`) buffer of all nodes in the `recipientNodes` list after a delay `forwardingDelay`, bypassing the communication channel.

#### 7.3.2.3 Conditional Attacks

This class of statements enables the execution of the list of events described through message primitives only if the specified `filter` condition is evaluated as `TRUE`. The filter condition is applied on packets intercepted at every layer of the communication stack of each node in the `list of nodes` argument. A conditional attack is described according to the following syntax, with keywords highlighted in bold.

```
from T nodes in <list of nodes> do {
  filter(<condition>)
    <list of events>
}
```

An example of a conditional attack is reported below in Listing 7.1.

**Listing 7.1** Example of a conditional attack in the ASL

```
1 list targetList = {1,2,5}
2 from 200 nodes in targetList do {
3    filter("TRA.sourcePort" == "1025" and "TRA.destinationPort" == "2000")
4       drop(original)
5 }
```

At simulation time $t = 200$ s, the involved nodes in `targetList` start intercepting packets traversing their communication stack. When a packet produces a positive match against the `filter` conditions, namely a transport-layer packet (`TRA`) with source port `1025` and destination port `2000`, the attack simulation engine discards that packet as specified by the message primitive `drop`. The adopted "dot notation" is further discussed in Sect. 7.3.2.5.

#### 7.3.2.4 Unconditional Attacks

This class of statements enables the periodical execution of the list of events described through message primitives, performed starting from an initial occurrence time '`T`' and then repeated according to a period '`P`'. Unlike conditional attacks, unconditional attacks are not related to the interception of packets by network nodes, but rather enforce the creation or duplication of network packets and their injection in the network. In particular, the user must specify the time '`t`' starting from which the attack takes place, and the occurrence period '`P`' according to which the attack has to be repeatedly reproduced over time. An unconditional attack is described according to the following syntax, with keywords highlighted in bold.

```
from T every P do {
  <list of events>
}
```

An example of an unconditional attack is reported below in Listing 7.2.

**Listing 7.2** Example of an unconditional attack in the ASL

```
1  list dstList = {1,2,5}
2  from 200 every 1 {
3     packet fakePacket
4     create(fakePacket, "APP.type", 1000)
5     change(fakePacket, "APP.name", "myFakePacket")
6     # Set TCP-related control info
7     change(fakePacket, "controlInfo.connId", 22)
8     change(fakePacket, "controlInfo.userId", 0)
9     # Set the layer where to inject the new packet
10    change(fakePacket, "sending.outputGate", "app_tcp_inf$o[0]")
11    put(fakePacket, dstList, TX, false, 0)
12 }
```

The attack starts at simulation time $t = 200$ s and is repeated every 1 s (line 2). A packet `fakePacket` is first declared as variable (line 3) and its creation as an application-layer packet (line 4). All packets have a specific type representing a simulation-based message. In the example, created malicious packets have type 1000 and only one information element, i.e., the field `name` (line 5). Furthermore, these newly created packets should be correctly filled in order to represent a complete OMNeT++ message (see Sect. 7.3.2.5). When invoking the `put()` primitive (line 11), the user specifies that the packet has to be injected in the transmission buffer `TX` of the nodes in the `dstList` defined at line 1, without any additional delay.

#### 7.3.2.5 Syntax Conventions

SEA++ is intended to cover all the layers in the communication stack, except for the physical layer. This means that the attack simulator can interact with and affect the application, transport (TCP/UDP protocol), network, and data link layer. The abbreviations used to refer the communication layers are listed in Table 7.1.

Additional reserved words used as keywords are: `RANDOM_IP`, `RANDOM_MAC`, `RANDOM_INT`, `RANDOM_SHORT`. They are used with the primitive `change()` to instruct the assignment of random values based on their type.

Regarding the examples in Sects. 7.3.2.3 and 7.3.2.4, note the use of the "dot notation" `packet.layer.field` to specify the `field` of the `packet` in the header of a `layer` (e.g., `layer.field` is translated as `TRA.sourcePort`). This implies that the user must be aware of the actual specific network protocols

**Table 7.1** ASL abbreviations for different communication layers

| Layer | Abbreviation |
|---|---|
| Application | `APP` |
| Transport | `TRA` |
| Network | `NET` |
| Data link | `MAC` |

adopted at each communication layer. Also, for each of them, the user must be aware of the packet header structure and fields, and the specific capabilities offered by the simulation platform. In particular, SEA++ relies on the objects `descriptors` provided by OMNeT++/INET in order to handle packets of a given communication layer and conveniently access and possibly alter their header fields.

In OMNeT++/INET, communication between protocol layers requires to specify additional information embedded into network packets, by means of `control info` objects. Then, in accordance with the considered "dot notation", the keyword `controlInfo.field` is used as the argument `field` in the `change()` primitive, in order to fill the `field` of `control info` objects. In particular, `controlInfo.payload` can be used in order to change the size of the packet specified as first argument in the `change()` primitive. With particular reference to the unconditional attack provided as an example in Sect. 7.3.2.4, the user must provide the specific information required by the Transmission Control Protocol (TCP) to correctly handle application-layer packets. The keyword `controlInfo` is used to provide such information (see lines 7–8).

Finally, there are two further keywords used with unconditional attacks, namely `sending.outputGate` and `attackInfo.fromGlobalFilter`. Both are used with the primitive `change()` for the argument `field`. The value provided for `sending.outputGate` for the argument `newContent` specifies in which layer the packet has to be injected. On the other hand, the value for `attackInfo.fromGlobalFilter` provided with the argument `newContent` is set to 1 for packets created specifically by the GEP (see Sect. 7.3.4). With further reference to the unconditional attack provided as example in Sect. 7.3.2.4, the value specified for `sending.outputGate` indicates that the packet `fakePacket` is injected between the application and the transport layer as an outgoing packet, i.e., the argument `newContent` is set to `app_tcp_inf$o` (see line 10).

### 7.3.3 *Attack Specification Interpreter (ASI)*

The ASI is a Python script that converts an *adl* file including the high-level attack description in ASL into an XML configuration file. The latter is then provided as input to the attack simulator, to be ultimately processed by the ASL. To ensure a successful translation of the *adl* file, the user must follow the syntax rules defined by the ASL and adopted by the ASI, with particular reference to the "dot notation" `packet.layer.field` discussed in Sect. 7.3.2.5.

The resulting XML configuration file is split into three distinct sections:

- The first section lists all the described physical attacks, each of which is composed by a single node primitive. That is, each physical attack indicates the involved node and the simulation time when the attack takes place.
- The second section lists all the cyber attacks described as conditional attacks. Each conditional attack indicates:

1. the set of involved nodes,
2. the simulation time when the attack starts,
3. the `filter` condition, and
4. the list of message primitives modeling the attack events.

- The third section lists all the cyber attacks described as unconditional attacks. Each unconditional attack indicates:

1. the simulation time when the attack starts,
2. the time period according to which the attack is reproduced, and
3. the list of message primitives modeling the attack events.

Listing 7.3 shows an example of an XML configuration file, produced by the ASI. It includes one attack for each of the three sections described above.

**Listing 7.3** Example of an XML configuration file describing three attacks

```
<?xml version="1.0"?>
<configuration>
  <Physical>
    <Attack>
      <start_time>20</start_time>
      <node>5</node>
      <action>
        <name>Disable</name>
      </action>
    </Attack>
  </Physical>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <filter>
      [:TRA.sourcePort:==:1025:]:[:TRA.destinationPort:==:2000:]:AND
      </filter>
      <action>
        <name>Drop</name>
        <parameters>packetName:original:threshold:0</parameters>
      </action>
    </Attack>
  </Conditional>
  <Unconditional>
    <Attack>
      <start_time>60</start_time>
      <frequency>0.1</frequency>
      <var>
        <name>"myFakePacket"</name>
        <value>myFakePacket</value>
        <type>STRING</type>
      </var>
      <var>
        <name>123</name>
        <value>123</value>
        <type>NUMBER</type>
      </var>
      <var>
        <name>4</name>
        <value>4</value>
        <type>NUMBER</type>
      </var>
      <var>
        <name>1250</name>
```

```
46        <value>1250</value>
47        <type>NUMBER</type>
48      </var>
49      <var>
50        <name>0</name>
51        <value>0</value>
52        <type>NUMBER</type>
53      </var>
54      <var>
55        <name>"10.0.0.3"</name>
56        <value>10.0.0.3</value>
57        <type>STRING</type>
58      </var>
59      <var>
60        <name>"app_udp_inf$o[0]"</name>
61        <value>app_udp_inf$o[0]</value>
62        <type>STRING</type>
63      </var>
64      <action>
65        <name>Create</name>
66        <parameters>packetName:fakePacket:APP.type:1001</parameters>
67      </action>
68      <action>
69        <name>Change</name>
70        <parameters>
71        packetName:fakePacket:field_name:APP.name:value:"myFakePacket"
72        </parameters>
73      </action>
74      <action>
75        <name>Change</name>
76        <parameters>
77        packetName:fakePacket:field_name:controlInfo.packetSize:value:1250
78        </parameters>
79      </action>
80      <action>
81        <name>Change</name>
82        <parameters>
83        packetName:fakePacket:field_name:controlInfo.destAddr:value:"10.0.0.3"
84        </parameters>
85      </action>
86      <action>
87        <name>Change</name>
88        <parameters>
89        packetName:fakePacket:field_name:controlInfo.destPort:value:123
90        </parameters>
91      </action>
92      <action>
93        <name>Change</name>
94        <parameters>
95        packetName:fakePacket:field_name:controlInfo.sockId:value:4
96        </parameters>
97      </action>
98      <action>
99        <name>Change</name>
100       <parameters>
101       packetName:fakePacket:field_name:controlInfo.interfaceId:value:0
102       </parameters>
103     </action>
104     <action>
105       <name>Change</name>
106       <parameters>
107       packetName:fakePacket:field_name:sending.outputGate:value:"app_udp_inf$o
              [0]"
108       </parameters>
109     </action>
110     <action>
111       <name>Put</name>
```

```
112       <parameters>
113       packetName:fakePacket:nodes:4:direction:TX:throughWC:false:delay:0
114       </parameters>
115     </action>
116   </Attack>
117  </Unconditional>
118 </configuration>
```

The XML configuration file from Listing 7.3 includes information for the ASE to reproduce the effects of the three different attacks described below using the ASL, namely a physical attack (see Listing 7.4), a conditional attack (see Listing 7.5), and an unconditional attack (see Listing 7.6).

In particular, the considered physical attack is described using the ASL in Listing 7.4, and results in lines 3–11 in the physical section of Listing 7.3. These include: (1) the simulation time when the attack occurs (line 5); (2) the involved node (line 6); and (3) the node primitive composing the attack (line 7).

**Listing 7.4** Physical attack described using the ASL

```
1 disable (5, 20)
```

The considered conditional attack is described using the ASL in Listing 7.5, and results in lines 12–24 in the conditional section of Listing 7.3. These include: (1) the simulation time when the attack starts (line 14); (2) the involved nodes (line 15); (3) the `filter` condition (line 16); and (4) the action composing the attack (line 19).

**Listing 7.5** Conditional attack described using the ASL

```
1 list targetList = {1,2,5}
2 from 200 nodes in targetList do {
3    filter("TRA.sourcePort" == "1025" and "TRA.destinationPort" == "2000")
4    drop(original,0)
5 }
```

The considered unconditional attack is described using the ASL in Listing 7.6, and results in lines 25–117 in the unconditional section of Listing 7.3. These comprise the following:

1. the simulation time when the attack starts (line 27),
2. the frequency according to which the actions of the attack are reproduced (line 28),
3. the declaration of the variables used in the attack description (lines 29–63),
4. and the actions composing the attack (lines 64–115).

**Listing 7.6** Unconditional attack described using the ASL

```
1 list attackNodesList = {4}
2 from 60 every 0.1 do {
3
4    # declare a packet
5    packet fakePacket
6
7    # create a new packet
8    create(fakePacket, "APP.type", "1001")
9
```

```
10    # fill the new packet properly
11    change(fakePacket, "APP.name", "myFakePacket")
12
13    change(fakePacket, "controlInfo.packetSize", 1250)
14
15    change(fakePacket, "controlInfo.destAddr", "10.0.0.3")
16    change(fakePacket, "controlInfo.destPort", 123)
17    change(fakePacket, "controlInfo.sockId", 4)
18    change(fakePacket, "controlInfo.interfaceId", 0)
19
20    change(fakePacket, "sending.outputGate", "app_udp_inf$o[0]")
21
22    put(fakePacket, attackNodesList, TX, FALSE, 0)
23 }
```

### 7.3.4 Attack Simulation Engine (ASE)

The ASE considers every node in the simulation scenario as an *Enhanced Network Node* module. The latter is in turn composed of: (1) an application module, that possibly includes multiple sub-modules modeling the actual application(s) running on the node; (2) an arbitrarily complex collection of communication protocols composing the node's stack; and, finally, (3) an LEP module. All sub-modules apart from the LEP can be off-the-shelf.

The LEP module is responsible for managing events related to physical and conditional attacks. In particular, the LEP module intercepts all application and network packets traversing the communication stack of a network node, as interposed between each pair of layers in the node's stack. Based on the attack description, the LEP can alter the packets' content, generate and inject new packets, or even discard them. Also, it can change the node's behavior, i.e., change its position, neutralize the node by making it inactive, or completely remove the node from the network, according to the specific events in the attack description.

Furthermore, SEA++ enables the simulation of complex attacks by means of a single GEP module. In particular, the GEP module takes care of unconditional attacks, by repeatedly executing the list of events specified in their description. The GEP module is connected to the LEP module of each *Enhanced Network Node*, hence enabling the reproduction and evaluation of complex distributed security attacks, for instance wormhole attacks.

Figure 7.2 shows the overall architecture of the ASE, with particular reference to two interconnected network nodes. In each *Enhanced Network Node*, the LEP is interposed between each pair of layers in the communication stack, hence acting as gate-bypass between each pair of modules providing the respective layers. Finally, the GEP is connected to the LEP modules of the two *Enhanced Network Nodes*.

SEA++ provides also support for Software-Defined Networking (SDN) environments [5, 9]. In particular, it considers the de facto standard *OpenFlow* [10] implemented in commercial SDN controllers and switches, and it builds on the simulation model initially proposed in [8].

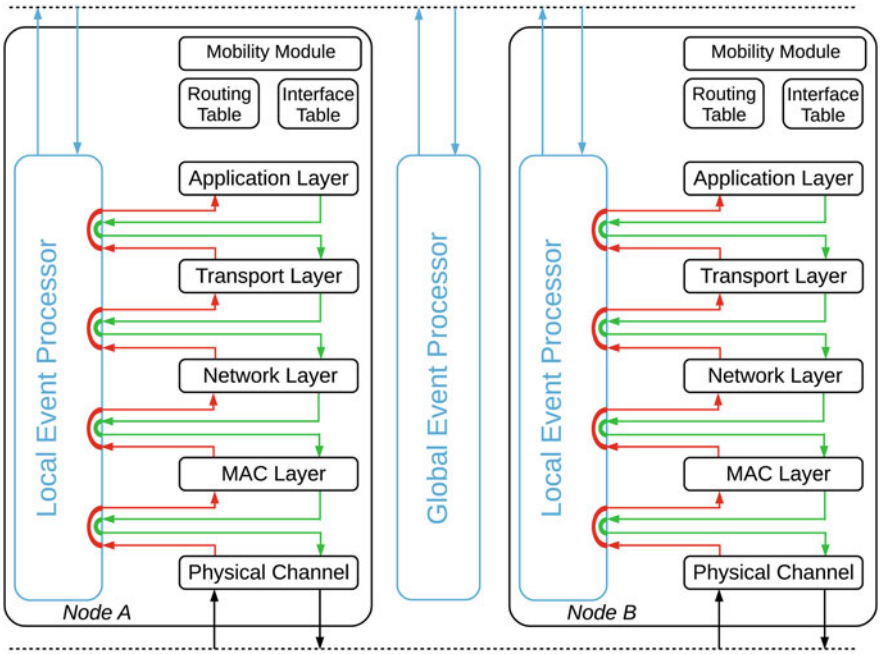



**Fig. 7.2** ASE architecture with two enhanced network nodes

Specifically, the LEP module has also been integrated into INET nodes modeling *OpenFlow* switches to enable the simulation of SDN-related security attacks. The separation of the *Control Plane* and *Data Plane* that is typical for SDN has also been modeled in the INET Framework. That is, the LEP interposes itself between the two planes, namely the data plane and the southbound *OpenFlow* interface. In this architecture, the LEP is responsible for intercepting network packets traversing the data plane, as well as *OpenFlow* messages which are exchanged between the SDN controller and the switch through the southbound *OpenFlow* interface. Figure 7.3 shows the overall architecture of an *Enhanced OpenFlow Switch Node*. Like the *Enhanced Network Nodes*, the LEP of the *Enhanced OpenFlow Switch Node* also communicates with the single GEP module. Note that SDN controllers can instead be simply represented as an *Enhanced Network Node* including a full communication stack. Further details on the SDN support in SEA++ are available in [16].

#### 7.3.4.1 Reproduction of Attack Effects

As discussed in Sect. 7.3.3, the ASI converts the high-level attack description in ASL from an *adl* file to an XML configuration file, which consists of three distinct sections covering physical attacks, conditional attacks, and unconditional attacks.

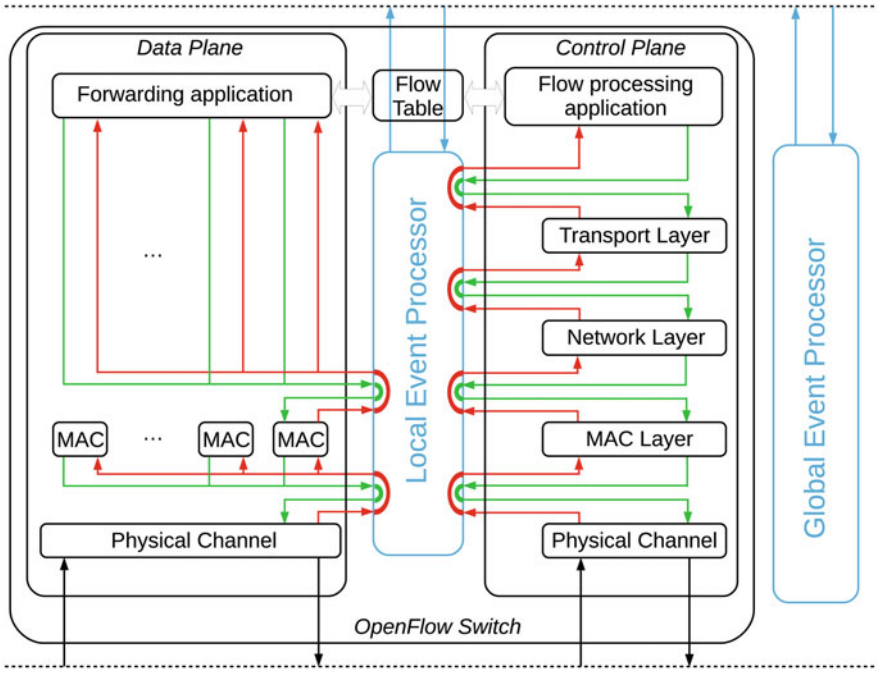



**Fig. 7.3** ASE architecture with an enhanced OpenFlow switch

At simulation startup, the ASE takes the XML file as input, parses it, and initializes three data structures separately for the three types of attack. In detail, the ASE creates the subsequently described lists, whose list elements are all chronologically ordered to reflect the respective attack's occurrence time.

- ASE creates a list $LP_n$ for each network node $n$ involved in a physical attack. In every list, each element includes the description of a physical attack.
- ASE creates a list $LC_n$ for each network node $n$ involved in a conditional attack. Again, each element in every list includes the description of a conditional attack.
- ASE creates a list $LU_n$ for each network node $n$ involved in an unconditional attack. Just like before, each element in every list includes the description of an unconditional attack.

After that, the ASE instantiates a set of timers for each network node $n$ involved in at least one of the lists defined above. In particular, each timer is associated to a specific attack involving that node. Then, the ASE starts all the timers in order to schedule the respective attack's (first) occurrence. That is, upon the expiration of a timer associated to a node $n$, the ASE retrieves the associated attack $A$ from the list where it is included, and performs the following actions:

- In case $A$ is a physical attack, the ASE executes the associated node primitive, by means of the LEP module of node $n$. Then, it removes $A$ from the attack list $LP_n$.

- In case $A$ is a conditional attack, node $n$ starts intercepting packets traversing its communication stack by means of its own LEP. In particular, node $n$ filters the intercepted packets according to the `filter` condition specified in $A$. Then, for each packet producing a positive match against the filter condition, node $n$ executes the message primitives listed in $A$. Note that the actual execution of such primitives may involve the GEP module, as well as the LEP module of other nodes than $n$.
- In case $A$ is an unconditional attack, the ASE starts executing the message primitives listed in $A$. Thereafter, the GEP repeatedly performs $A$, according to the specified occurrence period. This is practically achieved by resetting the timer associated to node $n$ and attack $A$ at the end of each repetition. Note that it is the prerogative of the GEP module to start the reproduction of unconditional attacks, then involving the LEP module of individual network nodes.

## 7.4 Explicative Example: Injection of Fake Packets

This section describes how to correctly set up the SEA++ framework. It also presents a step-by-step example on how to use the framework to evaluate the attack impact on a simple network scenario. The example presented in this section intends to provide a basic usage tutorial. It has no ambition to show how a comprehensive evaluation of attack impact can be conducted in a real-world networked system, nor to produce an attack ranking as part of a full risk assessment process.

In the considered example, we refer to a network composed of two client nodes and one server, and we specifically go through the steps to describe and reproduce an unconditional attack, in order to quantitatively evaluate its impact.

### *7.4.1 Framework Setup*

The current version of SEA++ relies on:

- a compiler for C++11,
- an interpreter for Python 2.7.6,
- the *libxml* library, and
- INET 2.6, which is based on OMNeT++ 4.x.

SEA++ has been developed and tested under Ubuntu Linux 14.04 LTS, while it is usable in older or newer versions after applying basic configuration changes. In the following, we assume that the installation and configuration of OMNeT++ 4.6 and INET 2.6 has been previously completed, and we focus on the setup of the actual SEA++ framework. The user installs and configures SEA++ according to the following steps when he/she finished installing and configuring OMNeT++/INET.

Additional details on the framework setup and configuration are available in the SEA++ user manual [17].

1. Obtain and install the *libxml* library, available at http://www.xmlsoft.org.
2. Download the SEA++ sources from the official Git repository at https://github.com/seapp/seapp_stable.
3. Extract the source archive and rename the main folder to *seapp_stable*.
4. Move to the SEA++ main folder and build the framework by running `make makefile` followed by `make`.

To build SEA++ on older Ubuntu versions, it is recommended to additionally:

1. upgrade the Python interpreter to version 2.7.6,
2. upgrade the C++compiler to version 4.7 and set it as default, and
3. change the SEA++ *makefile* to meet the requirements of the specific system's architecture.

After a successful build, the user can include additional network scenarios as a subfolder of */examples/seapp*.

## 7.4.2 Evaluation of Attack Impact

We refer to a simple network topology consisting of two clients and one UDP server connected through an Ethernet switch. In particular, we assume that an adversary has successfully compromised *client1* and uses it to inject newly generated bogus packets addressed to the server. This section describes how to: (1) describe the attack using the ASL; (2) define the network scenario in the presence of SEA++ components; and (3) evaluate the attack impact through simulation.

### 7.4.2.1 Attack Description

As a first step, the user describes the attack mentioned above by using the ASL. As shown in Listing 7.7, the attack is described as an *unconditional attack*, where new bogus packets are injected in the transmission buffer of the compromised *client1*, that then forwards them to the victim server node. Unlike with the direct injection in the reception buffer of the victim server, this makes it also possible to reproduce the actual transmission and reception of such packets in the network. Note that, as an *unconditional attack*, it is the GEP module of the ASE that takes care of the single attack events at simulation runtime (see Sect. 7.3.4.1).

**Listing 7.7** Attack description using the ASL

```
1 list attackNodesList = {4}
2
3 from 60 every 0.1 do {
4    #declare a packet
```

```
5     packet fakePacket
6
7     #create a new packet
8     create(fakePacket, "APP.type", "1001")
9
10    #fill the new packet properly
11    change(fakePacket, "APP.name", "myFakePacket")
12    change(fakePacket, "controlInfo.packetSize", 1250)
13    change(fakePacket, "controlInfo.destAddr", "10.0.0.3")
14    change(fakePacket, "controlInfo.destPort", 123)
15    change(fakePacket, "controlInfo.sockId", 4)
16    change(fakePacket, "controlInfo.interfaceId", 0)
17
18    #instructions to Global Filter
19    change(fakePacket, "sending.outputGate", "app_udp_inf$o[0]")
20    put(fakePacket, attackNodesList, TX, FALSE, 0)
21 }
```

In detail, the considered attack is carried out through the node with ID 4, namely *client1* in the simulation scenario. To this end, the list `attackNodesList` in line 1 includes the identifier of *client1*. Then, the attack starts at simulation time $T = 60$ s, and the events listed in its body are periodically repeated every 0.1 s (lines 3–21). In particular, at each loop repetition, the GEP performs the following actions.

First, the '`fakePacket`' variable of type '`packet`' is declared (line 5), and then used as first argument of the `change()` primitive to actually initialize the new packet (line 8). In particular, the new packet is specified as an application-level packet, by setting its `APP.type` field to the predefined message type 1001. This makes it possible to correctly create a `cPacket` message representing `fakePacket`.

After that, the primitive `change()` is further invoked to fill (meta-)information related to `fakePacket` and allow OMNeT++/INET to correctly handle the associated `cPacket` message (lines 11–16). In particular, the application name `APP.name` is set to `myFakePacket`, essentially to enable selective packet tracing (line 11). Also, information required to correctly dispatch the packet are specified as values of `controlInfo` subfields (lines 12–16). In particular, the packet size is set to 1250 bytes (line 12), i.e., the same size of legitimate application packets in the considered network scenario. Besides, the `controlInfo` subfields are filled with information related to the UDP transport layer, as the one immediately traversed by the outgoing (application-level) packet (lines 13–16).

Finally and consistent with the syntax conventions discussed in Sect. 7.3.2.5, a further indication is provided to the GEP module through the `change()` primitive (line 19). That is, the user specifies the stack layer in *client1* where `fakePacket` has to be injected. Then, the `put()` primitive instructs the GEP module to inject `fakePacket` in the transmission buffer of `client1` (line 20).

The described attack would continue for the whole simulation duration. This results in *client1* sending 10 additional bogus packets per second to the victim server, the intended *final effect* of the described attack.

Before proceeding with the network simulation in the presence of the described attack, the user has to convert the *adl* attack description in ASL into an XML file providing a parsable collection of attack events. To this end, the user simply runs the

ASI as shown in Listing 7.8, i.e., providing the *adl* file as input. The resulting output is the XML attack configuration file, which is provided as input to the ASE through the description of the network scenario in the *omnetpp.ini* file (see Sect. 7.4.2.2). The ASI is available as a Python script in the */seapp_stable/interpreter* directory.

**Listing 7.8** Usage of the ASI to convert the *adl* description file into the XML configuration file

```
../../../interpreter/interpreter/interpreter.py -i simple_attack.adl -o
     simple_attack.xml
```

### 7.4.2.2 Description of the Network Scenario

The user is now requested to define the topology of the network scenario. This is done by filling an OMNeT++/INET Network Topology Description (NED) file and storing it in the */examples/seapp* directory. A NED file example is shown in Listing 7.9. Note that an already existing NED file can be considered as a starting point from previous simulation scenarios and then be extended accordingly.

**Listing 7.9** Network scenario as defined in the NED file

```
1  package inet.examples.seapp.simpleTopo;
2
3  import inet.nodes.inet.StandardHost;
4  import inet.nodes.ethernet.EtherSwitch;
5  import inet.networklayer.autorouting.ipv4.IPv4NetworkConfigurator;
6  import inet.util.ThruputMeteringChannel;
7  import inet.globalfilter.GlobalFilter;
8
9  network Scenario
10 {
11    parameters:
12       string attackConfigurationFile = default("none");
13       @display("bgb=600,300");
14
15    types:
16       channel ethernetline extends ThruputMeteringChannel
17       {
18          delay = 1us;
19          datarate = 100Mbps;
20          thruputDisplayFormat = "u";
21       }
22
23    submodules:
24       configurator: IPv4NetworkConfigurator
25       {
26          @display("p=40,40");
27       }
28       globalFilter: GlobalFilter
29       {
30          @display("p=350,50;");
31       }
32       client1: StandardHost
33       {
34          @display("p=200,100;i=device/laptop");
35       }
36       client2: StandardHost
37       {
38          @display("p=200,200;i=device/laptop");
```

```
39        }
40        server: StandardHost
41        {
42           @display("p=500,150;i=device/server");
43        }
44        switch: EtherSwitch
45        {
46           @display("p=350,150");
47        }
48
49     connections allowunconnected:
50        switch.ethg++ <--> ethernetline <--> client1.ethg++;
51        switch.ethg++ <--> ethernetline <--> client2.ethg++;
52        switch.ethg++ <--> ethernetline <--> server.ethg++;
53
54        globalFilter.nodes++ <--> client1.global_filter;
55        globalFilter.nodes++ <--> client2.global_filter;
56        globalFilter.nodes++ <--> server.global_filter;
57  }
```

In particular, it is important to observe the following steps:

- Include the string parameter *attackConfigurationFile* to the network (line 12). This is initialized to `none`, but it will be overwritten later on within the *omnetpp.ini* file, when specifying the *xml* file including the attack description.
- Import the `GlobalFilter` class (line 7) providing the GEP module.
- Declare the actual GEP submodule (line 28).
- Connect the GEP module to every network node (lines 54–56).

The next step consists of defining the traffic model in the network scenario by filling the OMNeT++ file *omnetpp.ini*. An example of such an *omnetpp.ini* file is shown in Listing 7.10. During the simulation, each client node runs the application `UDPBasicApp` available in INET and sends 5 packets per second to the server node. The server node runs the application `UDPSink` available in INET, without replying back to the clients. According to this traffic model, the server receives 10 packets per second overall.

**Listing 7.10** Content of the *omnetpp.ini* file

```
1  [General]
2  debug-on-errors = true
3  sim-time-limit = 120s
4
5  network = Scenario
6
7  *.configurator.networkAddress = "192.168.2.0"
8
9  #Traffic Configuration
10 *.server.numUdpApps = 1
11 *.server.udpApp[0].typename = "UDPSink"
12 *.server.udpApp[0].localPort = 123
13
14 *.client*.numUdpApps = 1
15 *.client*.udpApp[0].typename = "UDPBasicApp"
16 *.client*.udpApp[0].localPort = 100
17 *.client*.udpApp[0].destPort = 123
18 *.client*.udpApp[0].messageLength = 1250 bytes
19 *.client*.udpApp[0].sendInterval = 0.2s
20 *.client*.udpApp[0].destAddresses = "server"
21
```

```
22 [Config Simple_attack]
23 extends = General
24 **.attackConfigurationFile = "simple_attack.xml"
```

In particular, it is important to include the configuration `Simple_attack` (lines 22–24) to specify the XML file that includes the attack description (see Sect. 7.4.2.1). This configuration extends the traffic behavior described in the `General` section by including the attack description as a parsable XML file.

### 7.4.2.3 Simulation of Attack Effects

Once the attack and scenario description are completed, the user can finally run the actual simulation in SEA++, just as it is usually done in OMNeT++/INET and typically through the native Graphical User Interface (GUI).

Figure 7.4 shows the output during the simulation runtime. This also includes the events related to the attack under evaluation, such as the injection of fake packets `myFakePacket` from the GEP module to `client1` through the wrapper message `PutReq`, as well as the following transmission to the server node as the final destination. The application `UDPSink` running on the server node additionally measures the number of received packets per second. Once the simulation is completed, the user can collect the statistics and perform post-analysis on simulation results.

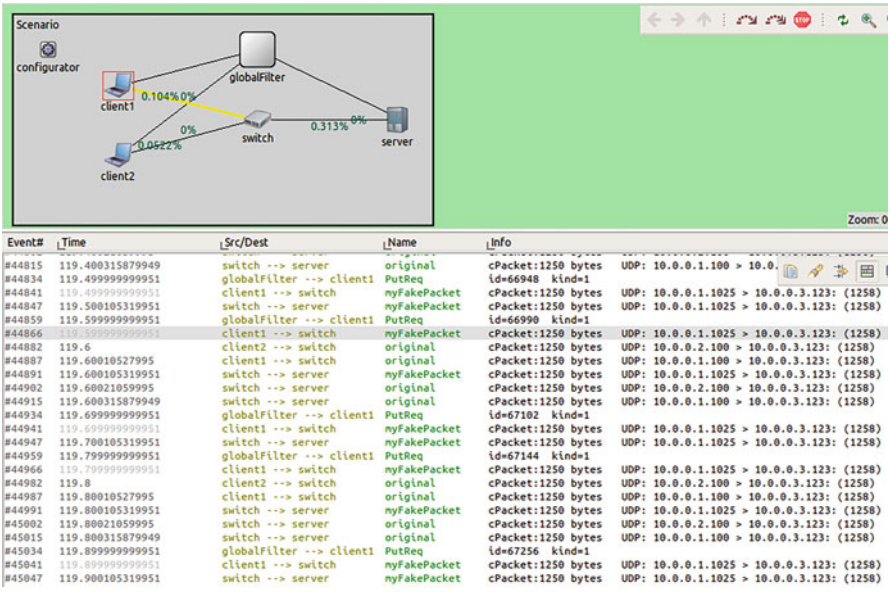



**Fig. 7.4** Simulation output on the OMNeT++/INET GUI during the injection attack

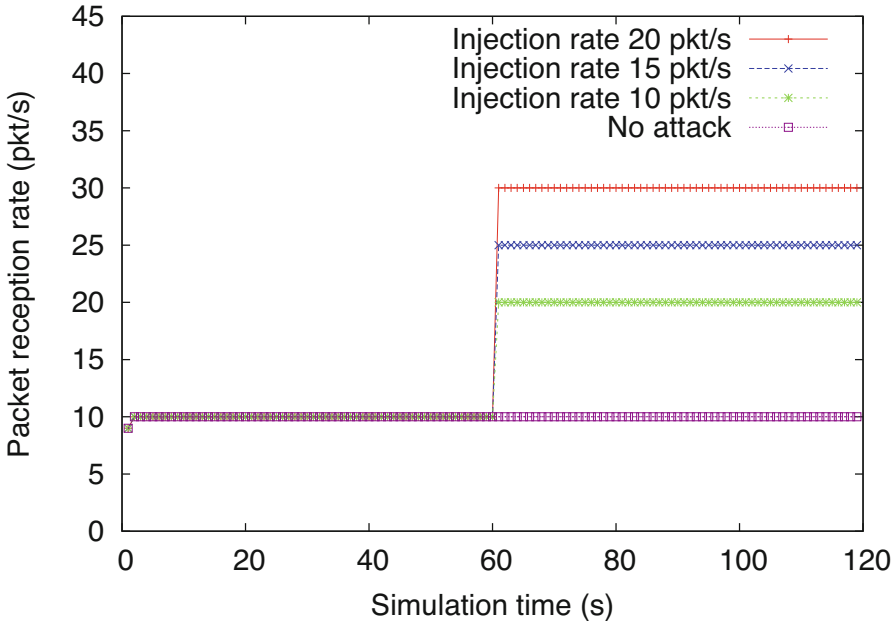



**Fig. 7.5** Packet reception rate on the server node

For instance, Fig. 7.5 shows the packet reception rate on the server node both in the baseline attack-free scenario (10 packets per second) and in the presence of the described attack (20 packets per second after $T = 60$ s). The graph also includes results produced by assessing the same attack, while considering higher attack injections rates, for a total reception rate on the server node of 25 of 30 packets per second. The attack injection rate can be easily changed in the attack description, by specifying a different time period in the `from` loop of the modeled unconditional attack (see Sect. 7.4.2.1, Listing 7.7—line 3).

## 7.5 Conclusion

This chapter presented SEA++, a simulation framework for assessing the impact of cyber-physical security attacks through simulative evaluation, in a way which is flexible and user-friendly. The framework is designed and implemented over OMNeT++/INET and allows network security architects to quantitatively evaluate the effects of security attacks against networks and applications in Information and Communications Technology (ICT) networked infrastructures, including support for SDN architectures. Therefore, SEA++ is especially convenient at the last stage of the security risk assessment process, as it makes it possible to clearly understand

the effects of successful attacks, rank them based on their severity, and accordingly prioritize security countermeasures to adopt.

The key advantage offered by SEA++ is the ability to reproduce the final actual effects of successful attacks, regardless of how they have been carried out. To this end, attacks to be evaluated are described through a high-level specification language. As a consequence, the user is not required to implement the actual adversary behavior or the actual attack execution, nor to modify any other software module. Besides, other than specific SEA++ modules seamlessly integrated into OMNeT++/INET, all other software modules can be off-the-shelf. The SEA++ framework is available as open-source software, and its core approach makes it portable to alternative simulation environments based on discrete events.

Finally, we foresee a number of potential interesting developments related to the SEA++ framework. First, feedback and requirements from the OMNeT++/INET community and use-case providers would be helpful to improve the Attack Specification Language as well as the core Attack Simulation Engine of SEA++. At the same time, it would be convenient to provide an end-user GUI, as well as tools for attack assessment in batches and for automatic analysis of simulation output.

Second, it would be good to harmoniously interconnect SEA++ with alternative simulation tools focused on different environments, technologies, or assessments. This would aim at a single and comprehensive simulation tool for the assessment of security attacks in complex and heterogeneous network environments. Such a tool would be composed of different synchronized sub-frameworks, and hence be able to perform more complex assessments through a single composite simulation. A first good candidate to consider is the ASF++ simulation framework, which is built on OMNeT++ Castalia and is based on the same rationale and approach of SEA++.

Third, SEA++ would greatly benefit from having security protocols available in OMNeT++/INET as off-the-shelf software modules, which would be selectable when composing communication stacks of network nodes. This includes implementations of protocols such as Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), Object Security for Constrained RESTful Environments (OSCORE), Internet Protocol security (IPsec), and Internet Key Exchange Version 2 (IKEv2). Similarly, SEA++ would greatly benefit from available software modules in OMNeT++/INET providing application protocols typical of Internet of Things (IoT) environments, such as Constrained Application Protocol (CoAP) and Lightweight Machine-to-Machine (LWM2M).

**Acknowledgements** The authors sincerely thank the anonymous reviewers as well as the editors Antonio Virdis and Michael Kirsche for their constructive feedback and comments.

## References


1. Bonaci, T., Bushnell, L., Poovendran, R.: Node capture attacks in wireless sensor networks: a system theoretic approach. In: The 49th IEEE Conference on Decision and Control (CDC 2010), pp. 6765–6772 (2010)
2. Boulis, T.: Castalia. https://github.com/boulis/Castalia (2018)
3. Dini, G., Tiloca, M.: ASF: an attack simulation framework for wireless sensor networks. In: The 8th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2012), pp. 203–210. IEEE, Ahmedabad (2012)
4. Dini, G., Tiloca, M.: On simulative analysis of attack impact in wireless sensor networks. In: 2013 IEEE 18th Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–8. IEEE, Ahmedabad (2013)
5. Open Networking Foundation: Software-defined networking: the new norm for networks, ONF White Paper (2012). https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf
6. Genge, B., Siaterlis, C., Hohenadel, M.: AMICI: an assessment platform for multi-domain security experimentation on critical infrastructures. In: Critical Information Infrastructures Security. Lecture Notes in Computer Science, vol. 7722, pp. 228–239. Springer, Berlin (2013)
7. Huang, Y.L., Cárdenas, A.A., Amin, S., Lin, Z.S., Tsai, H.Y., Sastry, S.: Understanding the physical and economic consequences of attacks on control systems. Int. J. Crit. Infrastruct. Prot. **2**(3), 73–83 (2009)
8. Klein, D., Jarschel, M.: An OpenFlow extension for the OMNeT++ INET framework. In: 6th International ICST Conference on Simulation Tools and Techniques (SimuTools '13), pp. 322–329 (2013)
9. Kreutz, D., Ramos, F.M.V., Veríssimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: a comprehensive survey. Proc. IEEE **103**(1), 14–76 (2015)
10. Pitt, D.: Open Networking Foundation. http://opennetworking.org (2012)
11. Queiroz, C., Mahmood, A., Tari, Z.: SCADASim–a framework for building SCADA simulations. IEEE Trans. Smart Grid **2**(4), 589–597 (2011)
12. Sánchez-Casado, L., Rodríguez-Gómez, R.A., Magán-Carrión, R., Maciá-Fernández, G.: NETA: evaluating the effects of NETwork attacks. MANETs as a case study. In: Advances in Security of Information and Communication Networks. Communications in Computer and Information Science, vol. 381, pp. 1–10. Springer, Berlin (2013)
13. Siaterlis, C., Garcia, A.P., Genge, B.: On the use of emulab testbeds for scientifically rigorous experiments. IEEE Commun. Surv. Tutorials **15**(2), 929–942 (2013)
14. Stoneburner, G., Goguen, A., Feringa, A.: Risk management guide for information technology systems - recommendations of the National Institute of Standards and Technology. Technology Report, National Institute of Standards and Technologies (2002). http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf
15. Tiloca, M., Racciatti, F., Dini, G.: Simulative evaluation of security attacks in networked critical infrastructures. In: 2nd International Workshop on Reliability and Security Aspects for Critical Infrastructure Protection (ReSA4CI 2015). Lecture Notes in Computer Science, LNCS, vol. 9338, pp. 314–323. Springer, Berlin (2015)
16. Tiloca, M., Stagkopoulou, A., Dini, G.: Performance and security evaluation of SDN networks in OMNeT++/INET. In: OMNeT++ Community Summit 2016, pp. 9–14 (2016)
17. Tiloca, M., Racciatti, F., Stagkopoulou, A., Dini, G.: SEA++, a tool for Simulative Evaluation of Attacks. https://github.com/seapp/seapp_stable (2017)
18. Wang, Y.T., Bagrodia, R.: SenSec: a scalable and accurate framework for wireless sensor network security evaluation. In: The 31st International Conference on Distributed Computing Systems Workshops (ICDCSW 2011), pp. 230–239 (2011)
19. Xu, Y., Chen, G., Ford, J., Makedon, F.: Detecting wormhole attacks in wireless sensor networks. In: Goetz, E., Shenoi, S. (eds.) Critical Infrastructure Protection, Post-Proceedings of the First Annual IFIP Working Group 11.10 International Conference on Critical Infrastructure Protection, IFIP, vol. 253, pp. 267–279. Springer, Berlin (2007)

# Part III
# Recent Developments

# Chapter 8
# Simulation Reproducibility with Python and Pweave


**Kyeong Soo (Joseph) Kim**

## 8.1 Introduction

*Reproducible research* is a key to a scientific method [14] and ensures repeating an experiment and the results of its analysis with a high degree of agreement among researchers. In a practical sense, we can say that a study is reproducible when it satisfies the following minimum criteria [7]:

- All methods are fully reported.
- All data and files used for the analysis are (publicly) available.
- The process of analyzing raw data is well reported and preserved.

Therefore, reproducible research is to ensure that with the same data and analysis scripts, one can generate the same results and thereby reach the same conclusions. When the results of a study are not reproducible; however, its claims—no matter what they are—cannot be accepted and used as a basis for further research.

Consider in this regard the *Schön scandal* [2], a notable example of unreproducible research related with data fraud. In 2001, Jan Hendrik Schön produced a series of high-profile research papers at a peculiar pace of publishing one paper every 8 days. In one of the papers published in *Nature*, which was withdrawn later, he claimed that he had produced a transistor on the molecular scale, i.e., a single-molecule transistor, that was regarded by many in the field as the holy grail of the molecular computer. Soon after Schön published the work, however, several physicists alleged that there were anomalies (e.g., two experiments carried out at very different temperatures had identical noise) and duplicates in his data,

K. S. (Joseph) Kim (✉)
Department of Electrical and Electronic Engineering, Xi'an Jiaotong-Liverpool University, Suzhou, Jiangsu, People's Republic of China
e-mail: kyeongsoo.kim@xjtlu.edu.cn



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_8

which triggered a formal investigation by a committee set up by Bell Labs in 2002. The committee found evidence of Schön's scientific misconduct in at least 16 allegations out of 24 considered [13]. The problem is that Schön had kept no laboratory notebooks and data for his groundbreaking experiments and he was unable to reproduce the claimed results. This scandal clearly shows the importance of handling experimental data and record keeping even after the publication and the need of reproducible research in carrying out any scientific research.

The detection of cosmic gravitational waves reported by the Laser Interferometer Gravitational-Wave Observatory (LIGO) team, on the other hand, provides a good example of reproducible research [1]. Together with the paper, they also published online an **IPython** [10] notebook and datasets so that readers can better understand their work, reproduce the results of their analysis, and lead into the same inferences based on them [15].

In many fields of science and technology, computer simulation provides a viable alternative to experiments based on a real system or its prototype in modeling and analyzing the performance of the system, especially when the cost and complexity of the full-scale implementation of the system is too high. Thanks to the rapid growth of computing power and development of simulation techniques, the gap between real experiments and computer simulations is being narrowed.

Because we can have more control of computer simulations compared to real experiments, it is possible to reproduce results even *identical to* those of the original research at different times and in different environments by different researchers. Even with the same simulation code and configurations (including random number seed values), however, results from computer simulations can vary due to the underlying differences in computer hardware (e.g., 32/64-bit platforms and CPU types) and software architectures (e.g., operating systems, compilers, and libraries). To address the issue of computer simulations' dependence on the underlying hardware and software architectures, an image of a virtual machine (e.g., **VirtualBox** [17]) could be released, which includes not only simulation platforms and actual code but also a complete operating system with supporting tools and libraries on a virtualized computer.

It is important to note that reproducing identical simulation results is not replacing the reliable processing of simulation results, where one repeats an experiment many times with different random number seeds and carries out statistical processing of the results for confidence intervals and other important measures in order to take into account the innate randomness in any experiments. In this chapter, we hence focus on how to guarantee the integrity of simulation code, configurations, and analysis scripts and to automatically run new simulations and update their analysis and presentation when detecting changes in any part of them for reproducible research by providing a detailed tutorial for reproducible simulation based on **Python** and **Pweave**; possible differences in results caused by computer hardware and software architectures, if any, should be handled through reliable statistical processing.

## 8.2 Tools for Reproducible OMNeT++ Simulation

In this section, we briefly review tools for reproducible OMNeT++ simulations before discussing the actual practices based on **Python** and **Pweave** in Sect. 8.3.

### *8.2.1 R and Sweave/knitr*

**R** [12] has been the language of choice for statistical processing and data analysis due to its huge code base and strong community of open-source developers as well as engineers and scientists. Listing 8.1 shows snippets of **R** source code importing OMNeT++ simulation results into a `DataFrame` object.

**Listing 8.1** R source code for importing OMNeT++ simulation results into a `DataFrame`

```
.resp <- readline("Process data from mixed configurations? (hit y or n) ")
if (.resp == "y") {
   .config <- readline("Type OMNeT++ configuration name: ")
   .mixed.rdata <- paste(.config, "RData", sep=".")
   if (file.exists(paste(.mixed.wd, .mixed.rdata, sep="/")) == FALSE) {
      .df <- loadDataset(paste(.mixed.wd, paste(.config, "-0.vec", sep=""), sep
           ="/"))
      .v <- loadVectors(.df, NULL)
      .vv <- .v$vectors
      .vd <- .v$vectordata
      .rk <- subset(.vv, name=="thruput (bit/sec)")$resultkey
      .df <- subset(.vd, is.element(resultkey, .rk))
      .df$resultkey <- as.factor(.df$resultkey)
      .df.name <- paste(.config, ".df", sep="")
      assign(.df.name, .df)
      save(list=c(.df.name), file=paste(.mixed.wd, .mixed.rdata, sep="/"))
   } else {
      load(paste(.mixed.wd, .mixed.rdata, sep="/"))
      .df.name <- paste(.config, ".df", sep="")
   }
   .df <- get(.df.name)
   .df$y <- .df$y / 1.0e6 # convert b/s -> Mb/s
   is.na(.df) <- is.na(.df) # remove NaNs
   .df <- .df[!is.infinite(.df$y),] # remove Infs
```

Still, **R** has the largest code base for a wide variety of statistical and graphical techniques. **R** also provides a tool called **Sweave** [6] (now replaced by **knitr** [5]) to weave documentation and the results of the execution of **R** code chunks (i.e., those enclosed by "`«echo = False, results = 'tex'»=`" and "`@`" in Listing 8.2) into a LaTeX source file for integrated documentation.

Compared to general-purpose programming languages like **Python**, however, **R** was created and has been developed as a specialized language for statistical processing and data analysis, so it is a bit limited in interfacing with the operating system, scraping the web, and other important programming tasks.

**Listing 8.2** **R** source code chunk inside a **Sweave** file for LaTeX

```
<<echo = False, results = 'tex'>>=
.df <- subset(.da_N1.df, select=c(1:8))
names(.df)[3:8] <- c(
"dly.mean", "dly.ci.width",
"thr.mean", "thr.ci.width",
"trf.mean", "trf.ci.width"
)
.tabledf <- xtable(.df, caption="Performance measures of FTP traffic for
     dedicated access with $N=1$.", label="tbl:da_N1_ftp")
digits(.tabledf)[2:9] <- c(0, 1, rep(-4, 6))
print(.tabledf,
    tabular.environment="longtable", caption.placement="top",
    include.rownames=FALSE, floating=FALSE, NA.string="NA")
@
```

### *8.2.2 Python and Pweave*

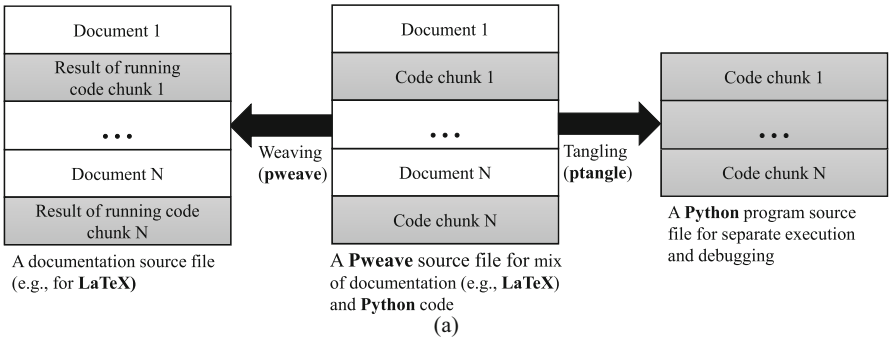
**Python** is one of the most popular programming languages in scientific computing, including artificial intelligence and machine learning, and recently takes over **R** in statistical processing and data analysis as well. Thanks to **pandas** [9] and **Pweave** [11] packages which provide a `DataFrame` object and **Sweave** functionalities, respectively, **Python** can now replace **R** for most statistical and data analysis tasks, while retaining its many advantages over **R** including being a fully featured programming language with easy syntax and higher speed and a diverse ecosystem for general scientific computing.

Figure 8.1 illustrates the integrated processing of document and **Python** code based on **Pweave**, which provides two separate programs **pweave**[1] and **ptangle** for weaving and tangling procedures; from a common **Pweave** file, **pweave** and **ptangle** generate a documentation source file (i.e., LaTeX file in this example) and **Python** source code, respectively. A detailed example of using **Pweave** for reproducible OMNeT++ simulation is provided in Sect. 8.4.

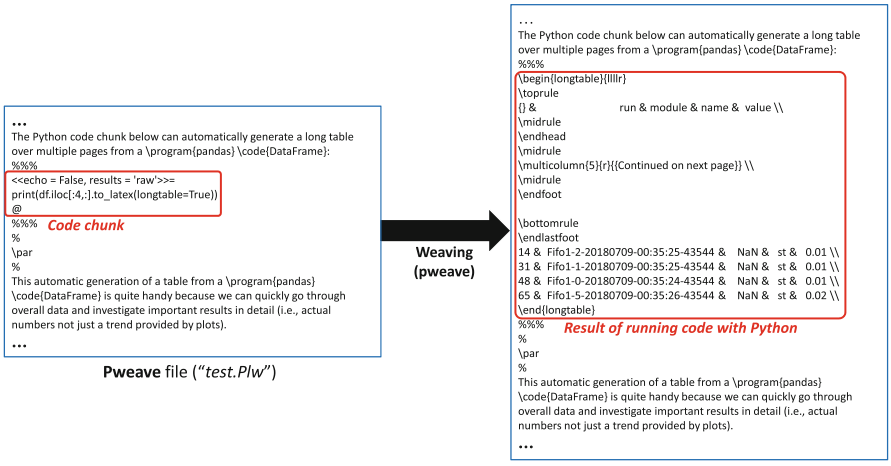
### *8.2.3 Python and Jupyter*

Project **Jupyter** [4] is an open-source project started in 2014 based on the **IPython** project. While **IPython** was developed mainly for the **Python** programming language to provide a powerful interactive shell and a web interface, **Jupyter** has evolved to support interactive data science and scientific computing across various other programming languages.

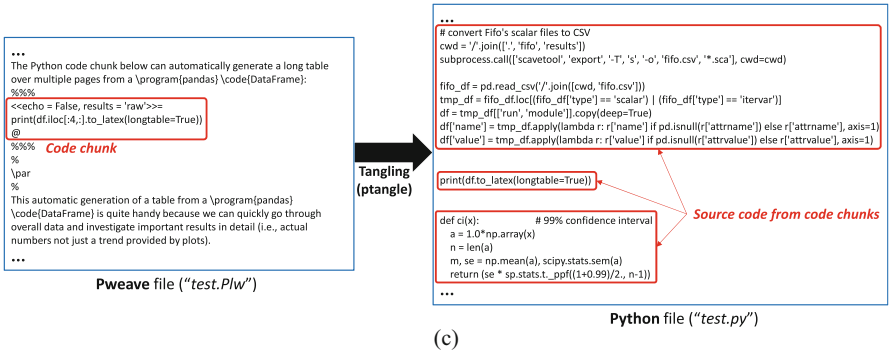
[1] **pweave** denotes an executable program, while **Pweave** denotes a **Python** package.




(a)




(b)




(c)

**Fig. 8.1** Integrated processing of document and **Python** code based on **Pweave**: (**a**) an overview of procedures and examples for (**b**) weaving (with **pweave**) and (**c**) tangling (with **ptangle**)

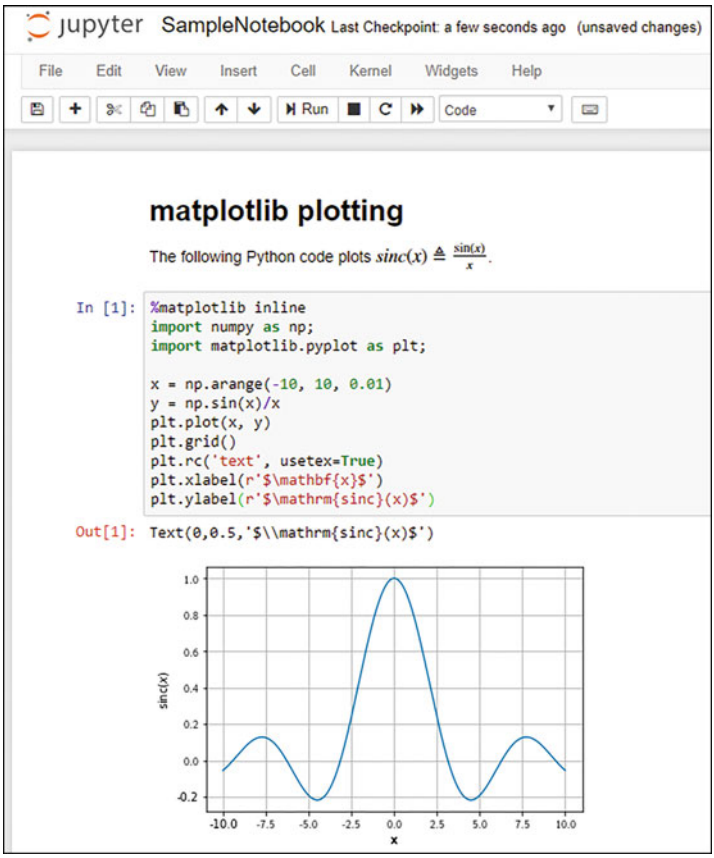



**Fig. 8.2** A **Jupyter** notebook example

Like **Sweave** and **Pweave**, **Jupyter** provides a notebook as an interactive web interface integrating text and code, which are individually stored and run in Markdown and code cells, respectively: texts stored in a Markdown cell can use the popular **Markdown** markup language to display them in a richer format in web browsers, thanks to **MathJax**. LaTeX-formatted equations are also supported. The code in a code cell, on the other hand, can be run interactively, and its outputs—not only numerical and text outputs but also graphs and tables—are immediately displayed below the cell as part of a notebook. In this way, the **Jupyter** notebook captures the whole computation process, i.e., developing, documenting, and executing code, as well as communicating the results. Figure 8.2 shows an example **Jupyter** notebook with embedded graph and math equation.

## 8.3 Reproducible OMNeT++ Simulations with Python and Pweave

### *8.3.1 Handling Simulation Code and Supporting Files*

Most of the problems in reproducing simulation results that have been published are related with the separation of documentation and simulation code and supporting files including simulation configuration files and analysis scripts. When the latter have gone through many revisions during and after the publication, even authors may find it difficult to identify the correct file versions.

In case of small-scale simulations, one solution is to include all OMNeT++ simulation code (i.e., **C++** (*.cc* and *.h*), message (*.msg*), and **NED** (*.ned*) files) with required configuration files (i.e., *.ini*) and analysis scripts in a **Pweave** file. As we will see in Sect. 8.4, this solution works best for OMNeT++ simulations where users use existing simulation models[2] and the focus of reproducible research is on the management of simulation configurations and results analysis.

If research is based on large-scale simulations and the simulation code as well as configuration files and analysis scripts should be maintained, however, including all the files in a document could not be a practical option due to their sheer size. In such a case, we can use a unique Identifier (ID) for each version of a file or a group of files (e.g., **Git** commit hashes and **SVN** release) [16]. Note that this option makes sense when the code and relevant files are managed in a version control system.

### *8.3.2 Updating Simulation Results Based on Configuration Files*

Raw simulation result files (e.g., *.sca* and *.vec* in OMNeT++) are usually excluded from the version control due to their possibly large size. This is more so when simulation code is hosted on a public repository like **GitHub** for open-source development with many collaborators, because usually there is more strict control on the sizes of individual files as well as the size of a repository. Therefore, it is a user's responsibility to generate simulation results on a local machine and check whether simulation results match with configuration files.

For small-scale simulations, we can adopt a **make**-like scheme based on timestamps of configuration files and result files, i.e., invoking a simulation when the timestamps of configuration files are newer than those of result files. Updating simulation results can be integrated into the processing of the **Pweave** file with embedded **Python** scripts as shown in Sect. 8.4.2.

[2] Or the development of the simulation models has already been finished and the resulting code will not change.

For large-scale simulations, however, the integration of the aforementioned automatic updating of simulation results as part of **Pweave** processing is not practical because it may take a much longer time to execute simulations; running large-scale simulations—often multiple simulation runs for parameter studies and/or repeating with different seeds for statistical processing—is usually done in a batch mode, which may last for hours, days, or even weeks. As already discussed in Sect. 8.3.1, using a unique ID assigned to each version of a result file could therefore be a solution to ensure a match between configuration and result files.

### *8.3.3 Analysis and Presentation of Simulation Results*

With **Pweave**, we can use many packages of **Python**—especially **NumPy**, **pandas**, and **Matplotlib**—for the analysis and presentation of simulation results within a document. A key data structure in this regard is **pandas** `DataFrame`: once we import simulation results into a `DataFrame`, we can apply advanced statistical processing techniques to the `DataFrame` and visualize the results in a much easier way through **pandas**'s high-level Application Programming Interface (API) compared to directly using **NumPy** and **Matplotlib**.

Note that you can refer to many online and offline resources available for the use of **Python** for data analysis and statistical processing with **Pweave** (e.g., [3] if you are already familiar with **R**). As for the specific information on the analysis of OMNeT++ simulation results with **Python**, the online tutorial by the OMNeT++ creator [8] can be a starting point, in addition to the general overview of OMNeT++ described in Chap. 1.

## 8.4 Example of a OMNeT++ `FIFO` Simulation

Having discussed the concept of reproducible research and the actual practices tailored for reproducible OMNeT++ simulation based on **Python** and **Pweave**, we now provide a specific example to demonstrate the concept and suggested practices. A working version of this chapter—i.e., a **Pweave** source file—is available online,[3] where the documentation part is prepared for LaTeX and the **Python** code chunk between "« . . . »=" and "@" is executed and its results are automatically embedded in the resulting LaTeX document. You can define various options for code chunks to control code execution and formatting as described in [11].[4] Figure 8.3 shows a

[3] Chapter **GitHub** repository: https://github.com/kyeongsoo/reproducible_research.

[4] The example provided in this section has been prepared and tested with OMNeT++ version 5.4 and **Pweave** version 0.30.2 running on **Python** version 3.6.6 (64-bit **Anaconda** distribution version 5.2 available online at https://www.anaconda.com/download/).

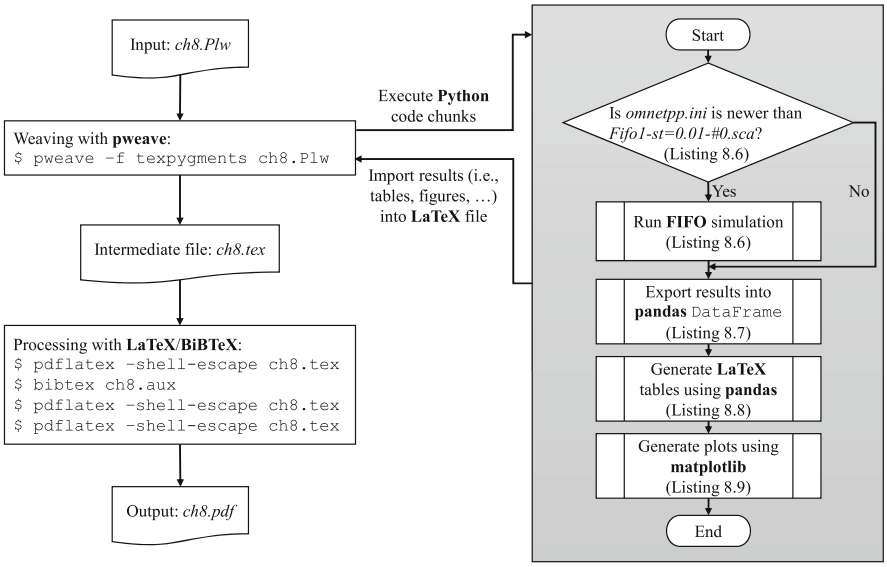



**Fig. 8.3** A workflow for generating a final PDF output file from a **Pweave** source file

workflow for generating a **PDF** output file from a **Pweave** source file, where each block of the flowchart shows a corresponding **Python** code chunk described later in this section.

Note that the current chapter has been produced after some modifications of the LaTeX source file automatically generated by **Pweave**; **Pweave** supports **Python**'s **pygments** and LaTeX's `minted` environment for code listing but not yet for the LaTeX `lstlisting` environment used within this book.

### *8.4.1 Simulation Configurations*

Listing 8.3 shows a **Python** code chunk for setting up the simulation of the `FIFO` sample model, which is part of the OMNeT++ simulation framework.

As described before, the lines 1 and 23 are not part of **Python** code but **Pweave** commands demarcating the code chunk so that the **Python** code (i.e., lines 2–22) can be processed by **Python** during the weaving procedure. Note that we set the `term` code chunk option to `False` in line 1; otherwise, the code will be executed one statement at a time and the output for each statement will be displayed. Also, note that we import all **Python** packages in the beginning (i.e., lines 4–10), which are needed not only for this code chunk but also for all other code chunks in this example; this is to avoid any potential conflicts among packages when importing **Python** packages in the middle of the program. Then, we set variables for paths and files in lines 13–22, which will be used in the code chunks described in Sects. 8.4.2 and 8.4.3.

**Listing 8.3** **Python** code chunk for setting up an OMNeT++ simulation

```
<<name = 'settings', term = False, wrap = True>>=
# import all the necessary packages here to avoid any issues
# resulting from splitted imports
import os
import subprocess
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy as sp
import scipy.stats

# set path to run Fifo simulation in DOS command prompt
omnetpp_root = os.environ['OMNETPP_ROOT']
path1 = '/'.join([omnetpp_root, 'bin'])
path2 = '/'.join([omnetpp_root, 'tools', 'win64', 'mingw64', 'bin'])
os.environ['Path'] = ';'.join([path1, path2, os.environ['Path']])

# set variables for input & output files and simulation program
ned = '/'.join(['.', 'fifo', 'Fifo.ned'])
ini = '/'.join(['.', 'fifo', 'omnetpp.ini'])
sca = '/'.join(['.', 'fifo', 'results', 'Fifo1-st=0.01-#0.sca'])
fifo = '/'.join(['.', 'fifo', 'fifo.exe'])
@
```

Listings 8.4 and 8.5 show the *FifoNet.ned* and *omnetpp.ini* files for the `FIFO` sample model, respectively.

**Listing 8.4** Content of the *FifoNet.ned* file for the `FIFO` sample model

```
// This file is part of an OMNeT++/OMNEST simulation example.
// Copyright (C) 1992-2015 Andras Varga

//
// Simple queueing network: generator + FIFO + sink.
//
network FifoNet
{
   submodules:
      gen: Source {
         parameters:
            @display("p=89,100");
      }
      fifo: Fifo {
         parameters:
            @display("p=209,100");
      }
      sink: Sink {
         parameters:
            @display("p=329,100");
      }
   connections:
      gen.out --> fifo.in;
      fifo.out --> sink.in;
}
```

**Listing 8.5** Content of the *omnetpp.ini* file for the `FIFO` sample model

```
[General]
network = FifoNet
sim-time-limit = 5h
#cpu-time-limit = 300s
repeat = 5
**.vector-recording = false
#debug-on-errors = true
#record-eventlog = true

[Config Fifo1]
description = "low job arrival rate"
**.gen.sendIaTime = exponential(0.2s)
**.fifo.serviceTime = ${st=0.01..0.05 step 0.01}s

[Config Fifo2]
description = "high job arrival rate"
**.gen.sendIaTime = exponential(0.01s)
**.fifo.serviceTime = 0.01s
```

### 8.4.2 *Running Simulations and Importing Results*

Listing 8.6 shows a **Python** code chunk, which invokes the OMNeT++ `FIFO` simulation only when *omnetpp.ini* is newer than one of the scalar files (i.e., lines 3–7) and then exports simulation results from all the scalar files to a Comma-Separated Values (CSV) file using the **scavetool** provided by OMNeT++ (i.e., lines 10–12).

**Listing 8.6** **Python** code chunk to automatically check simulation input files and programmatically run OMNeT++ simulation

```
<<name = 'update_results', term = False, wrap = True>>=
# run the simulation only if input files are newer than results
if (not os.path.isfile(sca)) or (os.path.getmtime(ini) >
os.path.getmtime(sca)):
    cwd = '/'.join(['.', 'fifo'])
    subprocess.call([fifo, '-u', 'Cmdenv', '-f', 'omnetpp.ini', '-c', 'Fifo1'],
                cwd=cwd, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)

# export simulation results from Fifo's scalar files to a CSV file
cwd = '/'.join(['.', 'fifo', 'results'])
subprocess.call(['scavetool', 'export', '-T', 's', '-o', 'fifo.csv', '*.sca'],
            cwd=cwd, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
@
```

Then, the simulation results can be imported into **pandas** `DataFrame` as shown in Listing 8.7.

**Listing 8.7** **Python** code chunk to import simulation results into **pandas** `DataFrame`

```
1 <<name = 'import_data', term = False, wrap = True>>=
2 fifo_df = pd.read_csv('/'.join([cwd, 'fifo.csv']))
3 tmp_df = fifo_df.loc[(fifo_df['type'] == 'scalar') | (fifo_df['type'] == '
      itervar')]
4 df = tmp_df[['run', 'module']].copy(deep=True)
5 df['name'] = tmp_df.apply(lambda r: r['name'] if pd.isnull(r['attrname']) else
      r['attrname'], axis=1)
6 df['value'] = tmp_df.apply(lambda r: r['value'] if pd.isnull(r['attrvalue'])
      else r['attrvalue'], axis=1)
7 df['value'] = pd.to_numeric(df['value'])
8 @
```

Note that, due to the changes in the export process of the **scavetool** from OMNeT++ v5.4, we need to merge "attrname" and "attrvalue" into "name" and "value" columns, respectively. For details of **pandas** API, please refer to its documentation.

The **Python** code chunk shown in Listing 8.8 automatically generates a long table over multiple pages (included for reference in the subsequent pages) from a **pandas** `DataFrame`, where we set the `echo` and `results` options to `False` and `"raw,"` respectively, in order to hide source code[5] and replace it with the raw output from the execution of the code chunk (i.e., LaTeX source code for Table 8.1).

**Listing 8.8** **Python** code chunk for generating a long table

```
1 <<echo = False, results = 'raw'>>=
2 print(df.to_latex(longtable=True))
3 @
```

This automatic generation of a table from a **pandas** `DataFrame` is quite handy because we can quickly go through overall data and investigate important results in detail (i.e., the actual numbers and not just a trend provided by plots). The suggested solution of embedding a long table within a **Pweave** document, however, is not perfect yet as there is no option in the `pandas.DataFrame.to_latex` API providing a caption and a label within a generated `longtable` environment.

### *8.4.3 Data Analysis and Presentation*

Listing 8.9 shows a **Python** code chunk that processes the `DataFrame` obtained in Sect. 8.4.2 and creates a bar plot with error bars showing the mean queuing time against the packet service time.

[5] Listing 8.8 is shown here for explanation using the LaTeX `lstlisting` environment.

**Table 8.1** Long table generated by the **Python** code chunk from Listing 8.8

| | Run | Module | Name | Value |
|---|---|---|---|---|
| 14 | Fifo1-1-20180709-00:35:25-43544 | NaN | st | 0.010000 |
| 31 | Fifo1-0-20180709-00:35:24-43544 | NaN | st | 0.010000 |
| 48 | Fifo1-10-20180709-00:35:27-43544 | NaN | st | 0.030000 |
| 65 | Fifo1-2-20180709-00:35:25-43544 | NaN | st | 0.010000 |
| 82 | Fifo1-4-20180709-00:35:26-43544 | NaN | st | 0.010000 |
| 99 | Fifo1-5-20180709-00:35:26-43544 | NaN | st | 0.020000 |
| 116 | Fifo1-3-20180709-00:35:25-43544 | NaN | st | 0.010000 |
| 133 | Fifo1-11-20180709-00:35:27-43544 | NaN | st | 0.030000 |
| 150 | Fifo1-6-20180709-00:35:26-43544 | NaN | st | 0.020000 |
| 167 | Fifo1-7-20180709-00:35:26-43544 | NaN | st | 0.020000 |
| 184 | Fifo1-8-20180709-00:35:27-43544 | NaN | st | 0.020000 |
| 201 | Fifo1-9-20180709-00:35:27-43544 | NaN | st | 0.020000 |
| 218 | Fifo1-12-20180709-00:35:28-43544 | NaN | st | 0.030000 |
| 235 | Fifo1-13-20180709-00:35:28-43544 | NaN | st | 0.030000 |
| 252 | Fifo1-14-20180709-00:35:28-43544 | NaN | st | 0.030000 |
| 269 | Fifo1-15-20180709-00:35:29-43544 | NaN | st | 0.040000 |
| 286 | Fifo1-17-20180709-00:35:29-43544 | NaN | st | 0.040000 |
| 303 | Fifo1-24-20180709-00:35:31-43544 | NaN | st | 0.050000 |
| 320 | Fifo1-18-20180709-00:35:29-43544 | NaN | st | 0.040000 |
| 337 | Fifo1-19-20180709-00:35:30-43544 | NaN | st | 0.040000 |
| 354 | Fifo1-22-20180709-00:35:30-43544 | NaN | st | 0.050000 |
| 371 | Fifo1-21-20180709-00:35:30-43544 | NaN | st | 0.050000 |
| 388 | Fifo1-20-20180709-00:35:30-43544 | NaN | st | 0.050000 |
| 405 | Fifo1-16-20180709-00:35:29-43544 | NaN | st | 0.040000 |
| 422 | Fifo1-23-20180709-00:35:31-43544 | NaN | st | 0.050000 |
| 425 | Fifo1-0-20180709-00:35:24-43544 | FifoNet.fifo | queueingTime:mean | 0.000271 |
| 429 | Fifo1-0-20180709-00:35:24-43544 | FifoNet.fifo | queueingTime:max | 0.022790 |
| 433 | Fifo1-0-20180709-00:35:24-43544 | FifoNet.fifo | busy:timeavg | 0.050264 |
| 436 | Fifo1-0-20180709-00:35:24-43544 | FifoNet.fifo | qlen:timeavg | 0.001361 |
| 439 | Fifo1-0-20180709-00:35:24-43544 | FifoNet.fifo | qlen:max | 3.000000 |
| 442 | Fifo1-0-20180709-00:35:24-43544 | FifoNet.sink | lifetime:mean | 0.010271 |
| 446 | Fifo1-0-20180709-00:35:24-43544 | FifoNet.sink | lifetime:max | 0.032790 |
| 450 | Fifo1-1-20180709-00:35:25-43544 | FifoNet.fifo | queueingTime:mean | 0.000264 |
| 454 | Fifo1-1-20180709-00:35:25-43544 | FifoNet.fifo | queueingTime:max | 0.018800 |
| 458 | Fifo1-1-20180709-00:35:25-43544 | FifoNet.fifo | busy:timeavg | 0.050042 |
| 461 | Fifo1-1-20180709-00:35:25-43544 | FifoNet.fifo | qlen:timeavg | 0.001319 |
| 464 | Fifo1-1-20180709-00:35:25-43544 | FifoNet.fifo | qlen:max | 2.000000 |
| 467 | Fifo1-1-20180709-00:35:25-43544 | FifoNet.sink | lifetime:mean | 0.010264 |
| 471 | Fifo1-1-20180709-00:35:25-43544 | FifoNet.sink | lifetime:max | 0.028800 |
| 475 | Fifo1-2-20180709-00:35:25-43544 | FifoNet.fifo | queueingTime:mean | 0.000272 |
| 479 | Fifo1-2-20180709-00:35:25-43544 | FifoNet.fifo | queueingTime:max | 0.025558 |

(continued)

**Table 8.1** (continued)

| | Run | Module | Name | Value |
|---|---|---|---|---|
| 483 | Fifo1-2-20180709-00:35:25-43544 | FifoNet.fifo | busy:timeavg | 0.050061 |
| 486 | Fifo1-2-20180709-00:35:25-43544 | FifoNet.fifo | qlen:timeavg | 0.001359 |
| 489 | Fifo1-2-20180709-00:35:25-43544 | FifoNet.fifo | qlen:max | 3.000000 |
| 492 | Fifo1-2-20180709-00:35:25-43544 | FifoNet.sink | lifetime:mean | 0.010272 |
| 496 | Fifo1-2-20180709-00:35:25-43544 | FifoNet.sink | lifetime:max | 0.035558 |
| 500 | Fifo1-3-20180709-00:35:25-43544 | FifoNet.fifo | queueingTime:mean | 0.000260 |
| 504 | Fifo1-3-20180709-00:35:25-43544 | FifoNet.fifo | queueingTime:max | 0.019135 |
| 508 | Fifo1-3-20180709-00:35:25-43544 | FifoNet.fifo | busy:timeavg | 0.049948 |
| 511 | Fifo1-3-20180709-00:35:25-43544 | FifoNet.fifo | qlen:timeavg | 0.001297 |
| 514 | Fifo1-3-20180709-00:35:25-43544 | FifoNet.fifo | qlen:max | 2.000000 |
| 517 | Fifo1-3-20180709-00:35:25-43544 | FifoNet.sink | lifetime:mean | 0.010260 |
| 521 | Fifo1-3-20180709-00:35:25-43544 | FifoNet.sink | lifetime:max | 0.029135 |
| 525 | Fifo1-4-20180709-00:35:26-43544 | FifoNet.fifo | queueingTime:mean | 0.000265 |
| 529 | Fifo1-4-20180709-00:35:26-43544 | FifoNet.fifo | queueingTime:max | 0.021754 |
| 533 | Fifo1-4-20180709-00:35:26-43544 | FifoNet.fifo | busy:timeavg | 0.049776 |
| 536 | Fifo1-4-20180709-00:35:26-43544 | FifoNet.fifo | qlen:timeavg | 0.001318 |
| 539 | Fifo1-4-20180709-00:35:26-43544 | FifoNet.fifo | qlen:max | 3.000000 |
| 542 | Fifo1-4-20180709-00:35:26-43544 | FifoNet.sink | lifetime:mean | 0.010265 |
| 546 | Fifo1-4-20180709-00:35:26-43544 | FifoNet.sink | lifetime:max | 0.031754 |
| 550 | Fifo1-5-20180709-00:35:26-43544 | FifoNet.fifo | queueingTime:mean | 0.001098 |
| 554 | Fifo1-5-20180709-00:35:26-43544 | FifoNet.fifo | queueingTime:max | 0.052863 |
| 558 | Fifo1-5-20180709-00:35:26-43544 | FifoNet.fifo | busy:timeavg | 0.099750 |
| 561 | Fifo1-5-20180709-00:35:26-43544 | FifoNet.fifo | qlen:timeavg | 0.005475 |
| 564 | Fifo1-5-20180709-00:35:26-43544 | FifoNet.fifo | qlen:max | 3.000000 |
| 567 | Fifo1-5-20180709-00:35:26-43544 | FifoNet.sink | lifetime:mean | 0.021098 |
| 571 | Fifo1-5-20180709-00:35:26-43544 | FifoNet.sink | lifetime:max | 0.072863 |
| 575 | Fifo1-6-20180709-00:35:26-43544 | FifoNet.fifo | queueingTime:mean | 0.001111 |
| 579 | Fifo1-6-20180709-00:35:26-43544 | FifoNet.fifo | queueingTime:max | 0.061320 |
| 583 | Fifo1-6-20180709-00:35:26-43544 | FifoNet.fifo | busy:timeavg | 0.100662 |
| 586 | Fifo1-6-20180709-00:35:26-43544 | FifoNet.fifo | qlen:timeavg | 0.005594 |
| 589 | Fifo1-6-20180709-00:35:26-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 592 | Fifo1-6-20180709-00:35:26-43544 | FifoNet.sink | lifetime:mean | 0.021111 |
| 596 | Fifo1-6-20180709-00:35:26-43544 | FifoNet.sink | lifetime:max | 0.081320 |
| 600 | Fifo1-7-20180709-00:35:26-43544 | FifoNet.fifo | queueingTime:mean | 0.001095 |
| 604 | Fifo1-7-20180709-00:35:26-43544 | FifoNet.fifo | queueingTime:max | 0.053629 |
| 608 | Fifo1-7-20180709-00:35:26-43544 | FifoNet.fifo | busy:timeavg | 0.100041 |
| 611 | Fifo1-7-20180709-00:35:26-43544 | FifoNet.fifo | qlen:timeavg | 0.005476 |
| 614 | Fifo1-7-20180709-00:35:26-43544 | FifoNet.fifo | qlen:max | 3.000000 |
| 617 | Fifo1-7-20180709-00:35:26-43544 | FifoNet.sink | lifetime:mean | 0.021095 |
| 621 | Fifo1-7-20180709-00:35:26-43544 | FifoNet.sink | lifetime:max | 0.073629 |
| 625 | Fifo1-8-20180709-00:35:27-43544 | FifoNet.fifo | queueingTime:mean | 0.001149 |

(continued)

**Table 8.1** (continued)

| | Run | Module | Name | Value |
|---|---|---|---|---|
| 629 | Fifo1-8-20180709-00:35:27-43544 | FifoNet.fifo | queueingTime:max | 0.060847 |
| 633 | Fifo1-8-20180709-00:35:27-43544 | FifoNet.fifo | busy:timeavg | 0.100764 |
| 636 | Fifo1-8-20180709-00:35:27-43544 | FifoNet.fifo | qlen:timeavg | 0.005787 |
| 639 | Fifo1-8-20180709-00:35:27-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 642 | Fifo1-8-20180709-00:35:27-43544 | FifoNet.sink | lifetime:mean | 0.021149 |
| 646 | Fifo1-8-20180709-00:35:27-43544 | FifoNet.sink | lifetime:max | 0.080847 |
| 650 | Fifo1-9-20180709-00:35:27-43544 | FifoNet.fifo | queueingTime:mean | 0.001126 |
| 654 | Fifo1-9-20180709-00:35:27-43544 | FifoNet.fifo | queueingTime:max | 0.054308 |
| 658 | Fifo1-9-20180709-00:35:27-43544 | FifoNet.fifo | busy:timeavg | 0.100372 |
| 661 | Fifo1-9-20180709-00:35:27-43544 | FifoNet.fifo | qlen:timeavg | 0.005653 |
| 664 | Fifo1-9-20180709-00:35:27-43544 | FifoNet.fifo | qlen:max | 3.000000 |
| 667 | Fifo1-9-20180709-00:35:27-43544 | FifoNet.sink | lifetime:mean | 0.021126 |
| 671 | Fifo1-9-20180709-00:35:27-43544 | FifoNet.sink | lifetime:max | 0.074308 |
| 675 | Fifo1-10-20180709-00:35:27-43544 | FifoNet.fifo | queueingTime:mean | 0.002684 |
| 679 | Fifo1-10-20180709-00:35:27-43544 | FifoNet.fifo | queueingTime:max | 0.101136 |
| 683 | Fifo1-10-20180709-00:35:27-43544 | FifoNet.fifo | busy:timeavg | 0.150412 |
| 686 | Fifo1-10-20180709-00:35:27-43544 | FifoNet.fifo | qlen:timeavg | 0.013455 |
| 689 | Fifo1-10-20180709-00:35:27-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 692 | Fifo1-10-20180709-00:35:27-43544 | FifoNet.sink | lifetime:mean | 0.032684 |
| 696 | Fifo1-10-20180709-00:35:27-43544 | FifoNet.sink | lifetime:max | 0.131136 |
| 700 | Fifo1-11-20180709-00:35:27-43544 | FifoNet.fifo | queueingTime:mean | 0.002658 |
| 704 | Fifo1-11-20180709-00:35:27-43544 | FifoNet.fifo | queueingTime:max | 0.112885 |
| 708 | Fifo1-11-20180709-00:35:27-43544 | FifoNet.fifo | busy:timeavg | 0.149972 |
| 711 | Fifo1-11-20180709-00:35:27-43544 | FifoNet.fifo | qlen:timeavg | 0.013286 |
| 714 | Fifo1-11-20180709-00:35:27-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 717 | Fifo1-11-20180709-00:35:27-43544 | FifoNet.sink | lifetime:mean | 0.032658 |
| 721 | Fifo1-11-20180709-00:35:27-43544 | FifoNet.sink | lifetime:max | 0.142885 |
| 725 | Fifo1-12-20180709-00:35:28-43544 | FifoNet.fifo | queueingTime:mean | 0.002623 |
| 729 | Fifo1-12-20180709-00:35:28-43544 | FifoNet.fifo | queueingTime:max | 0.100678 |
| 733 | Fifo1-12-20180709-00:35:28-43544 | FifoNet.fifo | busy:timeavg | 0.150370 |
| 736 | Fifo1-12-20180709-00:35:28-43544 | FifoNet.fifo | qlen:timeavg | 0.013148 |
| 739 | Fifo1-12-20180709-00:35:28-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 742 | Fifo1-12-20180709-00:35:28-43544 | FifoNet.sink | lifetime:mean | 0.032623 |
| 746 | Fifo1-12-20180709-00:35:28-43544 | FifoNet.sink | lifetime:max | 0.130678 |
| 750 | Fifo1-13-20180709-00:35:28-43544 | FifoNet.fifo | queueingTime:mean | 0.002661 |
| 754 | Fifo1-13-20180709-00:35:28-43544 | FifoNet.fifo | queueingTime:max | 0.091915 |
| 758 | Fifo1-13-20180709-00:35:28-43544 | FifoNet.fifo | busy:timeavg | 0.149858 |
| 761 | Fifo1-13-20180709-00:35:28-43544 | FifoNet.fifo | qlen:timeavg | 0.013292 |
| 764 | Fifo1-13-20180709-00:35:28-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 767 | Fifo1-13-20180709-00:35:28-43544 | FifoNet.sink | lifetime:mean | 0.032661 |
| 771 | Fifo1-13-20180709-00:35:28-43544 | FifoNet.sink | lifetime:max | 0.121915 |

(continued)

**Table 8.1** (continued)

| | Run | Module | Name | Value |
|---|---|---|---|---|
| 775 | Fifo1-14-20180709-00:35:28-43544 | FifoNet.fifo | queueingTime:mean | 0.002642 |
| 779 | Fifo1-14-20180709-00:35:28-43544 | FifoNet.fifo | queueingTime:max | 0.094572 |
| 783 | Fifo1-14-20180709-00:35:28-43544 | FifoNet.fifo | busy:timeavg | 0.149782 |
| 786 | Fifo1-14-20180709-00:35:28-43544 | FifoNet.fifo | qlen:timeavg | 0.013189 |
| 789 | Fifo1-14-20180709-00:35:28-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 792 | Fifo1-14-20180709-00:35:28-43544 | FifoNet.sink | lifetime:mean | 0.032642 |
| 796 | Fifo1-14-20180709-00:35:28-43544 | FifoNet.sink | lifetime:max | 0.124572 |
| 800 | Fifo1-15-20180709-00:35:29-43544 | FifoNet.fifo | queueingTime:mean | 0.004932 |
| 804 | Fifo1-15-20180709-00:35:29-43544 | FifoNet.fifo | queueingTime:max | 0.180621 |
| 808 | Fifo1-15-20180709-00:35:29-43544 | FifoNet.fifo | busy:timeavg | 0.200379 |
| 811 | Fifo1-15-20180709-00:35:29-43544 | FifoNet.fifo | qlen:timeavg | 0.024708 |
| 814 | Fifo1-15-20180709-00:35:29-43544 | FifoNet.fifo | qlen:max | 5.000000 |
| 817 | Fifo1-15-20180709-00:35:29-43544 | FifoNet.sink | lifetime:mean | 0.044932 |
| 821 | Fifo1-15-20180709-00:35:29-43544 | FifoNet.sink | lifetime:max | 0.220621 |
| 825 | Fifo1-16-20180709-00:35:29-43544 | FifoNet.fifo | queueingTime:mean | 0.005037 |
| 829 | Fifo1-16-20180709-00:35:29-43544 | FifoNet.fifo | queueingTime:max | 0.157060 |
| 833 | Fifo1-16-20180709-00:35:29-43544 | FifoNet.fifo | busy:timeavg | 0.199642 |
| 836 | Fifo1-16-20180709-00:35:29-43544 | FifoNet.fifo | qlen:timeavg | 0.025139 |
| 839 | Fifo1-16-20180709-00:35:29-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 842 | Fifo1-16-20180709-00:35:29-43544 | FifoNet.sink | lifetime:mean | 0.045037 |
| 846 | Fifo1-16-20180709-00:35:29-43544 | FifoNet.sink | lifetime:max | 0.197060 |
| 850 | Fifo1-17-20180709-00:35:29-43544 | FifoNet.fifo | queueingTime:mean | 0.005067 |
| 854 | Fifo1-17-20180709-00:35:29-43544 | FifoNet.fifo | queueingTime:max | 0.159440 |
| 858 | Fifo1-17-20180709-00:35:29-43544 | FifoNet.fifo | busy:timeavg | 0.201280 |
| 861 | Fifo1-17-20180709-00:35:29-43544 | FifoNet.fifo | qlen:timeavg | 0.025498 |
| 864 | Fifo1-17-20180709-00:35:29-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 867 | Fifo1-17-20180709-00:35:29-43544 | FifoNet.sink | lifetime:mean | 0.045067 |
| 871 | Fifo1-17-20180709-00:35:29-43544 | FifoNet.sink | lifetime:max | 0.199440 |
| 875 | Fifo1-18-20180709-00:35:29-43544 | FifoNet.fifo | queueingTime:mean | 0.005031 |
| 879 | Fifo1-18-20180709-00:35:29-43544 | FifoNet.fifo | queueingTime:max | 0.153422 |
| 883 | Fifo1-18-20180709-00:35:29-43544 | FifoNet.fifo | busy:timeavg | 0.200771 |
| 886 | Fifo1-18-20180709-00:35:29-43544 | FifoNet.fifo | qlen:timeavg | 0.025251 |
| 889 | Fifo1-18-20180709-00:35:29-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 892 | Fifo1-18-20180709-00:35:29-43544 | FifoNet.sink | lifetime:mean | 0.045031 |
| 896 | Fifo1-18-20180709-00:35:29-43544 | FifoNet.sink | lifetime:max | 0.193422 |
| 900 | Fifo1-19-20180709-00:35:30-43544 | FifoNet.fifo | queueingTime:mean | 0.005061 |
| 904 | Fifo1-19-20180709-00:35:30-43544 | FifoNet.fifo | queueingTime:max | 0.191037 |
| 908 | Fifo1-19-20180709-00:35:30-43544 | FifoNet.fifo | busy:timeavg | 0.199784 |
| 911 | Fifo1-19-20180709-00:35:30-43544 | FifoNet.fifo | qlen:timeavg | 0.025279 |
| 914 | Fifo1-19-20180709-00:35:30-43544 | FifoNet.fifo | qlen:max | 5.000000 |
| 917 | Fifo1-19-20180709-00:35:30-43544 | FifoNet.sink | lifetime:mean | 0.045061 |

(continued)

**Table 8.1** (continued)

| | Run | Module | Name | Value |
|---|---|---|---|---|
| 921 | Fifo1-19-20180709-00:35:30-43544 | FifoNet.sink | lifetime:max | 0.231037 |
| 925 | Fifo1-20-20180709-00:35:30-43544 | FifoNet.fifo | queueingTime:mean | 0.008278 |
| 929 | Fifo1-20-20180709-00:35:30-43544 | FifoNet.fifo | queueingTime:max | 0.243179 |
| 933 | Fifo1-20-20180709-00:35:30-43544 | FifoNet.fifo | busy:timeavg | 0.249125 |
| 936 | Fifo1-20-20180709-00:35:30-43544 | FifoNet.fifo | qlen:timeavg | 0.041244 |
| 939 | Fifo1-20-20180709-00:35:30-43544 | FifoNet.fifo | qlen:max | 5.000000 |
| 942 | Fifo1-20-20180709-00:35:30-43544 | FifoNet.sink | lifetime:mean | 0.058278 |
| 946 | Fifo1-20-20180709-00:35:30-43544 | FifoNet.sink | lifetime:max | 0.293179 |
| 950 | Fifo1-21-20180709-00:35:30-43544 | FifoNet.fifo | queueingTime:mean | 0.008352 |
| 954 | Fifo1-21-20180709-00:35:30-43544 | FifoNet.fifo | queueingTime:max | 0.189506 |
| 958 | Fifo1-21-20180709-00:35:30-43544 | FifoNet.fifo | busy:timeavg | 0.249678 |
| 961 | Fifo1-21-20180709-00:35:30-43544 | FifoNet.fifo | qlen:timeavg | 0.041706 |
| 964 | Fifo1-21-20180709-00:35:30-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 967 | Fifo1-21-20180709-00:35:30-43544 | FifoNet.sink | lifetime:mean | 0.058352 |
| 971 | Fifo1-21-20180709-00:35:30-43544 | FifoNet.sink | lifetime:max | 0.239506 |
| 975 | Fifo1-22-20180709-00:35:30-43544 | FifoNet.fifo | queueingTime:mean | 0.008423 |
| 979 | Fifo1-22-20180709-00:35:30-43544 | FifoNet.fifo | queueingTime:max | 0.200534 |
| 983 | Fifo1-22-20180709-00:35:30-43544 | FifoNet.fifo | busy:timeavg | 0.250056 |
| 986 | Fifo1-22-20180709-00:35:30-43544 | FifoNet.fifo | qlen:timeavg | 0.042125 |
| 989 | Fifo1-22-20180709-00:35:30-43544 | FifoNet.fifo | qlen:max | 5.000000 |
| 992 | Fifo1-22-20180709-00:35:30-43544 | FifoNet.sink | lifetime:mean | 0.058423 |
| 996 | Fifo1-22-20180709-00:35:30-43544 | FifoNet.sink | lifetime:max | 0.250534 |
| 1000 | Fifo1-23-20180709-00:35:31-43544 | FifoNet.fifo | queueingTime:mean | 0.008356 |
| 1004 | Fifo1-23-20180709-00:35:31-43544 | FifoNet.fifo | queueingTime:max | 0.198530 |
| 1008 | Fifo1-23-20180709-00:35:31-43544 | FifoNet.fifo | busy:timeavg | 0.249261 |
| 1011 | Fifo1-23-20180709-00:35:31-43544 | FifoNet.fifo | qlen:timeavg | 0.041656 |
| 1014 | Fifo1-23-20180709-00:35:31-43544 | FifoNet.fifo | qlen:max | 4.000000 |
| 1017 | Fifo1-23-20180709-00:35:31-43544 | FifoNet.sink | lifetime:mean | 0.058356 |
| 1021 | Fifo1-23-20180709-00:35:31-43544 | FifoNet.sink | lifetime:max | 0.248530 |
| 1025 | Fifo1-24-20180709-00:35:31-43544 | FifoNet.fifo | queueingTime:mean | 0.008404 |
| 1029 | Fifo1-24-20180709-00:35:31-43544 | FifoNet.fifo | queueingTime:max | 0.209697 |
| 1033 | Fifo1-24-20180709-00:35:31-43544 | FifoNet.fifo | busy:timeavg | 0.249950 |
| 1036 | Fifo1-24-20180709-00:35:31-43544 | FifoNet.fifo | qlen:timeavg | 0.042009 |
| 1039 | Fifo1-24-20180709-00:35:31-43544 | FifoNet.fifo | qlen:max | 5.000000 |
| 1042 | Fifo1-24-20180709-00:35:31-43544 | FifoNet.sink | lifetime:mean | 0.058404 |
| 1046 | Fifo1-24-20180709-00:35:31-43544 | FifoNet.sink | lifetime:max | 0.259697 |

**Listing 8.9** Python code chunk for creating a bar plot with error bars

```
<<Fig = True, f_pos = "htpb", caption = 'Mean queueing time vs. service time (
    with 99 percent confidence intervals).', wrap = True>>=
def ci(x):                  # 99% confidence interval
    a = 1.0*np.array(x)
    n = len(a)
    m, se = np.mean(a), scipy.stats.sem(a)
    return (se * sp.stats.t._ppf((1+0.99)/2., n-1))

pivoted = df.pivot(index='run', columns='name', values='value')
st_vs_qt = pivoted.pivot_table(index='st', values='queueingTime:mean')
errs = pivoted.pivot_table(index='st', values='queueingTime:mean', aggfunc=ci)
st_vs_qt.plot(kind='bar', rot=0, legend=None, yerr=errs, color='lightgray',
          edgecolor='black', linewidth=0.5,
          error_kw=dict(ecolor='black', elinewidth=1, capsize=5))
plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
plt.xlabel('Service Time')
plt.ylabel('Mean Queueing Time')
plt.show()
@
```

In line 1, we set the `Fig` option to `True` to include a `matplotlib` plot produced by the code chunk with setting `f_pos` and `caption` options to control the position and the caption of LaTeX `figure` environment.

Figure 8.4 shows the bar plot that is automatically included in the resulting LaTeX file after weaving the **Pweave** source file.

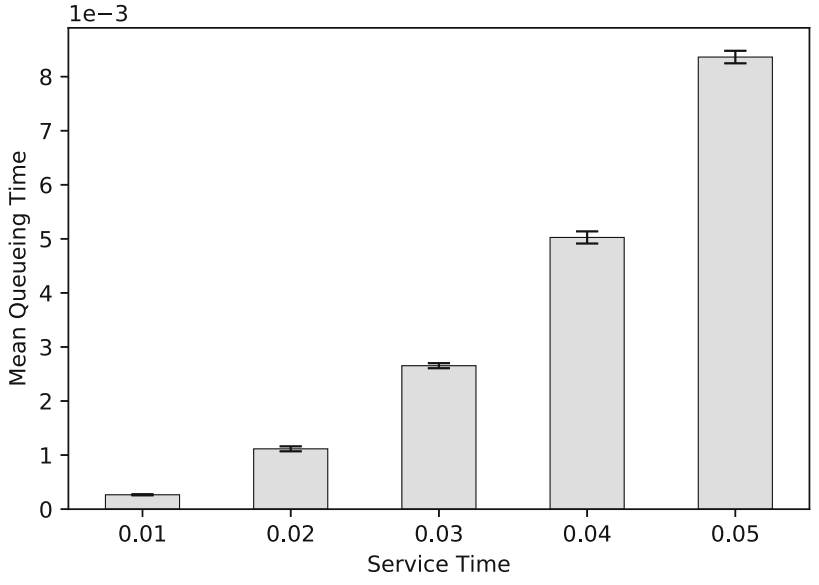



**Fig. 8.4** Mean queuing time vs. service time (with 99% confidence intervals)

## 8.5 Summary

This chapter discussed the concept of reproducible research and the actual practices of using **Python** and **Pweave** in facilitating reproducible OMNeT++ simulations. Taking the OMNeT++ `FIFO` sample model as an example, we have also demonstrated how to embed simulation configuration files and **Python** analysis code, import simulation data into **Pweave** `DataFrame` with automatic updating of simulation results, and analyze data and present the results in a LaTeX file. The **Pweave** source file of the example described in Sect. 8.4 has been prepared as a minimal template for future reproducible research based on OMNeT++ simulations.

**Acknowledgements** The author is grateful for the constructive comments and feedback from the editors Antonio Virdis and Michael Kirsche, the anonymous reviewers, and the financial support for this work from Xi'an Jiaotong-Liverpool University Research Development Fund (RDF) under Grant RDF-14-01-25.

## References

1. Abbott, B.P., et al.: Observation of gravitational waves from a binary black hole merger. Phys. Rev. Lett. **116**, 061102 (2016). https://link.aps.org/doi/10.1103/PhysRevLett.116.061102
2. Chang, K.: Panel says Bell Labs scientist faked discoveries in physics. The New York Times (2002)
3. Finkelstein, N.: Getting started with Python for R developers. http://n-s-f.github.io/2017/03/25/r-to-python.html. Accessed 03 May 2018
4. Jupyter: http://jupyter.org/. Accessed 03 May 2018
5. knitr: https://yihui.name/knitr/. Accessed 03 May 2018
6. Leisch, F., R-core: Sweave user manual (2018). https://stat.ethz.ch/R-manual/R-devel/library/utils/doc/Sweave.pdf
7. Marcelino, D.: Blog post: What is reproducible research? (2016). http://danielmarcelino.github.io/blog/2016/reproducible-research.html. Accessed 13 Mar 2018
8. OMNeT++ tutorials: result analysis with Python. https://docs.omnetpp.org/tutorials/pandas/. Accessed 03 May 2018
9. pandas: Python data analysis library. https://pandas.pydata.org/. Accessed 09 July 2018
10. Pérez, F., Granger, B.: IPython: a system for interactive scientific computing. Comput. Sci. Eng. **9**(3), 21–29 (2007)
11. Pweave: http://mpastell.com/pweave/. Accessed 03 May 2018
12. R Core Team: R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna (2013). http://www.R-project.org/. ISBN 3-900051-07-0
13. Report of the investigation committee on the possibility of scientific misconduct in the work of Hendrik Schön and coauthors. Bell Labs. (2002). https://media-bell-labs-com.s3.amazonaws.com/pages/20170403_1709/misconduct-revew-report-lucent.pdf
14. Scientific method: Oxford Dictionaries: British and World English (2016). https://en.oxforddictionaries.com/definition/scientific_method. Accessed 13 Mar 2018
15. Signal processing with GW150914 open data. LIGO open science center (2017). https://losc.ligo.org/s/events/GW150914/GW150914_tutorial.html. Accessed 27 Mar 2018
16. The Yale Law School Roundtable on Data and Code Sharing: Reproducible research: addressing the need for data and code sharing in computational science. Comput. Sci. Eng. **12**(5), 8–13 (2010). https://doi.org/10.1109/MCSE.2010.113
17. VirtualBox: https://www.virtualbox.org/. Accessed 03 May 2018

# Chapter 9
# Live Monitoring and Remote Control of OMNeT++ Simulations


**Janina Hellwege, Maximilian Köstler, and Florian Kauer**

## 9.1 Introduction

The conventional approach for performing an OMNeT++ evaluation is based on a three-phase process. First, the network is set up by adjusting parameters and, if necessary, writing C++ source code. Second, the simulation is started and will run for a predefined time span or until another predefined condition is fulfilled. During the simulation, measurement data is recorded in vector and scalar files. Finally, these files are analyzed to reach a conclusion. While being suitable for many applications, this approach has some weaknesses, because influencing a running simulation cannot be easily achieved. Especially for demonstrations and for teaching, qualitative statements about the dynamic behavior are often of high interest as well as the possibility to get immediate feedback, while in-depth quantitative analysis is of secondary importance. Furthermore, the predefinition of termination conditions is difficult in many scenarios. This is most relevant when running large simulation campaigns on remote clusters where human monitoring and intervention can shorten the simulation time significantly.

Conventionally, the various modules that build up an OMNeT++ simulation read their parameters from configuration files during the initialization phase after starting the simulation. This also enables the use of the same module implementation for instances with different parameters. When the simulation is running, parameters can be changed by using the method `handleParameterChange` that has to be implemented by the respective modules and can be used for all modules inheriting `cSimpleModule`. If a parameter changes but this change is not processed in a `handleParameterChange` method, the change will not affect the simulation.

J. Hellwege · M. Köstler · F. Kauer (✉)
Institute of Telematics, Hamburg University of Technology, Hamburg, Germany
e-mail: janina.hellwege@tuhh.de; maximilian.koestler@tuhh.de; florian.kauer@tuhh.de



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_9

This procedure is already implemented in OMNeT++, but issuing a parameter change in the **Qtenv** is not convenient, especially when changing multiple modules at once.

This motivates the proposed implementation of a remote interface based on web technologies that was first presented in [3]. It extends the existing possibilities to change parameters at runtime and allows for a convenient monitoring of live events. The communication takes place via the Web Application Messaging Protocol (WAMP) [4], which allows for flexible interfacing, including the possibility to build a Graphical User Interface (GUI) as a web front-end with HTML and JavaScript. This approach also allows to connect multiple simulations running on the same or different machines to a common user interface. This is especially interesting for comparing the behavior of different protocols side-by-side.

The rest of the chapter is structured as follows: Section 9.2 presents the architecture and the features of the proposed remote interface. It is followed by two tutorials. The first tutorial in Sect. 9.3 extends the well-known OMNeT++ *Tic Toc example* with remote monitoring and control. In the second tutorial in Sect. 9.4, the application for a complex wireless network scenario is presented.

## 9.2 Architecture

The basic architecture of our approach is shown in Fig. 9.1. The OMNeT++ project is extended with classes provided by the `WAMPInterfaceForOmnetpp`. These will communicate with the **Crossbar.io** router [2] by publishing events and providing the possibility to execute Remote Procedure Calls (RPCs). The router forwards these to a GUI or a Command Line Interface (CLI).

WAMP [4] is used as the communication protocol between the components . It provides a convenient publish/subscribe mechanism as well as a RPC service with web applications being the primary target. It is, however, possible to use WAMP in many other application scenarios since there are WAMP Application Programming Interfaces (APIs) for a range of programming languages available. It is also possible to couple clients written in different languages by using the WAMP protocol.

For message transport, any message-oriented, ordered, reliable, and bi-directional transport protocol can be used. *Websockets over TCP* is a common candidate for the connection between the router and a web interface. It is a routed protocol, so all clients connect to one router. This router hence decouples the clients from each other. In our architecture, the open-source **Crossbar.io** router is used.

For the publish/subscribe mechanism, clients subscribe to a topic on the router. Any client can then publish events to the topic that are forwarded to all clients subscribed to that topic. For the remote procedure call service, a client registers a certain procedure at the router. The other clients are able to call the function via the router. The router then invokes the function at the client that registered the function and forwards the result back to the calling client.

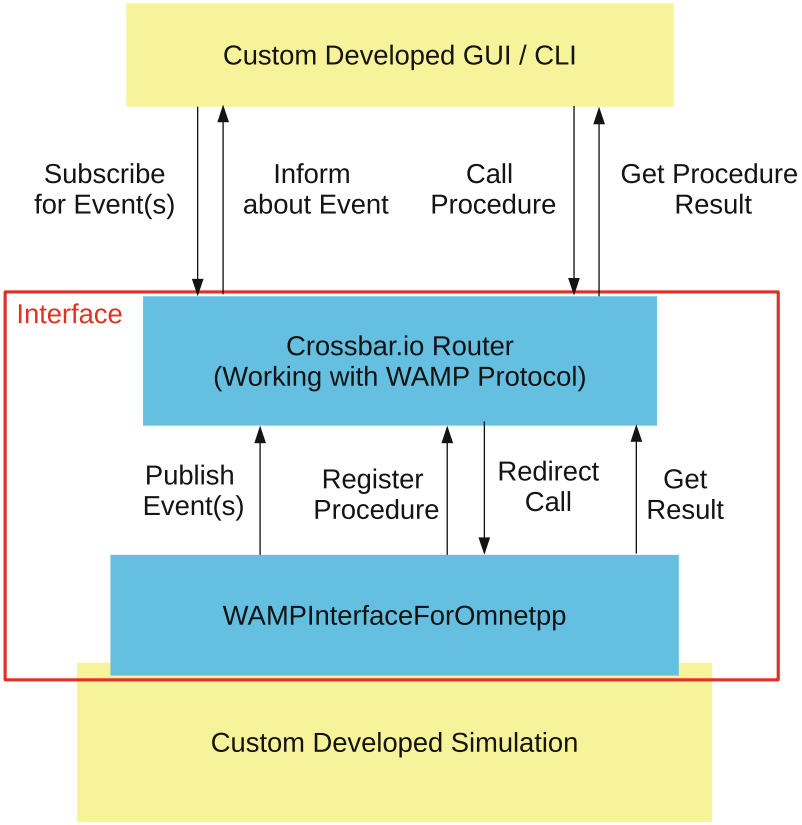



**Fig. 9.1** Architecture of the remote control approach

The WAMP protocol also provides security mechanisms, including the concept of realms. The router has to set up one or several realms and the clients join a realm to perform RPCs or to use the publish/subscribe mechanism. These realms are defined in the router configuration and can restrict which client is allowed to use which communication mechanism. Refer to [4] for details about the security mechanisms.

### 9.2.1 `LiveRecorder` for Publish/Subscribe

The `LiveRecorder` is a class template that can be used to publish events to the router. It is useful for generating live statistics for the user interface. For this, the template is instantiated and registered as a `ResultRecorder`. This can be connected to an OMNeT++ signal. When the signal is emitted, the `LiveRecorder`

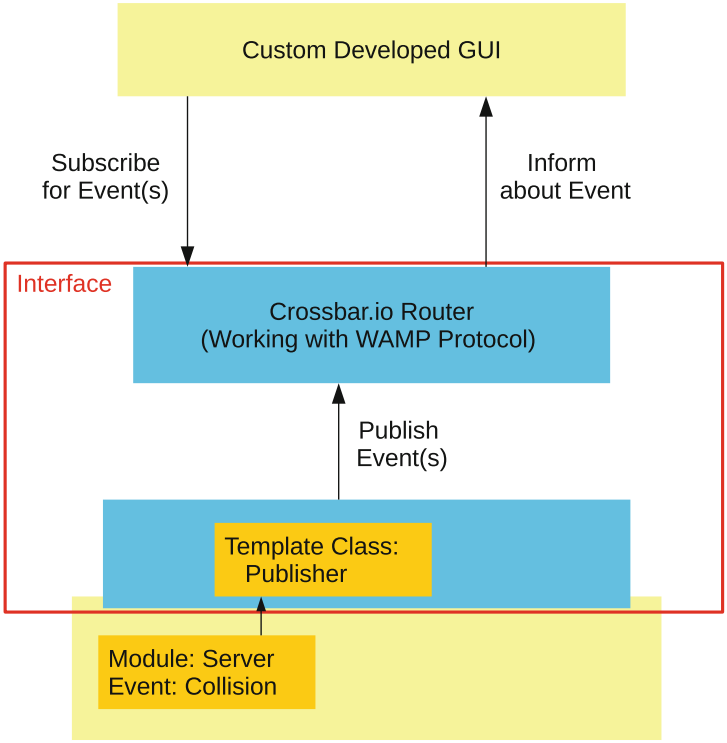



**Fig. 9.2** Publish/subscribe with the `WAMPInterfaceForOmnetpp`

collects the signal, opens a session to the WAMP router, and sends an event to the defined topic. Afterwards the session is closed again. In Fig. 9.2, the procedure is depicted. The user interface subscribes to a topic, so when an event is published by the simulation, it is forwarded to the user interface.

### 9.2.2 `SimulationCallee` for Remote Procedure Calls

The `SimulationCallee` class implements a module that can be added to the simulation via a Network Topology Description (NED) definition. It is used to implement RPCs. The procedures to be called are useful to get information about the simulation modules and to change parameters of modules. This is done by spawning a new thread that opens a session to the WAMP router. The RPC process is shown in Fig. 9.3. The available procedures are registered at the WAMP router so that the user interface can call them. The procedure is then invoked at the simulation and the result is forwarded to the user interface via the router.

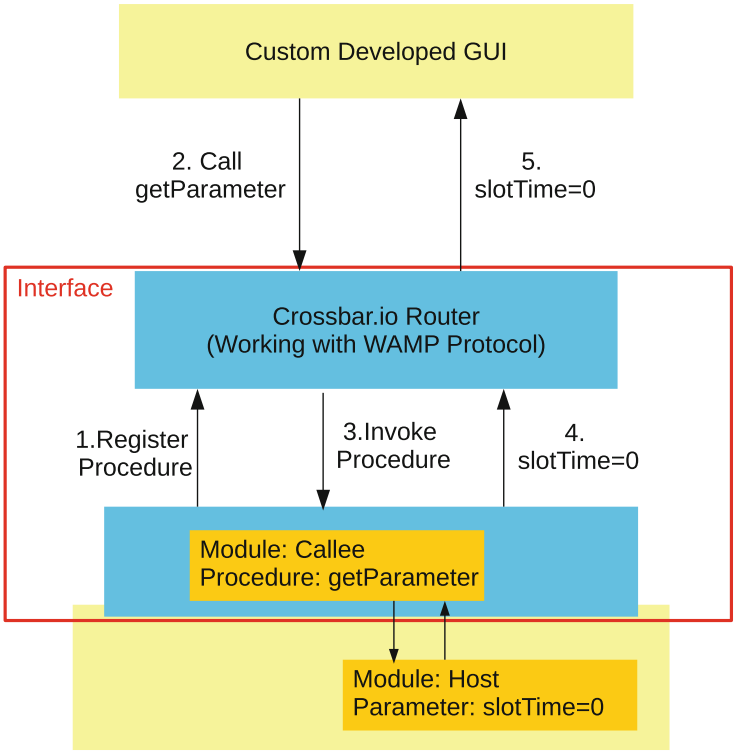



**Fig. 9.3** Remote procedure call with the `WAMPInterfaceForOmnetpp`

In the following, the set of predefined RPCs is presented.

**Exploring the Module and Parameter Tree** This includes a set of RPCs to get information about the modules of a simulation and their parameters. The `getSubmodules` procedure is used to return all submodule names of a given module, as well as their types. If no module name is given to the procedure, it returns the name and type of the main simulation module. To get the list of parameters, the `getParameterNames` procedure can be called. It returns the names, types, and units of the parameters of the module with the name used as the procedure argument.

**Read Parameters** To read the value of a parameter from the running simulation, the function `getParameter` of the `SimulationCallee` module is called remotely. It receives the name of the module and the name of the parameter as arguments and returns the result as tuple. In the module path, it is allowed to reference arrays. This is done by adding `[*]` after the modules name. The returned tuple will consist of one element per matching module. If the result of the `getParameter` function is an expression (for instance, `exponential(3s)`), it is returned with a preceding = to distinguish it from normal strings.

**Set Parameters** The procedure `setParameter` is available to set parameters of simulation modules. Compared to the get procedure, it has another argument for the new value of the parameter. In the module path, it is allowed to replace a module by an asterisk which means that the parameter is changed in all matching modules. Currently, the use of two asterisks to match a longer path of modules is not supported. Like with the `getParameter` function, it is possible to access arrays by adding `[*]` after the module name.

Since the `SimulationCallee` runs in a separate thread and cannot directly interact with the simulation, the information which parameter shall be changed is placed in a queue. To read from the queue, the `SimulationCallee` schedules an event periodically that then reads all entries from the queue and changes the parameters in the simulation. The period in which this event is called is defined by a parameter of the `SimulationCallee`. In order to apply the parameter changes to a module, the method `handleParameterChange` is to be implemented and has to handle all affected parameters.

**Controlling the Simulation Run** It is possible to end the simulation remotely from the WAMP router. For this, the `SimulationCallee` module provides a parameter `stopSimulation` that will end the simulation if it is set to true. It can be toggled by a `setParameter` RPC. It is, however, not possible to start a simulation remotely via this approach because no endpoint is available when the simulation does not run. However, by using a shell script that repeatedly starts the simulation after it has ended, a restart functionality can be implemented that is useful for investigating the initialization phase of a simulation.

### *9.2.3 User Interface*

For controlling the simulation remotely, a custom GUI or CLI has to be developed. There are many WAMP APIs available for a range of programming languages.[1] Since the presented implementation provides a simple interface to the simulation, no OMNeT++-specific client-side libraries are required. Though, some JavaScript abstractions are available for convenient development of own GUIs that can be found in the *scripts* folder of the GUIs used below.[2]

This includes the file *simulation_manager.js* with three functions that make subscriptions and parameter get and set operations easier. The function

```
1 setParameter(uri, mod, param, value)
```

can be used to set a parameter of the simulation. The arguments are the Uniform Resource Identifier (URI) of the host of the WAMP router, the module, the parameter that shall be changed, and the value this parameter shall get. The function

[1] Compare https://crossbar.io/autobahn/.

[2] For example, https://github.com/WAMPInterfaceForOmnetpp/AlohaGUI/tree/master/scripts.

```
getParameter(uri, mod, param, func, args)
```

can be used to get the value of one parameter in the simulation. It is especially useful while setting up the GUI. It also requires the URI, the module name, and the parameter name as arguments. Furthermore, the result function is provided and optional arguments are passed on to the result function in addition to the requested parameter value. The function

```
subscribe(uri, event_name, func, param)
```

can be used to subscribe to a topic on the WAMP router and to define what shall happen on an event. As arguments, it gets the URI of the WAMP router host, the topic name it shall subscribe to, the function that shall be called if this event occurs, and the parameters that are given to this function in addition to the event. The function creates a connection to the WAMP router and subscribes for an event. The connection stays open and waits for events. If an event occurs, the provided function is called.

In the file *parameter_slider.js*, a generic slider for simple parameters is defined. It creates a slider that can be used to change parameters in the simulation. The module is created with the following list of additional parameters that are given to the module constructor:

- the HTML container it is used for,
- the URI of the WAMP router host,
- the module which parameter shall be changed,
- the unit of the parameter value,
- the minimum value of the parameter,
- the maximum value of the parameter, and
- a correction factor that will be multiplied with the slider value before sending it to the simulation.

After the setup, the `getParameter` function from *simulation_manager.js* is called to get the initial value for the slider. Every time the slider position changes, the corresponding parameter is set to the new value via the `setParameter` function.

## 9.3 TicToc Tutorial

The *TicToc example* is a simple network consisting of two nodes that exchange the same message over and over again as shown in Fig. 9.4. It is also a default example coming with OMNeT++. The goal of this tutorial is to integrate the WAMP interface for OMNeT++ into the TicToc project. The interface will publish the information when a Tic has occurred or how many tics have occurred and respond to function calls that change a certain parameter in the simulation. With this functionality a GUI can be used to observe the simulation and change the parameter.

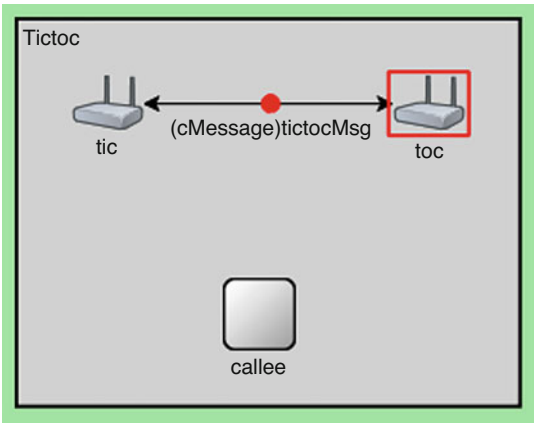



**Fig. 9.4** Screenshot of the running TicToc simulation

### *9.3.1 Installing the Prerequisites*

As a first step, install the `WAMPInterfaceForOmnetpp` project and integrate it into your OMNeT++ workspace. Installation instructions are available online.[3] As a next step, create a TicToc example project as described in the first two parts of the OMNeT++ TicToc tutorial website [5]. Furthermore, download the JavaScript web GUI for this tutorial.[4] If you open the *index.html* file in a web browser, you see the GUI consisting of a diagram with the number of tics over time and a slider which defines the time interval a node waits after a reception of a message until it sends the next one. The GUI is already programmed to connect to the WAMP server via the Autobahn JavaScript library. It subscribes to the topic `com.examples.subscriptions.topic1`, where the receive events will be published. Furthermore, if the slider is moved, it calls the remote procedure `setParameter` to change the time interval one node waits. Now you need to add the interface components to the TicToc example project to make the GUI responsive. For this, you first need to adapt the settings of the TicToc project to reference the `WAMPInterfaceForOmnetpp`. After that, you can start adding interface parts to your project. It is also advised to enable the real-time scheduler to get a realistic timing when observing the GUI by adding the following line to the *omnetpp.ini*:

```
1 scheduler-class = omnetpp::cRealTimeScheduler
```

[3] https://github.com/WAMPInterfaceForOmnetpp/WAMPInterfaceForOmnetpp.

[4] https://github.com/WAMPInterfaceForOmnetpp/TicTocGUI.

### 9.3.2 Adding the Publisher

This subsection presents the parts of the simulation that are required for publishing events. An event will be published every time a message is received at the Tic node during the simulation. A publisher is added and the results that can be seen in the GUI are explained.

First, we need a signal that is emitted every time a message reaches the Tic node. So we define a signal of type `long` in the *Txc.ned*, add it to the *Txc.h*, and register it in the *Txc.cc*. This signal shall be emitted when a message is received by the Tic node, so we add this functionality to the `handleMessage` method. Finally, to publish an event every time this signal is emitted, we need a recorder that publishes the event to the WAMP router. To create this recorder, we first have to include the file *LiveRecorder.h* into our *Txc.cc* file. Then, following the declarations and the `Define_Module` command, we define a character array. This character array holds the name of the topic we want to publish our events to. Because our GUI is programmed to listen to the topic `com.examples.subscriptions.topic1`, we have to use it here, too. Then we create a new result recorder and register it with `Register_ResultRecorder`. As a first argument, we insert an arbitrary recorder name; in this example, we take `tictoc_live`. For the second argument, we initialize an object of the class `LiveRecorder`. It can be found under the namespace `wampinterfaceforomnetpp` and takes the character array we defined as a template parameter. As the last step, the result recorder has to be connected to the signal in the *Txc.ned*. To send the total number of packets since the start, use `tictoc_live(sum)` as the `record` parameter. The resulting file contents are shown in the subsequent Listings 9.1–9.3.

**Listing 9.1** Contents of the *Txc.ned* file

```
1  package tictoc;
2  simple Txc
3  {
4     parameters:
5        bool sendInitialMessage = default(false);
6        @signal[arrival](type=long);
7        @statistic[arrivalSignal](title="the arrivals"; source=arrival; record=
               tictoc_live(sum); interpolationmode=none);
8     gates:
9        input in;
10       output out;
11 }
```

**Listing 9.2** Contents of the *Txc.h* file

```
1 #include <omnetpp.h>
2 namespace tictoc {
3 class Txc : public omnetpp::cSimpleModule
4 {
5   protected:
6     virtual void initialize();
7     virtual void handleMessage(omnetpp::cMessage *msg);
8     static omnetpp::simsignal_t arrivalSignal;
9 };}
```

**Listing 9.3** Contents of the *Txc.cc* file

```
#include "Txc.h"
#include "LiveRecorder.h"
using namespace tictoc;

omnetpp::simsignal_t Txc::arrivalSignal = registerSignal("arrival");
Define_Module(Txc);

char topic[] = "com.examples.subscriptions.topic1";
Register_ResultRecorder("tictoc_live", wampinterfaceforomnetpp::LiveRecorder<
      topic>());

void Txc::initialize() {
   if (par("sendInitialMessage").boolValue()) {
      omnetpp::cMessage *msg = new omnetpp::cMessage("tictocMsg");
      send(msg, "out");
   }
}

void Txc::handleMessage(omnetpp::cMessage *msg) {
   send(msg, "out");
   if (par("sendInitialMessage").boolValue()) {
      emit(arrivalSignal, 1);
   }
}
```

When running the simulation, the plot in the GUI will receive the events and plot them as shown in Fig. 9.5. The bars have timestamps at their bottom which give the time the receive event was send. Do not forget to start the **Crossbar.io** router first, as shown in Fig. 9.6. The slider will not have any functionality yet.

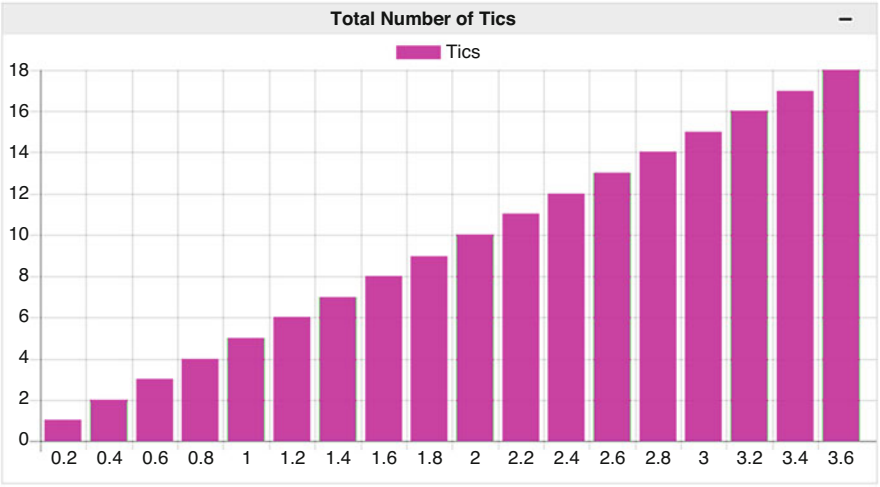



**Fig. 9.5** Tic plot in the graphical user interface

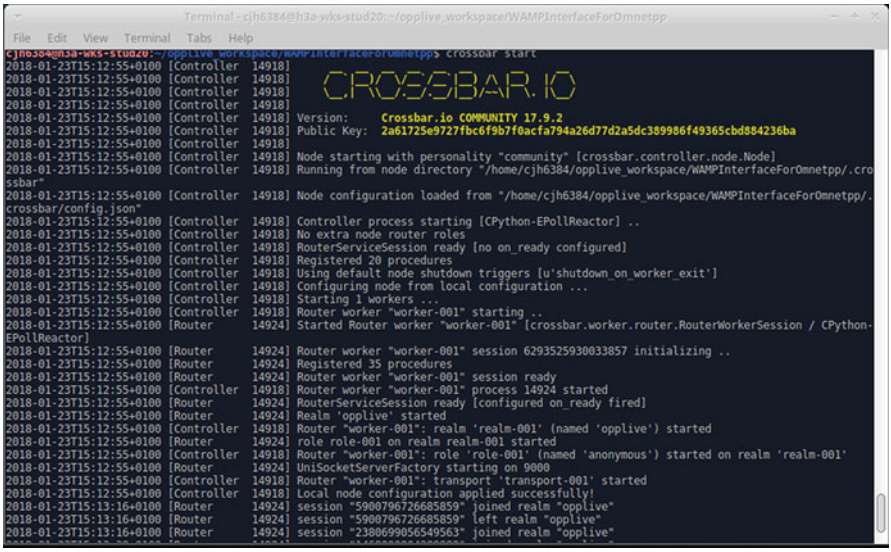



**Fig. 9.6** Running the **Crossbar.io** router

### *9.3.3 Adding the Function Callee*

This section shows how a function *callee* is added to the simulation. This is needed to make it possible to call the remote procedures to read or change parameters. Because the simulation does not yet have any parameters in it, a parameter is created first. The following parameter defines the time a node waits before resending the message.

```
1 parameters:
2   double sleepTime = default(1.0);
```

We also need a member variable of the `Txc` class that holds the value of the `sleepTime` parameter for the simulation. In order to make the nodes sleep for this time, we need to change the way the nodes resend the message. Instead of just returning the message, a timer is scheduled when a message arrives. If the timer is fired, the message is returned to the sender.

We furthermore have to implement the function `handleParameterChange`. It is declared for all modules inheriting from `cComponent` and defines what happens if a parameter is changed during the simulation run. In our case (and in most cases where we use this) it is necessary to reread the parameter value into the member variable.

With this modified version of the TicToc project, we only have to add the `SimulationCallee` module from the `WAMPInterfaceForOmnetpp` project to the *Tictoc.ned* as shown in the code snippet in Listing 9.4.

**Listing 9.4** Contents of the extended *Tictoc.ned* file

```
1 import wampinterfaceforomnetpp.SimulationCallee;
2 ...
3 submodules:
4   callee: SimulationCallee;
```

When running the final simulation code,[5] the slider can be used to modify the sleep interval. This will now have direct influence on the bar chart.

## 9.4 ALOHA Tutorial

The *Aloha example* is another one of the various sample projects that comes with the OMNeT++ simulation framework. It consists of a base station and 20 nodes trying to send data messages over a wireless medium to a base station via the Aloha medium access control protocol. Figure 9.7 depicts the Aloha simulation scenario during the simulation runtime.

The goal of this tutorial is to show how the `WAMPInterfaceForOmnetpp` can easily be integrated into complex existing projects without changing much in the simulation code. Obtain the Aloha example from the OMNeT++ sample directory and prepare it to reference the `WAMPInterfaceForOmnetpp`.

### *9.4.1 Understanding the Network*

We first have a look at the Aloha example to understand how it works. For this, first build and run the Aloha network simulation with one of the available predefined configurations. As you can see, there are possibilities to change the configuration of the network to slotted or unslotted Aloha with different slot time lengths. In the unslotted Aloha protocol, the nodes send to the base station when they have something to send, meaning there is no synchronization between them. In slotted Aloha, the time is slotted and nodes are only allowed to start sending at the beginning of a slot. The length of a time slot is important here, because it defines the throughput of the network. The slots should fit best to the length of a message transmission. Too big slots result in a lot of wasted time. Too small slots result in collisions of messages that are sent in successive slots, so that all these messages are eventually lost. For more information on the Aloha protocol refer to [1].

[5] Available online at: https://github.com/WAMPInterfaceForOmnetpp/TicTocSimulation.

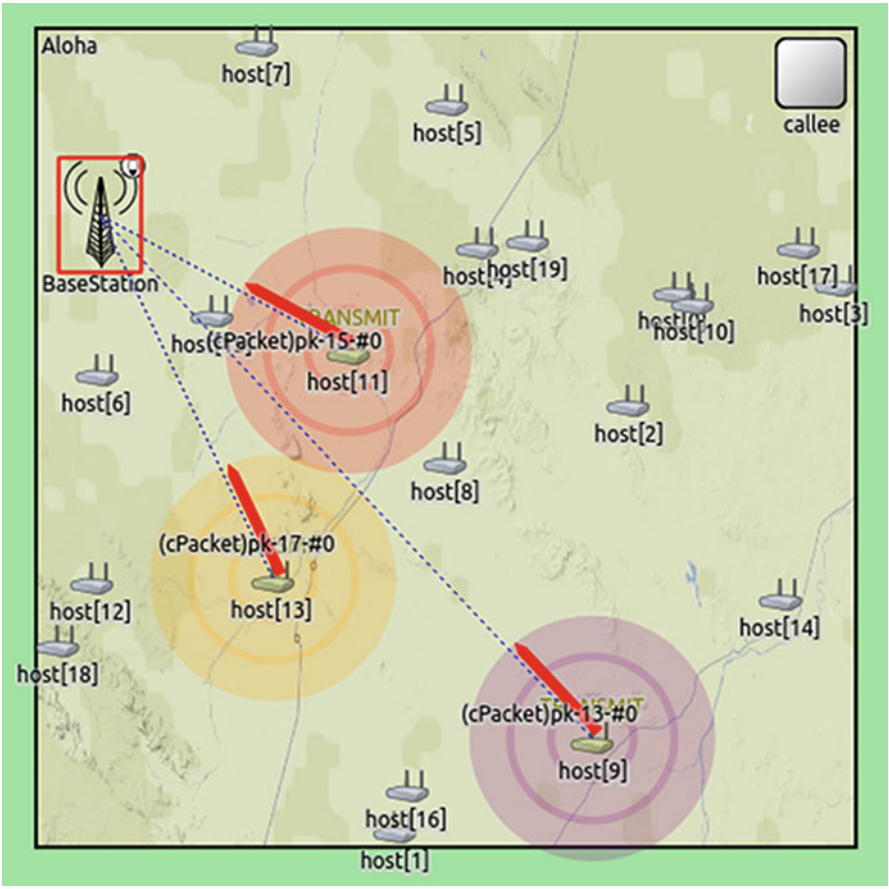



**Fig. 9.7** Screenshot of an exemplary Aloha simulation run

### *9.4.2 Looking into the Simulation Remotely*

Next, you will see how easy it is to get information about the internal structure of a simulation setup with our interface. With the `ModuleBrowser` GUI[6] it is possible to see all modules and parameters of a simulation. For this, add the `SimulationCallee` module to the *Aloha.ned* as described in Sect. 9.3.3. Start the WAMP router in order to use the GUI to explore the simulation. Click on *connect* and you are able to explore all modules and parameters of the simulation.

[6] https://github.com/WAMPInterfaceForOmnetpp/ModuleBrowser.

Additionally, a Python command line interface[7] is available to demonstrate an interface that does not rely on a web browser.

### 9.4.3 Making Parameter Changes Possible

Yet, if you change a parameter in the Graphical User Interface, the change does not effect the simulation. This is because the parameter is not reread by the simulation modules and therefore the initial value of the parameter is used the whole time. In this section we show how to apply the parameter changes in the simulation. To do so we need to implement the `handleParameterChange` function in all modules where it shall be possible to change parameters, for example, as shown in Listing 9.5. If you then change the parameters in the GUI, the changes also apply in the actual simulation.

**Listing 9.5** Implementation of the `handleParameterChange` function

```
1 void Host::handleParameterChange(const char *parname) {
2    iaTime = &par("iaTime");
3    pkLenBits = &par("pkLenBits");
4    slotTime = par("slotTime");
5    isSlotted = slotTime > 0;
6 }
```

### 9.4.4 Publishing Events

Now we examine a GUI specifically designed for the Aloha example.[8] In this GUI, the parameter we made changeable in the last step can be set by a slider. Furthermore, we want to publish events from the simulation so we can see the results in the diagram. This diagram shows how many packets were successfully received at the base station and how many packets are lost due to collisions. To publish events, we need to register two result recorders in the *Server.cc* as shown in Listing 9.6.

**Listing 9.6** Registration of result recorders in the *Server.cc* file

```
1 #include "LiveRecorder.h"
2 char collisions_topic[] = "com.examples.events.collisions";
3 Register_ResultRecorder("Collision_Recorder", wampinterfaceforomnetpp::
      LiveRecorder<collisions_topic>());
4 char receive_topic[] = "com.examples.events.received";
5 Register_ResultRecorder("Receive_Recorder", wampinterfaceforomnetpp::
      LiveRecorder<receive_topic>());
```

[7] https://github.com/WAMPInterfaceForOmnetpp/PythonInterface.

[8] Available online at: https://github.com/WAMPInterfaceForOmnetpp/AlohaGUI.

Now we need to get the correct signals from the simulation and connect our recorders to them. Open the *Server.ned* file. Here you can see that there is already a statistic defined for the collisions signal. The collision signal is emitted every time a collision occurs. The value that is sent with the signal says how many packets are present in the collision. With the following lines from Listing 9.7, the statistics for the `Collision_Recorder` and the `Receive_Recorder` are added.

**Listing 9.7** Adding statistics to the *Server.ned* file

```
1 @statistic[collisionMultiplicity](source=collision; record=vector?,histogram,
      Collision_Recorder; title="collision multiplicity");
2 @statistic[receivedPacks](source=receive; record=Receive_Recorder;
      interpolationmode = none);
```

The receive signal is emitted every time a reception starts and ends. If the reception starts, a 1 is given as a value with the signal. If it ends, a 0 is sent. The given GUI processes the signal accordingly.

### *9.4.5 Examining the Simulation Results*

The simulation code is now finished[9] and we can start the router, the simulation, and the GUI. The packets per 10 s are shown in the diagram. They are divided into the successfully received ones and the ones lost in collisions by color. The green ones are the successfully received packets and the red ones are those packets that are lost in a collision. At the bottom of the GUI we see some sliders with the different parameters in the simulation that are made changeable. First of all, the slot time can be adjusted. A slot time of zero means there are no slots and every message is sent immediately. The length of a message is also adjustable. A longer message results in a longer transmission time. Furthermore, the mean value of the distribution of the randomly chosen wait time can be changed. A bigger value means the time between two send messages on one client gets longer. You can play around with these parameters to see how they change the behavior of the simulation. Using this GUI, for instance, in a teaching environment allows for an intuitive access to the understanding of the Aloha protocol without requiring the students to write any code.

## 9.5 Conclusion and Future Work

The presented architecture with the `WAMPInterfaceForOmnetpp` provides a convenient possibility to monitor and control a running OMNeT++ simulation. It is useful for demonstration, for usage in teaching environments, and for managing

[9] Available online at: https://github.com/WAMPInterfaceForOmnetpp/AlohaSimulation.

large simulation campaigns. The use of the WAMP decouples the simulation from the GUI and allows, for example, an implementation with web technologies.

Two tutorials are provided to guide future users and to demonstrate how easily an existing simulation can be extended by remote monitoring and control. They demonstrate the possibilities enabled by the publish/subscribe mechanism as well as the RPC service. This includes, for instance, the collection of live statistics, the examination of the module tree, and the modification of simulation parameters at runtime. Thereby, the change of the statistics as the reaction to the live modification of parameters is very useful to get insights about the behavior of the analyzed protocols.

Possible extensions for our presented remote interface and the provided GUIs elements include the development of a larger set of widgets and abstractions to speed up the development of GUIs and to enrich the experience for the users. An example would be the possibility to change node positions on the fly to analyze the protocol behavior under dynamic modifications of the network topology.

## References

1. Abramson, N.: THE ALOHA SYSTEM: another alternative for computer communications. In: Proceedings of the AFIPS Fall Joint Computing Conference, pp. 281–285. ACM, New York (1970). https://doi.org/10.1145/1478462.1478502
2. Crossbar.io: Crossbar.io documentation (2017). https://crossbar.io/docs
3. Köstler, M., Kauer, F.: A remote interface for live interaction with OMNeT++ simulations. In: Proceedings of the 4th OMNeT++ Community Summit (2017). http://arxiv.org/abs/1709.02822
4. Oberstein, T.G., Goedde, A.: The web application messaging protocol (2017). http://wamp-proto.org/static/rfc/draft-oberstet-hybi-crossbar-wamp.html
5. OMNeT++: TicToc Tutorial for OMNeT++ (2017). https://omnetpp.org/doc/omnetpp/tictoc-tutorial/

# Chapter 10
# Simulation of Mixed Critical In-Vehicular Networks


**Philipp Meyer, Franz Korf, Till Steinbach, and Thomas C. Schmidt**

## 10.1 Introduction

The automotive market is growing in demand for innovative driver assistance systems, as well as highly automated or even autonomous driving units. In-vehicular communication networks that connect sensors and actuators with Electronic Control Units (ECUs) contribute the basis to these distributed, safety-critical, and highly complex systems. Consequently, their architecture and design are playing an increasingly important role. As of today, in-car communication concepts fall short in meeting the emerging requirements of future driving systems.

High bandwidth demands from distributed visual sensors—e.g., a raw data fusion of laser scanners and cameras—exceed the capacities of current data transmission systems by more than an order of magnitude. For example, a low-resolution camera stream of 7 Mbit/s already exceeds the Controller Area Network (CAN) bandwidth of 0.5 Mbit/s around 14 times. An increasing number of vehicular safety functions pose strict redundancy or Quality of Service (QoS) requirements like latency and jitter. With respect to this growing heterogeneity, current automotive communication architectures and technologies reach their limits. With timing and bandwidth aspects in mind, communication techniques are needed that provide a wide range of real-time communication services. Switched Ethernet is a promising candidate to overcome the challenges of future in-car networks [15] due to its high

P. Meyer (✉) · F. Korf · T. C. Schmidt
Department Informatik, HAW Hamburg, Hamburg, Germany
e-mail: philipp.meyer@haw-hamburg.de; franz.korf@haw-hamburg.de; t.schmidt@haw-hamburg.de

T. Steinbach
Ibeo Automotive Systems GmbH, Hamburg, Germany
e-mail: till.steinbach@ibeo-as.com



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_10

data capacities, its low cost of commodity components, and its flexibility in terms of protocols and topologies [27].

Communication architectures of today's vehicles are composed of different domain-specific technologies such as CAN, FlexRay, Local Interconnect Network (LIN), and Media Oriented Systems Transport (MOST) [19]. Cross-domain communication is enabled via a central gateway that inter-connects a majority of these buses. For premium cars, the simple structuring mechanism of a central gateway reaches its limits in terms of complexity and ability to control. Future developments of automotive services and communication require new concepts and solutions.

The One-Pair Ether-Net (OPEN) Alliance Special Interest Group, which is driven by the automobile industry, focuses on the standardization of certified automotive Ethernet that runs over one single pair of unshielded twisted wires, previously offered as BroadR-Reach by Broadcom [12, 13]. It is very well suited for the challenging Electromagnetic Compatibility (EMC) requirements compliant to the harsh environment in cars. This standard lays the foundation of automotive Ethernet variants. Since established automotive suppliers already offer this technology, Ethernet is a candidate for a new common communication architecture in vehicles [7, 22, 24].

When switching to Ethernet, the in-vehicle network will face a significant paradigmatic change. However, sudden changes in the network architecture of mass-produced cars are unfeasible due to costs and risks. That is why there will be a gradual transition to a flat Ethernet topology. First steps involve the migration of Ethernet into legacy bus topologies. A consolidation strategy with heterogeneous networks formed of an Ethernet core and legacy buses at the edges will allow to preserve investments in knowledge about legacy technologies. Such a mixed architecture forms the beginning of this stepwise transition towards a flat network topology consisting solely of Ethernet links [17].

The design of future vehicular networks requires new tools for experimentation, optimization, and debugging. Several commercial tools exist to analyze in-car networks. In the industry, **CANoe** (by Vector Informatik GmbH) is a popular tool that enables real-time cluster simulations of fieldbuses. As of today, **CANoe** does not provide functionality to simulate real-time Ethernet variants. **SymTA/S** is a commercial timing analyzer (not a network simulator) by Symtavision GmbH that supports Ethernet (standard and Audio Video Bridging (AVB) [10]) as well as common fieldbus technologies. It provides analytical models to calculate load and timing.

To evaluate future in-vehicle networks, suitable tools need to integrate the distributed system components. While today's tool chains focus on bit-correct simulation of fieldbus communication, future environments must enable the developer to analyze effects of congestion and jitter on the car's applications and assistance functions on a system level. For example, it can easily be explored how a third message source influences the traffic between two communication partners. System level simulation increases the understanding of the behavior of future automotive communication architectures and enables quantitative analysis at a higher abstraction level.

The OMNeT++ simulation framework (see Chap. 1) is a well-suited tool and a perfect base to implement such kind of system level simulation based on real-time Ethernet variants [23]. Besides its open-source simulation core, it allows to extend its Eclipse-based Integrated Development Environment (IDE) with custom plug-ins for specialized design and analysis tasks. In this chapter, we introduce both a uniform workflow as well as the required models and tools to design and evaluate future in-vehicle networks.

Experiences with the simulation during research on in-car network architectures showed that the configuration of these large networks is complex and tedious. Thus there was a demand to simplify the description of in-car network scenarios. This demand led to the development of a Domain-Specific Language (DSL) that supports the fast setup of simulations of in-car network architectures.

This DSL is called the Abstract Network Description Language (ANDL) and is integrated as a plug-in for OMNeT++. It enables the design of networks on an abstract layer. Additional benefits like autocompletion, syntax highlighting, validation, autoformatting, renaming, and scoping support the configuration process.

The remainder of this chapter is structured as follows. In Sect. 10.2, we discuss the problem space of mixed critical networks in cars. Our simulation environment, including tools and models, is introduced in Sect. 10.3. Section 10.4 shows an example of our simulation workflow followed by a case study about backbones in premium cars in Sect. 10.5. We conclude with an outlook in Sect. 10.6.

## 10.2 Mixed Critical In-Vehicle Networks

In a current premium car, there are up to 70 ECUs with more than 900 functions interconnected over a heterogeneous in-car network. While control loop applications have strong real-time requirements, other applications such as navigation, firmware updates, or multimedia streaming demand high bandwidth at relaxed timing constraints. With the introduction of high-quality sensors such as high-resolution driver assistance camera systems some functions require high date rates and rigid timing. For safety critical functions like autonomous driving, timing and data rates must be strictly guaranteed. This leads to a wide spectrum of soft and hard time-constrained domains, some of which are covering the entire topology.

The inter- and intra-domain communication is growing and the amount of data exchanged within the car is heavily increasing. In addition to on-board systems, a car will receive off-board data by its backend or by other cars, and infrastructure components such as traffic lights from the vicinity will use Car-to-Car (C2C) or Car-to-Infrastructure (C2I) communication.

Ethernet is the key technology discussed by the major Original Equipment Manufacturers (OEMs) to overcome the challenges of future in-car networking [15]. Consequently, the automobile industry is pushing standardization of a physical layer for automotive applications within the OPEN Alliance Special Interest Group. The 100 Mbit/s automotive certified physical layer is already available (commercially

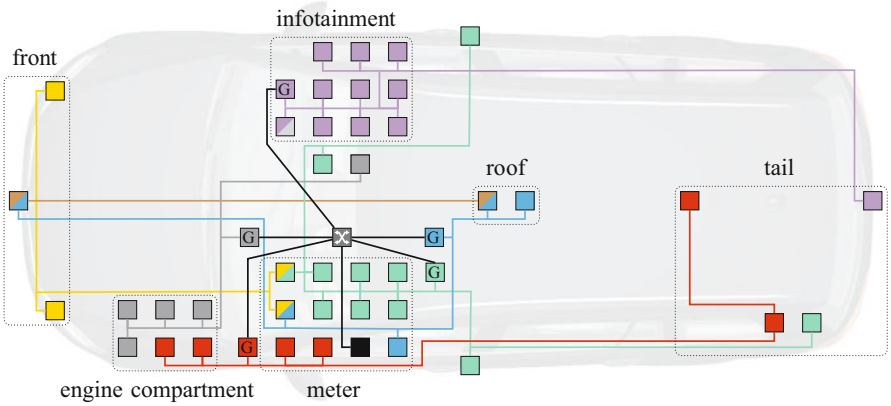



**Fig. 10.1** Domain decomposition of a traditional car network

available as BroadR-Reach, standardization by the IEEE under P802.3bw [12]), 1 Gbit/s automotive links are standardized under IEEE 802.3bp [13].

One possible direction for building future automotive communication is a homogeneous core network of switched Ethernet. Such a flat design reduces complexity by purely switching without the need for gateways between different technological domains in the car. On the other hand, OEMs need to protect their investments in fully developed and proven ECUs as well as their software components. In most cases, these components follow an integrated design that communicates via CAN. Changing to Ethernet hardly justifies the redesign of these components. As part of the migration to a pure Ethernet-based communication layer, gateways will connect CAN buses to the Ethernet backbone (see Fig. 10.1). Corresponding gateways must support a tunneling of CAN message over the Ethernet backbone to interconnect CAN buses of different domains. In addition, CAN message priorities must be preserved. Since CAN supports a maximum payload of 8 B and Ethernet offers a minimum payload of 46 byte, these gateways will allow the aggregation of CAN messages within an Ethernet frame.

Thinking these directions of design, it becomes evident that an in-car network can no longer be considered a collection of closed domains with fixed, offline-configured traffic. Instead, dynamic traffic and changing communication requirements must be foreseen in particular with the integration of new services entering the car via an Internet uplink. Examples for such applications include online software updates, car diagnoses, or updates of on-board information repositories such as navigation maps or meta-data. Such networked applications are easy to host on an Ethernet-based in-car backbone. Even an offloading to a cloud of data or computationally expensive tasks are currently discussed in the context of autonomous driving. Envisioned from these perspectives, future cars may even be characterized as mobile constituents of the Internet of Things (IoT), requiring the network to cope with ever arriving new challenges that include security and safety.

### *10.2.1 Time-Sensitive Networking Technologies*

Automotive communication consists of a collection of distinct services that strictly differentiate in quality, and potentially interfere within a flat common network [26]. Hence, standard switched Ethernet must be extended beyond simple traffic prioritization to provide real-time guarantees. Typical techniques are bandwidth limitation as in IEEE 802.1Qav [8], or rate-constraining as in TTEthernet or Avionics Full-Duplex Switched Ethernet (AFDX) [1] traffic. Further time-triggering techniques are the time-triggered traffic class of AS6802 [21] or the IEEE 802.1Qbv (*Enhancements for Scheduled Traffic*) [11]. The Time-Sensitive Networking (TSN) family of standards bundles all of these techniques.

The IEEE 802.1 AVB standard [10] is a core predecessor of TSN. It enables low latency streaming services and guaranteed data transmission in switched Ethernet networks. This real-time Ethernet extension originates from the multimedia domain where synchronization, jitter, and latency constraints of the applications are high. AVB Ethernet guarantees latencies under 2 ms over seven hops for its best traffic class. IEEE 802.1 AVB consists of different sub-standards required to guarantee the latency, synchronization performance, and compatibility with legacy Ethernet nodes.

*IEEE 802.1Qav* [8] specifies queuing and forwarding rules to guarantee the latency constraints for AVB and the support of legacy Ethernet frames. AVB defines two service classes with different guarantees:

1. stream reservation class-A with a maximum latency of 2 ms, and
2. stream reservation class-B with 50 ms over seven hops.

An AVB network is also able to deal with non-AVB frames. These frames are mapped to the best-effort class (see Fig. 10.2).

Prioritization, queuing, and scheduling mechanisms realize a guaranteed data transmission of AVB frames within strict latency bounds. A transmission of an AVB frame is controlled by using a Credit-Based Shaper (CBS). The AVB frame transmission is allowed when the number of available credits is larger or equal to 0. Implicitly, the CBS has a lower and upper bound to limit the data rate and

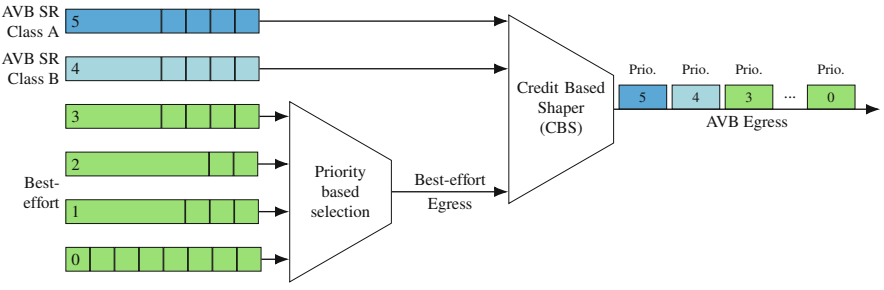



**Fig. 10.2** IEEE 802.1Qav transmission selection algorithms

burstiness of AVB data. The remaining bandwidth is available for non-AVB nodes. To ensure that AVB traffic always has the highest priority, the priority of legacy Ethernet frames by non-AVB nodes is re-mapped to the priorities of the best-effort traffic class. Furthermore, there is a signaling protocol specified in IEEE 802.1Qat [9] to reserve the required resources for AVB frames along the entire path between source and sink. The standard recommends that at most 75% of the total bandwidth shall be reserved for AVB data, while the remaining resources are freely available to best-effort traffic.

Another option of traffic shaping and media access policy for real-time communication in switched networks is time-triggered Ethernet. The TTEthernet protocol was standardized in 2011 by the Society of Automotive Engineers (SAE) [20] under AS6802 [21]. It is a compatible extension of IEEE switched Ethernet and uses topologies formed of full-duplex links. The TTEthernet media access strategies are similar to the IEEE 802.1Qbv (*Enhancements for Scheduled Traffic*) Ethernet amendment, developed by the IEEE TSN working group.

Time-triggered Ethernet variants are operating on an offline-configured schedule with dedicated transmission slots for all real-time messages shared among all network participants. This enables a *coordinated* Time Division Multiple Access (TDMA) media access strategy with deterministic transmission and predictable delays. TDMA prevents congestion on outgoing line cards and thereby enables isochronous communication with bounded low latency and jitter. To enable this access scheme, a fail-safe synchronization protocol has to provide a precise global time among all participants.

In addition to time-triggered, TTEthernet defines two other event-triggered message classes: rate-constrained is comparable to the link layer of the *ARINC-664 (AFDX)* protocol [1]. Bandwidth limits for each stream and sender enable the real-time guarantees. The so-called Bandwidth Allocation Gaps (BAGs) implement the bandwidth limits. The BAGs define the minimum distance of two consecutive frames of the same stream (called virtual link). The rate-constrained traffic is comparable to Ethernet AVBs stream reservation classes A and B. Similarly, it uses strict priorities for traffic with different real-time requirements.

The best effort traffic conforms to standard Ethernet messages transmitted with the lowest priority. The best-effort class is used for the transmission of cross-traffic. It allows the integration of hosts that are unaware of the time-triggered protocol and remain unsynchronized.

## 10.3 Simulation Environment

The simulation models introduced in this section were developed for the simulation of in-car networks but can be used for other systems as well. To simplify the installation, an OMNeT++ plug-in is provided that offers an automated installation process as well as an update procedure. Figure 10.3 gives an overview of the contributed simulation models and their place in the software stack of the toolchain.

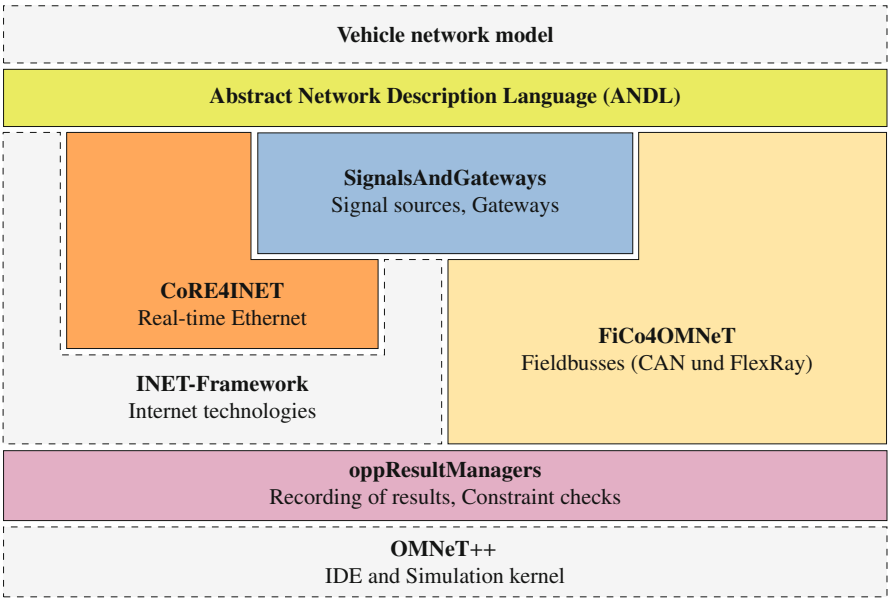



**Fig. 10.3** Overview of the contributed simulation environment

All simulation models use the IDE and the simulation kernel of OMNeT++. The optional *oppResultManagers* module framework writes simulation results directly to Packet Capture next generation (PCAPng) files or a database. Fieldbus Communication for OMNeT++ (FiCo4OMNeT) contributes simulation models for CAN and FlexRay fieldbuses. For both there are no dependencies to other simulation models. Communication over Real-time Ethernet for INET (CoRE4INET) provides simulation models for real-time Ethernet communication. It uses the Ethernet layer implemented in the INET Framework. INET models from higher layers can communicate via CoRE4INET models, too. The *SignalsAndGateways* model collection implements gateways. They support different strategies for the translation of communication between real-time Ethernet and fieldbuses. Thus, it depends on CoRE4INET and FiCo4OMNeT.

As simulation input, a vehicle network must be described using **.ini* and **.ned* files. It is time consuming to describe several variants of a vehicle network using these files. Hence, such a network should be modeled in the ANDL DSL. A compiler translates network designs given in ANDL to the corresponding **.ini* and **.ned* files used by CoRE4INET, FiCo4OMNeT, and SignalsAndGateways models.

All simulation models and plug-ins are published as open-source software and can be downloaded from our website [4].

### 10.3.1 Domain Specific Language for Automotive Networks

Configuring the simulation of large heterogeneous networks is complex and lengthy. To reduce this effort and to let the developer focus on the design task, we developed a Domain-Specific Language for the description of heterogeneous in-vehicle network designs. It is called Abstract Network Description Language and provides an easy and assisted way to design a network in an Eclipse environment. It is implemented as an Eclipse plug-in and thus fits into the OMNeT++ IDE. The plug-in provides syntax highlighting as well as context-aware code completion. For typical vehicle networks that require few TDMA-based communications, a scheduling algorithm generates a feasible schedule for TDMA traffic. It should be used as starting point for improving the TDMA schedule [14].

ANDL is implemented as an OMNeT++ plug-in using Eclipse's *Xtext* technology [2, 5]. Xtext is a framework for development of programming- and domain-specific languages. Using a grammar that has been extended by some specific elements, the DSL will be described. Based on this input, Xtext generates a parser and a code editor that will be plugged into the OMNeT++ IDE. Using a set of Java classes, which have been generated by Xtext, the compilation from ANDL to **.ini* and **.ned* of OMNeT++ will be defined.

### 10.3.2 CoRE4INET

CoRE4INET is a suite of real-time Ethernet simulation models. Currently, it supports the AS6802 protocol suite, traffic shapers of AVB Ethernet, IEEE 802.1Q, and models for mapping Internet Protocol (IP) traffic to real-time traffic classes.

The center of the CoRE4INET models is the implementation of media access strategies for different traffic classes. By combining these strategies, new traffic shapers can be designed that are able to forward real-time traffic of different standards on the same physical link. For example, it is possible to combine time-triggered traffic of AS6802 with credit-based shaping of Ethernet AVB to form a new Time-Aware Shaper (TAS) that can handle both classes in parallel [16]. This allows to evaluate new concepts that are currently under standardization or are even not yet assessed.

For incoming traffic, the models contain traffic selection and constraint checks. To simulate time-triggered behavior and time-synchronization, CoRE4INET provides models for oscillators, timers, and schedulers. Oscillators allow to implement the behavior of inaccurate clocks with their unique influence on real-time communication. Finally, CoRE4INET contains application models for simple traffic patterns and traffic bursts.

Selected simulation models were checked against analytical models of the different specifications and evaluated in empirical tests using real-world hardware [22].

### 10.3.3 FiCo4OMNeT

CAN [28] is a fieldbus widely used in automobiles. Future vehicle networks require mixed operation of Ethernet and CAN. FlexRay [6] is used in a few premium vehicles. A migration from FlexRay to real-time Ethernet is expected for the next generation of these vehicles. Hence, FiCo4OMNeT provides simulation models for CAN and FlexRay.

Exploiting the fact that all FlexRay nodes are connected to the same bus, the static segment of FlexRay provides a TDMA-based communication. FiCo4OMNeT implements this behavior using a clock and an oscillator model, which are simpler than the one of CoRE4INET. In order for the two models to remain independent, FiCo4OMNeT provides a simple clock- and oscillator model. Similar to CoRE4INET, it contains application models for CAN and FlexRay applications with simple traffic patterns. The fieldbus models in FiCo4OMNeT were originally checked against results of the **CANoe** simulation environment [3], an industry standard software for the analysis of CAN bus communication.

### 10.3.4 SignalsAndGateways

Using gateways, the SignalsAndGateways simulation model interlinks between real-time Ethernet and fieldbuses. These gateways are specific network nodes that translate between legacy bus technologies and (real-time) Ethernet. To be as flexible as possible, a gateway consists of the following three submodules.

Path Finding

The router module decides to which components an information is forwarded. It receives messages in their original representation (CAN or Ethernet frame). Based on forwarding rules, it selects the path that the message will take. Using the header information of the message, a forwarding rule defines a CAN bus or an Ethernet node that should receive the information of the message. A statically defined routing table stores all forwarding rules. If there is no entry in the routing table for a message, it is dropped. Otherwise it will be sent to all destinations given by the forwarding rules that match to the message.

There is no limit of buses and links a gateway can be connected to. The gateway can also translate between fieldbus technologies, thus it is also applicable to legacy designs with multiple buses interconnected over a central gateway.

Buffering

Gateways support aggregation strategies to improve bandwidth utilization of different technologies. CAN messages, for example, have a maximum payload of 8 B, while Ethernet messages have a minimum payload of 46 B. If an Ethernet frame encapsulates only one CAN message, the rest of the minimum payload would be padded and bandwidth would be wasted. Aggregation strategies implemented in the buffer modules allow to release frames in groups, according to different strategies. These strategies are implemented in the buffer modules, too.

Aggregation strategies have a huge impact on the latency of messages passing a gateway. All strategies delay frames to collect multiple messages before aggregating them into a single frame. The most popular strategy implemented in the buffer is the pooling strategy with holdup time. The holdup time of a message defines the maximum acceptable delay for this message. Each message is assigned to a pool, while multiple different messages share the same pool. To each message a holdup time is assigned. On arrival of a frame in the pool, its holdup time is compared with the holdup time of the pool. If the holdup time of the frame is shorter, the holdup time of the pool is adjusted accordingly. When the holdup time of the pool is expired, all messages in the pool are released together in one frame. The modular architecture of the gateway allows to easily add more aggregation strategies.

Transformation

Transformation modules implement the translation between different communication technologies. The strategies transparently map information between fieldbuses and Ethernet. Currently, there is a simple mapping between fieldbus frames and raw (layer 2) Ethernet frames. The modular gateway architecture allows to easily add sophisticated mappings, e.g., when higher layer application protocols shall be used.

Similar to real-world gateways, gateway nodes can host applications that are not related to gateway functionality. Thus, gateways can be added to control units that also host application software.

### *10.3.5 Result Manager*

The OMNeT++ IDE already contains result analysis tools. We extended those built-in tools to simplify the analysis in specialized use-cases and developed interfaces to interconnect the OMNeT++ simulation with established industry products.

oppResultManagers is a set of modules for OMNeT++ simulations. Instead of simulation models, it contains so-called ResultManagers, which are responsible for writing out simulation results. The OMNeT++ vector and scalar files, as well as the eventlog are built-in instances of ResultManagers. The oppResultManagers project adds ResultManagers, for example, for PCAPng, SQLite, postgreSQL, and Constrained Check. It is also possible to use several managers in parallel.

## 10.4 Simulation Process

The simulation process (see Fig. 10.4) starts with the network design. ANDL is used to describe the required nodes, as well as the desired network topology, and the mapping of messages to different traffic classes. Afterwards, our toolchain automatically generates an executable simulation configuration that is run using the simulation models for real-time Ethernet and fieldbuses. After the simulation run, the results are analyzed with various result analyzers that are built into the OMNeT++ IDE, provided as additional plug-ins, or interconnected using databases and specialized output formats such as PCAPng. This section presents an example workflow of the simulation process—from network description to result analysis.

### *10.4.1 Network Modeling*

The first step is the network description. This is done with the ANDL domain-specific language. Listing 10.1 shows an example network consisting of two CAN buses interconnected over a real-time Ethernet backbone described in the ANDL.

The definition of the scenario starts with the required `devices` of the network. The `connections` section arranges previously defined devices into a network topology. This section shows also an additional way to instantiate `ethernetLink`. It can be created in the specific link definition without the need for defining a name.

The topology can be divided in several segments with different configurations for messages. In the example, there is one segment for the Ethernet part called `backbone` and one segment for the CAN bus part called `canbus`. A message traversing the border of a segment will be translated from the representation of the sending segment into the representation of the receiving one. The last part of the definition is the actual communication taking place. There is one message transmitted from `cn1` to `cn2` and one message transmitted from `en1` to `en2` in the example. The mapping of each message defines how the message is represented in the different segments. In the example, the message `msg1` is a CAN frame with Identifier (ID) `37` on the bus and a time-triggered message with the critical traffic ID `102` on the real-time Ethernet backbone.

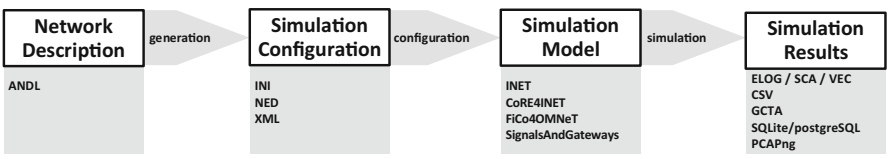



**Fig. 10.4** Workflow of simulation projects—from network description to result analysis

**Listing 10.1** ANDL code example with comments

```
types std {       // Types can be defined and reused
 ethernetLink ETH { // Definition for Ethernet link
  bandwidth 100Mb/s; // Link has bandwidth of 100MBit/s
 }
} //it is also possible to define types in a separate file

network smallNetwork { // Network name is smallNetwork
 inline ini {    // Inline ini for special parameters
```ini
record-eventlog = false
```
 }               // Parameters are inserted into .ini
 devices {       // Define all devices in the network
  ethernetLink eth1 extends std.ETH; // First Ethernet cable
  canLink cb1;   // First CAN bus
  canLink cb2;   // Second CAN bus
  node cn1;      // First CAN node
  node cn2;      // Second CAN node
  node en1;      // First Ethernet node
  node en2;      // Second Ethernet node
  gateway gw1 {  // Gateway for first CAN bus
   pool gw1_1; // Pool for Aggregation of CAN frames
  }
  gateway gw2;   // Gateway for second CAN bus
  switch s1;     // Real-time Ethernet Switch
 }
 connections {   // Physical connections (Segments = groups)
  segment backbone { // Ethernet Backbone part
   en1 <--> eth1 <--> s1;  // Ethernet Link
   en2 <--> {new std.ETH} <--> s1; // Ethernet Link
   gw1 <--> {new std.ETH} <--> s1; // Ethernet Link
   gw2 <--> {new std.ETH} <--> s1; // Ethernet Link
  }
  segment canbus { // CAN bus part (buses share config)
   cn1 <--> cb1; // CAN node connected to first bus
   gw1 <--> cb1; // Gateway connected to first bus
   cn2 <--> cb2; // CAN node connected to second bus
   gw2 <--> cb2; // Gateway connected to second bus
  }
 }
 communication { // Communication in the network
  message msg1 { // First message definition
   sender cn1;  // First CAN node is sender
   receivers cn2; // Second CAN node is receiver
   payload 6B;  // Message payload is 6 Bytes
   period 1ms;  // 1ms cyclic transmission
   mapping {    // mapping to traffic class, id, gw strategy
    canbus: can{id 37;}; // Message ID 37 on CAN
    gw1: pool gw1_1{holdUp 2ms;}; // Aggregation time
    gw2;        // gw2 also responsible for the msg path
    backbone: tt{ctID 102;}; // TT traffic on backbone
   }
  }
  message msg2 { // Second message definition
   sender en1;  // First Ethernet node is sender
   receivers en2; // Second Ethernet node is receiver
   payload 500B; // Message payload is 500 Bytes
   period 125us; // 125us cyclic transmission
   mapping {    // mapping to traffic class
    backbone: avb{id 1;}; // AVB traffic on backbone
   }
  }
 }
}
```

Beside the features shown in this example, ANDL provides more parameters to describe traffic flows or aggregation strategies. Commonly used components can be defined in include files, e.g., an Ethernet link with 100 Mbit/s, and used in several places. Furthermore, ANDL provides inheritance. It is hence possible to define primitive stencils for components that are later refined during the instantiation.

Currently, ANDL supports only the most common simulation parameters. For more sophisticated configurations `inline ini` code can be used. Parameters defined in the `inline ini` sections are directly copied into the resulting *.ini* files in an additional `with_inline_ini` configuration. Hence, `inline ini` definitions override generated definitions, which are placed in the `General` configuration.

### *10.4.2 Experimentation*

In comparison to the compact description in ANDL, the size of the generated OMNeT++ simulation configuration (*.ini/.ned/.xml*) has more than 250 lines. Nevertheless, all relevant parameters for the in-car network designer are available in ANDL. The resulting network is depicted in Fig. 10.5.

It is shown that the generation process has created the defined topology. Stimuli, TDMA scheduling, and gateway strategies are generated, too. A simulation can be run immediately. Furthermore, all OMNeT++ features like distribution functions and configurations for concurrent simulations runs can be done in the `inline ini` part of the ANDL description. Using ANDL in this way supports a fast experimentation process. Changes on topology, stimuli, or gateway strategies

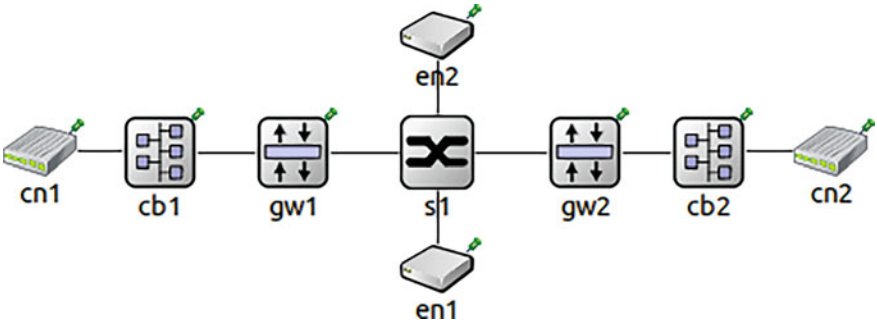



**Fig. 10.5** ANDL generated network consisting of two CAN buses and a real-time Ethernet backbone with two gateways, two Ethernet nodes, and one switch

are done in ANDL and simulated in OMNeT++ for an interactive analysis and comparison of different in-car network settings.

### *10.4.3 Result Analysis*

All known OMNeT++ tools to analyze and visualize result data can be used to view in-car protocol specific results, too. To find the corresponding data for a device defined in an ANDL description a user simply uses the defined name to filter the result set. All user-defined ANDL names are adopted to the OMNeT++ simulation configuration. An example for a protocol-specific result is the credit of the AVB shaper. Figure 10.6 shows an extract of the credit vector for port 1 in switch `s1`.

The oppResultManagers models enable distributed analysis of simulation results. It is realized using database ResultManagers. Listing 10.2 shows how to enable database ResultManagers in a **.ini* configuration file of the simulation.

**Listing 10.2** Enabling database ResultManagers

```
outputscalarmanager-class="cPostgreSQLOutputScalarManager"
outputvectormanager-class="cPostgreSQLOutputVectorManager"
postgresqloutputmanager-connection="dbname=testdb user=testuser password=
      testuser port=15432"
```

The exemplary **postgreSQL** manager allows to write simulation results to a central database server in the network, while simulations are executed on a

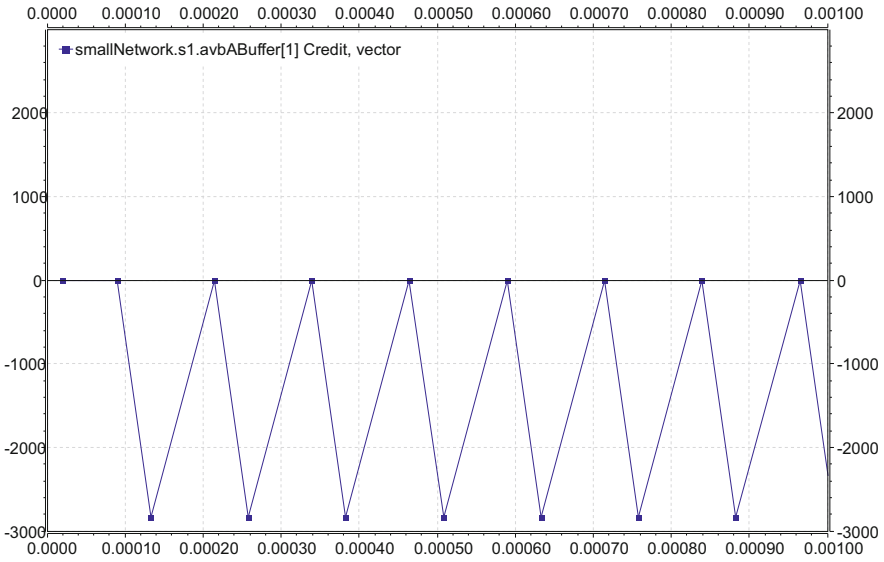



**Fig. 10.6** AVB credit vector (*s1* port 1) as seen in OMNeT++

distributed cluster of nodes. Several users can access the results concurrently without the necessity to distribute the result files. This allows to transfer the load of the simulation as well as result analysis from the user's workstations towards strong servers and large centralized storage systems. The drawback of this solution is a slight performance decrease due to the overhead of sending results over the network as well as delays due to the databases lock mechanisms when it is accessed concurrently. Using a database system, OMNeT++ simulations can be easily attached to a wide range of analysis tools such as **R** [18] using a database driver. With this approach, a ResultManager can be selected to fit the specific result analysis requirements.

## 10.5 Case Study: Automotive Backbone for Premium Cars

This section presents a system-level simulation case study based on an anonymized communication matrix of the Volkswagen Golf 7 that supports the modular transversal toolkit platform of Volkswagen. It is extended by high bandwidth communications that transport measurements of two cameras and two lidars to support features like sensor fusion based on raw data. This case study demonstrates the relevance of system-level simulation within the field of future in-car network designs, gives an impression of how to use and handle the simulation tools, and finally provides selected results obtained by this case study. All results are the outcome of OMNeT++ simulations. The case study was supported by the German Federal Ministry of Education and Research (BMBF) under the project RECBAR [25].

### *10.5.1 Case Study and Metrics*

To get results for a comparison, the first simulation analyzes the current CAN-based communication structure of premium cars (see Fig. 10.7). It is based on a central CAN gateway that connects the domain-specific CAN buses and realizes the exchange of CAN frames between these buses. The design consists of seven public and two private CAN buses. All ECUs connected to these buses and the corresponding periodic CAN traffic will be simulated (cf. Tables 10.1 and 10.2). CAN traffic based on acyclic messages will not be simulated. Even if the CAN identifiers have been anonymized, the prioritization remains in accordance with the original Volkswagen modular transversal toolkit communication matrix.

In the second simulation, the central CAN gateway is replaced by an Ethernet switch and CAN-to-Ethernet gateways (see Fig. 10.8). The final simulation analyzes a network that consists of a real-time Ethernet backbone using three real-time switches and several additional nodes with high bandwidth applications such as

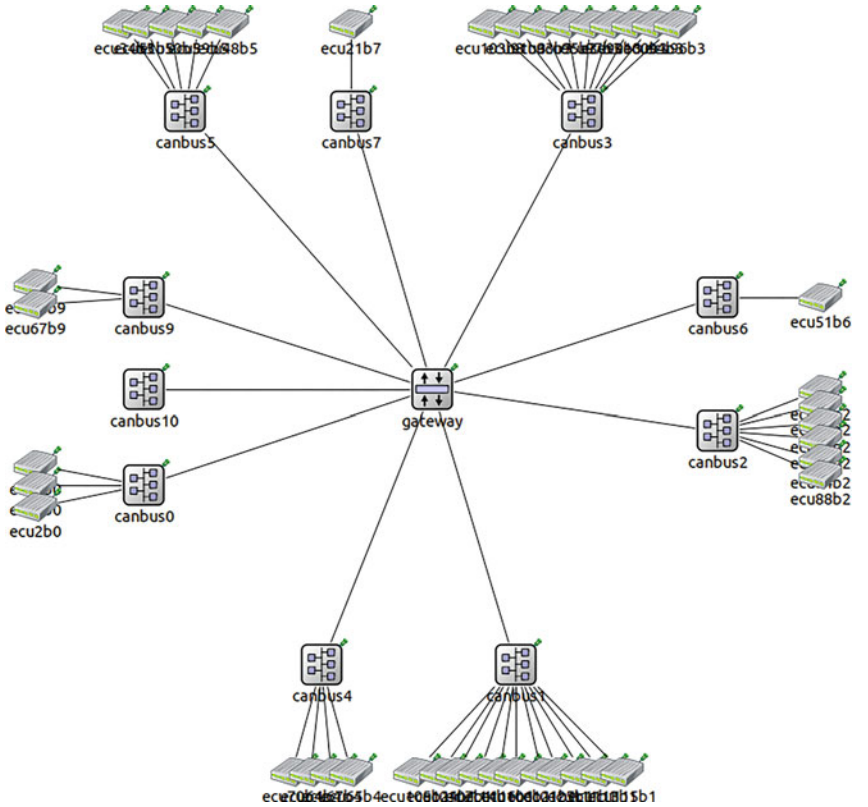



**Fig. 10.7** Central CAN gateway design

**Table 10.1** Number of ECUs per CAN bus

| CAN bus | Number of ECUs | CAN bus | Number of ECUs |
|---|---|---|---|
| canbus0 | 3 | canbus1 | 11 |
| canbus2 | 6 | canbus3 | 9 |
| canbus4 | 4 | canbus5 | 5 |
| canbus6 | 1 | canbus7 | 1 |
| canbus9 | 2 | canbus10 | 0 |

high definition cameras and laser scanners (see Fig. 10.12). They represent the first steps of the gradual transition to a flat Ethernet topology.

The simulation records the following metrics:

- *Utilized bandwidth*: This is the bandwidth used on all Ethernet links and CAN buses, including all protocol overheads. Results can be obtained from the scalars `bitsPerSec` of the `canBusLogic` module and `bits/sec sent` or `bits/sec rcvd` of the `mac` module.

**Table 10.2** Number of CAN messages per period

| Period [ms] | Number of messages | Period [ms] | Number of messages |
|---|---|---|---|
| 10 | 23 | 20 | 38 |
| 25 | 1 | 30 | 1 |
| 40 | 33 | 50 | 17 |
| 60 | 2 | 80 | 10 |
| 100 | 69 | 150 | 10 |
| 160 | 1 | 200 | 50 |
| 450 | 2 | 500 | 49 |
| 1000 | 38 | 2000 | 3 |

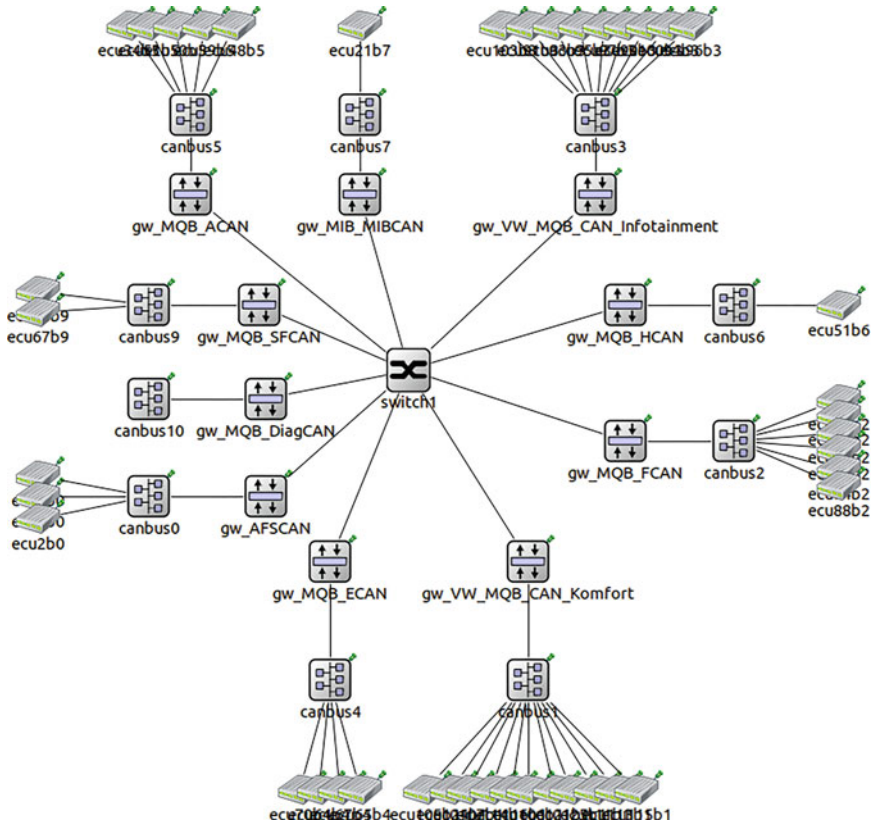



**Fig. 10.8** One Ethernet switch design

- *Latency*: The simulation records End-to-End (E2E) latency across the entire route between the sending and the receiving ECU. It includes the time spent within gateways. The time starts at the point in time when a data source provides a package for sending. It ends at the time of arrival of the package at a data sink.

If applicable, the latency on the Ethernet network is investigated—without the CAN bus transmission. This highlights the effects of Ethernet configurations.

The latency will be recorded for each ID at each receiver and each station on the path from the sender to receiver. Since a large amount of data is generated in the simulation, this section gives only the maximum and minimum latencies.
Many modules in our simulation models have the `rxLatency` vector results. For E2E latency, we are looking at the consuming applications.

- *Jitter*: This metric is defined for periodic messages. It is the absolute value of the maximum difference of the latency belonging to consecutive messages. The jitter is ignored for messages sent from different points in the network to the same sink.
- *Queue Length*: The simulation observes the length of all queues in the network. Moreover, the packets lost due to buffer overflow will be monitored. This is also important for the design of the target hardware.
  For obtaining the queue length statistics we look at the `QueueLength` vectors.

### *10.5.2 Central CAN Gateway Design*

This scenario depicts the series configuration of the vehicle and simulates the initial situation without Ethernet backbone. This represents the actual communication topology of many current cars. The recorded metrics serve as reference for further architectural variants and configurations. Figure 10.7 shows the configured OMNeT++ simulation. Each CAN bus is connected to the central CAN gateway which routes inter-domain traffic. There are a total of 416 different messages on the buses, of which 216 are transported via the gateway. The central CAN gateway delays messages by 60 µs. This value corresponds to an average value measured in the real vehicle.

For simulation model validation the utilized bandwidth was determined analytically based on the cycle times given in the communication matrix. Table 10.3 compares analytical and simulation results. Due to a different recording technique the deviation is up to 2339%. The analytical approach is based on the cycle times, solely. In contrast, the simulation takes only packages into account that are transmitted over the bus. Packages that are still waiting for transmission in the gateway queue or packages that are waiting for the access to the bus are not included here. Simulated bandwidth therefore lies slightly below the analytically calculated.

E2E latencies for all CAN messages were determined with Table 10.4 depicting the results. It shows the great influence of CAN message priority on latency. The reason is the CAN bus arbitration, which always favors the message with the lowest CAN ID. While the maximum latency of the highest prioritized message (lowest CAN ID value) is less than 1 ms, it rises to almost 18 ms for the lowest prioritized message (highest CAN ID value). This effect affects the average latency, too.

The same effect occurs for the jitter of CAN messages. Typically, the jitter of low prioritized CAN messages is higher. For the analysis of the jitter, a distinction must

**Table 10.3** Utilized bandwidth: analytical vs. simulation results

| Bus | Analytical result [kbit/s] | Simulation result [kbit/s] | Deviation [ % ] |
|---|---|---|---|
| canbus0 | 97.919 | 97.918 | 0.001 |
| canbus1 | 128.130 | 127.329 | 0.625 |
| canbus2 | 237.788 | 232.225 | 2.339 |
| canbus3 | 63.125 | 63.121 | 0.006 |
| canbus4 | 113.238 | 113.232 | 0.005 |
| canbus5 | 221.631 | 220.509 | 0.506 |
| canbus6 | 5.215 | 5.214 | 0.019 |
| canbus7 | 2.801 | 2.800 | 0.036 |
| canbus9 | 184.602 | 184.589 | 0.007 |

**Table 10.4** Exemplary end-to-end latencies

| CAN-ID | Maximum end-to-end latency [µs] | Average end-to-end latency [µs] |
|---|---|---|
| 17 | 946.707 | 572.445 |
| 331 | 8465.906 | 845.920 |
| 510 | 17,974.989 | 1168.952 |

**Table 10.5** Comparison of minimal and maximal jitter

| | Local | | Via gateway | |
|---|---|---|---|---|
| Bus | Min [ms] | Max [ms] | Min [ms] | Max [ms] |
| canbus0 | 0.574 | 7.511 | 4.301 | 5.866 |
| canbus1 | 1.910 | 30.629 | 0.983 | 16.295 |
| canbus2 | 1.504 | 15.827 | 0.961 | 23.196 |
| canbus3 | 1.702 | 9.430 | 0.935 | 23.995 |
| canbus4 | 2.154 | 9.735 | 0.957 | 24.505 |
| canbus5 | 0.564 | 19.383 | 1.868 | 20.044 |
| canbus6 | No local traffic | | 8.017 | 19.383 |
| canbus7 | No local traffic | | 5.278 | 15.920 |
| canbus9 | 4.346 | 25.641 | 0.959 | 23.707 |

be made between messages that are transmitted locally on a CAN bus and messages transmitted via the central gateway. For each CAN bus, Table 10.5 gives the jitter for CAN messages consumed on this bus. As a result, messages that are forwarded via the central gateway do not necessarily have a higher jitter. Rather, the jitter is a mix of effects in the gateway, arbitration on multiple buses, and message prioritization. For example, on `canbus9` the minimum jitter of local messages is higher than messages that traverse the gateway due to a higher prioritized CAN message.

For an architecture with a central CAN gateway these measurements show that the arbitration of the CAN messages has a significantly higher impact on jitter and latency than the delay caused by the gateway (60 µs). This is a good starting position for architectures based on an Ethernet backbone.

### *10.5.3 One Ethernet Switch Design*

This architecture is an intermediate step to an architecture with an Ethernet backbone. The central CAN gateway is replaced by an Ethernet switch and CAN-Ethernet gateways (see Fig. 10.8). This simulation uses the same stimuli (CAN messages) as the previous simulation.

Each CAN bus is connected to its own CAN-Ethernet gateway. In accordance with the behavior of the CAN central gateway, each CAN message, which flows via the central CAN gateway in the original design, flows via the Ethernet switch in this architecture. On the Ethernet side it is sent from the source gateway to the destination gateway. A CAN message may be forwarded to several CAN buses. In this case, the simulation implements this behavior by sending a separate frame for each destination. Alternatively, it could be realized using multicast.

The Ethernet frames will be sent via the standard Ethernet protocol (best effort) using 100 Mbit/s links. In the current simulation, the processing delay of a gateway is 40 µs. This value is based on measurements of an ARM-9-based prototype gateway.

The maximum payload of a CAN frame is 8 B. The minimal payload of an Ethernet frame is 46 B. Without aggregation of multiple CAN messages within one Ethernet frame, Ethernet bandwidth would be wasted.

#### 10.5.3.1 One Ethernet Switch Design Without Aggregation

In general, the utilized bandwidth on CAN buses does not change compared to the central Ethernet gateway design. The bandwidth used on the Ethernet links is always below 1 Mbit/s, but above the utilized bandwidth on CAN buses. This is due to the lack of aggregation of CAN messages. By padding and the larger size of the Ethernet frames compared to CAN frames, the necessary bandwidth increases on the Ethernet links. It should be noted that no multicast is used and therefore messages that are transmitted to multiple buses are also sent multiple times.

The maximum E2E latency is 25.289 ms. The minimum E2E is 126 ms (local on a CAN bus). The delay on the Ethernet network is always less than 10 % of the total E2E latency. The latency on the Ethernet from transmitter to receiver via a switch (8 µs hardware delay) is between 19.92 and 60.24 µs.

The queues in front of the output links of the switch store a maximum of two frames. The queues of the gateways store frames that are waiting to be processed by the CAN side of the gateway. These queues store a maximum of six Ethernet frames.

In summary, this simulation shows that the use of an Ethernet switch has little effect on the transmission of CAN messages. The latency and jitter metrics continue to be significantly influenced by CAN bus arbitration.

#### 10.5.3.2 One Ethernet Switch Design with Aggregation

This section investigates the aggregation of CAN messages into one Ethernet frame. It is based on the approach that incoming CAN messages will be buffered in the gateway, so that ultimately several CAN messages will be transmitted in one Ethernet frame to reduce the utilized bandwidth on the Ethernet link. On the other hand, the delay in a gateway increases the CAN message latencies.

The gateway implements the aggregation of CAN messages as follows: to each CAN message that arrives at a gateway a hold-up time is assigned. It defines how long a message can be buffered in the gateway until it is sent to the Ethernet link.

CAN messages will be aggregated in buffers called pools. A pool stores all messages that will be aggregated in the same Ethernet frame. As soon as the hold-up time of one CAN message expires, all buffered CAN messages of the corresponding pool will be passed to the transformation module of the gateway. For each target gateway the transformation module sends an Ethernet frame that contains the CAN messages stored in the pool. Figure 10.9 shows how such a pool works. At time $t_3$, the hold-up time of CAN message 2 expires. Therefore, all messages that have arrived so far will be forwarded at this time. The pool aggregation can be configured in two ways. One way is to use ANDL like in Listing 10.1 in Sect. 10.4.1. Another way is to manipulate the generated *.xml* file. Both ways are demonstrated in the SignalsAndGateways model examples folder.

Three different aggregation scenarios will be simulated.

1. Within this scenario hold-up times are based on the CAN-ID and period of the CAN message. Each hold-up time is calculated from the period and the percentage given in Table 10.6. All CAN messages are stored in the same pool. Hence, high-priority CAN messages are delayed less than low-priority messages.
2. This scenario differs from the first one as messages with high priority CAN messages (CAN-IDs < 101) get a hold-up time of 1 ms. This increases the

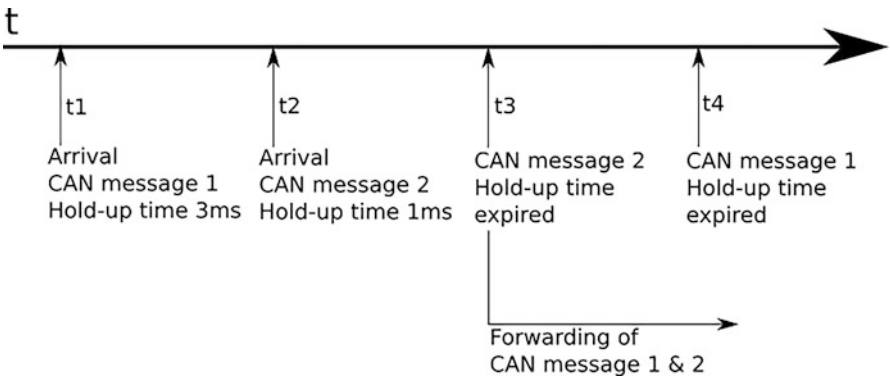



**Fig. 10.9** Aggregation of CAN messages with an pool

**Table 10.6** Initial pool configuration

| CAN-ID | Hold-up time for scenario 1 |
|---|---|
| < 101 | 0 ms |
| 101–200 | 25% of the period of CAN-ID |
| 201–300 | 50% of the period of CAN-ID |
| 300< | 75% of the period of CAN-ID |

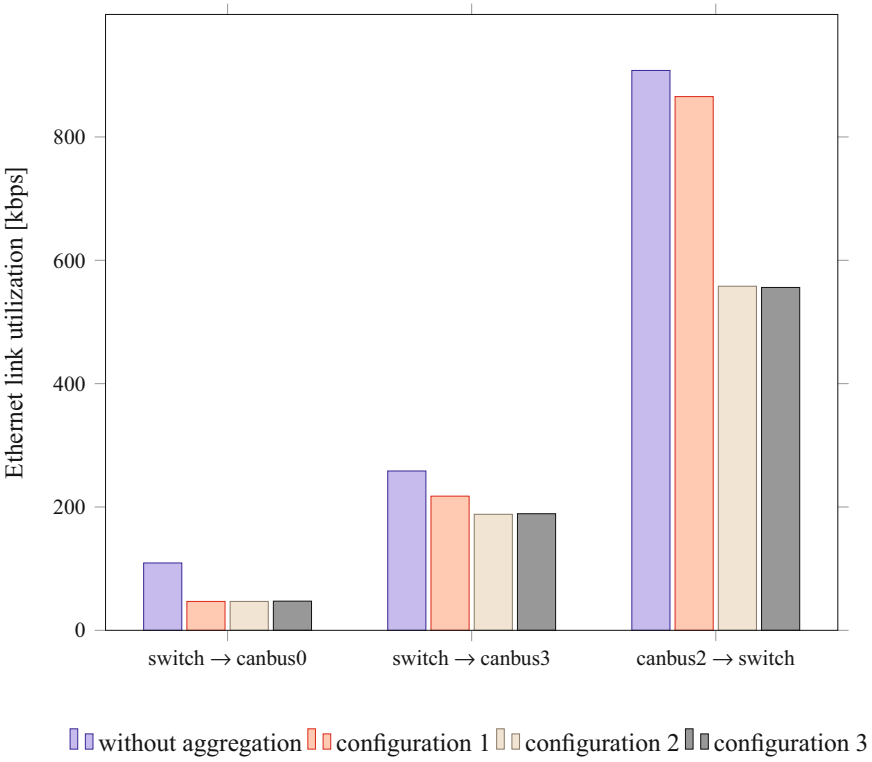



**Fig. 10.10** Utilized bandwidth on three Ethernet links

likelihood that high priority CAN messages are aggregated with other CAN messages.

3. This scenario uses two pools. CAN messages with similar hold-up times belong to the same pool.

Figure 10.10 represents the utilized bandwidth for three Ethernet links. As expected, it shows that aggregation reduces the Ethernet bandwidth utilization. Depending on the structure of the bus traffic, the utilization is reduced by more than 50 %. The difference between the three configurations is small. It is worth noting that no configuration provides the lowest utilized bandwidth for all links. In contrast to the Ethernet links aggregation does not change the utilized bandwidth on CAN buses.

**Table 10.7** Maximum end-to-end latency for some CAN-IDs on `canbus1`

| CAN-ID | Central CAN gateway [us] | Without aggregation [us] | With aggregation | | |
|---|---|---|---|---|---|
| | | | Configuration 1 [us] | Configuration 2 [us] | Configuration 3 [us] |
| 17 | 946.707 | 984.859 | 990.248 | 1984.378 | 1987.647 |
| 331 | 8465.906 | 8658.725 | 12,835.177 | 13,643.712 | 16,676.314 |
| 510 | 17,974.989 | 18,415.130 | 23,217.447 | 24,470.504 | 754,554.309 |

As expected, aggregation increases the E2E latency. Table 10.7 presents latencies of aggregated messages. Configuration 1 provides the best performance. For CAN-ID 17 the latency is similar to the one without aggregation. The cause is the hold-up time value of 0 ms for CAN-IDs < 101. For configuration 2, this latency increases by the configured hold-up time of 1 ms. In the third configuration, the latency of CAN-ID 510 increases significantly. This is due to the subdivision into two different pools depending on the hold-up time. The probability that the pool forwarding is triggered by a CAN message with a faster expired hold-up time is much lower. This can be verified by inspecting the `HoldUpTime` vector in the gateway buffer module.

Regarding jitter, CAN messages that are not transported via a gateway have little difference between the configuration with and without aggregation. This difference is based on bursts generated by gateways as follows. If an Ethernet frame containing several CAN messages arrives at a gateway, it fills the message object buffers of the CAN bus interface with CAN frames of different priority at the same time.

Figure 10.11 gives the min and max jitter for messages transported via gateways for three CAN buses. Aggregation leads to a significant higher max jitter compared to the configuration without aggregation. In particular, this concerns low prioritized ID messages, due to the bursts on the destination bus and their long hold-up time.

In terms of aggregation performance, the number of messages that are aggregated in an Ethernet frame is of special interest. Table 10.8 represents this metric. With up to 23 aggregated CAN messages, some pools are very large. However, far fewer CAN messages will be aggregated on average. Aggregation configurations hence provide potential for further optimization. A general statement about an optimal pool strategy cannot be given.

In summary, the results of this subsection show a trade-off between bandwidth and latency/jitter. The respective aggregation strategy has a massive impact on latency, jitter, and bandwidth. For some aggregation strategies, the gain in bandwidth is particularly efficient in relation to the effects on latency and jitter.

### *10.5.4 Real-Time Ethernet Backbone Design*

This scenario simulates the Ethernet backbone communication architecture of the real RECBAR prototype car [25] (see Fig. 10.12). In addition to the previous

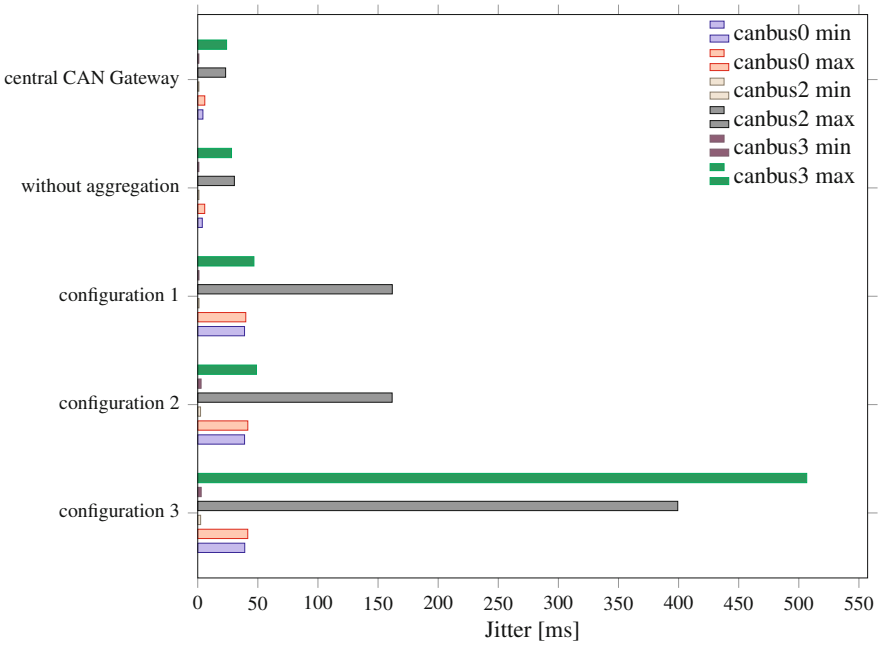



**Fig. 10.11** Minimal and maximal jitter on three Ethernet links

**Table 10.8** Number of CAN messages within a pool

| | Configuration 1 | | Configuration 2 | | Configuration 3 | |
|---|---|---|---|---|---|---|
| CAN bus | Max | Average | Max | Average | Max | Average |
| canbus0 | 5 | 1.44 | 5 | 1.44 | 3 | 1.41 |
| canbus1 | 23 | 5.03 | 23 | 5.20 | 12 | 4.59 |
| canbus2 | 14 | 1.23 | 16 | 2.81 | 16 | 2.80 |
| canbus3 | 13 | 9.09 | 13 | 9.08 | 9 | 6.81 |
| canbus4 | 17 | 2.49 | 19 | 5.15 | 19 | 5.16 |
| canbus5 | 18 | 1.19 | 21 | 2.70 | 17 | 2.69 |
| canbus6 | Gateway does not send | | | | | |
| canbus7 | Gateway does not send | | | | | |
| canbus9 | Gateway does not send | | | | | |

communication matrix, additional network participants have been added. These components are a front- and a rear-camera, two lidars in the front of the car, a logging ECU, and an ECU for sensor fusion based on raw data. `ECU1` periodically sends synchronization frames for the global time synchronization. The backbone contains three Ethernet switches. All links have a bandwidth of 100 Mbit/s.

We use the flexibility gained by the ANDL domain-specific language to add these additional devices to the network configuration. Just the lines shown in


**Fig. 10.12** Ethernet backbone within the RECBAR car

Listing 10.3 are added to the existing ANDL model to extend the topology. In addition, new message definitions must be added or extended with new receivers. The generation of this extended ANDL results in a new target topology and new message definitions.

**Listing 10.3** ANDL code example with comments

```
1  ...
2  connections {
3     segment backbone {
4        lid1 <--> {new std.ETH100M} <--> switch0;
5        lid2 <--> {new std.ETH100M} <--> switch0;
6        cam1 <--> {new std.ETH100M} <--> switch0;
7        switch0 <--> {new std.ETH100M} <--> switch1;
8        ecu1 <--> {new std.ETH100M} <--> switch1;
9        gateway0 <--> {new std.ETH100M} <--> switch1;
10       gateway1 <--> {new std.ETH100M} <--> switch1;
11       gateway2 <--> {new std.ETH100M} <--> switch1;
12       gateway3 <--> {new std.ETH100M} <--> switch1;
13       gateway4 <--> {new std.ETH100M} <--> switch1;
```

```
14        gateway5 <--> {new std.ETH100M} <--> switch1;
15        gateway6 <--> {new std.ETH100M} <--> switch1;
16        gateway7 <--> {new std.ETH100M} <--> switch1;
17        gateway9 <--> {new std.ETH100M} <--> switch1;
18        gateway10 <--> {new std.ETH100M} <--> switch1;
19        switch1 <--> {new std.ETH100M} <--> switch2;
20        cam2 <--> {new std.ETH100M} <--> switch2;
21        log <--> {new std.ETH100M} <--> switch2;
22        fusi <--> {new std.ETH100M} <--> switch2;
23     }
24     segment canbus {
25        ecu3b0 <--> canbus0;
26        ...
27     }
28  }
29  ...
```

**Table 10.9** End-to-end latency for some CAN-IDs on `canbus1` of the RECBAR car

| CAN-ID | Maximum latency [µs] | Average latency [µs] |
|---|---|---|
| 17 | 532.264 | 331.641 |
| 331 | 4191.718 | 380.729 |
| 510 | 8625.544 | 595.964 |

The CAN gateways are connected to the central switch. In addition to the previous CAN communication, the CAN messages are also sent to a logging ECU that is connected to `switch2`. Due to low camera resolution and compressed data transmission each camera stream requires only 7 Mbit/s bandwidth. We use 2D four layer laser scanners (lidar). A single lidar requires 2.5 Mbit/s bandwidth. These streams are sent to the logging unit and a fusion calculator.

The communication is based on the rate-constrained traffic class (AFDX). This traffic class uses multicast. The biggest link load occurs between `switch2` and the logging unit (about 20.2 Mbit/s).

In contrast to the previous simulations, rate constrained traffic will now be used. Therefore, the utilized bandwidth decreases due to multicast forwarding. Again, this can be investigated through the `QueueSize` vectors. When a frame is forwarded to multiple destinations just one frame is queued inestead of one per target.

Comparing the numbers of Table 10.7 and Table 10.9 shows that this configuration reduces the E2E latency of CAN messages. This is due to the use of multicast. At the sending gateway no more frames, which different unicast destination addresses and the same payload, appear at the same time.

In summary, the simulation results of this scenario show that latency and jitter are more affected by CAN bus arbitration than by the Ethernet backbone architecture. It can be seen that multicast addressing saves the bandwidth of corresponding Ethernet links and reduces the latency, too.

## 10.6 Conclusion and Outlook

The automotive industry is currently re-thinking the communication technologies in cars, thereby developing a strong preference towards real-time Ethernet. The design and evaluation of future Ethernet-centric architectures, but more delicately the transition from current legacy buses and gateways to a distributed switching layer, requires extensive experimentation and careful evaluation of every design step. This work largely profits from available simulation tools and platforms that allow for a rapid assessment of design choices with high accuracy.

We presented an environment for modeling and simulating of future in-car networks based on the OMNeT++ simulator and the INET Framework. This suite includes simulation models for various real-time extensions of Ethernet including AVB and TSN, field-buses including CAN and FlexRay, gateways, as well as tools for modeling vehicular networks. This rich environment enables researchers from academia and developers from industry to thoroughly investigate network concepts and designs that are composed of the current and the emerging link-layer technologies.

As a special support for engineers during their design process, the domain-specific language ANDL has been defined. It allows to describe design variant on an abstract level and supports a fast exploration of different design variants. We illustrated its utility with a case study based on realistic automotive data and demonstrated the practicability of this approach.

In future work, we will proceed in three directions: (1) Within the automotive industry, detailed support of layer 3 and 4 protocols (like IEEE 1722, Transmission Control Protocol (TCP), and User Datagram Protocol (UDP)) on top of the QoS-enhanced link layer is of growing interest. Therefore, an interface has to be implemented into our models, so that existing models of higher layers are easily adaptable. (2) The TSN working group investigates frame preemption, and we plan to integrate different variants and investigate them with our models. (3) Current applications allow stimuli generation based on random numbers. Domain-specific reactive behavior could make the stimuli generation even more realistic. Corresponding concepts, adapted to real-time protocols, need to be introduced.

## References

1. Aeronautical Radio Incorporated: Aircraft Data Network. Standard 664. ARINC, Annapolis (2002)
2. Bettini, L.: Implementing Domain Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing, Birmingham (2016)
3. Buschmann, S., Steinbach, T., Korf, F., Schmidt, T.C.: Simulation based timing analysis of FlexRay communication at system level. In: Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques, pp. 285–290. ACM-DL, New York (2013)
4. CoRE Working Group: CoRE Simulation Models for Real-Time Networks. https://sim.core-rg.de/trac

5. Eclipse Foundation Inc.: Xtext. https://www.eclipse.org/Xtext/index.html
6. El Salloum, C., Bilic, K.: FlexRay, chap. 6, pp. 121–152. CRC Press, Boca Raton (2012)
7. Hillebrand, J., Rahmani, M., Bogenberger, R., Steinbach, E.: Coexistence of time-triggered and event-triggered traffic in switched full-duplex Ethernet networks. In: International Symposium on Industrial Embedded Systems, 2007. SIES '07, pp. 217–224 (2007). https://doi.org/10.1109/SIES.2007.4297338
8. Institute of Electrical and Electronics Engineers: 802.1Qav - forwarding and queuing enhancements for time-sensitive streams. IEEE Standard for Information Technology. IEEE, New York (2009)
9. Institute of Electrical and Electronics Engineers: IEEE 802.1Qat - IEEE standard for local and metropolitan area networks - virtual bridged local area networks - amendment 14: Stream Reservation Protocol (SRP). Standard IEEE 802.1Qat-2010. IEEE, Piscataway (2010)
10. Institute of Electrical and Electronics Engineers: IEEE 802.1BA - IEEE standard for local and metropolitan area networks - Audio Video Bridging (AVB) Systems. Standard IEEE 802.1BA-2011. IEEE, Piscataway (2011)
11. Institute of Electrical and Electronics Engineers: 802.1Qbv - bridges and bridged networks - amendment: enhancements for scheduled traffic. Draft Standard P802.1Qbv/D1.0. IEEE, Piscataway (2013)
12. Institute of Electrical and Electronics Engineers: Standard for Ethernet amendment 1: Physical layer specifications and management parameters for 100 Mb/s operation over a single balanced twisted pair cable (100BASE-T1). Standard IEEE Std 802.3bw-2015. IEEE, Piscataway (2015)
13. Institute of Electrical and Electronics Engineers: Standard for Ethernet amendment 4: physical layer specifications and management parameters for 1 Gb/s operation over a single twisted-pair copper cable. Standard IEEE Std 802.3bp-2016. IEEE, Piscataway (2016)
14. Kamieth, J., Steinbach, T., Korf, F., Schmidt, T.C.: Design of TDMA-based in-car networks: applying multiprocessor scheduling strategies on time-triggered switched Ethernet communication. In: 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014), pp. 1–9. IEEE Press, Piscataway (2014). https://doi.org/10.1109/ETFA.2014.7005119
15. Matheus, K., Königseder, T.: Automotive Ethernet. Cambridge University Press, Cambridge (2015)
16. Meyer, P., Steinbach, T., Korf, F., Schmidt, T.C.: Extending IEEE 802.1 AVB with time-triggered scheduling: a simulation study of the coexistence of synchronous and asynchronous traffic. In: 2013 IEEE Vehicular Networking Conference (VNC), pp. 47–54. IEEE Press, Piscataway (2013). https://doi.org/10.1109/VNC.2013.6737589
17. Müller, K., Steinbach, T., Korf, F., Schmidt, T.C.: A real-time Ethernet prototype platform for automotive applications. In: 1st IEEE International Conference on Consumer Electronics - Berlin (ICCE-Berlin 2011), pp. 221–225. IEEE Press, Piscataway (2011)
18. Reimann, C., Filzmoser, P., Garrett, R., Dutter, R.: Statistical Data Analysis Explained - Applied Environmental Statistics with R. Wiley, New York (2008)
19. Robert Bosch GmbH: Bosch Automotive Electrics and Automotive Electronics - Systems and Components, Networking and Hybrid Drive. Springer, Berlin (2013)
20. SAE - AS-2D Time Triggered Systems and Architecture Committee: Time-Triggered Ethernet (AS 6802) (2009). http://www.sae.org
21. Society of Automotive Engineers - AS-2D Time Triggered Systems and Architecture Committee: Time-Triggered Ethernet AS6802. SAE Aerospace (2011). http://standards.sae.org/as6802/
22. Steinbach, T., Korf, F., Schmidt, T.C.: Comparing time-triggered Ethernet with FlexRay: an evaluation of competing approaches to real-time for in-vehicle networks. In: 8th IEEE International Workshop on Factory Communication Systems (WFCS 2010), pp. 199–202. IEEE Press, Piscataway (2010)
23. Steinbach, T., Dieumo Kenfack, H., Korf, F., Schmidt, T.C.: An extension of the OMNeT++ INET framework for simulating real-time Ethernet with high accuracy. In: SIMUTools 2011 – 4th International OMNeT++ Workshop, pp. 375–382. ACM, New York (2011)

24. Steinbach, T., Lim, H.T., Korf, F., Schmidt, T.C., Herrscher, D., Wolisz, A.: Tomorrow's in-car interconnect? A competitive evaluation of IEEE 802.1 AVB and time-triggered Ethernet (AS6802). In: 76th IEEE Vehicular Technology Conference: VTC2012-Fall, pp. 1–5. IEEE Press, Piscataway (2012)
25. Steinbach, T., Müller, K., Korf, F., Röllig, R.: Real-time Ethernet in-car backbones: first insights into an automotive prototype. In: 2014 IEEE Vehicular Networking Conference (VNC), pp. 137–138. IEEE Press, Piscataway (2014). https://doi.org/10.1109/VNC.2014.7013331
26. Steinbach, T., Lim, H.T., Korf, F., Schmidt, T.C., Herrscher, D., Wolisz, A.: Beware of the hidden! How cross-traffic affects quality assurances of competing real-time ethernet standards for in-car communication. In: 2015 IEEE Conference on Local Computer Networks (LCN), pp. 1–9. IEEE Press, Piscataway (2015)
27. Steinbach, T.: Ethernet-basierte Fahrzeugnetzwerkarchitekturen für zukünftige Echtzeitsysteme im Automobil, Springer Vieweg, Wiesbaden (2018). doi:10.1007/978-3-658-23500-0
28. Wolfhard, L. (ed.): CAN System Engineering From Theory to Practical Applications, 2nd edn. Springer, London (2013)

# Chapter 11
# LIMoSim: A Framework for Lightweight Simulation of Vehicular Mobility in Intelligent Transportation Systems


**Benjamin Sliwa and Christian Wietfeld**

## 11.1 Introduction

Upcoming smart-city-based Intelligent Transportation Systems (ITS) will be dominated by the convergence of mobility and communication with challenges arising on both sides [20]. On the one hand, the deployment of safe autonomous driving requires coordination among the traffic participants by means of Ultra-Reliable Low-Latency Communication (URLLC). On the other hand, novel communication systems need to operate in highly dynamic environments where different technologies coexist and compete for the available radio resources [4].

Therefore, instead of treating the two worlds separately, upcoming communication systems will have to become *mobility-aware* in order to integrate the system-immanent dynamics of the network topology into the decision processes. Figure 11.1 illustrates example use-cases of *anticipatory mobile networking* [2] that exploit mobility knowledge for determining routing paths, performing handover and resource reservation as well as predictive steering of pencil beams in millimeter Wave (mmWave) systems [6]. In experimental evaluations [16, 17], we have integrated channel information as well as mobility knowledge in a machine-learning-based data rate prediction process that is used to schedule the transmission times of sensor data transmissions with respect to the achievable throughput.

Similarly, in *communication-aware* mobility applications, vehicles adjust their path planning according to the available connectivity. While this type of behavior has already received great attention in the context of Unmanned Aerial Vehicle (UAV) networks, it will also be of relevance for cars acting as mobile sensor nodes for upcoming crowd sensing-based applications like distributed weather forecast [3].

B. Sliwa (✉) · C. Wietfeld
Communication Networks Institute, TU Dortmund University, Dortmund, Germany
e-mail: benjamin.sliwa@tu-dortmund.de; christian.wietfeld@tu-dortmund.de



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_11

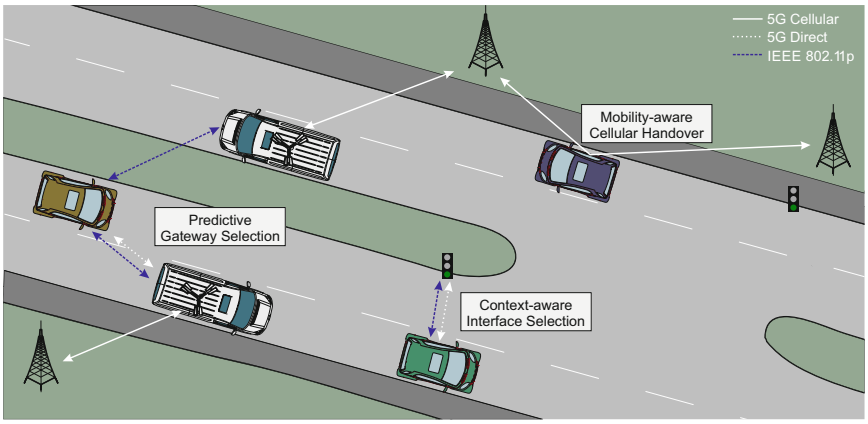



**Fig. 11.1** Example scenarios for anticipatory mobile networking. ©[2018] IEEE. Reprinted, with permission, from [14]

In order to address these arising challenges, researchers and developers require tools that are able to simulate both worlds in a unified manner. Furthermore, easy access to data of both components is of tremendous importance in order to provide the required information for developing mobility-aware communication models for next-generation mobile networks.

While full-featured traffic simulators like Simulation of Urban MObility (SUMO) [9] provide a multitude of highly accurate mobility models and evaluation scenarios, they have a clear focus on the demands of the traffic physicists community and are rather intended for static large-scale analysis and optimization of traffic flow problems. Although approaches for joint simulation of vehicular mobility and communication exist, they rely on Inter-Process Communication (IPC)-based coupling where multiple specialized simulators running in separated processes are synchronized using a dedicated communication protocol that involves additional overhead.

Contrastingly, Lightweight Information and Communications Technology-centric Mobility Simulation (LIMoSim) was developed to bring both worlds together in a more natural way with a clear communications-oriented perspective on ITS. In this context, the term *lightweight* refers to a reduced complexity about the variety of the different models and parameters for vehicular motion, on the one hand, and an integrated single-process simulation setup, on the other hand. The main intent of the framework is to bring together vehicular motion and *anticipatory* communication systems using a shared codebase approach. In addition, the mobility behavior is simulated without any dependencies to the actually used communication technology and can therefore be coupled with WiFi-based systems as well as Long Term Evolution (LTE), IEEE 802.15.4, and Long Range Wide Area Network (LoRaWAN) using the available OMNeT++ extension frameworks such as INET, SimuLTE, or Framework for Long Range (FLoRa) [12].

From an application point of view, LIMoSim can be considered as a moving platform that can be equipped with any desired communication technology. In contrast to previous work, where we presented the architecture of LIMoSim and evaluated its simulation performance [14] and described the embedding of the framework into OMNeT++ [15], this chapter focuses on giving an overview about LIMoSim and its possible applications using different case-studies with a tutorial-like approach.

The remainder of this chapter is structured as follows. After giving an overview about relevant state-of-the-art traffic simulators and principles, we present the architecture of the proposed LIMoSim and provide details about its individual components such as the mobility implementation, the map representation, and the integration into OMNeT++. Afterwards, a reference scenario for the evaluation is presented and different short tutorials are provided that address specialized aspects like accessing mobility information and using mobility prediction.

## 11.2 Related Work

The simulation of vehicular traffic with the main goals of traffic forecast and traffic flow optimization has been a research topic for a long time. A variety of simulators have been proposed such as SUMO, Multi-Agent Transport Simulation Toolkit (MATSim-T) [1], and PTV Vissim [5] that differ in the provided level of detail, multi-modality, variety of models, and type of license. All of them model vehicular mobility without considering communication technologies at all—which has become an essential component of modern cars. However, some of the established frameworks have recognized the need for Information and Communications Technology (ICT)-enabled vehicular traffic simulation and offer interfaces for run-time data exchange with and external control through third-party tools. For the communication networks community, the traffic simulator SUMO is the most popular framework for simulating vehicular mobility as it can be coupled with most network simulators through its Transmission Control Protocol (TCP)-based Traffic Control Interface (TraCI).

Using IPC for coupling multiple specialized simulators as illustrated in Fig. 11.2 is a standard approach to bring highly specialized tools together. It is also widely used in channel modeling. While this *multi-scalar* method guarantees a high level of accuracy through usage of verified models that have been created by their respective communities, it has a number of disadvantages that should be considered.

- Since multiple processes are executed in parallel, simulation setups become quite complicated, especially if multiple simulation servers are used in parallel. Moreover, often additional scripts need to be running in the background in order to manage the life cycles of the processes.
- The need for runtime synchronization between the different processes reduces the simulation performance and its applicability for large-scale evaluation [8].

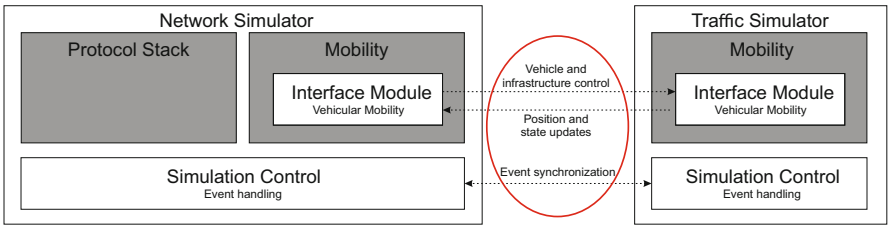



**Fig. 11.2** IPC-based coupling using a dedicated coupling protocol. ©[2018] IEEE. Reprinted, with permission, from [14]

- Since the different frameworks are developed independently, their compatibility requires constant attention with each published update on both sides.
- Data cannot be accessed on a codebase-level and needs to be requested through the coupling interface. Moreover, the exchange is bound to the specification of the coupling protocol and requires modifications on both sides if it is not contained in the specification.

In conclusion, IPC-based coupling is a powerful mechanism for bringing together different worlds, however it blows up the evaluation setup complexity and does not provide methods for data exchange in a native (code-based) way. As always, there is no tool that perfectly satisfies all requirements. Instead, the choice of the simulation framework is highly depending on the requirements of the application scenario.

Contrastingly, LIMoSim is intended to provide system-level vehicular mobility for medium-scale scenarios. Unlike other approaches that aim to provide a full model of a connected car with a specific communication technology, it purely focuses on providing the *mobile platform* that can be used in combination with any communication technology using other extension frameworks.

## 11.3 Framework Architecture

This section explains the general architecture of LIMoSim as well as the hierarchical mobility model for the simulation of the different components of vehicular mobility. Likewise to OMNeT++, the simulation is performed in an event-based way and the mobility is updated periodically as well as by external events. LIMoSim consists of two main modules that can be executed separately from each other, depending on the application requirements: the purely C++-based simulation kernel and the LIMoSim User Interface (UI) written in Qt-C++ and OpenGL. Figure 11.3 shows the general architecture of the simulation framework.

Using the *standalone* mode, the simulation kernel is controlled from the LIMoSim UI and makes use of a dedicated event scheduler. This mode is intended to be used for fast evaluations of mobility-only algorithms without the overhead

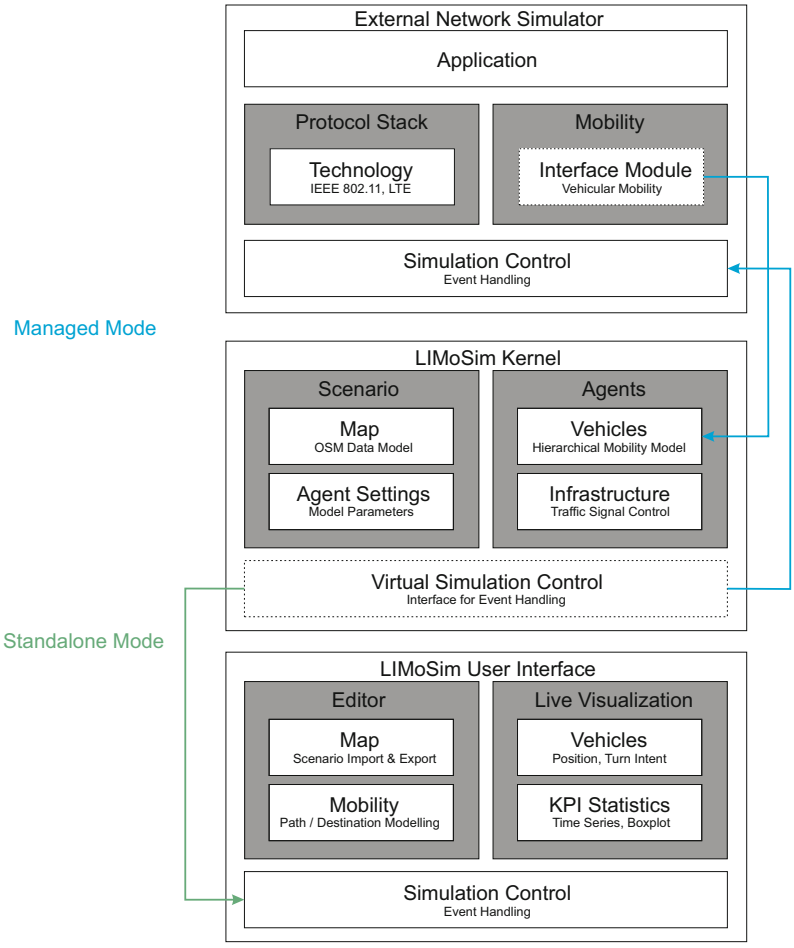



**Fig. 11.3** Architecture of the proposed simulation framework LIMoSim consisting of the two main modules simulation kernel and UI. ©[2018] IEEE. Reprinted, with permission, from [14]

for the communication stack. In contrast to that, the *managed* mode does not have any dependencies to the UI components and works without using an own event scheduler. Instead, the source code of the simulation kernel is embedded into the workspace of the INET Framework of OMNeT++ and the event handling is performed directly by the OMNeT++ event queue (see Sect. 11.3.3). This approach enables native interactions between the different OMNeT++ frameworks and the mobility information provided by LIMoSim in a shared codebase manner (see Sect. 11.4.2 for an example how this kind of coupling can be exploited for data exchange). Moreover, the coupling is performed transparently for all other extension frameworks without adding additional dependencies to LIMoSim. For

integrating LIMoSim into simulation setups, the respective hosts simply assign the *LIMoSimCar.ned* module as their mobility submodule in the *omnetpp.ini* configuration file.

### *11.3.1 Agent-Based Mobility Modeling*

In contrast to *macroscopic* mobility modeling, which describes the behavior of vehicle groups, LIMoSim follows a *microscopic* approach and models the behavior of individual entities in order to be compliant with the agent-based simulation approach of OMNeT++. Since vehicular motion is a complex process with the goal to fulfill multiple targets in parallel—reaching a destination, respecting the traffic rules, and keeping a safety distance to the leading car—the overall mobility behavior for each vehicle is modeled by a hierarchical approach as shown in Fig. 11.4 which is represented by the *LIMoSimCar.ned* module that is derived from *MovingMobilityBase.ned*. It is important to note that no changes are made to the actual host module in order to avoid interference with further extension frameworks like SimuLTE (see Chap. 5) that provide their own logical host module.

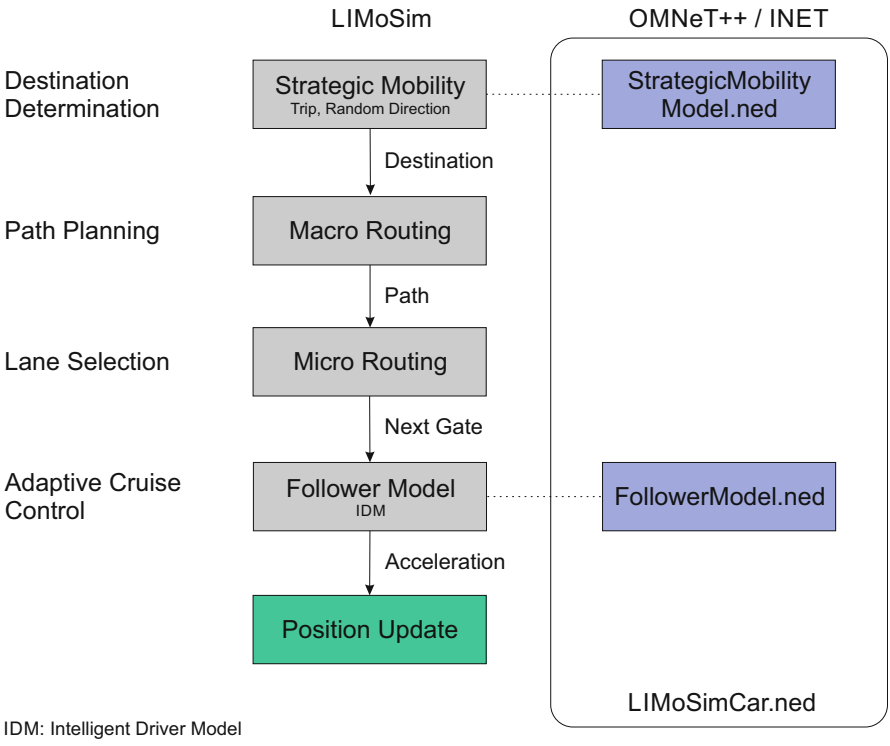



**Fig. 11.4** Hierarchical mobility model. Source: Adapted from [15]

The hierarchical mobility model consists of multiple layers similar to the *strategic*, *tactical*, and *operational* levels proposed by Hoogendoorn [7]. On the top layer, the *strategic* aspect of the mobility behavior is modeled, representing the determination of the destination and the motivation for the motion itself. This can be done with a purely random model or deterministically by defining a *trip* containing multiple destinations to provide a closer model for actual human behavior like driving to different grocery stores and back home afterwards.

Once the destination is determined, the routing path is computed using *Dijkstra's algorithm* and represented as a list of waypoints that the vehicle approaches sequentially. This part represents the macro component of the routing processes that is performed on the level of gates (see Sect. 11.3.2) with respect to the road directionality. The micro routing part then handles how the actual gates can be reached from a lane-level perspective. The whole routing process is repeated if the vehicle deviates from the defined route, e.g., because of traffic obstructions. For the on-lane behavior, the Intelligent Driver Model (IDM) is used to model the adaptive cruise control with respect to the other traffic participants. The model has been proven to provide a better representation of real-world driver behavior than other state-of-the-art models in [11]. Furthermore, the Minimizing Overall Braking Induced by Lane change (MOBIL) [19] decides about the vehicle's lane change decisions.

Figure 11.5 shows an example usage of the IDM and illustrates the concepts of *free flow* and *following* behavior. In each simulation step, the acceleration $a_{\text{IDM}}$ is determined with Eqs. (11.1) and (11.2) using the current distance $s$, the velocity $v$, and the velocity difference to the leader car $\Delta v$. The goal of the model is to approach the desired speed $v_0$ that is derived from the allowed road speed and the

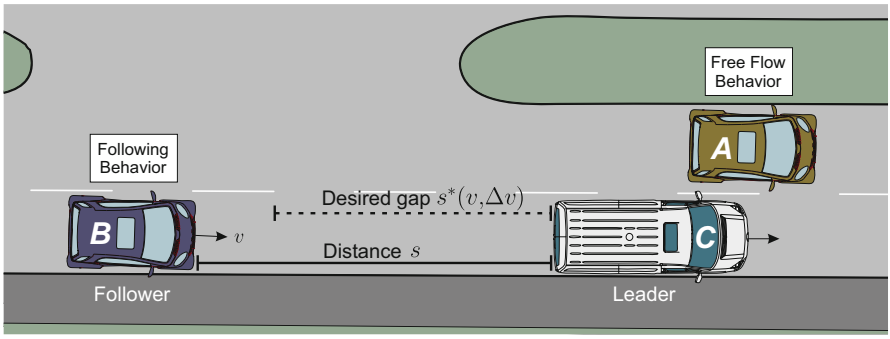



**Fig. 11.5** Example usage of the IDM. Car *A* is not influenced by other vehicles on the same lane and determines its acceleration using the free flow part of the model. Contrastingly, car *B* adjusts its acceleration to reach the desired gap to the leader car

driver's behavior. The exponent $\delta$ describes how fast acceleration decreases with respect to the vehicle's speed.

$$a_{\text{IDM}}(s, v, \Delta v) = a \left[ \underbrace{1 - \left(\frac{v}{v_0}\right)^{\delta}}_{\text{Free flow behavior}} - \underbrace{\left(\frac{s^*(v, \Delta v)}{s}\right)^2}_{\text{Following behavior}} \right] \tag{11.1}$$

Different vehicle types are described by their respective system parameters: maximum acceleration $a$, comfortable deceleration $b$, jam distance $s_0$, and desired time gap $T$. The *desired gap* $s^*$ (see Eq. (11.2)) represents the safety distance for the current speed difference between the two vehicles and is derived from the minimum vehicle distance, the speed dependent distance term, and the intelligent braking strategy.

$$\underbrace{s^*(v, \Delta v)}_{\text{Desired distance}} = \underbrace{s_0 + vT}_{\text{Safety distance}} + \underbrace{\frac{v\Delta v}{2\sqrt{ab}}}_{\text{Braking strategy}} \tag{11.2}$$

Within LIMoSim, IDM is also applied for modeling how a vehicle approaches intersections. Traffic signals are treated as static vehicles as long as their state is "red" or "yellow." When the updated acceleration is available, the velocity of the car is updated and, finally, a new position is computed based on the current position, the direction, and the new velocity.

From the perspective of a communication simulator, not all entities having an influence on the vehicular traffic need to be modeled with a surrogate. Instead, non-communicating entities like regular traffic signals and interfering traffic vehicles are only modeled within the simulation kernel of LIMoSim and are hidden to OMNeT++ in order to reduce the complexity of the simulation setup.

### *11.3.2 Represent Map Data with the OpenStreetMap Data Model*

LIMoSim has a native integration for OpenStreetMap (OSM) data in order to allow an easy setup. Unlike other mobility simulators, additional preprocessing steps performed by the user are not required but can be applied in order to reduce the file size by filtering out unnecessary information from the data which speeds up the initial map import process.

Although OSM data uses the World Geodetic System 1984 (WGS84), LIMoSim internally operates on Cartesian coordinates in order to avoid the need for run-time coordinate transformations for being compliant with the Cartesian model of

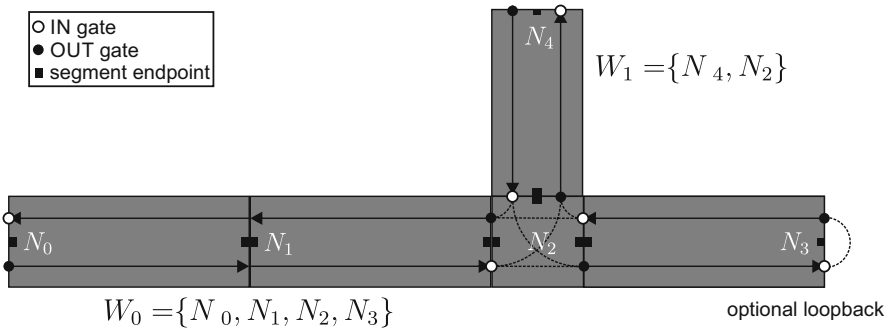



**Fig. 11.6** Example road excerpt using the OSM data model. Gates are owned by intersections and end nodes and specify the direction of the traffic flow

OMNeT++. Each time, an OSM file *map.osm* is loaded for the first time, an optimized representation of the map *map.osm.limo* is created that filters out unnecessary meta-data and contains the geographical data in their Cartesian representation. In addition to the real-world map data, LIMoSim supports generic maps that can be created with an integrated editor inside the UI or via direct code editing.

The OSM data model is not only utilized for importing map information, it is also used internally as a hierarchical structure that represents the different components that form a street network (see Fig. 11.6). *Nodes* form the most basic elements of the model and are used to represent meaningful locations within the map. They consist of at least a coordinate and an Identifier (ID) variable and can be extended with further, optional parameters. Lists of nodes are called *Ways* and are used to model more complex structures. Within OSM, they are not only used to represent street segments with the same properties (e.g., speed limitation and number of lanes) but also for marking the outlines of parking lots and buildings. Multiple ways that have common nodes in their node set form *Intersections* that enable transitions from one way to the other with respect to the available lanes and according to the traffic rules. Intersections do not need to be explicitly defined as they are computed automatically. They can be attached with traffic signals for controlling the directional traffic flow, otherwise the regular traffic rules are applied.

As Sect. 11.3.1 points out, the vehicle routing macro level is performed using *gates* as reference points that derive the overall routing graph. Gates refer to a segment endpoint and are either placed on unconnected segments or are part of an intersection. Depending on the traffic direction, they are defined as either *input* or *output* and can be connected to multiple other gates of the opposite type. Unconnected endpoints can optionally be configured in *loopback* mode so that vehicles are automatically inserted on the other side of the road after passing the endpoint.

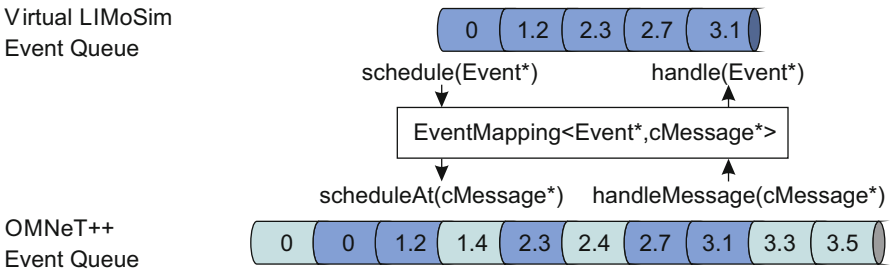



**Fig. 11.7** Synchronization of OMNeT++ and LIMoSim event queues. Source: Adapted from [15]

### *11.3.3 Embedding LIMoSim into OMNeT++*

In order to facilitate the separation of the *managed* mode and the *standalone* mode (see Sect. 11.3), the simulation kernel itself does not have any OMNeT++ dependencies. Consequently, the OMNeT++ event handling mechanisms cannot be directly used as the C++ classes cannot be derived from `cModule`. Instead, the event handling modules within LIMoSim are derived from the `LIMoSim::EventHandler` class that implements the basic methods for the event-based behavior.

For the standalone mode, LIMoSim uses its own event handling mechanism with a dedicated event scheduler. Since the latter cannot be applied in managed mode, the LIMoSim objects use a virtual event queue that maps the LIMoSim events to OMNeT++ messages, as it is illustrated in Fig. 11.7.

Whenever a new event is created in the LIMoSim domain in managed mode, a new `cMessage` is created in the OMNeT++ domain by the `EventScheduler` interface and a mapping entry between these objects is stored. Upon calling the `handleMessage()` method, the actual LIMoSim event is retrieved from the mapping and its handler method is called within the owner object.

Since the `LIMoSim::EventHandler` singleton is the only module that is actually using the `cMessage`-based event handling mechanism of OMNeT++, non-communicating entities such as interference traffic or traffic lights do not need to be explicitly modeled with an OMNeT++-representative to make use of event-based behavior.

## 11.4 Use Cases and Tutorials

In this section, example use cases focusing on specific aspects of the mobility simulator are discussed. We do not consider IEEE 802.11p here, as it is covered by the frameworks Vehicles in Network Simulation (Veins) (see Chap. 6) and Artery (see Chap. 12). Instead, we emphasize the mobility platform character of LIMoSim.

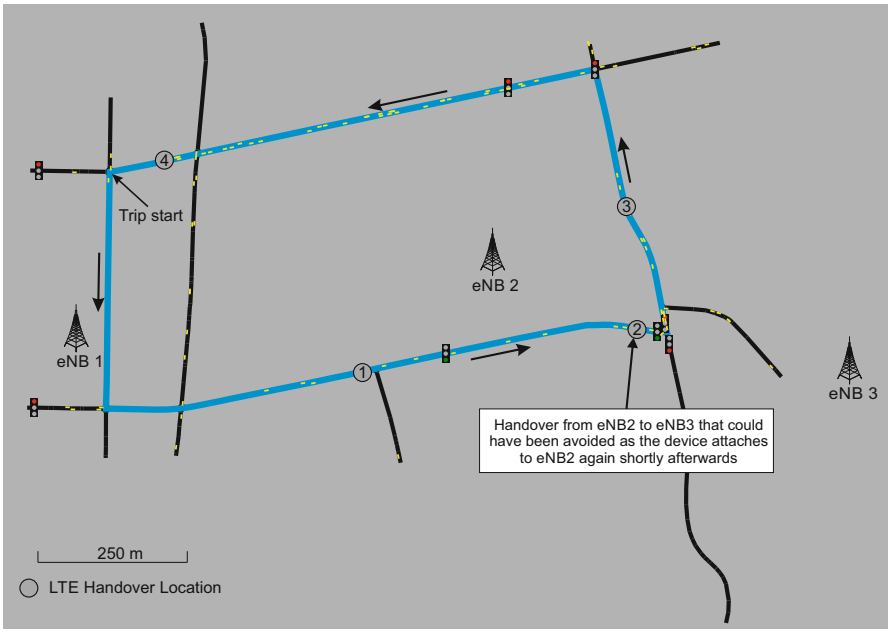



**Fig. 11.8** Map of the reference scenario illustrating the road network topology and the base station locations. As cellular handovers decrease the QoS of active connections, unnecessary handovers should be avoided leveraging navigation system knowledge and mobility prediction. Source: Adapted from [15]

### *11.4.1 Integration of Real-World Map Data and Initial Positioning*

Figure 11.8 shows the map of a reference scenario that contains multiple road intersections and traffic signals. It is located around the campus of the university of Dortmund in Germany. As LTE-applications will be discussed in the following, three evolved Node Bs (eNBs) are positioned with respect to their real-world locations based on information provided from the federal network agency.

The extension for the *omnetpp.ini* file is shown in Listing 11.1 while Listing 11.2 gives an excerpt from the converted map data file that is based on OSM.

**Listing 11.1** Initial positioning within the *omnetpp.ini* file

```
1 **.host.mobilityType = "LIMoSimCar"
2 **.host.mobility.map = "map.osm"
3 **.host[0].mobility.strategicModel = "Trip"
4 **.host[0].mobility.strategicModel.trip = "
      677230875,275672221,3569208993,477807"
5 **.host[0].mobility.way = "337055293"
6 **.host[0].mobility.segment = 4
7 **.host[0].mobility.lane = 0
8 **.host[0].mobility.offset = 1m
```

**Listing 11.2** Excerpt of the TU Dortmund street map data

```
<osm generator="LIMoSim">
   <node id="677231620" x="173.659" y="766.422"/>
   <node id="677231627" x="176.378" y="766.378"/>
   <node id="627846556" x="188.113" y="766.189"/>
   <node id="677231621" x="202.428" y="765.933"/>
   <node id="52919181" x="238.261" y="765.267"/>
   <node id="477807" x="254.202" y="767.967"/>
   <node id="3441521491" x="320.05" y="780.956"/>
   <way id="337055293">
      <nd ref="677231620"/>
      <nd ref="677231627"/>
      <nd ref="627846556"/>
      <nd ref="677231621"/>
      <nd ref="52919181"/>
      <nd ref="477807"/>
      <nd ref="3441521491"/>
      <tag k="highway" v="primary"/>
      <tag k="name" v="Emil-Figge-Straße"/>
      <tag k="maxspeed" v="50"/>
   </way>
</osm>
```

The scenario map itself is not represented as a dedicated OMNeT++ module in order to avoid the necessity for requiring a LIMoSim dependency inside the simulation network. Instead, the map creation is handled transparently by the mobility module. The first `LIMoSimCar` that gets instantiated creates the `LIMoSim::Simulation` singleton and loads the map according to the parameter specified in the *omnetpp.ini* file.

The initial positioning procedure is based on the corresponding values for the OSM data model. If no position parameters are defined or if they contain elements that cannot be mapped to the actual street network, the car is positioned randomly.

### *11.4.2 Accessing Mobility Data from the Communication Side*

As pointed out in Sect. 11.1, upcoming vehicular communication technologies will integrate mobility information into their decision processes and therefore require methods to access this kind of information. An example how LIMoSim provides its mobility data for the communication side is shown in Listing 11.3.

**Listing 11.3** Accessing mobility information from the communication side

```
cModule *host = getParentModule(); // depends on the layer of the module!
LIMoSimCar *mobility = dynamic_cast<LIMoSimCar*>
   (host->getSubmodule("mobility"));

if (mobility)
{
   LIMoSim::Car *car = mobility->getCar();
   double velocity_mps = car->getVelocity();
   double acceleration_mpss = car->getAcceleration();
}
```

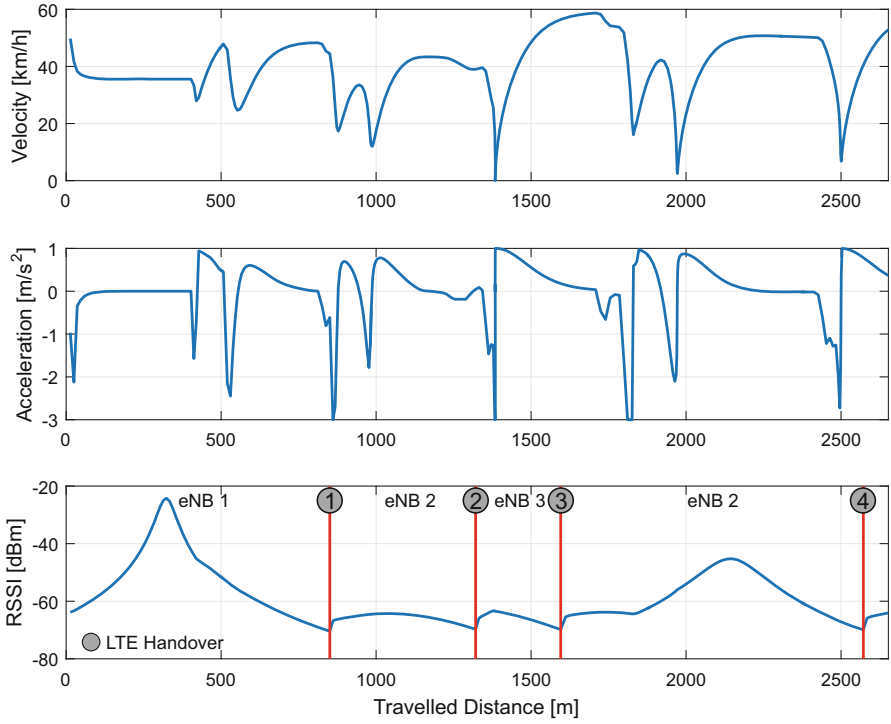



**Fig. 11.9** Example temporal behavior of velocity, acceleration, and measured RSSI. The corresponding handover locations are shown in Fig. 11.8. Source: Adapted from [15]

Since the `LIMoSim::Car` class is the hub for the whole hierarchical mobility model within the simulation kernel of LIMoSim, it can be further used to get pointers to the models that operate on the different layers to access or modify more specific parameters like the current acceleration with IDM. Figure 11.9 shows an example trace of the mobility behavior and measured signal strength for a single car. Additionally, performed handovers are marked and can be set into relation to their actual locations as shown in Fig. 11.8. It can be seen that the device is only attached to eNB 3 for a short duration and reattached to eNB 2 shortly after the handover. As handovers are cost-intense processes that decrease the QoS-capabilities of active communication links, unnecessary handovers should be proactively avoided.

In the considered vehicular scenario, information from the navigation system can be leveraged to move from *context-aware* to *context-predictive* communication and elaborate, if performing a handover is reasonable with respect to the predicted trajectory of the vehicle. Similarly, resource reservation and forwarder selection might also benefit from integrating knowledge about the anticipated future behavior of the mobility network participant. In [13], we leveraged application layer knowledge about the anticipated trajectory to increase the robustness of mesh routing paths in mobile robotic networks. LIMoSim provides two different approaches to access predicted information, both mirroring an equivalent sensor module in the real world.

The *extrapolation-based* scheme mirrors a simple prediction using the car's Global Positioning System (GPS) only. For a defined prediction horizon $\tau$, the future position $\mathbf{P}(t+\tau)$ is computed based on the current position $\mathbf{P}(t)$, the current velocity $v$, and the current angular direction $\lambda$ using Eq. (11.3). While this method can be implemented easily into real-world systems, it does not consider direction changes. Therefore, its prediction accuracy is severely impacted by the turns a vehicle performs on its route.

$$\mathbf{P}(t+\tau) = \mathbf{P}(t) + \begin{pmatrix} \sin\left(\frac{\Pi}{2}\right) \cdot \cos\left(\frac{\lambda \cdot \Pi}{180}\right) \\ \sin\left(\frac{\Pi}{2}\right) \cdot \sin\left(\frac{\lambda \cdot \Pi}{180}\right) \end{pmatrix} \cdot v \cdot \tau \tag{11.3}$$

In contrast to that, the *trajectory-based* scheme assumes access to the vehicle's navigation system and planned route. Within the prediction processes, the total movement potential is computed and the vehicle is virtually moved on the different street segments until the total movable distance is reached. Therefore, this scheme is able to consider the direction changes on the path. However, both schemes remain agnostic towards the acceleration dynamics of the vehicle which are affected by the interaction with other traffic participants and traffic signals.

It should be noted that trajectory information may also be leveraged even if no navigation system access is explicitly provided by exploiting the regularities of the human behavior. In [18], the authors show that human trajectories can be predicted with a high grade of accuracy as people usually use the same ways regularly while following their daily routines. A simulative comparison of the achieved accuracy with respect to the vehicle's velocity is shown in Fig. 11.10. For $\tau = 10\,\text{s}$, both schemes perform almost equally. With an increasing prediction horizon, the error dimension of the extrapolation-based approach raises dramatically and becomes even impractical at higher speeds.

These two approaches can be considered as a starting point for developing more advanced prediction methods, e.g., based on machine learning processes that consider knowledge about the anticipated traffic flow and the street capacity.

### *11.4.3 Collecting Statistical Information with LIMoSim*

In order to enable the collection of statistical information in the *standalone* mode, LIMoSim has its own statistics module that can also be used in the *managed* mode as an alternative to the scalar and vector files of OMNeT++. Similar to OMNeT++, the size of generated log files can be highly reduced by preselecting only relevant performance indicators. Moreover, since the Comma-Separated Values (CSV) format is used, the collected data can directly be processed by **Python**- or **MATLAB**-based postprocessing scripts. In many cases, the analysis of the collected data is not performed on all data points but with a time reference (for example once per second). Therefore, the size of the log files can be further reduced by means

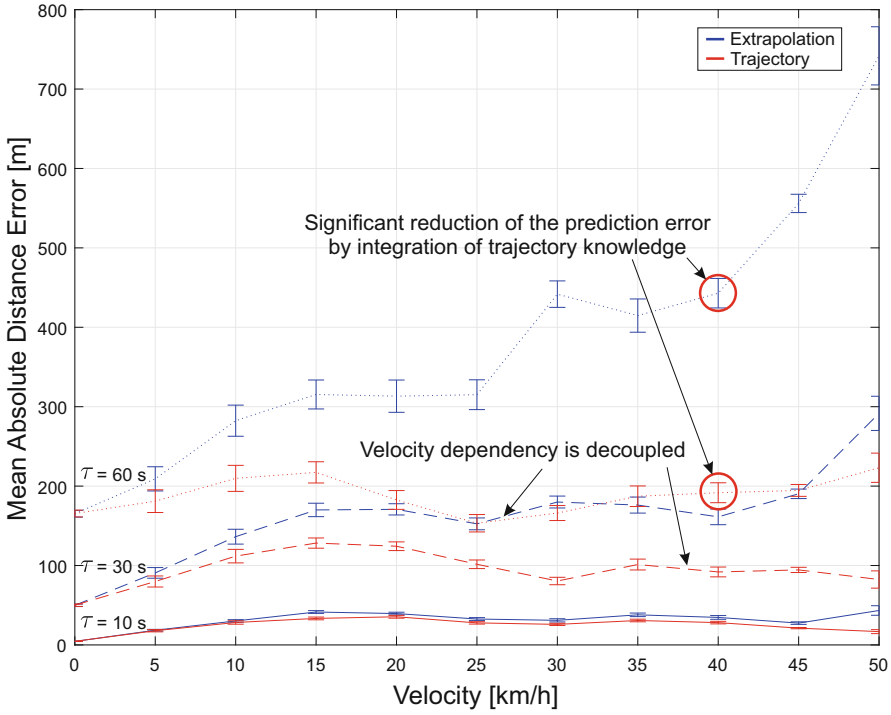



**Fig. 11.10** Accuracy evaluation of the mobility prediction schemes in the considered scenario with different prediction horizons. While there is no significant difference for short-time forecasts ($\tau = 10$ s), the trajectory-based scheme performs much better for higher prediction horizons. The curves show the 0.95 confidence interval of the mean value

of online data aggregation. LIMoSim offers multiple operators for this task that aggregate data with a defined time reference (e.g., averaging and min/max).

Listing 11.4 shows an example how the statistics handling can be integrated into simulations. During the initialization phase, the valid keys are registered according to the `statisticsKey` parameter of the mobility module that also specifies how the values will be arranged in the resulting CSV file.

**Listing 11.4** Statistics with LIMoSim

```
// omnetpp.ini
**.host.mobility.statisticsKey = "x,y,distance,speed,acceleration"

// update
Statistics *statistics = StatisticsManager::getInstance()->getStatistics(id);
statistics->add(StatisticsEntry("x", position.x, Stats::LAST));
statistics->add(StatisticsEntry("y", position.y, Stats::LAST));
statistics->add(StatisticsEntry("distance", distance_m, Stats::SUM));
statistics->add(StatisticsEntry("speed", currentSpeed_mps, Stats::MEAN));
statistics->add(StatisticsEntry("acceleration", currentAccerleration_mpss,
     Stats::MEAN));
```

The `getStatistics()` method takes an identifier for the agent as a parameter. If it is statically set to be equal for all agents, the statistic values are aggregated globally instead of agent-based. All values that are not contained in the key definition will be automatically discarded by the `StatisticsManager` module.

## 11.5 Conclusion and Further Research

In this chapter, we presented the vehicular-mobility simulation framework LIMoSim and demonstrated its application for simulating LTE-enabled ITS. The proposed mobility simulation framework focuses on selected well-known analytical models and provides support for real-world map data from OSM. For bringing together vehicular mobility and communication simulation, LIMoSim relies on a shared codebase coupling approach that allows lean information exchange between both sides in a native way without any IPC-related overhead and facilitates the execution of the whole simulation setup within a single system process. LIMoSim is intended to provide the moving vehicular platform for arbitrary communication technologies and can therefore be transparently coupled with third party extensions frameworks (e.g., SimuLTE for providing cellular connectivity) without requiring additional changes to the latter. In future work, we want to extend LIMoSim with multi-modal simulation models to enable the simulation of cooperative cars and UAVs in future ITS scenarios as described in [10].

**Acknowledgements** Part of the work on this chapter has been supported by the German Research Foundation (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis," project B4.

## References

1. Balmer, M., Rieser, M., Meister, K., Charypar, D., Lefebvre, N., Nagel, K.: MATSim-T: architecture and simulation times. In: Multi-Agent Systems for Traffic and Transportation Engineering, pp. 57–78. IGI Global, Hershey (2009)
2. Bui, N., Cesana, M., Hosseini, S.A., Liao, Q., Malanchini, I., Widmer, J.: A survey of anticipatory mobile networking: context-based classification, prediction methodologies, and optimization techniques. IEEE Commun. Surv. Tutorials **19**(3), 1790–1821 (2017)
3. Calafate, C.T., Cicenia, K., Alvear, O., Cano, J.C., Manzoni, P.: Estimating rainfall intensity by using vehicles as sensors. In: 2017 Wireless Days, pp. 21–26 (2017). https://doi.org/10.1109/WD.2017.7918109
4. Djahel, S., Doolan, R., Muntean, G.M., Murphy, J.: A communications-oriented perspective on traffic management systems for smart cities: challenges and innovative approaches. IEEE Commun. Surv. Tutorials **17**(1), 125–151 (2015). https://doi.org/10.1109/COMST.2014.2339817
5. Fellendorf, M., Vortisch, P.: Microscopic traffic flow simulator VISSIM. In: Fundamentals of Traffic Simulation, pp. 63–93. Springer, New York (2010)

6. Heimann, K., Tiemann, J., Boecker, S., Wietfeld, C.: On the potential of 5G mmWave pencil beam antennas for UAV communications: an experimental evaluation. In: 22nd International ITG Workshop on Smart Antennas (WSA 2018) (2018)
7. Hoogendoorn, S.P., Bovy, P.H., Daamen, W.: Microscopic pedestrian wayfinding and dynamics modelling. Pedestrian Evacuation Dyn. **123**, 154 (2002)
8. Hu, W., Feng, Z., Chen, Z., Harkes, J., Pillai, P., Satyanarayanan, M.: Live synthesis of vehicle-sourced data over 4G LTE. In: Proceedings of the 20th ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems, MSWiM '17, pp. 161–170. ACM, New York (2017). https://doi.org/10.1145/3127540.3127543
9. Krajzewicz, D., Erdmann, J., Behrisch, M., Bieker, L.: Recent development and applications of SUMO - simulation of urban mobility. Int. J. Adv. Syst. Meas. **5**(3&4), 128–138 (2012)
10. Menouar, H., Guvenc, I., Akkaya, K., Uluagac, A.S., Kadri, A., Tuncer, A.: UAV-enabled intelligent transportation systems for the smart city: applications and challenges. IEEE Commun. Mag. **55**(3), 22–28 (2017). https://doi.org/10.1109/MCOM.2017.1600238CM
11. Pourabdollah, M., Bjärkvik, E., Fürer, F., Lindenberg, B., Burgdorf, K.: Calibration and evaluation of car following models using real-world driving data. In: 2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC), pp. 1–6 (2017). https://doi.org/10.1109/ITSC.2017.8317836
12. Slabicki, M., Premsankar, G., Francesco, M.D.: Adaptive configuration of LoRa networks for dense IoT deployments. In: IEEE/IFIP Network Operations and Management Symposium (NOMS) (2018)
13. Sliwa, B., Behnke, D., Ide, C., Wietfeld, C.: B.A.T.Mobile: leveraging mobility control knowledge for efficient routing in mobile robotic networks. In: IEEE GLOBECOM 2016 Workshop on Wireless Networking, Control and Positioning of Unmanned Autonomous Vehicles (Wi-UAV) (2016)
14. Sliwa, B., Pillmann, J., Eckermann, F., Habel, L., Schreckenberg, M., Wietfeld, C.: Lightweight joint simulation of vehicular mobility and communication with LIMoSim. In: IEEE Vehicular Networking Conference (VNC) (2017)
15. Sliwa, B., Pillmann, J., Eckermann, F., Wietfeld, C.: LIMoSim: a lightweight and integrated approach for simulating vehicular mobility with OMNeT++. In: 4th OMNeT++ Community Summit (2017). Best Contribution Award
16. Sliwa, B., Liebig, T., Falkenberg, R., Pillmann, J., Wietfeld, C.: Efficient machine-type communication using multi-metric context-awareness for cars used as mobile sensors in upcoming 5G networks. In: 2018 IEEE 87th IEEE Vehicular Technology Conference (VTC-Spring) (2018). Best Student Paper Award
17. Sliwa, B., Liebig, T., Falkenberg, R., Pillmann, J., Wietfeld, C.: Machine learning based context-predictive car-to-cloud communication using multi-layer connectivity maps for upcoming 5G networks. In: 2018 IEEE 88th IEEE Vehicular Technology Conference (VTC-Fall), Chicago (2018)
18. Song, C., Qu, Z., Blumm, N., Barabási, A.L.: Limits of predictability in human mobility. Science **327**(5968), 1018–1021 (2010). https://doi.org/10.1126/science.1177170
19. Treiber, M., Kesting, A.: Traffic flow dynamics. Traffic Flow Dynamics: Data, Models and Simulation (2013). https://doi.org/10.1007/978-3-642-32460-4
20. Zanella, A., Bui, N., Castellani, A., Vangelista, L., Zorzi, M.: Internet of things for smart cities. IEEE Internet Things J. **1**(1), 22–32 (2014). https://doi.org/10.1109/JIOT.2014.2306328

# Chapter 12
# Artery: Large Scale Simulation Environment for ITS Applications


**Raphael Riebl, Christina Obermaier, and Hendrik-Jörn Günther**

## 12.1 Introduction

The development of Artery has been initiated by the need for a simulation environment supporting vehicular communication complying to European specifications. Back at that time, Vehicles in Network Simulation (Veins) has been already the de facto standard for simulating Vehicular Ad Hoc Network (VANET) communication with OMNeT++. However, Veins is focused on IEEE Wireless Access in Vehicular Environments (WAVE) and supported only a single application type per simulation setup. Artery originally addresses exactly these issues: running multiple applications per vehicle and facilitating communication with the European standard specifications.

While in the beginning the link between Artery and Veins has been very strong, i.e., Artery was merely an extension of Veins, today one can run Artery without a single line of code from Veins. Veins support has not been dropped entirely though, as it is now one possibility for the radio model choice. The other option is to employ the INET Framework for simulating the radio-related aspects such as the wireless medium, the propagation process and its effects, and the actual transmitting and receiving Network Interface Cards (NICs). In a nutshell, Artery's relation to Veins evolved from being an extension to a full-grown sibling.

Before digging deep into Artery and its various features, this section introduces the peculiarities of European Vehicle-to-Everything (V2X) communication

R. Riebl (✉) · C. Obermaier
Technische Hochschule Ingolstadt, Ingolstadt, Germany
e-mail: raphael.riebl@thi.de; christina.obermaier@thi.de

H.-J. Günther
Volkswagen Group of America, Auburn Hills, MI, USA
e-mail: hendrik-joern.guenther@vw.com



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_12

standards. As a sideline, some commonalities and differences between Artery and Veins (see Chap. 6) are highlighted, though both could be simply labeled as V2X simulation frameworks.

### 12.1.1 Intelligent Transport Systems in Europe

Wireless communication between vehicles and other road participants, often referred to as V2X communication, can be realized based on various technological approaches. In Europe, this kind of communication is standardized by the ETSI under supervision of its Technical Committee for Intelligent Transportation Systems (ITS). The European Union even allocated a dedicated frequency spectrum for the purpose of increasing safety, efficiency, and comfort in road traffic. This frequency band is located at 5.9 GHz and hence the communication in this band according to the ETSI standards is called ITS-G5.

At the lower layers, namely the physical and the link layer, ITS-G5 borrows a lot from IEEE 802.11p. The 802.11 amendment with the "p" suffix introduced new features to the well-known and omnipresent Wireless Local Area Network (WLAN) standard enabling its use for V2X communication. The most significant change is the Outside the Context of a Basic Service Set (OCB) mode enabling direct communication between vehicles without requiring additional infrastructure. Hence, vehicles are forming a completely decentralized network without being required to ask for permission entering it and sending messages in it. This type of network established by vehicles is summarized by the term VANET: all communication is spontaneous and possible links can vanish as quickly as they emerged ("ad-hoc").

One has to note, though, that there exist VANETs using different protocols while sharing the ad-hoc principle. Although the European ITS-G5 and its U.S. counterpart WAVE share the 802.11 layers, i.e., identical radio hardware can be used in both regions, their upper layers differ significantly. In this context, Veins belongs to the American world of standards with its models for IEEE 1609 (also known as WAVE). Artery, in turn, puts its emphasis on the European communication architecture [1].

### 12.1.2 Modeling ITS-G5

Let us take a closer look at the European communication stack and the respective layers' responsibilities. A basic understanding of these ITS layers will make it easier to understand Artery's architecture introduced in Sect. 12.2.

**Physical and Medium Access** As mentioned before, the radio-related layers follow IEEE 802.11. The INET Framework includes an elaborated model for 802.11 which can be configured for VANET communication. Based on INET's `Ieee80211Nic` the following adoptions are required:

- Enable ad-hoc communication by enforcing the `Ieee80211MgmtAdhoc` link management entity instead of infrastructure-based communication.
- Set operation mode to "p" so that the associated modulation schemes are used.
- Adjust radio channel bandwidth to 10 MHz and carrier frequency to 5.9 GHz.
- Radios with activated OCB show a slightly different backoff behavior with respect to contention window sizes and prioritization as per [7, Table 9-138].

Those adjustments are already accomplished when Artery's `VanetNic` is used. Furthermore, it extends INET's Medium Access Control (MAC) layer implementation by another feature: *channel busy reports*. As shown later on, nodes in a VANET are expected to adapt their transmission behavior depending on the current channel congestion. The level of congestion is announced by these channel busy reports, which are issued every 100 ms. Artery implements this feature by an extended radio receiver module `VanetRx` emitting a `ChannelLoad` OMNeT++ signal containing the ratio of time the channel has been busy during the last 100 ms interval.

**Network and Transport** The network and transport layers enhance the communication features and increase the developer's comfort when crafting various applications over a single medium as provided by `VanetNic`. NICs like `VanetNic` provide only the fundamental *link-layer communication* between stations, i.e., vehicles can only exchange data when in radio communication range. Those link-layer packets can be addressed to one (MAC unicast) or all (MAC broadcast) nearby stations.

The network layer's main purpose is to route packets and send them to their destination which may be farther away than the radio link distance. In ubiquitous Internet traffic routing is dealt by the Internet Protocol Version 4 (IPv4) and Internet Protocol Version 6 (IPv6) protocols, whereas in the context of ITS it is handled by GeoNetworking (GN) [4]. Broadly speaking, Internet Protocol (IP) protocols are optimized for routing packets over long distances to a specific remote host identified by its IP address. Due to the nature of ITS use cases, we seldom want to communicate with only one clearly identified vehicle ("The car with registration number..." ) but with a group of vehicles in a specific geographic area. Hence, GN is used at the network layer instead of classical IP. A non-exclusive summary of features provided by GN on top of a link layer's capabilities comprises:

- extending the communication range by using other vehicles as intermediate relaying nodes (multi-hop communication),
- dissemination of packets into geographically scoped destination areas, e.g., to all vehicles within a given $400 \times 400$ m rectangle,
- delaying a transmission until another vehicle is within communication range via GN's Store-Carry-Forward (SCF), and
- cryptographically secured packet content to ensure integrity and authenticity.

Basic Transport Protocol (BTP) [5] introduces the concept of port numbers to the ITS communication stack. Each application is assigned a unique port number which

enables the identification of a packet's source and destination application. Thus, each vehicle can run as many distinct applications as port numbers are available. The ETSI [3] maintains a list of well-known BTP port numbers.

Artery leverages Vanetza to provide BTP and GN features, as well as Decentralized Congestion Control (DCC) and security according to ETSI's communication architecture. Vanetza is not strictly bound to OMNeT++. It is merely an ordinary C++ library published under an open-source license. Thus, Vanetza can be used in simulations as well as in other software and even on embedded hardware. Artery handles all the details of integrating Vanetza into OMNeT++. Please refer to Sect. 12.2.1.4, [12], and Vanetza's website[1] for information about the internals of Vanetza.

**Facilities and Applications** *Facilities* sit on top of the BTP transport layer according to the ETSI ITS architecture. One can look upon facilities as a collection of helper tools to provide contextual information for ITS applications. For example, a facilities' Local Dynamic Map (LDM) is a kind of knowledge database collecting information received from other vehicles. Local vehicle data, on the other hand, is provided by its Vehicle Data Provider (VDP). The primary source for VDP are the vehicle parameters received from the SUMO mobility simulator but it can estimate further data elements such as curvature too.

All those helpers of facilities are hosted by Artery's middleware OMNeT++ module. This middleware is also in charge of establishing and running a configurable set of ITS services per vehicle. As of today, Artery includes the *Day One* services Cooperative Awareness (CA) and Decentralized Environmental Notification (DEN) out-of-the-box. In the context of Artery, ITS applications are realized as Artery services (e.g., `CaService`).

**Mobility** There is one striking difference between classical computer networks and VANETs: vehicles are moving much faster than a desktop computer, a laptop, or a server. If we want to simulate a VANET sufficiently realistic, we cannot neglect the inherent movement of its vehicular network nodes. While OMNeT++/INET provides some mobility features out-of-the-box to move nodes randomly or on predefined paths, the mobility of vehicles is another simulation domain on its own. Fortunately, Simulation of Urban MObility (SUMO) is a comprehensive traffic simulator, freely available and comparatively easy to couple with other tools such as OMNeT++ via its Traffic Control Interface (TraCI) protocol. Artery—in contrast to Veins—bundles SUMO's official C++ Application Programming Interface (API) and thus enables developers to make use of all TraCI features. Our integration of SUMO into Artery emphasizes extensibility and avoids tight coupling of classes by emitting several OMNeT++ signals. Any other OMNeT++ module can listen on those signals to get notified about SUMO simulation steps and single vehicle changes such as insertion, updates, and deletion of OMNeT++ modules associated with SUMO vehicles. SUMO and TraCI are covered in detail in Sect. 12.2.1.7.

[1] Vanetza project website: http://www.vanetza.org.

### 12.1.3 Setting up Artery

A significant difference of Artery compared to other well-known OMNeT++ frameworks is its **CMake**-based build process. Thus, setting up Artery and getting it to run differs from the usual "Import Project" procedure. It is certainly not complicated either but offers a lot of possibilities such as flexible integration of third-party libraries (compare the dependency graph of Artery depicted in Fig. 12.1 for example). In fact, since we were reluctant to reinvent the wheel or to dully copy code we wanted to make use of third-party libraries. From experience we can say that this speeds up the development process significantly and allows to focus on the actual simulation problem instead of "boilerplate code".

One problem occurring when trying to integrate external components into any C++ project, however, is the variance where those components are located on individual systems. In particular, we need to know the path to include headers and the location of the pre-compiled libraries to link them with our code. Using **Eclipse**—i.e., the OMNeT++ Integrated Development Environment (IDE)—can be quite frustrating because those paths need to be configured on each single system. Since this is a common problem in C/C++ development, smart developers have invented **CMake** which eases the exhausting task of finding and configuring build dependencies.

Artery itself depends on the well-known Boost libraries, Vanetza, and OMNeT++. A few dependencies are directly tracked in Artery's *extern* directory, i.e., this directory includes copies of external projects known to work with Artery in their respective version. Thus, one must not worry about matching versions

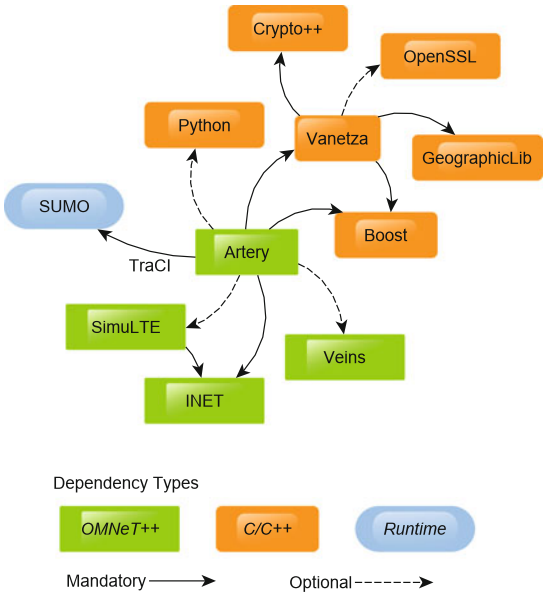



**Fig. 12.1** Dependency graph of Artery incorporating other OMNeT++ models as well as ordinary C/C++ libraries. SUMO is not part of Artery itself but required during runtime

between Artery, OMNeT++/INET, and Veins by default. Other dependencies, such as Boost and OMNeT++, are expected to be installed on the system and are looked up automatically.

An installation guide of Artery is included in its repository, we refer to it for installation details. Some internals of integrating OMNeT++ and **CMake** are discussed in a previous publication [10]. Tips and tricks regarding build configurations and available build options are presented in the appendix of this chapter.

## 12.2 Artery at the Core

In this section, we will take you through the process of actually working with the Artery simulation environment. After finishing this section, you will be familiar with all basic features of Artery and you should be able to start developing your own applications and hence start your own simulations. To make things a bit more interesting, we opted for building a small simulation scenario which can be observed everyday throughout the world: a police vehicle needs to pass a section of a street and requires that other vehicles clear the lane. This scenario is depicted in Fig. 12.2.

As depicted, the police vehicle's path (vehicle 0) is blocked by a group of vehicles on the highway. An attentive driver of vehicle 1 would have to recognize the approaching police vehicle and is required to change to the adjacent lane as soon as possible, as indicated by the arrow with the dashed line. This situation is usually stressful for all involved parties. For the driver of vehicle 1, the situation is stressful as he needs to figure out a safe way to get out of the police vehicle's way as soon as possible. For the police officers, the situation is stressful as they are approaching at a high relative speed.

Since you are currently reading about V2X communications, we dare to ask the question how we can relieve all parties of at least some of the stress. A direct communication link between the vehicles provides the option of explicitly warning the driver of vehicle 1 (and all other vehicles in the vicinity) of the approaching police vehicle. Drivers on non-affected lanes can be issued a warning and a recommendation to stay on the current lane. The driver of vehicle 1 is informed about the approaching police vehicle 0 well in advance and is therefore given more

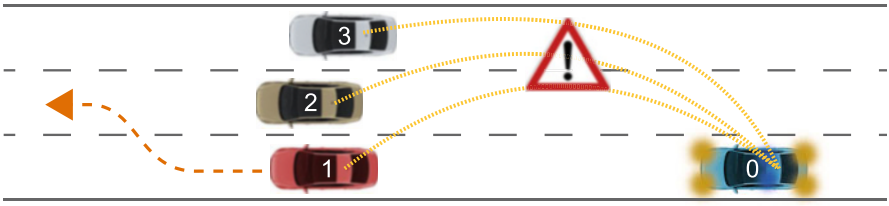



**Fig. 12.2** Simulation scenario: an approaching police vehicle 0 requires right-of-way. The depicted IDs refer to the vehicle identification used within the SUMO scenario

time to change to an adjacent lane. Simultaneously, the police officers can be assured that other vehicles are warned about their presence as well.

In the following, we will now assume the role of an application developer who has to specify, implement, and test the application. Obviously, there are two parts in this application:

1. The *transmitting part* on the police vehicle 0 has to broadcast a message about its approach and its current position.
2. The *receiving part* on all other vehicles 1–3 has to react to a received message from the police vehicle 0 in order to issue a warning to the driver.

The next subsections provide details for all steps required to build the application. It should be noted that since both parts are unique to the role of the vehicle within the scenario, we will implement two separate applications (a transmitting and a receiving application) to accomplish the task.

Artery thereby provides all required tools to focus on building user-defined applications, including the development of new message formats which may be exchanged between the vehicles. Note that although Artery focuses on the development of applications, the focus of the simulations can also be on any other aspect of the communication stack. Being an open-source framework, any layer can be modified for specific research questions.

After completing this section, you will have knowledge about these topics:

- the basic architecture and components of the Artery simulation environment including its basic triggering and timing mechanisms to be used by user-defined algorithms [Sect. 12.2.1],
- create and parametrize a simulation scenario for Artery [Sect. 12.2.2],
- write and build your own application (called *Services* within the Artery nomenclature) [Sects. 12.2.3.1 and 12.2.3.2],
- run simulations with your own application [Sect. 12.2.4], and
- record and analyze user-defined application-specific data [Sect. 12.2.5].

### *12.2.1 Architecture of Artery*

To concentrate on the development of V2X applications, Artery provides a framework to simplify the interaction with the communication stack. Although we are focusing on the European ETSI ITS G5 stack based on the IEEE 802.11p standard in this section, note that the stack itself may be changed to support other technologies as well (e.g., cellular V2X as outlined in Chap. 13). The various components that we integrated to Artery are shown in Fig. 12.3.

The depicted stack is instantiated for every vehicle within the simulation. On top of the two lower layers (namely, either Veins or INET and Vanetza) operates Artery's middleware that serves as an abstraction and data provisioning layer for the applications which are called *Services* in the nomenclature of Artery [11].

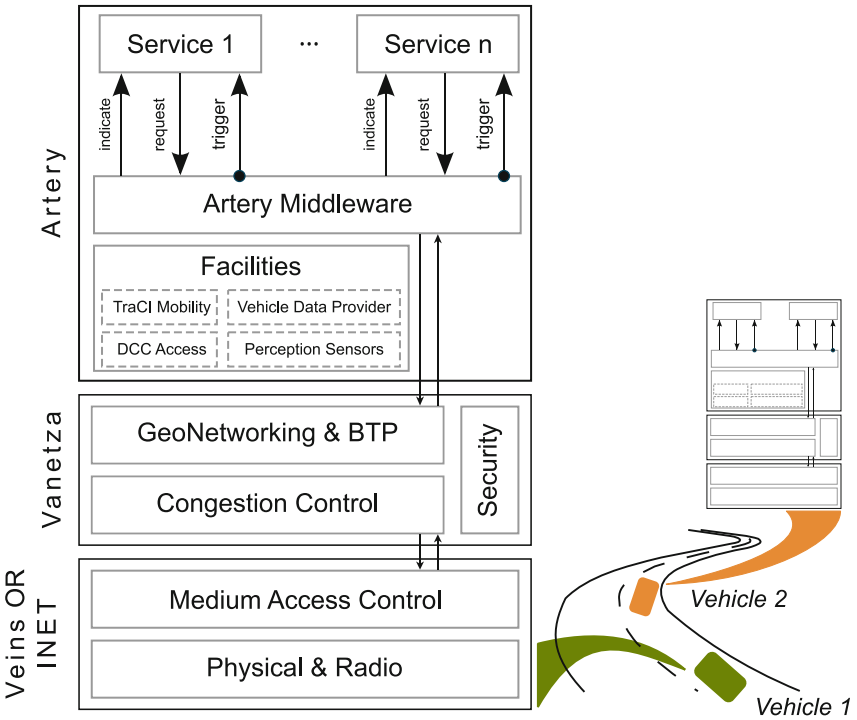



**Fig. 12.3** Artery architecture and multiple instantiation for every vehicle within the simulation

#### 12.2.1.1 Middleware

The key element of Artery is the middleware which serves as the interaction hub between the applications and the communication stack. The middleware provides the only (and therefore easy to maintain) interface to the lower layers and is also responsible for the life cycle and initialization of the ITS station's services.

To assign applications to vehicles within the simulation, an XML configuration is loaded by the middleware using the parameter `middleware.services`, which is accessible via the global simulation initialization and configuration file (usually named *omnetpp.ini*). The XML services file enables simulation-specific parameterization of service-penetration rates and alike. This is particularly useful, in case a specific application shall only be active for certain vehicle types. In the scenario depicted in Fig. 12.2 for which we are developing an application in this section, this becomes a handy feature as well, as it enables us to only equip the police vehicle with the transmitting application while all other vehicles will only have the receiving application. This approach also enables the development of different versions of the same application in order to assess its interoperability when operating on the same message set, e.g., for backward compatibility tests.

More information about the options provided by the external configuration file and how to parametrize it for the scenario depicted in Fig. 12.2 are given in Sect. 12.2.2.

In accordance with stack specifications, each service listed in the external configuration file can be linked with a single port number. This number is used for multiplexing transmitted or received messages to the corresponding services. A service generating the standardized CA Message (CAM), for example, is assigned the port number 2001 [3]. It should be noted that the assignment of ports may be arbitrary in case of non-standardized services or not required at all, e.g., in case you are developing an application which does not rely on message reception or transmission. The port number is used by a port-dispatcher within the middleware to either provide transport information when transmitting a message to the lower layer or to forward the message to the correct service upon message reception from the lower layer.

As stated above, next to the message multiplexing between the lower layers and the applications, the middleware is responsible for the life cycle management of the applications. This is particularly important, when you think about how and when vehicles are usually introduced to the simulation (e.g., by SUMO). Usually, a traffic simulation like SUMO also operates on a discrete time line. Vehicle positions and their dynamic states are all updated simultaneously within the simulator. This behavior gets reflected in OMNeT++. Vehicles will enter and leave the simulation at the same time stamp although in real life, we do not synchronize on specific points in time to turn on our vehicle (even though you and your neighbor might leave the house at the same time to drive to work). To avoid effects introduced by synchronized vehicle behavior, the Artery middleware introduces a separate life cycle management for every vehicle. Although the positions and dynamic states of the vehicles within OMNeT++ get updated simultaneously (i.e., at the SUMO update step which can be configured in the **.sumocfg* of the scenario), Artery ensures that individual (i.e., random offset but cyclic) update and triggering intervals for vehicle applications are observed. Figure 12.4 depicts the behavior introduced by Artery's middleware.

Though both vehicles 1 and 2 are updated by SUMO simultaneously, the applications are triggered by each vehicle's Artery middleware at different points in time. Hence, in case you need a cyclic triggering of your application algorithm, the `trigger()` method of each service can be used. Note that each service can also have its own triggers, using OMNeT++'s concept of self-messages. Upon removal of vehicle 2 from the simulation, the applications of vehicle 1 are still updated at their initial update offset from the main SUMO update interval, as indicated.

As stated above, the middleware is responsible for routing all application messages from and to the lower layer. Whenever an application wishes to transmit a message (e.g., after the service encoded the message), it can pass the payload to the middleware, using the `request` method, as depicted in Fig. 12.3. Message reception is indicated to a service by the `indicate()` callback which gets called as soon as a message has been received by the Intelligent Transportation System-Station (ITS-S). Note that both, the `request()` and `indicate()` methods are not restricted to the service cycle-time introduced by the middleware. This means

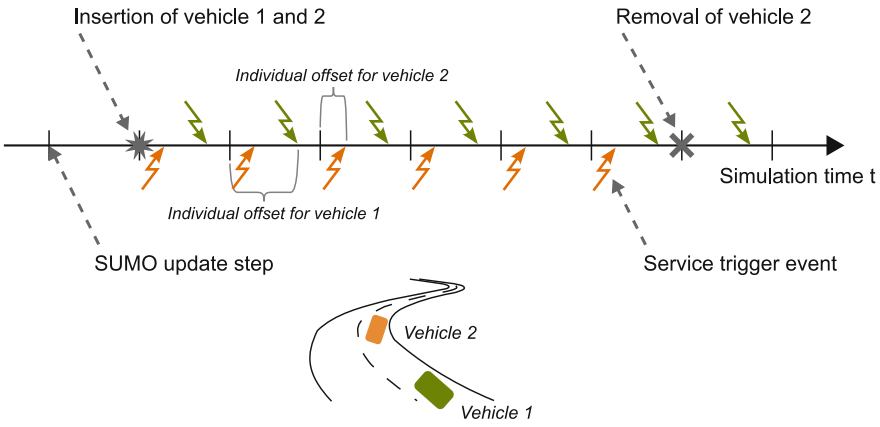



**Fig. 12.4** Life cycle management of Artery

that you do not have to wait until your service is triggered by the middleware in order to react on a received message. Instead, the `indicate()` method may also serve as an entrance-point to your algorithm's state machine. It should be noted that the middleware is capable of receiving two variants of messages: either you can create messages based on the Abstract Syntax Notation One (ASN.1) or rely straight on OMNeT++'s `cPacket` objects and its message compiler. Of course, OMNeT++ packets are only suitable for rapid prototyping in a pure simulation environment. If Artery is going to be coupled with external devices (e.g., Hardware-in-the-Loop (HIL) testbeds), its packets' binary representation must be compatible to real "over-the-air" packets. Then it is beneficial to use standardized message formats such as ASN.1's packet encoding rules.

Last, the middleware also provides access to other Artery components such as the `Facilities`, which are detailed in Sect. 12.2.1.3. You may even consider adding your own objects to the middleware which should be accessible by your services. The two extensions detailed in Sects. 12.3 and 12.4 make use of this feature.

#### 12.2.1.2 Services

Application logic and ITS functions, e.g., connected Advanced Driver Assistance System (ADAS) features, are realized as services in Artery. In essence, there are three major variants to attach custom logic to a service:

1. triggered by timed events by overriding `trigger()` or scheduling dedicated OMNeT++ messages,
2. reacting on packet receptions by overriding `indicate(...)`, and
3. listening to signals emitted by sibling modules and services.

Of course, any subset or mix of aforementioned variants is possible. The most suitable way to go depends on the particular type of applications that is going to implemented.

Artery ships with implementations of the basic CA and DEN services known from standardization. Those services emit unique signals on generation and reception of a corresponding message which can be used by other services. Each service can easily make use of OMNeT++'s signaling mechanism by subscribing to a particular signal through calling its `subscribe` method. Exemplary usage of this feature is demonstrated in Listing 12.1. Using signals instead of direct method calls has the major advantage of having only loose coupling between services, i.e., the service emitting the signal does not need to know anything about services interested in this signal and vice versa. Only the signal's name and emitted type need to be known.

**Listing 12.1** Exemplary usage of service signals

```
// those lines may be in YourService.cc
using namespace omnetpp;

// this is only required if we want to cast received object type
#include "artery/application/CaObject.h"
// no need to include CaService.h at all!

// we know CA service emits "CamReceived" with cObject* (CaObject*)
static const simsignal_t camRxSignal =
  cComponent::registerSignal("CamReceived");

// subscribe to signal at any time (in most cases at initialization)
void YourService::initialize()
{
  ItsG5BaseService::initialize();
  subscribe(camRxSignal);
}

// CamReceived signal emits a cObject*
void YourService::receiveSignal(cComponent* source, simsignal_t sig, cObject*
     obj, cObject*)
{
  if (sig == camRxSignal) {
    // optionally cast emitted object to derived type (CaObject in this case)
    CaObject* ca = check_and_cast<CaObject*>(obj);
    // do some action
  }
}
```

#### 12.2.1.3 Facilities

In terms of the ITS architecture, the *Facility layer* is located just between the transport layer and the ITS applications. The components of the *Facility layer*, the *Facilities*, are intended to support the realization of applications by providing access to information sources, easing usage of communication interfaces and exchanging data between basic services. Hence, those *Facilities* can be seen as general purpose supporting tools for ITS applications.

Artery implements *Facilities* as a generic object container, i.e., arbitrary C++ objects can be registered at its `Facilities` object. Those registered objects are then available to all users of the Artery `Middleware`, e.g., any service can refer to the respective station's `Facilities` via `getFacilities()`. By default any station, i.e., any OMNeT++ node equipped with Artery's `Middleware` module, exposes the following entities as *Facilities*:

- A `Timer` object to convert between ITS time and OMNeT++ simulation time. The time base is configured by the middleware's `datetime` parameter, i.e., simulation time zero corresponds to `datetime` and both are incremented uniformly.
- An LDM that stores information received via V2X such as recent CAMs for a predefined time.
- A DCC scheduler and state machine to control and react on channel congestion.

Vehicle stations, i.e., those equipped with a `VehicleMiddleware`, offer `VehicleDataProvider` and `VehicleController` in addition. The former grants read-access to vehicle data directly given by SUMO e.g., speed, position, and heading/direction. Furthermore, `VehicleDataProvider` offers some "derived" data that is not directly available otherwise, e.g., yaw rate and estimation of curvature (inverse of curve radius). The latter, `VehicleController` grants direct read and write access to the associated SUMO vehicle via TraCI. When you need to change a vehicle's behavior, e.g., its speed or route, this controller becomes handy.

#### 12.2.1.4 Vanetza

As already outlined in Sect. 12.1.2, the networking protocols above the link layer and below the *Facilities* are not traditional simulation models but a real implementation named *Vanetza*. Although Vanetza does not depend on OMNeT++, it has been designed to fit into an OMNeT++ environment from the beginning. Briefly speaking, the conglomerate of GN, BTP, Security, and DCC is covered by Vanetza. Since Vanetza is a plain C++ library, it can be linked into the Artery binary straightforwardly, i.e., all classes provided by Vanetza can be employed by OMNeT++ modules then. Fortunately, the cumbersome integration of external libraries with OMNeT++ is already dealt with by Artery's build system. We can therefore simply focus on the usage aspects of Vanetza for VANET simulations.

Artery's `Middleware` does the code-wise lifting of the communication stack by integrating the time-related protocol aspects as well as the packet data flow with OMNeT++'s event processing.

**Router and Runtime** The GN router instance is a key object when using Vanetza. This router is represented by the `vanetza::geonet::Router` class, which is instantiated and configured by Artery's `Middleware`. OMNeT++ configuration parameters that are directly affecting Vanetza are prefixed with `vanetza` such as `vanetzaEnableSecurity` which allows to control the presence of security

features, for instance. Another essential object is the `vanetza::Runtime`, which is used by Vanetza to determine the current time and execute time-based actions, e.g., cleaning expired entries from its GN location table. The interweaving of *Runtime* and OMNeT++ is threefold:

1. Runtime is initialized with `Timer's` base time so Vanetza and OMNeT++ have a synchronized starting point.
2. Runtime is updated at each `Middleware::handleMessage` call to keep the `Middleware` and Vanetza synchronized.
3. An OMNeT++ self-message is scheduled for the next scheduled Vanetza action in `Middleware::scheduleRuntime`.

**Data Request (Transmitting Packets)** When an Artery service wants to transmit a packet it passes its message (i.e., the BTP payload) to the `request` method provided by the service's base class. Behind the scenes, this message is passed on to `vanetza::btp::RequestInterface` provided by `Middleware`, which invokes Vanetza's packet handling routines. Finally, Vanetza passes the processed packet with attached network protocol headers and alike to the link layer. From Vanetza's point of view the link layer is represented by the `vanetza::access::Interface`. This interface is registered at the router during initialization of the middleware. The middleware's implementation of this interface issues the transmission by INET or Veins via the respective `RadioDriverBase`.

**Data Indications (Receiving Packets)** Packets received by the link layer are passed to the middleware via its `radioDriverIn` gate. Messages arriving at this gate are fed into Vanetza GN router in `Middleware::handleLowerMsg` for processing network and transport protocol headers. As soon as Vanetza is done with this processing and it is determined that the packet is relevant for the local station, the extracted BTP payload is passed up. Vanetza knows the matching receiving service because `Middleware::initializeServices` registers each listening service at its local `vanetza::btp::PortDispatcher`. Consequently, the service with the matching BTP port number receives the original packet via its `indicate` method.

As a rule of thumb, customizations concerning packet routing shall be done in Vanetza directly using plain C++ with the additional benefit that those customizations could then be deployed to other environments as well, for instance, to real V2X devices. Various internal actions by Vanetza can be observed by means of a so-called hook mechanism. For example, you can assign custom C++ code to `Hooks` exposed by the router, which are called when packets are dropped for one or another reason.

**Listing 12.2** Example: Registering a custom hook at Vanetza router

```
// you may append these lines to Middleware::initializeMiddleware()
mGeoRouter.forwarding_stopped =
[this](vanetza::geonet::Router::ForwardingStopReason reason) {
  if (reason == vanetza::geonet::Router::ForwardingStopReason::HOP_LIMIT) {
    EV_INFO << "GN packet dropped, reached hop limit\n";
  }
};
```

#### 12.2.1.5 Physical Layer Modeling: Veins vs. INET

When work on Artery began back in 2014 it started as a collection of additional modules for Veins. Consequently, the model of the wireless communication on the physical level has been directly inherited from Veins. The Veins' model, however, is centered around WAVE including a fixed channel switching scheme between a control and a service channel back and forth. Switching away from the control channel is not used by ETSI ITS-G5 at all, however. The periodic reporting of channel busy ratio measurements was missing, which is required as input for ITS-G5's congestion control. Hence, the lower radio layer (Physical Layer (PHY)) of Veins has been combined with a customized upper radio layer (MAC), which can be found in Artery's *src/artery/mac* directory by the name `MacItsG5`.

Later on, the INET Framework has been adopted by Artery as an alternative 802.11 model. The necessary adaptation code to integrate INET into Artery is located at *src/artery/inet*. Primarily, the original INET modules are configured with sane defaults for VANET communication, e.g., the contention windows and inter-frame spacings are overridden for OCB according to IEEE standardization in *VanetNic.ned*. A customized `inet::ieee8011::Rx` class—`VanetRx`—enhances INET's model to report channel busy ratio.

Nowadays, our recommendation is to use the model based on INET if there is no explicit reason to use the Veins radio model. Artery's middleware itself does not depend on neither Veins nor INET directly. The module interface `artery.nic.IRadioDriver` abstracts from the actual radio model. Artery ships driver implementations for INET (`InetRadioDriver`) and Veins (`VeinsRadioDriver`). This abstraction layer also allows to use an entirely distinct radio layer model.

#### 12.2.1.6 Mobility

Moving network nodes are usually realized by some kind of OMNeT++ *mobility* module. Both, Veins and INET facilitate such a mobility concept inherited from their common ancestor MiXiM.[2] The particular interfaces, however, are not entirely compatible to each other without further ado. Artery supports both—INET's `IMobility` and Veins' `BaseMobility`—and couples them with its `artery.traci.Mobility`. By this approach no code for coupling with SUMO via TraCI needs to be duplicated while maintaining compatibility with the respective frameworks. Whenever an update of a SUMO vehicle's state is received the corresponding mobility submodule—either `Inet` or `VeinsMobility`—is updated as well. Each Artery *car* possesses such a mobility submodule that emits a common update signal. The `VehicleMiddleware` receives those signals and refreshes,

[2]The MiXiM project has been discontinued, and its contents have been merged into the INET Framework. New projects should be based on a recent version of INET instead of MiXiM.

for instance, the *Facilities'* VDP. This also updates the position of the GN router subsequently, which needs to know its own position for geographical routing.

#### 12.2.1.7 SUMO and TraCI

As Artery evolved it switched from Veins' re-implementation of the TraCI API to the official C++ API by the SUMO project. Hence, every SUMO/TraCI feature supported upstream is also available when using Artery. Artery ships with a copy of SUMO' original C++ API at *src/traci/sumo*.

The coupling on the OMNeT++ side is realized by the `traci.Manager` module located in *src/traci/*. This manager module is composed of four submodules:

1. `Core`: essential TraCI life cycle incorporating initialization of the connection, timing update steps, and finally closing the connection. This module emits corresponding OMNeT++ signals `traci.init`, `traci.step`, and `traci.close`. Additionally, it checks if the remote SUMO instance provides the required TraCI API version.
2. `Launcher`: helper for establishing Core's TraCI connection either by connecting to an already running SUMO instance (`ConnectLauncher`) or launching a new SUMO process (`PosixLauncher`). By default, `PosixLauncher` starts the SUMO command-line version, but by setting its `sumo` parameter to `sumo-gui` the graphical version can be used as well.
3. `Node Manager`: controls the life cycle of individual vehicles by inserting, updating, and removing OMNeT++ modules according to the respective vehicle state in SUMO. These life cycle events are accompanied by the OMNeT++ signals `traci.node.add`, `traci.node.update`, and `traci.node.remove`.
4. `Module Mapper`: the node manager consults the module mapper which module type shall be used for a particular vehicle, usually `artery.inet.Car` or `artery.veins.Car`.

Access to the underlying TraCI API is possible either by fetching it directly from `Core::getLiteAPI()` or each vehicle's `VehicleController`. Extending functionality is usually easily possible by either listening to one or more of the various emitted signals or by sub-classing the aforementioned submodules. Except for `traci.Core`, the employed `traci.Launcher`, `traci.NodeManager`, and `traci.ModuleMapper` can be configured by setting the respective `typename` OMNeT++ attribute.

### *12.2.2 Driving Scenario*

With this basic knowledge about the internal workings of Artery from the previous section, we can start creating the scenario to experiment with our envisioned police

service. For this purpose, we create a new scenario directory *scenarios/highway-police* along the already existing scenarios. We are going to fill this directory gradually in the rest of this section.

#### 12.2.2.1 SUMO Roads and Traffic

First of all, we need some streets where vehicles can drive along. Creating maps and associated traffic demand is an art of its own and would burst this chapter. For experimenting with our service giving a police car space for overtaking, we create only a very simple highway segment. You can do this with the help of SUMO's **netedit** or even entirely by hand as shown in Listing 12.3 that creates a single-direction highway with three lanes of 1 km length. Please note the custom map projection (`projParameter` attribute) for converting Cartesian coordinates ($x$, $y$) to geodetic coordinates (longitude, latitude).

**Listing 12.3** Road network (*highway.net.xml*)

```
<?xml version="1.0" encoding="UTF-8"?>
<net version="0.27" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/net_file.xsd">

   <location netOffset="0.00,0.00" convBoundary="0.00,0.00,1000.00,0.00"
        origBoundary="-10000000000,-10000000000,10000000000,10000000000"
        projParameter="+proj=tmerc +ellps=WGS84 +datum=WGS84 +lat_0=49 +lon_0
        =11 +units=m +no_defs"/>

   <edge id="hw0" from="hwStart" to="hwEnd" priority="1" type="highway">
      <lane id="hw0_0" index="0" speed="33.33" length="1000.00" width="3.00"
           shape="1000.00,7.75 0.00,7.75"/>
      <lane id="hw0_1" index="1" speed="33.33" length="1000.00" width="3.00"
           shape="1000.00,4.65 0.00,4.65"/>
      <lane id="hw0_2" index="2" speed="33.33" length="1000.00" width="3.00"
           shape="1000.00,1.55 0.00,1.55"/>
   </edge>

   <junction id="hwEnd" type="dead_end" x="0.00" y="0.00" incLanes="hw0_0 hw0_1
         hw0_2" intLanes="" shape="0.00,9.25 0.00,0.05"/>
   <junction id="hwStart" type="dead_end" x="1000.00" y="0.00" incLanes=""
        intLanes="" shape="1000.00,0.05 1000.00,9.25"/>
</net>
```

As a second step, vehicles are added to the scenario as shown in Listing 12.4. All of them follow the same route `route0`, which is the only possible one on this simple map. Three ordinary passenger cars start immediately with their maximum speed, each on its own lane. The police car is inserted 3 s later on the left-most lane. This police car is allowed to exceed speed limits by 50% (`speedFactor`). The passenger cars are willing to keep right (`lcKeepRight`) if possible.[3]

[3]Correct `lcKeepRight` behavior requires SUMO 0.31 or newer, otherwise vehicles keep their lanes.

**Listing 12.4** Vehicle routes (*highway.rou.xml*)

```
1 <?xml version="1.0"?>
2 <routes>
3   <vType id="car" vClass="passenger" length="5" maxSpeed="50" lcKeepRight="10"
        color="0,0.67,0"/>
4   <vType id="police" vClass="authority" length="5" maxSpeed="60" speedFactor="
        1.5" minGap="0.5" color="0,0,1"/>
5   <route id="route0" edges="hw0"/>
6   <vehicle id="car0" type="car" route="route0" depart="0" departLane="free"
        departSpeed="max"/>
7   <vehicle id="car1" type="car" route="route0" depart="0" departLane="free"
        departSpeed="max"/>
8   <vehicle id="car2" type="car" route="route0" depart="0" departLane="free"
        departSpeed="max"/>
9   <vehicle id="police0" type="police" route="route0" depart="3" departLane="2"
        departSpeed="max"/>
10 </routes>
```

Both, the road network and traffic demand, are combined to a scenario via the SUMO configuration file in Listing 12.5. This configuration references the files *highway.net.xml* and *highway.rou.xml* and sets up SUMO's timing. Simulations using this configuration start at second zero and time will progress in 0.1 s steps.

**Listing 12.5** SUMO configuration (*highway.sumocfg*)

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="http://sumo.sf.net/xsd/sumoConfiguration.
      xsd">
3    <input>
4       <net-file value="highway.net.xml"/>
5       <route-files value="highway.rou.xml"/>
6    </input>
7    <time>
8       <begin value="0"/>
9       <step-length value="0.1"/>
10   </time>
11 </configuration>
```

You can run this scenario now standalone with SUMO via *File→Open Simulation* and observe what happens. The passenger cars drive shoulder to shoulder and thus block all three lanes. Since the police car is unable to pass them, it has to slow down. Figure 12.5 shows the start of the standalone simulation.

#### 12.2.2.2 Artery Parameters

V2X communication is now expected to solve this situation. Subsequently, the police car (SUMO vehicle `police0`) will be equipped with a special `PoliceService` as shown in lines 3–6 in Listing 12.6. All other vehicles except the police car will be equipped with `ClearLaneService`, which is going to react upon messages received from the police. Both services are listening to the (arbitrarily chosen) port number 8001. The respective services are assigned to vehicles based on their SUMO vehicle identifier using name pattern filters (lines 5 and 9). Only vehicles matching the given pattern are assigned the respective

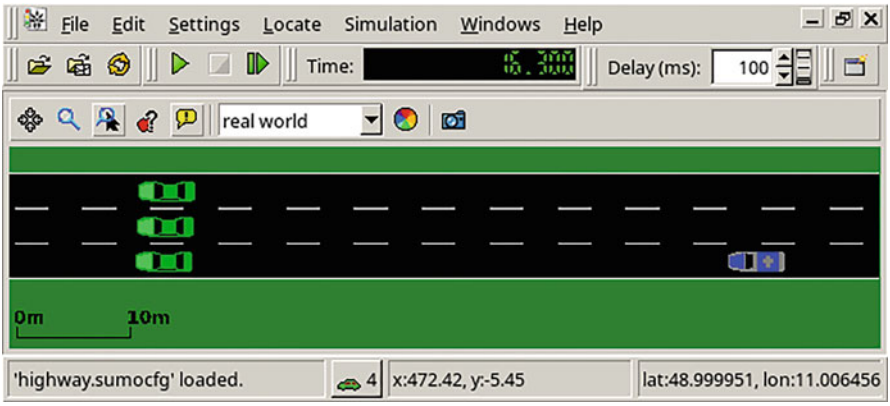



**Fig. 12.5** Screenshot of SUMO running the created highway scenario (standalone)

service. By adding the attribute `match="inverse"` the pattern is inverted, i.e., only vehicles not matching the pattern are equipped. While the exact vehicle name is used as a pattern in this example, in fact any regular expression can be used. For instance, `pattern="police.*"` would match any vehicle whose name is starting with `police`, including `police0` but also `policeMotorcycle` and so on.

**Listing 12.6** Service configuration (*services.xml*)

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <services>
3     <service type="PoliceService">
4        <listener port="8001" />
5        <filters><name pattern="police0" /></filters>
6     </service>
7     <service type="ClearLaneService">
8        <listener port="8001" />
9        <filters><name pattern="police0" match="inverse" /></filters>
10    </service>
11 </services>
```

The configuration in Listing 12.7 is the last piece. With `artery.inet.World`, we use a basic Artery network with radio modules derived from INET. Furthermore, the network's TraCI manager is advised to launch a separate SUMO process (line 5) using our previously created SUMO configuration (line 6). By default, Artery only accepts TraCI connections of the same version as its bundled SUMO API. Setting `*.traci.core.version` from default 0 to −1 tells Artery to accept any TraCI version. Assigning a positive number would enforce exactly this version.

The middleware's time base is set to 19th March 2018 at 10 a.m., so time 0.0 in OMNeT++ and SUMO refers to this time point. Every 100 ms each middleware schedules a self-message to call its `Middleware::update()` and `trigger()` of all its services. Last but not least, the *services.xml* introduced in Listing 12.6 is referenced.

**Listing 12.7** OMNeT++ configuration (*omnetpp.ini*)

```
[General]
network = artery.inet.World

*.traci.core.version = -1
*.traci.launcher.typename = "PosixLauncher"
*.traci.launcher.sumocfg = "highway.sumocfg"

*.node[*].middleware.updateInterval = 0.1s
*.node[*].middleware.datetime = "2018-03-19 10:00:00"
*.node[*].middleware.services = xmldoc("services.xml")
```

### *12.2.3 Creating Services*

The *services.xml* is already referring to the two new services, *PoliceService* and *ClearLaneService*. Their implementation will now be outlined in this section.

#### 12.2.3.1 Development of Police Vehicle's Service

First, let us create the police car's service demanding for a free lane to pass the other vehicles. Strictly speaking, ETSI standardization of the CA service [2] already covers this use case by appending an "Emergency Vehicle Container" to CAMs sent by the police car. As we want to keep things simple in this example, we will instead design our own OMNeT++ message for educational purposes.

For now, there is only one critical information necessary to be propagated by the police car: Which lane shall be cleared by other vehicles? A lane can be uniquely identified in SUMO by a pair of edge name and lane index, so these two data fields are essential for our `PoliceClearLane` message. Put the message definition given in Listing 12.8 in a file named *PoliceClearLane.msg* next to our SUMO files, i.e., at *scenarios/highway-police*. Of course, you can place source files wherever you find them fitting, e.g., also somewhere in *src/artery*. For this tutorial section, however, it is a good idea to keep all custom files together for the sake of simplicity. This *PoliceClearLane* message will be transmitted by our `PoliceService`, which itself consists of three files: Network Topology Description (NED) module description and C++ source and header files (cf. Listings 12.11 and 12.10, respectively).

**Listing 12.8** `PoliceClearLane` message definition

```
packet PoliceClearLane
{
   string edgeName;
   int laneIndex;
};
```

The NED module description (cf. Listing 12.9) is fairly short, it only tags the `PoliceService` module to be an `artery.application.ItsG5Service`.

**Listing 12.9** Content of the *PoliceService.ned* file

```
1 import artery.application.ItsG5Service;
2
3 simple PoliceService like ItsG5Service {}
```

This relationship between *PoliceService* and *ItsG5Service* is also reflected in C++, hence our `PoliceService` class inherits from Artery's `ItsG5Service` (see line 6 in Listing 12.10). Two methods of the `ItsG5Service` are going to be overridden: `trigger()` will be used to periodically send out *PoliceClearLane* messages and `initialize()` will initialize the `mVehicleController` attribute. `traci::VehicleController` allows us to determine the edge and lane a vehicle is currently driving on, the information to be included in the sent messages.

**Listing 12.10** Content of the *PoliceService.h* file

```
1  #ifndef POLICESERVICE_H_
2  #define POLICESERVICE_H_
3
4  #include "artery/application/ItsG5Service.h"
5
6  class PoliceService : public ItsG5Service
7  {
8     public:
9        void trigger() override;
10
11    protected:
12       void initialize() override;
13
14    private:
15       const traci::VehicleController* mVehicleController = nullptr;
16 };
17 #endif /* POLICESERVICE_H_ */
```

In line 16 of Listing 12.11, we make use of *Facilities* and fetch a read-only reference to the `VehicleController`. As a general rule, every time `initialize` of a service is overridden the original method from the base class has to be called as well (line 15). In `PoliceService::trigger`, we first prepare access to SUMO's vehicle API (lines 22ff.) and use this `vehicle_api` to fill a new *PoliceClearLane* message (lines 25–27). We make up an arbitrarily chosen byte length for this message of 40 bytes. Note that those 40 bytes only refer to the length on application layer, i.e., the total length of the packet will be considerably longer since every layer adds further headers.

**Listing 12.11** Content of the *PoliceService.cc* file

```
1 #include "PoliceService.h"
2 #include "police_msgs/PoliceClearLane_m.h"
3 #include "artery/traci/VehicleController.h"
4 #include <vanetza/btp/data_request.hpp>
5 #include <vanetza/dcc/profile.hpp>
6 #include <vanetza/geonet/interface.hpp>
```

```
 7
 8 using namespace omnetpp;
 9 using namespace vanetza;
10
11 Define_Module(PoliceService)
12
13 void PoliceService::initialize()
14 {
15    ItsG5Service::initialize();
16    mVehicleController = &getFacilities().get_const<traci::VehicleController>();
17 }
18
19 void PoliceService::trigger()
20 {
21    Enter_Method("PoliceService trigger");
22    const std::string id = mVehicleController->getVehicleId();
23    auto& vehicle_api = mVehicleController->getLiteAPI().vehicle();
24
25    auto msg = new PoliceClearLane();
26    msg->setEdgeName(vehicle_api.getRoadID(id).c_str());
27    msg->setLaneIndex(vehicle_api.getLaneIndex(id));
28    msg->setByteLength(40);
29
30    btp::DataRequestB req;
31    req.destination_port = host_cast<PoliceService::port_type>(getPortNumber());
32    req.gn.transport_type = geonet::TransportType::SHB;
33    req.gn.traffic_class.tc_id(static_cast<unsigned>(dcc::Profile::DP1));
34    req.gn.communication_profile = geonet::CommunicationProfile::ITS_G5;
35    request(req, msg);
36 }
```

Lines 30–34 configure a `btp::DataRequestB` object that controls the transmission, but is not part of the packet. The BTP destination port is set in line 31 by retrieving the port number defined in *services.xml* using `getPortNumber` provided by Artery. The remaining transmission parameters are hard-coded: the message is going to be sent via Single-Hop Broadcast (SHB) (like CAMs) and DCC profile DP1 (like normal DEN Messages (DENMs)). Finally, the message and its request parameters are passed to the next lower layer (BTP). Now the police car can tell surrounding vehicles which lane to clear so it can reach its destination faster.

#### 12.2.3.2 Development of Corresponding Reacting Vehicle Service

Though the police car can now express its intention, not much changed because other vehicles do not react upon police messages yet. Hence, those other vehicles are equipped with a *ClearLaneService* which is supposed to handle those messages. Similar to *PoliceService*, this services comprises a NED module and C++ code. Its header file printed in Listing 12.12 should already look quite familiar except for the `indicate(...)` method. This method is going to be called whenever a packet is received at the service's listening port defined in *services.xml*.

**Listing 12.12** Content of the *ClearLaneService.h* file

```
1 #ifndef CLEARLANESERVICE_H_
2 #define CLEARLANESERVICE_H_
3
4 #include "artery/application/ItsG5Service.h"
5
```

```
6  class ClearLaneService : public ItsG5Service
7  {
8     protected:
9        void indicate(const vanetza::btp::DataIndication&, omnetpp::cPacket*)
               override;
10       void initialize() override;
11
12    private:
13       traci::VehicleController* mVehicleController = nullptr;
14 };
15 #endif /* CLEARLANESERVICE_H_ */
```

Initialization of the service is almost identical to *PoliceService*. However, in *ClearLaneService* a modifiable `VehicleController` is retrieved from *Facilities* because we want to change the vehicle's behavior by reducing the vehicle speed when necessary. Necessity to change behavior is checked on reception of a message in `ClearLaneService::indicate(...)`. This method is called with two arguments: a `btp::DataIndication` conveying information about lower layers' processing concerning the packet passed as second argument. Initially, the packet is a generic `omnetpp::cPacket` so we need to cast it down to the specific `PoliceClearLane` message type in line 19. Then, we can read the message's content and check if the receiving vehicle is driving on the same edge as the transmitting police car at all (line 23). If this is the case, those vehicles occupying a different lane than the police car are going to reduce their speed to a maximum of 25 m/s. By taking this approach, vehicles on the police car's lane can pass by other vehicles and change to another lane as soon as there is a suitable gap. In order to avoid memory leaks, the received message needs to be deleted at the end.

**Listing 12.13** Content of the *ClearLaneService.cc* file

```
1  #include "ClearLaneService.h"
2  #include "police_msgs/PoliceClearLane_m.h"
3  #include "artery/traci/VehicleController.h"
4
5  using namespace omnetpp;
6  using namespace vanetza;
7
8  Define_Module(ClearLaneService)
9
10 void ClearLaneService::initialize()
11 {
12    ItsG5Service::initialize();
13    mVehicleController = &getFacilities().
14       get_mutable<traci::VehicleController>();
15 }
16
17 void ClearLaneService::indicate(const vanetza::btp::DataIndication& ind,
       omnetpp::cPacket* packet)
18 {
19    Enter_Method("ClearLaneService indicate");
20    auto clearLaneMessage = check_and_cast<const PoliceClearLane*>(packet);
21
22    const std::string id = mVehicleController->getVehicleId();
23    auto& vehicle_api = mVehicleController->getLiteAPI().vehicle();
24    if (vehicle_api.getRoadID(id) == clearLaneMessage->getEdgeName()) {
```

```
25        if (vehicle_api.getLaneIndex(id) != clearLaneMessage->getLaneIndex()) {
26            mVehicleController->setMaxSpeed(25 * units::si::meter_per_second);
27        }
28    }
29    delete clearLaneMessage;
30 }
```

### 12.2.4 Run Simulation

With the code for our `PoliceService` and `ClearLaneService` readily placed in *scenarios/highway-police* one thing is left before running the simulation: we need to advise the build system to consider our files when building Artery. For this purpose, we place a *CMakeLists.txt* next to our other files that contain the instructions for Artery's build system based on **CMake**.

In line 1 in Listing 12.14, a new feature named `police` is added to Artery consisting of the two C++ source files of the previously created services. This feature depends on the *PoliceClearLane* message, so we tell **CMake** to generate C++ code from the message file. The generated code will be placed in a *police_msgs* folder, identical to the `#include` in line 2 of Listings 12.13 and 12.11. The last line adds so-called *run targets* for this exemplary scenario (named *highway_police*) to the build system.

**Listing 12.14** Content of *CMakeLists.txt* in the highway-police scenario

```
1 add_artery_feature(police ClearLaneService.cc PoliceService.cc)
2 generate_opp_message(PoliceClearLane.msg TARGET police DIRECTORY police_msgs)
3 add_opp_run(highway_police NED_FOLDERS ${CMAKE_CURRENT_SOURCE_DIR})
```

Let us run the scenario: go to the build directory of Artery (usually *build* in the Artery root directory) and call **make run_highway_police** from there. The *Makefile* generated by **CMake** will build Artery along with the added C++ sources. When done, **opp_run** will be invoked to start OMNeT++ with the created *highway_police* scenario. Since *scenarios/highway-police* includes NED files, we append this folder to the run target's NED folders. The build system will automatically determine the NED folders of all dependencies (e.g., Artery core, INET, Veins, etc.) and pass them to **opp_run**. You can find more details about run targets' features in the chapter appendix, including options for debugging and locating memory leaks.

Congratulations, you just ran your first custom Artery scenario! Now try to make the SUMO Graphical User Interface (GUI) visible as well. Hint: look for the `*.traci.launcher.sumo` parameter and set it in *omnetpp.ini* to **sumo-gui**. Compared to running the *highway.sumocfg* scenario standalone in SUMO, you should notice that the police car is able to pass the three vehicles in our OMNeT++ simulation effortlessly. Mission accomplished!

### 12.2.5 Data Analysis

While you can run and watch simulations already, usually one is interested in the data generated by OMNeT++. In principle, Artery outputs data the same way as other OMNeT++ models. Processing data after simulation, however, may require specific steps due to VANET characteristics. We therefore present some ideas how to analyze gathered data with a focus on filtering and aggregating data regarding time and space.

**Configuring Data Recording** There are numerous tools available which have not been designed for OMNeT++ specifically. Thus, it is often beneficial to record OMNeT++ data in *SQLite* databases instead of OMNeT++'s custom file format. Since OMNeT++ 5.1, you can advise OMNeT++ to record vectors and scalars using *SQLite* by the *omnetpp.ini* settings given in Listing 12.15.

**Listing 12.15** Activating SQLite-based recording

```
1 outputvectormanager-class="omnetpp::envir::SqliteOutputVectorManager"
2 outputscalarmanager-class="omnetpp::envir::SqliteOutputScalarManager"
```

Beside configuring *how* data is recorded, you can also control *what* is recorded. Especially long-running simulations with hundreds of vehicles may generate several gigabytes of data. Huge data files, however, can slow down any analysis significantly. Therefore, it is a good idea to think about the actually required data set before running large simulations.

OMNeT++ vectors are time series of data so the temporal aspect is inherently included in any recorded vector file. Often we also need to know the position of vehicles for data evaluation, e.g., we may want to analyze only data of vehicles while they are in certain *region of interest*. The module `artery.inet.Car` is already prepared to record the position information from its mobility submodule: it defines two `@statistic` vectors named `posX` and `posY` which only need to be enabled in *omnetpp.ini* (cf. Listing 12.16).

**Listing 12.16** Enable `artery.inet.Car` position recording

```
1 *.node[*].posX.result-recording-modes = vector
2 *.node[*].posY.result-recording-modes = vector
```

From previous sections you have learned that Artery makes heavy use of OMNeT++ signals. For example, `artery.application.CaService` emits a `CaObject` conveying the full CAM whenever one has been generated (signal `CamSent`) or received (signal `CamReceived`). Recording of some CAM data fields, such as the included station identifier or its differential timestamp, is already prepared by accompanying `@statistic` parameters in *CaService.ned*. Please refer to Artery's NED files for a comprehensive overview of all available signals. Furthermore, in the source file *src/artery/application/CaObject.cc* you can find examples of OMNeT++ *result filters* that demonstrate how to extract information from objects like `CaObject`.

**Post-processing** With simulation data stored in SQLite databases, you can process this data with **R** *dplyr*[4] or **Python** *pandas*,[5] among others. A full post-processing example based on **R** can be found at *scenarios/car2car-grid/tools.R*. Listing 12.17 highlights some essential fragments which may be useful for your post-processing needs with a few adaptations.

Lines 1–6 load the vector database file (*your_sim.vec* in this example), enable some additional *SQL* features, and tell *SQLite* to store temporary tables in memory for maximum speed. The tables *vector* and *vectordata* written by OMNeT++ are accessible via **R** variables `vec` and `vecdata`, respectively. Since dealing with vector names in one table and actual recorded vector data in another table is cumbersome, both tables are joined in a temporary table *vt*. Names and data entries from each table are matched via the common *vectorId* table column in line 9. Furthermore, the raw simulation time of column *simtimeRaw* is converted to floating-point seconds (by default, OMNeT++ `SimTime` has a time resolution of picoseconds, i.e., $10^{-12}$ s). The *collect* in line 12 instructs **R** to load the complete table content into its memory. If you are not low on memory, you should do this for best performance.

With all vector data at our hands in *vt* it is now possible to extract vehicles' position data (lines 15–18). Having $x$- and $y$-coordinates of each vehicle at each time step in the same row makes position-based filtering easy later on. Hence, we select those coordinates (line 15 and 16) and join them for equal time and originating module, i.e., vehicle's mobility module, into `pos`.

Since Artery cars are composed of various modules, it also handy to have a common attribute which distinguishes modules that belong to the same vehicle. You can derive such an exemplary `node` attribute from the module path, for instance: `World.node[3].middleware` and `World.node[3].mobility` belong both to node 3. For each distinct module path we calculate this node attribute in lines 21ff. and then have a lookup-up table `nodes` which can be used to quickly determine the node number for each module. This is demonstrated in line 25 where `vt3` contains only vectors belonging to the vehicle `node[3]`.

**Listing 12.17** Post-processing with **R** *dplyr*

```
1  # open OMNeT++ database and its vector tables
2  db <- DBI::dbConnect(RSQLite::SQLite(), 'your_sim.vec')
3  RSQLite::initExtension(db)
4  DBI::dbSenqQuery(db, 'PRAGMA temp_store = MEMORY;')
5  vec <- tbl(db, 'vector')
6  vecdata <- tbl(db, 'vectordata')
7
8  # join vec and vecdata to a table containing vector names and its data
9  vt <- left_join(vec, vecdata, by = 'vectorId') %>%
10   mutate(simtime = simtimeRaw / 10^12) %>%
11   select(name = vectorName, module = moduleName, simtime, value) %>%
12   collect(n = Inf)
13
```

[4] **R** *dplyr* package website: https://dplyr.tidyverse.org.

[5] **Python** data analysis library *pandas* website: https://pandas.pydata.org/.

```
# build position table of nodes, i.e. (x, y) in columns
posx <- vt %>% filter(name == 'posX:vector')
posy <- vt %>% filter(name == 'posY:vector')
pos <- inner_join(posx, posy, by = c('simtime', 'module')) %>%
  select(module, simtime, x = value.x, y = value.y)

# extract node number N from "*.node[N].*" strings as common vehicle identifier
nodes <- vt %>% distinct(module) %>% rowwise() %>%
  mutate(node = as.integer(str_match(module, ".*node\\[([0-9]+)\\].*")[1,2]))

# all vectors belonging to "node[3]"
vt3 <- left_join(vt, nodes, by = 'module') %>% filter(node == 3)

# all vectors of nodes within a rectangle
pos_roi <- left_join(pos, nodes, by = 'module') %>%
  filter(x > 20, x < 300, y > 100, y < 400) %>% select(-module)
vt_nodes <- left_join(vt, nodes, by = 'module')
vt_roi <- inner_join(pos_roi, vt_nodes, by = c('simtime', 'node'))
```

The remaining lines of Listing 12.17 filter vector data from `vt` depending on vehicle positions. `vt_roi` contains only data from vehicles while they are within a defined Region of Interest (ROI), in this case the rectangle with corner points at (20,100) and (300,400). Of course, there are many more nifty tools available with **R** and *dplyr* not covered here, e.g., aggregating columns by some criteria using `group_by` and `summarize`. You should have grasped some ideas how to process your own simulation data, though.

## 12.3 Local Perception Sensors

Now that you know how to run a basic simulation using your own modules and analyze the generated result files, you may start developing your own simulations. Primarily, Artery focuses on the development and the analysis of applications leveraging inter-vehicle communications. However, many connected vehicle applications incorporate further vehicle features such as sensor data. Typical examples of such applications are Cooperative Adaptive Cruise Control (CACC) or *platooning*.

Assume that you would like to analyze various aspects of CACC in more detail—quite soon, you would realize that a CACC system depends on two components: the *plain* Adaptive Cruise Control (ACC) component as well as the *connectivity* component. Integrating the communication aspects into your CACC application should be simple enough with Artery, as we have just shown in Sect. 12.2. So far, you do not know about the perception of other vehicles without communication yet, a key aspect when implementing the ACC component. In today's vehicles, this is usually achieved by mounting some type of local perception sensor, such as a radar, camera, or laser scanner on the vehicle. These sensors perceive the surrounding environment and processing software extracts information about detected objects, such as distance and speed of another vehicle.

Luckily, an extension to Artery provides a flexible architecture to equip vehicles with one or multiple local perception sensors. Since CACC and platooning applications are research topics on their own, we will opt for a simplified example to

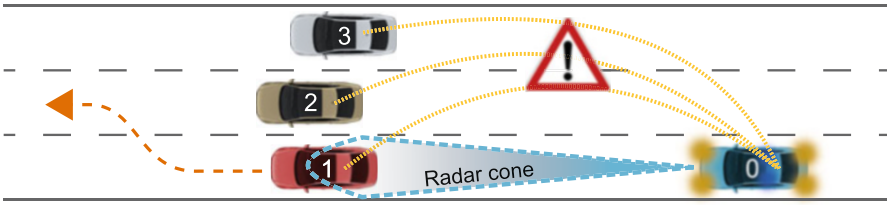



**Fig. 12.6** Extended simulation scenario: the police car (0) is equipped with a radar sensor for detecting vehicles ahead

demonstrate usage of perception sensors with Artery. We will hence introduce the components of the local perception sensors' architecture by adapting the police car's message rate depending on the presence of other vehicles in front of it.

Up to now, the `PoliceService` emits a warning message at each update interval, i.e., each `trigger` occurrence. This is an obvious waste of channel resources if no vehicle is in front of the police car. Hence, we can reduce resource usage by sending only at every fifth trigger in this case. Otherwise, when vehicles are detected in front, the police car still urgently demands for a clear lane like before. Though we could rely on CA messages to check for other vehicles' presence, we want to use a radar sensor mounted at the police car's front bumper as depicted in Fig. 12.6.

In the following subsections, we will guide you through the required extension to the `PoliceService` to make use of local perception sensors. Section 12.3.1 briefly provides an overview on the various components that Artery provides to realize simulation of local perception sensors. We will then tie those components to the previously introduced `World` and `Car` modules in Sect. 12.3.2. It is then time to add a few more lines to the `PoliceService`, as explained in Sect. 12.2.3.1.

### *12.3.1 Perception Architecture*

At the core of Artery's *Local Perception Sensor* extension stands the idea of a so-called *Environment Model*. This model can be interpreted as a database for each vehicle, containing the actual information about detected objects. In the following, we refer to other detected traffic participants as *objects*.

You need to know three types of components that constitute the foundation of local perception for simulated vehicles:

1. *Global Environment Model* keeps track of all dynamic *objects* (vehicles) as well as static *obstacles* (e.g., buildings) on the whole map.
2. *Local Environment Model* tracks a subset of *objects* visible to the respective vehicle hosting this module.

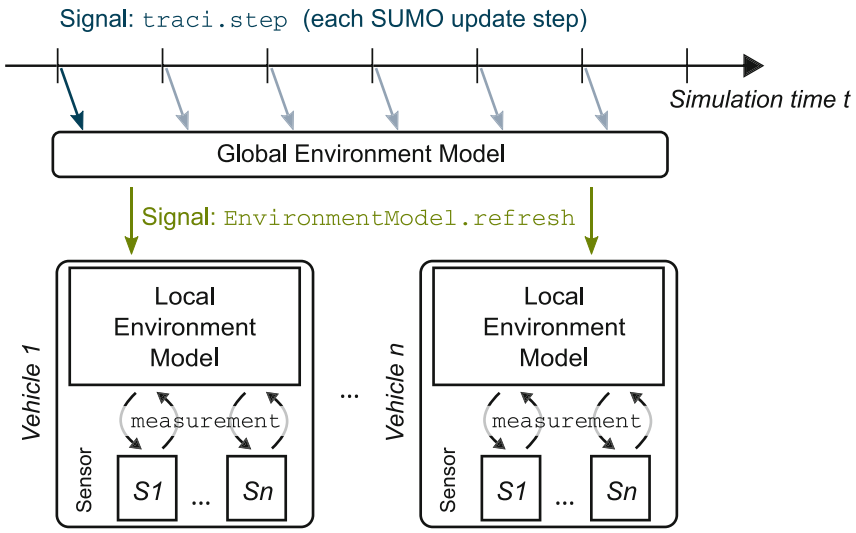



**Fig. 12.7** Interaction between Artery's perception components

3. *Sensor* modules define a vehicle's perception capabilities such as the observable field-of-view. Sensors are closely linked to their sibling *Local Environment Model*.

Figure 12.7 depicts the high-level interactions between those components. The following paragraphs provide a condensed overview on the internal workings of each. A detailed description is available in [6]. All sources of Artery's *Environment Model* are co-located in the *src/artery/envmod* directory. By setting the **CMake** build variable `WITH_ENVMOD` to `ON`, this directory will be included in the build process and thus enable the *Environment Model* extension.

#### 12.3.1.1 Local Perception Sensor

As an application developer, the local perception sensors are probably the most important modules that you need to be aware of. Depending on the configured sensor set for a particular vehicle, objects may be detected or not. Configuring the sensor sets is very similar to the configuration of services you already know from Sect. 12.2.1.2 and Listing 12.6. This sensor configuration is often stored in an XML file named *sensors.xml* along with *services.xml*. You can make use of the same filters to equip only a subset of vehicles with particular sensors as we have introduced in Listing 12.6. Additionally, each sensor's *visualization* options can be configured in this XML file. These options control if the sensor's cone, detected objects and obstacles, and corresponding line-of-sights shall be drawn in the OMNeT++ GUI. We provide a detailed step-by-step tutorial including configuration options in Sect. 12.3.2.

Each perception sensor is an OMNeT++ module derived from the C++ interface class `artery::Sensor`. This interface exhibits only a single setter method `setVisualization` which is called during initialization to pass the aforementioned *visualization* options. We recommend to use `artery::BaseSensor` as base class for your custom sensor class relieving you from dealing with this aspect. However, there are several *getter* methods you are required to implement defining the sensor's characteristics:

- `FieldOfView` defines the sensor's cone in terms of range and opening angle.
- `SensorPosition` specifies the logical mounting position relative to the vehicle's body. The sensor's field-of-view will be moved and rotated by the *Environment Model* according to the host vehicle's outline, position, and heading.
- A validity period determining the maximum time an object is still considered to be detected even when it is currently shadowed by another object/obstacle or has left the sensor's field of view.
- A string categorizing the sensor type which may be used for grouping and filtering sensors, e.g., all types of radar sensors return a common "Radar" category.

While these *getters* describe a sensor's properties, its behavior is realized by its `measurement` method. When vehicles have moved a SUMO step onward, this method gets called for each sensor (see Fig. 12.7 and Sect. 12.3.1.3). It is then this method's duty to check which objects this sensor can perceive currently. Each sensor reports its list of perceived objects and obstacles to the vehicle's Local Environment Model (LEM) (see Sect. 12.3.1.2).

Artery comes with a set of predefined sensors. A `RadarSensor` class is used as base of a front and rear-radar sensor, respectively. They differ only in their `SensorPosition`. From a functional perspective, one may treat information received via V2X communications also as objects. Hence, the `CamSensor` utilizes received CA messages and adds the transmitting vehicles to the list of perceived objects. For such purposes, the *Environment Model* manages several identifiers for each object in a central `IdentityRegistry` module. While radar sensors use the SUMO vehicle name to distinguish objects, this SUMO identifier is unknown on reception of a CAM and vice versa. Still, an object's unique `Identity` can be looked up via this registry in an easy fashion.

#### 12.3.1.2 Local Environment Model

The LEM is the single point of contact for your application with Artery's perception components. Whenever object information is available, the vehicle's perception sensors (see Sect. 12.3.1.1) report updated measurements which are stored and fused in the LEM. Note that data fusion is performed by associating objects' identifiers such as SUMO names, ITS station IDs, and network addresses. As such, the LEM represents a database within each vehicle, tracking the currently detected objects in so-called object lists. These object lists refer to the actual

`EnvironmentModelObject` via a `std::weak_ptr`, i.e., there is only one common object representing a particular vehicle. All these objects are managed by the Global Environment Model (GEM) and LEMs just use this pointer type to keep resource usage low.

During initialization, the LEM looks up the GEM instance, registers it and itself at the middleware's *Facilities*. In a second step, it creates the local perception sensors for this vehicle according to the sensor XML configuration, e.g., *sensors.xml*. Due to the flexible *Facilities* there is no need to "hardwire" any *Environment Model* components. Instead, an Artery *Service* can look these up via *Facilities* and access them when required. We will make use of the object list provided by the LEM in Sect. 12.2.3.1.

Though `EnvironmentModelObjects` and thus commonly observable object properties (outline, position, etc.) are shared by GEM and LEMs, each LEM maintains individual tracking information. These tracking data comprise the first and last time point this object has been detected by a specific local perception sensor. Thus, an application may, for example, only consider objects tracked by at least two distinct sensors further on. An object is removed from LEM when it has not been detected by any sensor for the respective sensor's validity period.

Object detection is performed each time SUMO updates the vehicles' state. For this purpose, each LEM subscribes to the `EnvironmentModel.refresh` signal emitted by the GEM. When this signal occurs, LEMs iterate over all their local sensors and invoke the respective `measurement` method, as depicted in Fig. 12.7. Consequently, the object states in each LEM are updated by the accompanying sensors. Regardless of the vehicle and sensor type, the list of detected objects is generated by consulting the GEM.

#### 12.3.1.3 Global Environment Model

The backbone of Artery's perception architecture is the GEM. What the LEM is to an application developer, the GEM is to the LEM: the single-point of contact of each vehicle's environment model. Being a global component within the simulation model, it reacts upon `traci.step` OMNeT++ signals which are emitted whenever SUMO performs a simulation step. The rationale behind monitoring SUMO updates is that vehicle (object) states only need to be updated whenever the traffic simulation provides new data. Between two consecutive SUMO steps the vehicle's physical state does not change at all and the sensor's environment therefore does not change either.

The GEM maintains the global view on all (potentially moving) objects and static obstacles such as buildings. These obstacles are retrieved once from the SUMO map as soon as the TraCI connection is established. Though obstacles are never included in object lists, they may obstruct line-of-sights between sensors and objects, e.g., vehicles at a closed intersection.

While obstacles are readily returned as polygons by SUMO, the position of each vehicle corresponds just to the middle of its front bumper. Consequently, the GEM has to create and maintain a geometric representation of each vehicle. This geometric representation is the vehicle's outline depending on the vehicle's length, width, position, and orientation. Since positions and orientations may change after every SUMO update, the objects need to get updated accordingly in GEM's database. An updated database is then signaled by `EnvironmentModel.refresh`. This signal triggers all vehicle sensors to perform measurements again, as depicted in Fig. 12.7.

Whenever a sensor mounted to a vehicle performs a measurement, it queries the GEM for all objects within the sensors configured field-of-view. The GEM then performs line-of-sight checks for these objects eliminating objects from the result set shadowed by another vehicle or an obstacle between the sensor's origin and this object. It should be noted that the GEM is a construct that can only exist in a simulation environment. However, it allows for a more efficient, centralized representation of objects. In summary, the GEM maintains the physical state of the whole map whereas LEMs add the local "perspective" on this state.

#### 12.3.1.4 Ideas for Customizations

Before diving into the implementation of local perception sensors, let us take a moment to review the concept. As stated above, local perception sensors in Artery can be attached to each vehicle, specifying a mounting position and a sensor type. Multiple sensors can be added to vehicles, serving applications relying on side sensors such as blind spot detection and alike. Each sensor contributes information to an LEM, a database representing an abstract enumeration of detected objects in each vehicle's vicinity. Even object information received via V2X communication can be added to the environment model by special sensor types. Within the simulation environment, the central information base is the GEM, acting as the backbone of the perception system. The GEM maintains a database containing a geometric representation of all vehicles currently in the simulation. At each simulation step, the database is updated and provides current line-of-sight information for each sensor to every vehicle. This speeds up the calculation process, as a central instance calculates current visibility conditions, rather than every instance of an LEM.

Nevertheless, Artery's perception system is based on several simplifying assumptions, leaving ample space for further extensions and customization.

**Prediction** Artery does currently not provide object prediction mechanisms. Instead, object information are only updated at each SUMO simulation step. This is, in most cases, a reasonable simplification facing a microscopic traffic simulation, where vehicle states do not change between simulation steps. However, if messages including sensor objects are going to be transmitted at a considerably higher rate than SUMO updates occur, it may become necessary to predict intermediate object changes.

**Noise** Positions in Artery (and SUMO) are always "perfect," i.e., as accurate as numerically representable. Hence, the current sensor implementations provide *exact* actual object information, as all state variables can be computed from simulation ground-truth data. In reality, every sensor is subjected to measurement errors, leading to inaccuracies that have to be accounted for. Due to the modular architecture of the environment model, sensor noise could be integrated into the sensor model itself.

**Fusion** Combining several measurements and even sensors, the area of *sensor fusion* becomes even more relevant [8, 9]. Artery provides a very simplified fusion mechanism, associating measurements based on know vehicle IDs. In reality, vehicles do not exhibit a unique ID that can be measured by a perception sensor—opening the field of data fusion based on object features and predictions. Intentionally ignoring those IDs in favor of physical properties such as vehicle dimensions, one may investigate sensor fusion algorithms using Artery.

### *12.3.2 Extending the Model to Use Local Perception Sensors*

Let us now use local perception sensors in our extended scenario depicted in Fig. 12.6. We want to adapt the rate at which `PoliceClearLane` messages are disseminated: the highest rate when the police car detects other vehicles ahead, a reduced rate otherwise. Using Artery's perception capabilities requires a few changes to the existing scenario `artery.inet.World` which we used before (see Listing 12.18). Inheriting the original scenario, we have to add the `GlobalEnvironmentModel` as well as the `IdentityRegistry` which is the central database behind the perception system. Vehicles register their identifiers, i.e., their SUMO name, ETSI ITS station ID, and GeoNetworking address, so it becomes easy to determine if you are dealing with the same vehicle though you know only one of its identifiers.

**Listing 12.18** Updated *World.ned* file for Artery's perception components

```
1  package artery.envmod;
2
3  import artery.envmod.GlobalEnvironmentModel;
4  import artery.envmod.LocalEnvironmentModel;
5  import artery.inet.World;
6  import artery.utility.IdentityRegistry;
7
8  network World extends artery.inet.World
9  {
10    parameters:
11       **.globalEnvironmentModule = default("environmentModel");
12
13    submodules:
14       environmentModel: GlobalEnvironmentModel {
15          parameters: @display("p=140,20");
16       }
17
18       idRegistry: IdentityRegistry {
```

```
19          parameters: @display("p=180,20");
20      }
21 }
```

Gathering local sensor data is achieved by the `LocalEnvironmentModel`, added to each vehicle node. You can either use the bundled `artery.envmod.Car` or derive your own vehicle module type similar to Listing 12.19. Just make sure your custom vehicle also possesses a middleware module, which is a prerequisite.

**Listing 12.19** Updated *Car.ned* file for Artery's perception components

```
1 package artery.envmod;
2
3 import artery.envmod.LocalEnvironmentModel;
4 import artery.inet.Car;
5
6 module Car extends artery.inet.Car
7 {
8    submodules:
9       environmentModel: LocalEnvironmentModel {
10         @display("p=214,57");
11         middlewareModule = default(".middleware");
12      }
13 }
```

In the next step, we need to define that only the police vehicle should be equipped with a front-facing radar sensor. Create a *sensors.xml* file like the one in Listing 12.20. Notice how we are using the same filter as used in *services.xml*. The visualization parameters are optional and may be interesting in case you would like to visualize the perception sensors which may be helpful for debugging. Figure 12.8 shows the drawn sensor cones, line-of-sights, and detected objects in the respective cones influenced by these settings.

**Listing 12.20** Sensor configuration (*sensors.xml*)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <sensors>
3    <sensor type="artery.envmod.sensor.FrontRadar">
4       <visualization>
5          <showObjectsInSensorRange value="true" />
6          <showObstaclesInSensorRange value="false" />
7          <showSensorCone value="true" />
8          <showLineOfSight value="true" />
9       </visualization>
10   </sensor>
11 </sensors>
```

Let us review the changes that we need to make to the simulation configuration. We amend the original *omnetpp.ini* from Listing 12.7 by an additional configuration section `envmod` as given by Listing 12.21. For the most part, we simply put the previously created customizations into place. The last two lines configure the shape of the police car's radar sensor to approximate a long-range radar with a narrow opening angle of 20°. Since it is a front radar, any other vehicle within ±10° relative to the police car's longitudinal axis and not farther than 200 m can be detected.

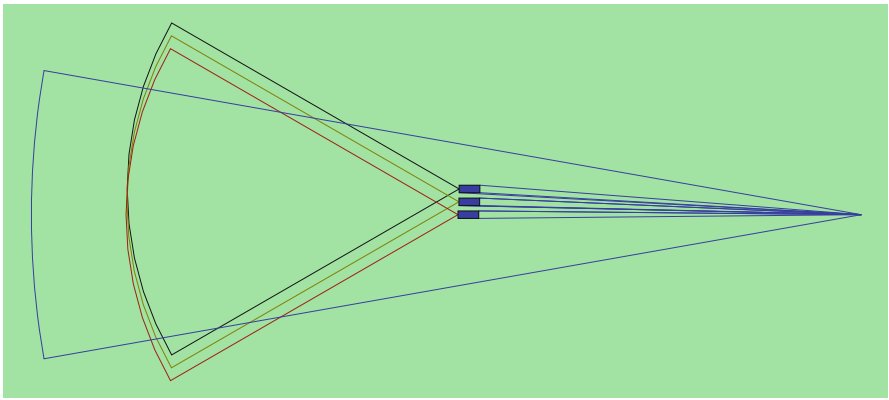


**Fig. 12.8** Canvas of `GlobalEnvironmentModel` with line-of-sight checks

**Listing 12.21** OMNeT++ configuration (extension of Listing 12.7)

```
1 [Config envmod]
2 network = artery.envmod.World
3 *.traci.mapper.vehicleType = "artery.envmod.Car"
4 *.node[*].middleware.services = xmldoc("services-envmod.xml")
5 *.node[*].environmentModel.sensors = xmldoc("sensors.xml")
6 *.node[4].environmentModel.FrontRadar.fovRange = 200.0 m
7 *.node[4].environmentModel.FrontRadar.fovAngle = 20.0
```

We updated the simulation network to the updated `artery.envmod.World` from Listing 12.18. Also notice that instead of using Artery's default car, we are now using the extended car type `artery.envmod.Car` which extends the default car with the LEM. The sensor configuration *sensors.xml* is provided to the `environmentModel` module and we also prepared our updated configuration to load an alternative set of Artery services, listed in *services-envmod.xml*. This file is identical to Listing 12.6 except it refers to `PoliceServiceEnvmod` instead of the plain `PoliceService`.

### *12.3.3 Extending the PoliceService Application*

Only a few minor changes to the original `PoliceService` are necessary to emit `ClearLaneMessages` at a high message rate only if another vehicle has been spotted. Obviously, we need to fetch a reference to the LEM during initialization (line 11 in Listing 12.22). This reference is then used when evaluating the service's `trigger` invocation: If the police car's environment model knows about at least one object from its sensors, it will call the base class' `PoliceService::trigger`. Otherwise, the base class' trigger is only called every fifth time. This is controlled by the `skippedTrigger` variable counting how often our customized `trigger` did not invoke the base class.

**Listing 12.22** `PoliceServiceEnvmod` implementation

```
#include "PoliceServiceEnvmod.h"
#include "artery/envmod/LocalEnvironmentModel.h"
using namespace omnetpp;
Define_Module(PoliceServiceEnvmod)

void PoliceServiceEnvmod::initialize()
{
   PoliceService::initialize();
   localEnvmod = &getFacilities().get_const<artery::LocalEnvironmentModel>();
   skippedTrigger = 0;
}

void PoliceServiceEnvmod::trigger()
{
   Enter_Method("PoliceServiceEnvmod trigger");
   const auto& objects = localEnvmod->allObjects();
   if (objects.size() > 0 || skippedTrigger >= 4) {
      PoliceService::trigger();
      skippedTrigger = 0;
   } else {
      ++skippedTrigger;
   }
}
```

## 12.4 Scripting for Dynamically Evolving Scenarios

Besides the regular information exchange between vehicles, which is achieved by sending CA messages and beacons, ITS aims for distributing information about various incidents. Therefore, there is a specific message type called DEN Message (DENM) that is generated when such an event was spotted by a vehicle.

Various types of incidents exist, like dangerous weather conditions or accidents, which should be distributed upon detection. In a common traffic scenario simulated by SUMO those events either do not exist or are not modeled easily. Modeling them directly in C++ code is not favorable either as those DEN use cases tend to vary a lot (comparable to SUMO scenarios). Hence, Artery provides a Python-based scripting interface called *Storyboard* for modeling those traffic situations.

### *12.4.1 Storyboard Concept*

To model a specific traffic situation (hereafter called *Story*) two main components are involved as presented in Fig. 12.9. On the one hand, there are *Triggering Conditions*. They define when and on which car a certain story becomes active. On the other hand, there are *Effects* describing what happens after a story became active. The *Storyboard* evaluates the conditions for the simulated vehicles periodically and

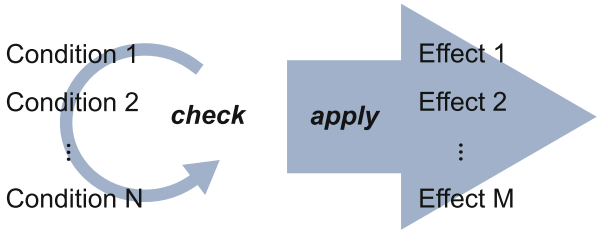



**Fig. 12.9** Storyboard mechanism

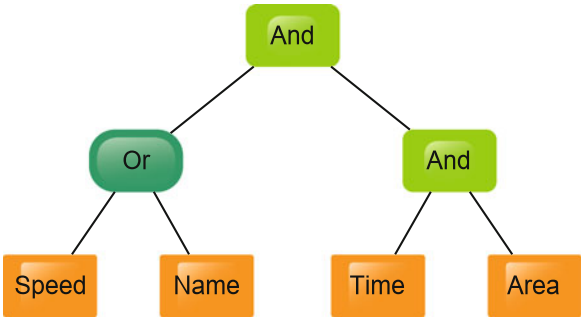



**Fig. 12.10** Condition tree with three layers

applies the effects accordingly. For example: if one wants to model an adverse bad weather condition in a specific area, one could define an *Area Condition* for the affected map region. In addition, a *Speed Effect* could be used as the associated story effect forcing all vehicles to slow down while driving in this area.

Since not every story is as straightforward as the aforementioned one, various triggering conditions as well as effects can be combined. Conditions can be linked using *And Conditions* as well as *Or Conditions* resulting in an arbitrary deep condition tree. Figure 12.10 shows an example of a condition tree created by combining four conditions using *And Conditions* and an *Or Condition*.

In this particular example, effects are applied to a vehicle when it enters a dedicated area (*Area*) within a defined time span (*Time*) but only if it has a certain SUMO identifier (*Name*) or exceeds a specific velocity (*Speed*). After the condition becomes true, various effects can be applied ranging from speed changes to emission of OMNeT++ signals. A full list of bundled conditions and effects is presented in Tables 12.1 and 12.2. Extending those is as easy as sub-classing the respective base class and adding a small Python binding to *storyboard/Binding.cc*.

**Table 12.1** List of available triggering conditions

| Condition | Description |
|---|---|
| AndCondition | Combines two conditions using the logic operator `AND` |
| OrCondition | Combines two conditions using the logic operator `OR` |
| CarSetCondition | Selects a subset of all cars based on SUMO names |
| LimitCondition | Evaluates to *true* only for the first N vehicles. Because of lazy evaluation of condition trees, the first N-evaluated vehicles are not the first N-simulated vehicles necessarily |
| PolygonCondition | Checks if vehicle positions are inside a polygon |
| SpeedConditionGreater, SpeedConditionLess | Becomes *true* if a vehicle drives faster or slower than the defined velocity |
| SpeedDifferenceConditionFaster, SpeedDifferenceConditionSlower | Triggers if a speed difference between vehicles is exceeded. *Effects* can access the corresponding list of vehicles |
| TimeCondition | Used to specify the time in which the story is active |
| TtcCondition | Becomes true if the TTC between two vehicle undercuts a limit (depends on speed and predicted path) |

**Table 12.2** List of available effects

| Effect | Description |
|---|---|
| SpeedEffect | Changes the speed of affected vehicles while the story is active |
| StopEffect | Stops the vehicle entirely while the story is active |
| SignalEffect | Emits an OMNeT++ signal (StoryboardSignal) at the vehicle's middleware. `StoryboardSignal` transports the defined string |

### *12.4.2 Dynamic Activation of Police Service*

In Sect. 12.2, we have created a rather static scenario: while the vehicles are changing lanes quite dynamically, the external circumstances are rather static because the driver's intentions are fixed. To put it straight, the police car is always in hurry though even the most courageous policeman will not be on a mission all the time. So, let us leverage the *Storyboard* to add some salt to this scenario: instead of sending out *PoliceClearLane* messages right from the start, we assume the police car receives a call during the simulation which requires it to move faster.

Like mentioned before, stories can be created using a Python script that is loaded and executed by the OMNeT++ `Storyboard` module. Listing 12.23 is one proposal to virtually turn on the police car's siren after 10 s. Two conditions are combined via an `AndCondition` so only if both are `true` the story's `signalEffect` is applied. The former `timeCondition` limits the story to the time window starting at OMNeT++ simulation time 10 s. The latter `carSetCondition` selects one particular vehicle, the `police0`. Consequently, the `SignalEffect` emits a `StoryboardSignal` at $t = 10$ s in `police0`'s middleware to which a modified `PoliceService` can be subscribed.

**Listing 12.23** Example storyboard script from the *story.py* file

```
import storyboard
import timeline

def createStories(board):
   # condition triggering after 10 simulated seconds
   timeCondition = storyboard.TimeCondition(timeline.seconds(10))
   # select police car
   carSetCondition = storyboard.CarSetCondition("police0")
   # create signal effect
   signalEffect = storyboard.SignalEffect("siren on")
   # combine conditions
   condition = storyboard.AndCondition(timeCondition, carSetCondition)
   # create story by linking effect and conditions together
   story = storyboard.Story(condition, [signalEffect])
   # register story at storyboard
   board.registerStory(story)
```

The modifications to `PoliceService` are minor and can be implemented in the original source code. Just to keep the original code untouched, we created a derived `PoliceServiceStoryboard` module which highlights the changes related to the *Storyboard*. As Listing 12.24 shows, the original `initialize` and `trigger` methods from the `PoliceService` base class are reused. During initialization the extended service subscribes to the `StoryboardSignal` and disables its siren as indicated by its member attribute `activatedSiren`. Only if the `activatedSiren` attribute is `true` the original `PoliceService::trigger` is invoked to transmit the recurring `PoliceClearLane` message. This internal flag turns `true` as soon as the matching *Storyboard* signal is received by the `receiveSignal` method (see `omnetpp::cListener`). To ensure that we need to turn on the siren we check that the received signal is (1) a signal named `"StoryboardSignal"`, (2) of type `StoryboardSignal`, and (3) contains the `"siren on"` string from Listing 12.23.

**Listing 12.24** Police Service prepared for Storyboard interaction

```
#include "PoliceServiceStoryboard.h"
#include <artery/application/StoryboardSignal.h>
using namespace omnetpp;

// register signal to receive signal from Storyboard
static const simsignal_t storyboardSignal = cComponent::registerSignal("
      StoryboardSignal");

Define_Module(PoliceServiceStoryboard)

void PoliceServiceStoryboard::initialize()
{
   PoliceService::initialize();
   subscribe(storyboardSignal);
   activatedSiren = false;
}
```

```
16
17 void PoliceServiceStoryboard::trigger()
18 {
19    Enter_Method("PoliceServiceStoryboard trigger");
20    if (activatedSiren) {
21       PoliceService::trigger();
22    }
23 }
24
25 void PoliceServiceStoryboard::receiveSignal(omnetpp::cComponent*, omnetpp::
       simsignal_t sig, omnetpp::cObject* sigobj, omnetpp::cObject*)
26 {
27    // start the PoliceService after receiving Storyboard Signal
28    if (sig == storyboardSignal) {
29       auto storysig = dynamic_cast<StoryboardSignal*>(sigobj);
30       if (storysig && storysig->getCause() == "siren on") {
31          activatedSiren = true;
32       }
33    }
34 }
```

We are almost done with our accommodations. Do not forget to mention your new source file *PoliceServiceStoryboard.cc* in *CMakeLists.txt* along with the sources from Sect. 12.2.4. Furthermore, the **CMake** variable `WITH_STORYBOARD` needs to be set to "ON" in your build directory so the optional *Storyboard* feature is enabled. The *omnetpp.ini* also needs to be slightly adjusted by adding a `Config` section, listed in Listing 12.25. Those changes boil down to activating the optional *Storyboard* module in `artery.inet.World` by setting its `withStoryboard` parameter, telling this module the name of the desired Python script (*story.py* without extension), and a changed *services* deployment. The referred *service-storyboard.xml* differs from Listing 12.6's *services.xml* only in the service equipped on the police car—`PoliceServiceStoryboard` instead of plain `PoliceService`.

**Listing 12.25** Extended OMNeT++ configuration

```
1 [Config storyboard]
2 *.withStoryboard = true
3 *.storyboard.python = "story"
4 *.node[*].middleware.services = xmldoc("services-storyboard.xml")
```

When you run the simulation setup again using the *Config storyboard*, you will notice that the simulation will take a few seconds longer because the passenger cars will not immediately give way as soon as the police car is present.

It is also possible to overlap stories in time and place. The storyboard maintains the order of all stories and takes care of successfully restoring all changed vehicle parameters (like the speed of a vehicle). Also, the storyboard is very modular to allow for an easy creation of new conditions and effects. For example, you can create a new effect called *FogLightEffect* issuing a TraCI command to turn on the fog light. Just derive a new class from *Storyboard*'s `Effect` class, implement its abstract methods, and add two lines of binding to the *Binding.cc* file.

## 12.5 Outlook

Though there is much more to discover in Artery, a chapter like this cannot cover all features. However, we want to refer to Chap. 13 that demonstrates how to combine two OMNeT++ frameworks, SimuLTE and Artery, to a joint cellular V2X simulation.

What we did not discuss is how to operate Artery in testbeds. In fact, Artery in its present state already supports two distinct variants to link its simulation environment with external components. On the one hand, ITS functions can be attached to simulated vehicles' middleware by means of the *Transfusion Service*. V2X messages can be forwarded back and forth to remote software via TCP/IP socket communication. If you are interested in this application layer coupling, please have a look at `artery.transfusion.TransfusionService`. On the other hand, you can also run a vehicle's whole software stack remotely. Packets can be intercepted at the link-layer and forwarded to another device, e.g., the device under test in a HIL setup. In this case, Artery provides the simulated communicating environment for a real V2X unit. In either case, a custom OMNeT++ scheduler implementation realizes the real-time behavior and facilitates communicating to external components.

We hope you enjoyed this introduction to Artery and we welcome your feedback and contributions regarding Artery.

## Appendix

### *Running, Debugging, and Finding Memory Leaks*

Artery provides debug and memory check targets for every scenario integrated via the **add_opp_run** macro. Those targets are prefixed by `run_`, `debug_`, and `memcheck_` before the target name is passed to the macro.

**Debugging** Debug targets (`debug_*`) are only available when Artery is built in debug mode, i.e., `CMAKE_BUILD_TYPE` is set to `Debug`.

| | |
|---|---|
| `GDB_COMMAND` | Path to the GNU Debugger, usually found automatically if installed system-wide. |
| `RUN_FLAGS` | These flags are appended to the **opp_run** invocation and shared by `run_*` and `debug_*` targets. |

**Valgrind** Three **CMake** options influence the behavior of memory check runs (`memcheck_*` targets):

| | |
|---|---|
| `VALGRIND_COMMAND` | Path of the **Valgrind** executable, usually found automatically if installed system-wide. |

| | |
|---|---|
| `VALGRIND_FLAGS` | Options passed to **Valgrind** itself, e.g., enabling tracking down the source of a memory leak with `-track-origins=yes`. |
| `VALGRIND_EXEC_FLAGS` | Flags added to the **opp_run** invocation. This is useful to run the simulation with the faster command-line interface (`-u Cmdenv`) and to select a specific configuration (`-c my_opp_config`). These flags are in addition to the `RUN_FLAGS` for ordinary simulation runs. |

### *Common Pitfall: Wrong Longitude and Latitude*

Artery uses TraCI to convert Cartesian coordinates to geodetic ones. SUMO needs to be built with the enabled *PROJ* feature for this to work. You can query your local SUMO executable by calling **sumo** with the option `--version` and check the second line starting with "Build features:". Alternatively, open the "About SUMO" dialog in the *Help*→*About* menu using the SUMO GUI. Furthermore, your SUMO map needs to define a map projection. When you are using a map imported from OpenStreetMap there is already a proper map projection defined. However, synthetic maps such as those generated by SUMO's **netgenerate** define no map projection. You can easily fix your map by editing the `*.net.xml` file: look for the `<location ...projParameter="!" />` tag and replace the empty `!` projection by a more suitable projection, for instance, `+proj=tmerc +ellps=WGS84 +datum=WGS84 +units=m +no_defs` as a minimal working projection. You can also shift the map origin with additional parameters (e.g., `+lat_0=49 +lon_0=11`).

## References

1. ETSI: Intelligent Transport Systems (ITS); Communications Architecture. European standard, ETSI (2010). V1.1.1
2. ETSI: Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service. European standard, ETSI (2014). V1.3.2
3. ETSI: Intelligent Transport Systems (ITS); GeoNetworking; Port Numbers for the Basic Transport Protocol (BTP). Technical specification, ETSI (2016). V1.1.1
4. ETSI: Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 4: Geographical addressing and forwarding for point-to-point and point-to-multipoint communications; Sub-part 1: Media-Independent Functionality. European standard, ETSI (2017). V1.3.1
5. ETSI: Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 5: Transport Protocols; Sub-part 1: Basic Transport Protocol. European standard, ETSI (2017). V2.1.1

6. Günther, H.J., Timpner, J., Wegner, M., Riebl, R., Wolf, L.: Extending a holistic microscopic IVC simulation environment with local perception sensors and LTE capabilities. Vehicular Communications (2017). http://dx.doi.org/10.1016/j.vehcom.2017.01.003. http://www.sciencedirect.com/science/article/pii/S2214209616300638
7. IEEE Computer Society: Telecommunications and information exchange between systems; Local and metropolitan area networks; Specific requirements; Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE Standard for Information technology, IEEE Standards Association (2016)
8. Khaleghi, B., Khamis, A., Karray, F.O., Razavi, S.N.: Multisensor data fusion: a review of the state-of-the-art. Inf. Fusion **14**(1), 28–44 (2013)
9. Matzka, S., Altendorfer, R.: A comparison of track-to-track fusion algorithms for automotive sensor fusion. In: Multisensor Fusion and Integration for Intelligent Systems, pp. 69–81. Springer, Cham (2009)
10. Riebl, R., Facchi, C.: Regain control of growing dependencies in OMNeT++ simulations. In: Proceedings of the 2nd OMNeT++ Community Summit (2015). http://arxiv.org/abs/1509.03561
11. Riebl, R., Günther, H.J., Facchi, C., Wolf, L.: Artery - extendig Veins for VANET applications. In: Models and Technologies for Intelligent Transportation Systems (MT-ITS) (2015)
12. Riebl, R., Obermaier, C., Neumeier, S., Facchi, C.: Vanetza: Boosting research on inter-vehicle communication. In: Proceedings of the 5th GI/ITG KuVS Fachgespräch Inter-Vehicle Communication, pp. 37–40 (2017). https://opus4.kobv.de/opus4-fau/files/8528/fg-ivc-2017-report.pdf

# Chapter 13
# Simulating LTE-Enabled Vehicular Communications


**Raphael Riebl, Giovanni Nardini, and Antonio Virdis**

## 13.1 Introduction

Modern inter-vehicle communication systems are labeled with various names, including Dedicated Short-Range Communication (DSRC), Vehicle-to-Everything (V2X), Vehicle-to-Vehicle (V2V), and Vehicle-to-Infrastructure (V2I), just to list a few of them. Terms like V2X and its siblings refer to the communicating parties, e.g., vehicles with other vehicles or traffic infrastructure or something else, and are principally independent of the underlying radio technology. Some communication standards have outlined such a heterogeneous radio approach for Intelligent Transportation Systems (ITS) since many years [1]. DSRC, ITS G5, or Cellular V2X (C-V2X) then represent a particular realization of V2X communication. Some of them are using Wireless Local Area Network (WLAN) technology (e.g., DSRC and ITS G5), others use cellular networks (e.g., Long Term Evolution (LTE) and 5G) or cellular communication modes not strictly requiring any base stations (e.g., C-V2X). On the one hand, simulation of WLAN-based vehicular communication is covered in detail in Chaps. 6 and 12. On the other hand, simulation of cellular communication in general is presented in Chap. 5. This section merges those topics into a combined simulation model, exploiting cellular communication in the context of vehicular communication, using both infrastructure (evolved Node B (eNB)-relayed) and direct (UE-to-UE) communications, which exploit LTE's Device-to-Device (D2D) capabilities. Ideally, those wireless communications converge to a

R. Riebl (✉)
Technische Hochschule Ingolstadt, Ingolstadt, Germany
e-mail: raphael.riebl@thi.de

G. Nardini · A. Virdis
University of Pisa, Pisa, Italy
e-mail: g.nardini@ing.unipi.it; antonio.virdis@unipi.it



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_13

hybrid system leveraging the specific benefits of both. For this purpose, we will use *Artery* to model aspects related to vehicular communications and *SimuLTE* for cellular ones. The simulation model presented in this chapter is distinct from previous attempts combining V2X with cellular networks such as *Veins LTE* [3, 4] or *ArteryLTE* [2, 5, 6].

The rest of this contribution is organized as follows: in Sect. 13.2, we provide background on vehicular communication. Then, the subsequent sections describe two variants of a V2X application warning other road users about black ice: Sect. 13.3 describes LTE-enabled vehicular communication based on *infrastructured* transmissions and Sect. 13.4 introduces *D2D* communications in place of the former = infrastructured transmissions.

## 13.2 General Aspects

As presented in Chaps. 6 and 12, dedicated protocol suites exist for the communication between vehicles. In the United States, the proposed Dedicated Short-Range Communication (DSRC) system is based on Wireless Access in Vehicular Environments (WAVE) and the European counterpart is based on ETSI specified Intelligent Transportation System (ITS) protocols. The two have in common that they are not using Internet Protocol (IP) communication as fundamental networking technique in contrast to the Internet. Instead, both are stand-alone Vehicular Ad Hoc Networks (VANETs) which require no supporting infrastructure. In contrast, users of cellular networks want to access Internet-based services. Consequently most data traffic in cellular networks is nowadays based on IP. Fortunately, nobody has to buy into VANET or cellular communication exclusively. Both technologies have their distinct advantages and disadvantages for particular application types.

With this in mind, we can have a closer look at the challenges when integrating two distinct OMNeT++ frameworks—SimuLTE and Artery—into a combined simulation setup. Section 13.2.1 highlights some considerations when simulating vehicles with SimuLTE, whereas Sect. 13.2.2 deals with multi-radio vehicles.

### *13.2.1 Management of Vehicles in SimuLTE*

Within the SimuLTE environment, vehicles are User Equipments (UEs) of the cellular network, which can communicate with a back-end server located in the Internet and with other vehicles via the cellular infrastructure or via D2D transmissions. From a pure communication perspective, this means that no specific operations are required to support V2X communications within the LTE protocol stack implemented by SimuLTE. However, one has to take care of a few aspects concerning node mobility and dynamic node creation/destruction. A well-known system to simulate vehicles' mobility is Simulation of Urban MObility (SUMO)

(see Chap. 12). SUMO allows vehicles to enter and leave the simulation at runtime, thus any framework exploiting it needs to handle the dynamic creation and deletion of modules.

Within SimuLTE, this affects the behavior of the `initialize` and `finish` functions of the UEs. In the former, UEs attach themselves to their serving eNB and register with the `Binder`. As described in Chap. 5, the latter is used to keep information about the nodes present in the whole cellular network. In order to support mobility provided by SUMO, the `initialize()` functions within the entire protocol stack of the LTE Network Interface Card (NIC) of the `Ue` module are organized so that they can be invoked at any time rather than only at the beginning of the simulation. Likewise, the `finish()` functions are implemented so as to perform all the operations related to the de-registration of the UE from both the `Binder` and the eNB (e.g., clearing all the buffers related to that UE).

As far as eNB-UE association is concerned, SimuLTE allows one to decide whether to select the serving eNB statically (i.e., from the configuration file) or dynamically (i.e., selecting the best eNB according to a given criterion during the initialization of the `Ue` module). The latter mode is useful in a multicell scenario where vehicles can appear at sparse locations on the simulation playground, hence the best-serving eNB may not be known a priori. To enable dynamic eNB selection, the `dynamicCellAssociation` parameter is provided in the Network Topology Description (NED) files of both the `eNodeB` and `Ue` modules. If this feature is enabled, UEs compute the Signal-to-Interference-plus-Noise-Ratio (SINR) perceived from every eNB in the network during the initialization phase and select the one with the highest value as serving eNB. Then, mobility across different eNBs is handled by the existing handover mechanism provided by SimuLTE.

Moreover, vehicles entering the simulation need to obtain an IP address to communicate via the LTE network. For this purpose, SimuLTE delegates the assignment of IP addresses to INET's `IPv4NetworkConfigurator` module. The latter assigns IPs only at the beginning of the simulation and it does not handle dynamically created modules. For this reason, the NED structure of a module representing a vehicle in the LTE network has to be endowed with INET's `HostAutoConfigurator` module, which assigns a valid IP address during the initialization stage of the newly created node.

When a vehicle leaves the simulation conversely, some packets in the network may still be destined to that vehicle, e.g., those packets transmitted by a back-end server just before the vehicle's module has been destroyed. Such packets need to be removed from the simulation to avoid abnormal behaviors or even faults. For this reason, the existence of a vehicle's LTE radio is checked as soon as a reference to it is required (e.g., at the physical layer when transmitting the packet over the air interface). This is accomplished by checking whether the vehicle is still registered to the `LteBinder`. If not, the packet is deleted from the simulation.

### 13.2.2 Attaching an LTE Radio to Vehicles in Artery

Vehicles in Artery are defined by means of the `artery.inet.Car` module. They are equipped only with WLAN NICs configured for operation in a VANET, i.e. without an IP stack. To support LTE communication, we extend the car module to include the LTE NIC card. Thus, a simple LTE-enabled vehicle is created as `artery.lte.Car`, by adding the following functions:

- `HostAutoConfigurator` for the assignment of IP addresses,
- Internet Protocol Version 4 (IPv4) routing table and network layer from INET,
- Configurable number of User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) applications similar to other INET hosts,
- An LTE NIC from SimuLTE.

In order to support the concurrent existence of LTE and WLAN interfaces in one network node, a slight SimuLTE modification is required. In fact, SimuLTE's physical layer looks for gates named `radioIn` at network nodes to deliver LTE frames. However, nodes adhering to INET's `INetworkNode` interface define a gate vector for the same purpose. We modify the code so that gates named `lteRadioIn` are searched for, hence enabling co-existence of both simulation models.

Although not further discussed for the remainder of this chapter, one can keep up deploying Artery services such as `artery.application.CaService` on those vehicles just as described in Chap. 12. Since LTE and ITS-G5 are using distinct frequency bands there is no mutual interference to be considered.

## 13.3 V2X Services Relying on a Centralized Back-End

*Connected services* are integrated in many of today's cars. These services offer convenience features such as over-the-air map updates for the car's navigation system or finding available parking spaces. Since 2018, every new vehicle model sold in the European Union needs to be equipped with *eCall*, an automatic emergency call system activated after accidents. In either case those services leverage the existing cellular network and work only when the respective vehicles are actually located within network coverage. Hence, they do not belong to the category of V2X services albeit they use wireless communication to be connected with remote entities.

Aforementioned connected services can usually tolerate temporary network outages. Map updates can be completed later on and are not time critical because the previous map version can still be used. Guidance to parking lots can continue even if the information "123 of 400 parking bays free" is already a few minutes old. Features like *eCall* are more critical due to safety aspect, however, only a small amount of data is transmitted sporadically. A temporary unavailability of connectivity in some areas is unfavorable for this use case but tolerated nowadays.

The communication architecture is similar, if the service's use case is intended to increase either driver comfort or safety. The vehicle establishes a data connection to a central entity via cellular communication. This central entity then serves required data (e.g., up-to-date maps) or triggers actions like alarming rescue forces.

The OMNeT++ ecosystem provides the tools required to model aforementioned use cases. First, SimuLTE enables one to create a cellular network to simulate cellular-based communication among UEs and eNBs. Second, INET is up to the task of modeling a subset of the Internet representing the wired communication from base stations to central servers. Finally, Artery provides the means to integrate a connected service within a simulated vehicle.

In this section we will extend Artery's vehicle module introduced in Chap. 12 using an LTE modem. This enables users to realize V2X services using direct communication based on 802.11 radios and Internet-based services using LTE concurrently. As an exemplary use case we show how to develop a black ice warning application: vehicles report loss of traction due to black ice to a central server application. Other vehicles query the server periodically for any reported warnings in the area they are driving. If the server application responds with at least two warnings for the given area, the asking vehicle will cut its speed by half.

### *13.3.1 Network Definition*

Extending Artery with LTE requires not only to include an `artery.lte.Car`, as we showed in Sect. 13.2.2, but also to add the LTE infrastructure in its `World` network, as we show in Fig. 13.1. On the right side we see the modules related to SimuLTE: two eNBs are connected via a gateway *pgw* and a *router* to a central server hosting the application's back-end, e.g., aggregated information gathered on the streets. The `channelControl` and `binder` modules are helper modules required for running SimuLTE, as discussed in Chap. 5. On the left side we find modules related to vehicles and vehicular communication: the `traci` module adds vehicles to the simulation at runtime, thus no vehicles are included in the `World` on its own (see Chap. 12 for more details). Furthermore, `radioMedium` is a mandatory module for wireless communication based on OMNeT++/INET and thus WLAN-based direct communication by Artery's vehicles. The `configurator` is also a global helper module provided by OMNeT++/INET for managing IP networks. Plain Artery simulations do not require this module if only V2X is considered, but our LTE-enabled vehicles will use IP for their data connections to the *server*.

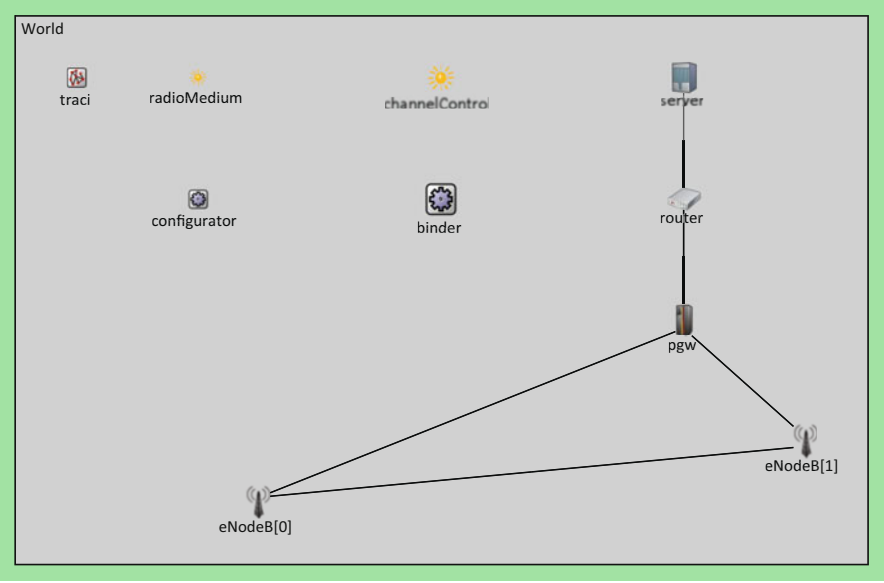



**Fig. 13.1** LTE world network `artery.lte.World`

### *13.3.2 Parameters Configuration*

Listing 13.1 highlights the main settings required to operate Artery with cellular communication. First, the network `artery.lte.World` is used as a starting point for a minimal working LTE network for use with Artery. This network is derived from `artery.inet.World`, an OMNeT++/INET network for V2X communication only. While `artery.lte.World` employs `artery.lte.Car` modules as vehicles already by default, users can override it by manipulating the parameter `traci.mapper.vehicleType`. For this example we stick with the LTE-enabled cars bundled with Artery in its *src/artery/lte* folder.

**Listing 13.1** Configuring Artery with SimuLTE extension (excerpt)

```
network = artery.lte.World

# This is also implicitly set by artery.lte.World
traci.mapper.vehicleType = "artery.lte.Car"

# LTE UEs (vehicles)
*.node[*].lteNic.dlFbGen.feedbackComputation = xmldoc("lte_channel.xml")
*.node[*].lteNic.phy.channelModel = xmldoc("lte_channel.xml")
*.node[*].lteNic.phy.dynamicCellAssociation = true
*.node[*].lteNic.phy.enableHandover = true
*.node[*].masterId = 1
*.node[*].macCellId = 1

# LTE base stations
*.eNodeB[*].lteNic.phy.feedbackComputation = xmldoc("lte_channel.xml")
```

```
16 *.eNodeB[*].lteNic.phy.channelModel = xmldoc("lte_channel.xml")
17 *.eNodeB[*].lteNic.phy.dynamicCellAssociation = true
18 *.eNodeB[*].lteNic.phy.enableHandover = true
19 *.eNodeB[*].cellInfo.broadcastMessageInterval = 0.5 s
20
21 # LTE applications of vehicles
22 *.node[*].numUdpApps = 2
23 *.node[*].udpApp[0].typename = "BlackIceReporter"
24 *.node[*].udpApp[1].typename = "BlackIceWarner"
25 *.node[*].udpApp[*].centralAddress = "server"
26 *.node[*].udpApp[*].middlewareModule = "^.middleware"
27
28 # Central back-end
29 *.server.numUdpApps = 1
30 *.server.udpApp[0].typename = "BlackIceCentral"
31
32 # Storyboard (black ice region)
33 *.withStoryboard = true
34 *.storyboard.python = "blackice"
```

The configuration of the LTE NICs is the same as for the one described for SimuLTE. As we anticipated in Sect. 13.2.1, the `dynamicCellAssociation` parameter is used to associate vehicles entering the network with the best-serving eNB, as shown in Listing 13.1 in lines 8–9 and 16–17. To manage the mobility of vehicles, handover is enabled by setting the parameter `enableHandover` for both the cars and the eNBs. The `broadcastMessageInterval` allows one to specify the rate used by eNBs to broadcast a message to devices under their control, and let them check whether a handover is due.

Regarding the application layer, any `UdpApps` can be instantiated on top of a vehicle's LTE NICs. In this use case, each vehicle is configured to run two applications: `BlackIceReporter` sends a warning to the network's central *server* when traction loss occurred and `BlackIceWarner` will periodically check if warnings apply to the vehicle's position. Furthermore, to allow interaction with Artery objects, which are accessible via its middleware, both applications are told where to look for this module, which is a sibling module to these UDP apps.

The central server runs a single UDP application `BlackIceCentral`, which stores black ice warnings reported by vehicles and respond to their queries. No direct communication between the vehicles happens in this case: warnings and queries are sent to the server via eNB in the Uplink (UL), whereas a response to a vehicle's query is sent down to a single receiver in the Downlink (DL).

To decide whether a vehicle is currently suffering from traction loss due to black ice, Artery's Storyboard feature is used, as we show in Listing 13.2. For each vehicle driving within the given rectangle faster than 16.78 m/s a signal will be emitted in its middleware. Applications like the `BlackIceReporter` can simply listen for this *traction loss* signal.

**Listing 13.2** Defining black ice region with storyboard

```
1 import storyboard as sb
2
3 def createStories(board):
4     region = sb.PolygonCondition([
5         sb.Coord(300, 300), sb.Coord(700, 300),
6         sb.Coord(700, 700), sb.Coord(300, 700)])
```

```
7    fast = sb.SpeedConditionGreater(16.67)
8    blackice = sb.AndCondition(region, fast)
9    story = sb.Story(blackice, [sb.SignalEffect("traction loss")])
10   board.registerStory(story)
```

### 13.3.3 Modifying the Code

Now that the basic configuration is in place, we will show how to implement a black ice application. Communication between vehicles and the central back-end relies on three packet types, which are listed in Listing 13.3. `BlackIceReporter` is going to send a `BlackIceReport` as soon as it detects a traction loss. This message contains the current position, speed and time, and is one-way only, i.e., the back-end does not answer to it but silently adds the received information to its database. `BlackIceQuery` and `BlackIceResponse` instead are used to build a simple query-response communication: vehicles ask for active black ice warnings in their region defined by the circle in `BlackIceQuery`. The back-end simply responds with the number of applicable warnings for the received query.

**Listing 13.3** BlackIce packet types

```
1  packet BlackIceReport
2  {
3     double positionX;
4     double positionY;
5     double speed;
6     simtime_t time;
7  }
8
9  packet BlackIceQuery
10 {
11    double positionX;
12    double positionY;
13    double radius;
14 }
15
16 packet BlackIceResponse
17 {
18    int warnings;
19 }
```

Listing 13.4 shows a snippet from the `BlackIceReporter` application. During initialization, an `inet::UDPSocket` is connected to the address and port of the central back-end. When the subscribed `StoryboardSignal` is received containing "traction loss," a `BlackIceReport` is created in `sendReport()`. This message is filled using data provided by Artery's `VehicleController` and finally sent over the previously established socket connection.

**Listing 13.4** Excerpt from BlackIceReporter

```
1  static const simsignal_t storyboardSignal =
2     cComponent::registerSignal("StoryboardSignal");
3
4  void BlackIceReporter::initialize()
5  {
```

```
6     socket.setOutputGate(gate("udpOut"));
7     auto centralAddress = inet::L3AddressResolver().resolve(
8        par("centralAddress"));
9     socket.connect(centralAddress, par("centralPort"));
10
11    auto mw = inet::getModuleFromPar<artery::Middleware>(
12       par("middlewareModule"), this);
13    mw->subscribe(storyboardSignal, this);
14    vehicleController = mw->getFacilities().
15       get_const_ptr<traci::VehicleController>();
16 }
17
18 void BlackIceReporter::receiveSignal(cComponent*, simsignal_t sig,
19       cObject* obj, cObject*)
20 {
21    if (sig == storyboardSignal) {
22       auto sigobj = check_and_cast<StoryboardSignal*>(obj);
23       if (sigobj->getCause() == "traction loss") {
24          sendReport();
25       }
26    }
27 }
28
29 void BlackIceReporter::sendReport()
30 {
31    Enter_Method_Silent();
32    using boost::units::si::meter;
33    using boost::units::si::meter_per_second;
34    auto report = new BlackIceReport("reporting black ice");
35    report->setPositionX(vehicleController->getPosition().x / meter);
36    report->setPositionY(vehicleController->getPosition().y / meter);
37    report->setSpeed(vehicleController->getSpeed() / meter_per_second);
38    report->setTime(simTime());
39    socket.send(report);
40 }
```

`BlackIceCentral` binds to two UDP sockets to the respective port numbers, one for receiving reports and the other for receiving queries. The `handleMessage` function hands received UDP packets over to `processPacket` of Listing 13.5. This method dispatches the packets according to the port number either to the `processReport` or to the `processQuery` functions. The former simply stores a copy of the received `BlackIceReport` for future reference, e.g., when a query needs to be answered. Consequently, `processQuery` iterates over all stored reports and counts all known reports for the queried region. This number is then filled into a `BlackIceResponse` and sent back to the querying vehicle.

**Listing 13.5** Excerpt from BlackIceCentral

```
1  void BlackIceCentral::processPacket(cPacket* pkt)
2  {
3     auto ctrl = pkt->getControlInfo();
4     if (auto udp = dynamic_cast<inet::UDPDataIndication*>(ctrl)) {
5        if (udp->getDestPort() == reportPort) {
6           processReport(*check_and_cast<BlackIceReport*>(pkt));
7        } else if (udp->getDestPort() == queryPort) {
8           processQuery(*check_and_cast<BlackIceQuery*>(pkt),
9              udp->getSrcAddr(), udp->getSrcPort());
10       } else {
11          throw cRuntimeError("Unknown UDP destination port %d",
12             udp->getDestPort());
13       }
14    }
```

```
15    delete pkt;
16 }
17
18 void BlackIceCentral::processReport(BlackIceReport& report)
19 {
20    ++numReceivedWarnings;
21    reports.push_back(report);
22 }
23
24 void BlackIceCentral::processQuery(BlackIceQuery& query,
25    const inet::L3Address& addr, int port)
26 {
27    ++numReceivedQueries;
28    int warnings = 0;
29    for (auto& report : reports) {
30       double dx = query.getPositionX() - report.getPositionX();
31       double dy = query.getPositionY() - report.getPositionY();
32       if (dx * dx + dy * dy < query.getRadius() * query.getRadius()) {
33          ++warnings;
34       }
35    }
36
37    auto response = new BlackIceResponse("black ice response");
38    response->setWarnings(warnings);
39    querySocket.sendTo(response, addr, port);
40 }
```

As a last step, the `BlackIceWarner` periodically polls the server for black ice hazards. The setup of the UDP socket and the fetching `VehicleController` from the middleware are similar to Listing 13.4.

**Listing 13.6** Excerpt from BlackIceWarner

```
1  void BlackIceWarner::initialize()
2  {
3     // setup UDP socket (similar to BlackIceReporter)
4     // ...
5
6     // fetch (mutable) traci::VehicleController from middleware
7     // ...
8
9     pollingRadius = par("pollingRadius");
10    pollingInterval = par("pollingInterval");
11    pollingTrigger = new cMessage("poll black ice central");
12    scheduleAt(simTime() + uniform(0.0, pollingInterval), pollingTrigger);
13
14    numWarningsCentral = 0;
15    WATCH(numWarningsCentral);
16 }
17
18 void BlackIceWarner::handleMessage(cMessage* msg)
19 {
20    if (msg->isSelfMessage()) {
21       pollCentral();
22    } else if (msg->getKind() == inet::UDP_I_DATA) {
23       processResponse(*check_and_cast<BlackIceResponse*>(msg));
24       delete msg;
25    } else {
26       throw cRuntimeError("Unrecognized message (%s)%s",
27          msg->getClassName(), msg->getName());
28    }
29 }
30
31 void BlackIceWarner::pollCentral()
32 {
```

```
33    using boost::units::si::meter;
34    auto query = new BlackIceQuery("poll for black ice");
35    query->setPositionX(vehicleController->getPosition().x / meter);
36    query->setPositionY(vehicleController->getPosition().y / meter);
37    query->setRadius(pollingRadius);
38    socket.send(query);
39    scheduleAt(simTime() + pollingInterval, pollingTrigger);
40 }
41
42 void BlackIceWarner::processResponse(BlackIceResponse& response)
43 {
44    EV_INFO << "Black ice warnings: " << response.getWarnings() << "\n";
45    if (response.getWarnings() >= 2 && !reducedSpeed) {
46       vehicleController->setSpeedFactor(0.5);
47       reducedSpeed = true;
48       ++numWarningsCentral;
49    } else if (response.getWarnings() == 0 && reducedSpeed) {
50       vehicleController->setSpeedFactor(1.0);
51       reducedSpeed = false;
52    }
53 }
```

During initialization a local timer is scheduled (line 11) with a random offset. Since vehicles are added by SUMO at fixed intervals, this offset scatters the vehicles' queries so they are not synchronized. After sending the query, the local timer has to be re-scheduled according to the module's `pollingInterval`. If the server's response includes at least two warnings and the vehicle is still driving at full speed, `BlackIceWarner` reduces the vehicle's speed behavior and increments the local warning counter. As soon as the vehicle leaves the *danger zone*, i.e., no warnings apply for the polled region anymore, the vehicle restores its previous movement speed.

### *13.3.4 Results*

When running the aforementioned simulation setup on a grid map of $1 \times 1\,\text{km}^2$ similar to Fig. 13.2, fast vehicles in the middle region will generate black ice warnings. You can observe this by looking at `BlackIceCentral` while running the OMNeT++ *Qtenv* GUI: `numReceivedQueries` and `numReceivedWarnings` are incremented whenever `World.server.udpApp[0]` has received a corresponding message from the LTE network. Similarly, the `World.node[*].udpApp[1]` of endangered vehicles is incremented when the server responds with warnings in their area. You can add `recordScalar` calls for these variables to optionally record them in your result files as well.

## 13.4 V2X Services Using LTE D2D

In the previous section we showed how to simulate vehicles communicating with a back-end through infrastructure-based LTE communications. We now show how the same black ice scenario can be realized endowing UEs with D2D capabilities, i.e.,

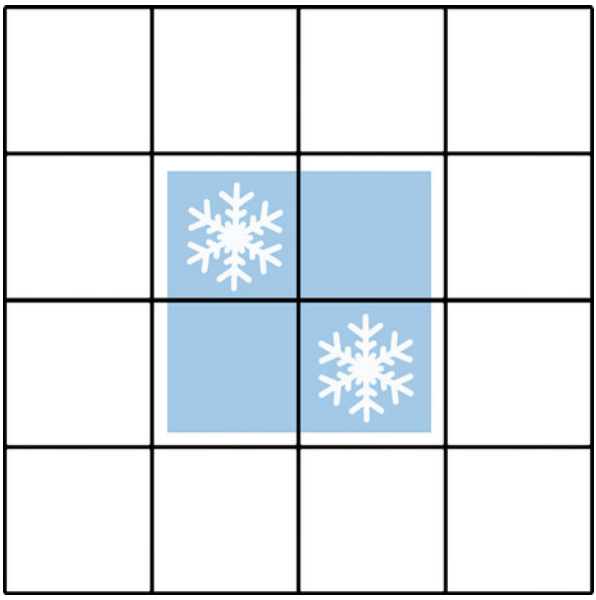


**Fig. 13.2** Suggested SUMO grid map for evaluating the *BlackIceWarner*

allowing them to communicate directly without the eNB acting as a relay for every communication. D2D mode in LTE is in fact gaining a wider attention recently, due to benefits such as a reduced end-to-end latency and better frequency reuse within the same cell. ETSI ITS protocols are not strictly bound to any radio technology.

To investigate how the black ice warning application from Sect. 13.3 could be adopted for LTE D2D, we will remove the central server component and move its logic towards the UEs, i.e., our connected vehicles.

### *13.4.1 Network Definition*

In this example, we consider the network from Fig. 13.1. In fact, endowing vehicles with LTE D2D capabilities does not require additions or modifications to the network definition. We use the two eNBs from Sect. 13.3.1 and the scenario comprising the SUMO map, the traffic demand, and the *Storyboard* script described so far. We will instead configure the V2X application logic of the LTE network for *one-to-many* D2D communications.

### *13.4.2 Parameters Configuration*

To enable D2D, we extend the *omnetpp.ini* configuration file from Sect. 13.3.2 with a `Config` section labeled `BlackIce-D2DMulticast`, as shown in

Listing 13.7. The first five lines enable D2D capabilities for both the eNBs and the vehicles. Their LTE NICs are set to the D2D counterparts, i.e., `LteNicEnbD2D` and `LteNicUeD2D`, respectively. As outlined in Chap. 5, SimuLTE requires a fixed Modulation and Coding Scheme (MCS) for one-to-many D2D transmissions. With reference to Listing 13.7, the D2D mode for the Adaptive Modulation and Coding (AMC) module is selected and the parameter `usePreconfiguredTxParams` is set. The parameter `d2dCqi` allows one to indicate the Channel Quality Indicator (CQI) that will be used for transmissions (within a range of 1–15).

**Listing 13.7** Configuration of D2D-enabled LTE NICs

```
[Config BlackIce-D2DMulticast]
*.eNodeB[*].nicType = "LteNicEnbD2D"
*.node[*].nicType = "LteNicUeD2D"
**.amcMode = "D2D"
**.d2dCqi = 7
**.usePreconfiguredTxParams = true

*.server.numUdpApps = 0
*.node[*].numUdpApps = 1
*.node[*].udpApp[0].typename = "BlackIceWarnerD2D"
*.node[*].udpApp[0].middlewareModule = "^.middleware"
*.node[*].udpApp[0].mcastAddress = "224.0.0.23"
*.node[*].configurator.mcastGroups = "224.0.0.23"
```

As all the application logic is distributed in the UEs, the *server* is not equipped with any UDP application. Instead, each vehicle is equipped with a `BlackIceWarnerD2D` application, i.e., an `UdpApp` combining the functionality of all three UDP applications from Sect. 13.3. Warnings are disseminated in the network via multicast transmissions: a vehicle originating a black ice warning will transmit a UDP packet using a multicast IP address as the destination via `mcastAddress`. Multicast addresses are a predefined subset of the IPv4 address space ranging from `224.0.0.0` to `239.255.255.255`.

To receive said warnings, other vehicles need to join the multicast group identified by that IP address in advance. The `HostAutoConfigurator` `mcastGroups` parameter includes a space-separated list of multicast IP addresses for this purpose. Our black ice warning application uses the arbitrary address `224.0.0.23`.

### *13.4.3 Modifying the Code*

As already indicated in the previous section, the D2D black ice warning is a melange of the three applications working hand-in-hand in the centralized setup. Thus, `BlackIceWarnerD2D` shares several of their concepts:

- report warnings when traction loss occurs like `BlackIceReporter`,
- reduce speed when a warning is received like `BlackIceWarner`, and
- check if a warning applies to the own driving area like `BlackIceCentral`.

The main difference, though, resides in how warnings are "stored" in the system. In the previous example, this was done by the central back-end, which stored the received warnings and disseminated them among vehicles. Now this has to be taken care of by each vehicle. In the following, we describe a simple solution to this purpose, which can serve as a baseline for smarter and more efficient communication patterns and algorithms for this use case. The application in Listing 13.8 does the following: as soon as a vehicle receives a warning concerning the area it is driving in, it will reduce its speed immediately and will restore maximum speed after 20 s, if no further warning is received. Two parameters, `warningRadius` and `warningDuration`, allow to configure the range of a warning and how long it is considered valid, respectively.

Similarly to the case of `BlackIceReporter`, `BlackIceReport` will emit a traction loss signal when the vehicle is driving too fast in the defined black ice region. Thus, also `BlackIceWarnerD2D` has to subscribe to the *Storyboard's* signal during initialization. Instead of sending those reports to the back-end, however, they will be sent to a multicast group, as specified in line 59.

Since direct communication is now employed, peering vehicles can receive `BlackIceReports` directly and process them in `processReport`. First, a vehicle checks if such a report applies to its own position by computing the distance between itself and the sending vehicles. If this distance does not exceed the `warningRadius` (line 67), it will reduce its speed (line 75) and schedule a self-message to cancel this reduction after `warningDuration` (lines 77 and 39). When receiving further reports in the meantime, the timer will be reset (line 73).

**Listing 13.8** BlackIceWarner application tailored for direct communication

```
1  void BlackIceWarnerD2D::initialize(int stage)
2  {
3     cSimpleModule::initialize(stage);
4     if (stage != inet::INITSTAGE_APPLICATION_LAYER)
5        return;
6
7     mcastAddress = inet::L3AddressResolver().resolve(par("mcastAddress"));
8     mcastPort = par("mcastPort");
9     socket.setOutputGate(gate("udpOut"));
10    socket.bind(mcastPort);
11
12    // LTE multicast support
13    inet::IInterfaceTable *ift = inet::getModuleFromPar<inet::IInterfaceTable>(
14       par("interfaceTableModule"), this);
15    inet::InterfaceEntry *ie = ift->getInterfaceByName("wlan");
16    if (!ie)
17       throw cRuntimeError("No interface named wlan");
18    socket.setMulticastOutputInterface(ie->getInterfaceId());
19    socket.joinMulticastGroup(mcastAddress);
20
21    // application's supporting code
22    auto mw = inet::getModuleFromPar<artery::Middleware>(
23       par("middlewareModule"), this);
24    mw->subscribe(storyboardSignal, this);
25    vehicleController = mw->getFacilities().
26       get_mutable_ptr<traci::VehicleController>();
27
28    // application logic
29    removeSpeedReduction = new omnetpp::cMessage("remove speed reduction");
```

```
30    warningRadius = par("warningRadius");
31    warningDuration = par("warningDuration");
32    numWarningsPeer = 0;
33    WATCH(numWarningsPeer);
34 }
35
36 void BlackIceWarnerD2D::handleMessage(cMessage* msg)
37 {
38    if (msg->isSelfMessage()) {
39       vehicleController->setSpeedFactor(1.0);
40    } else if (msg->getKind() == inet::UDP_I_DATA) {
41       processReport(*check_and_cast<BlackIceReport*>(msg));
42       delete msg;
43    } else {
44       throw cRuntimeError("Unrecognized message (%s)%s",
45          msg->getClassName(), msg->getName());
46    }
47 }
48
49 void BlackIceWarnerD2D::sendReport()
50 {
51    Enter_Method_Silent();
52    using boost::units::si::meter;
53    using boost::units::si::meter_per_second;
54    auto report = new BlackIceReport("black ice warning");
55    report->setPositionX(vehicleController->getPosition().x / meter);
56    report->setPositionY(vehicleController->getPosition().y / meter);
57    report->setSpeed(vehicleController->getSpeed() / meter_per_second);
58    report->setTime(simTime());
59    socket.sendTo(report, mcastAddress, par("mcastPort"));
60 }
61
62 void BlackIceWarnerD2D::processReport(BlackIceReport& report)
63 {
64    using boost::units::si::meter;
65    double dx = report.getPositionX() - vehicleController->getPosition().x /
          meter;
66    double dy = report.getPositionY() - vehicleController->getPosition().y /
          meter;
67    if (dx * dx + dy * dy > warningRadius * warningRadius) {
68       return;
69    }
70
71    ++numWarningsPeer;
72    if (removeSpeedReduction->isScheduled()) {
73       cancelEvent(removeSpeedReduction);
74    } else {
75       vehicleController->setSpeedFactor(0.5);
76    }
77    scheduleAt(simTime() + warningDuration, removeSpeedReduction);
78 }
```

### *13.4.4 Results*

Both, `BlackIceWarner` and `BlackIceWarnerD2D`, count the occurrence of applicable warnings in `numWarningsCentral` and `numWarningsPeer`, respectively. Comparing those numbers aggregated over all vehicles in the scenario, however, is similar to comparing apples and oranges because of the entirely different communication pattern. The performance of the two warning mechanisms can instead be compared by counting the occurrence of traction losses. Traction

**Table 13.1** Black ice warnings and traction losses

| Centralized communication | Direct communication |
|---|---|
| `pollingRadius` = 100 m | `warningRadius` = 100 m |
| `pollingInterval` = 1.0 s | `warningDuration` = 10.0 s |
| $\Sigma$ `numWarningsCentral` = 45 | $\Sigma$ `numWarningsPeer` = 34 |
| $\Sigma$ traction loss signals = 11 | $\Sigma$ traction loss signals = 27 |

losses will still occur, otherwise no vehicle would generate a warning. Exemplary numerical results from the presented setup are presented in Table 13.1.

As we anticipated, the proposed example can be used to experiment with the considered scenario, e.g., verifying how the performance is affected by the transmission power used for D2D communication, how to efficiently set `warningDuration` to harmonize between driving slow for too long and speeding up too early, or what is the impact of the `pollingRadius` and `warningRadius` parameters.

## 13.5 Conclusions

This chapter described how to simulate vehicles equipped with LTE and WLAN radios. Inter-vehicle communications are currently one of the most active research topics and realistic simulations of both communications and V2X services are important in order to evaluate the performance of driver-assistance systems and self-driving cars, for instance. Among the several available frameworks, we chose to integrate SimuLTE (see Chap. 5) and Artery (see Chap. 12) for simulating the two aspects above within an integrated framework. We showed how applications running on the LTE stack can make use of various features provided by Artery, e.g., its *Facilities* layer and scripting dynamic scenarios using the *Storyboard* (cf. Sect. 12.4).

We first described how to integrate Artery and SimuLTE together to obtain LTE-enabled cars. Afterwards, we detailed a black ice scenario, where cars detect and notify traction losses. We showed two possible approaches to the problem, the first one with infrastructure-based LTE communication, where traction losses are notified to a remote server through the serving eNB. A second approach exploited D2D communication to let vehicles communicate directly the detected traction losses. For each approach, we detailed how to configure the network and how to configure the main system parameters.

## References

1. ETSI: Intelligent Transport Systems (ITS); Communications Architecture (V1.1.1). European Standard (Telecommunications series), ETSI (2010)
2. Günther, H.J., Timpner, J., Wegner, M., Riebl, R., Wolf, L.: Extending a holistic microscopic IVC simulation environment with local perception sensors and LTE capabilities. Veh. Commun. **9**, 211–221 (2017). https://doi.org/10.1016/j.vehcomm.2017.01.003

3. Hagenauer, F., Dressler, F., Sommer, C.: Poster: A simulator for heterogeneous vehicular networks. In: Proceedings of the 6th IEEE Vehicular Networking Conference (VNC 2014), pp. 185–186. IEEE, Piscataway (2014)
4. Hagenauer, F., et al.: Veins LTE [online]. http://veins-lte.car2x.org
5. Timpner, J., Wegner, M., Günther, H.J., Wolf, L.: High-resolution vehicle telemetry via heterogeneous IVC. In: Proceedings of the First International Workshop on Internet of Vehicles and Vehicles of Internet, pp. 19–24. ACM, New York (2016)
6. Wegner, M., Timpner, J., et al.: ArteryLTE—An Inter-Vehicle Communication (IVC) simulation framework [online]. https://github.com/ibr-cm/artery-lte

# Chapter 14
# Simulating Opportunistic Networks with OMNeT++


**Asanga Udugama, Anna Förster, Jens Dede, and Vishnupriya Kuppusamy**

## 14.1 Introduction and Motivation

Opportunistic Networks (OppNets) are defined as *"the set of applications and services running on end user devices (e.g., smartphones, tablets and similar digital devices) that use direct communication opportunities to exchange information with each other"* [3]. They are generally motivated on one side from human gossiping and on the other side from unavailability of traditional infrastructure-based communications. In the next paragraphs, we present a short overview of what OppNets are and describe how to simulate them with OMNeT++.

### *14.1.1 What are Opportunistic Networks?*

OppNets are used in a number of application areas ranging from communications during disasters to social networking. They operate on any device-to-device communication technology such as Bluetooth and IEEE 802.15.4. A key component of a node in OppNets is the forwarding (i.e., data dissemination) protocol used to propagate information in networks. There are a number of such protocols developed by researchers to efficiently disseminate information throughout a network [21, 23, 24].

Figure 14.1 shows an example use case of OppNets. A fire breaks out at a shop in the middle of the city center. The visitors and shoppers who are present at that location see this emergency and quickly move away. The smart devices of

A. Udugama · A. Förster · J. Dede (✉) · V. Kuppusamy
Sustainable Communication Networks, University of Bremen, Bremen, Germany
e-mail: adu@comnets.uni-bremen.de; afoerster@comnets.uni-bremen.de;
jd@comnets.uni-bremen.de; vp@comnets.uni-bremen.de



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_14

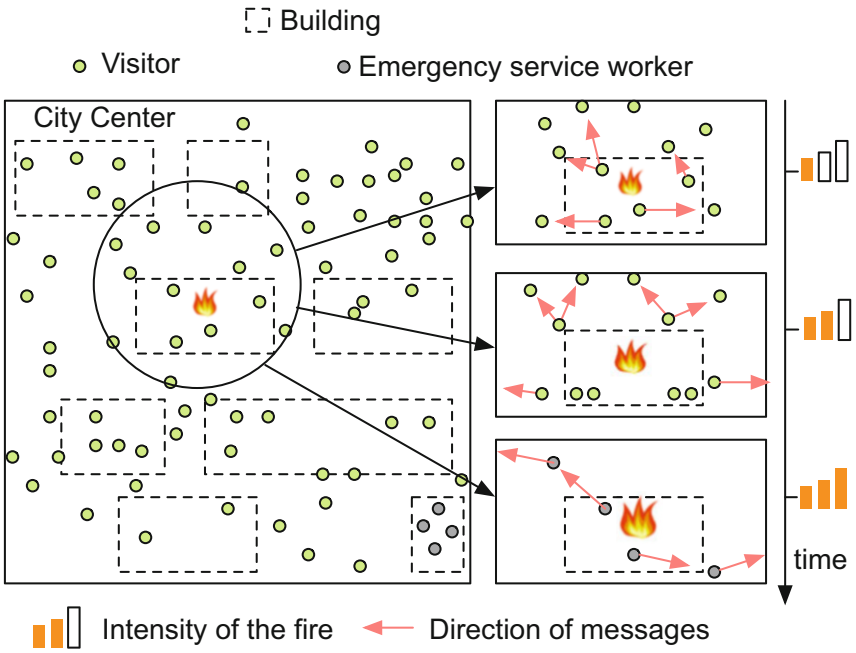



**Fig. 14.1** Dissemination of information in an emergency using OppNets

these people propagate information related to this emergency over OppNets. This information reaches the emergency service workers who are then able to respond to the emergency.

### *14.1.2 OppNets Simulation Requirements*

OppNets can be considered similar to Wireless Sensor Networks (WSNs), Mobile Ad Hoc NETworks (MANETs), or Vehicular Ad Hoc Networks (VANETs). However, they have special requirements and properties, which hinders the use of existing simulation platforms or frameworks. For example, they largely depend on human mobility, which is not covered at all in WSNs and only partially in MANETs or VANETs. Furthermore, the OppNets community has developed a large suite of data forwarding protocols, which are not used nor implemented for other systems. This makes it necessary to design a dedicated simulation framework for OppNets.

The main requirements towards simulation are the following:

- Human mobility models that realistically model the behavior of people in their everyday lives, including their day-night and workday-weekend rhythm, usage of different transport vehicles, repeatability in their mobility patterns, etc.

- A simple and efficient link technology model, which abstracts away the details of bit-level wireless communications and physical channels. Optimally, both an abstract and a precise model should be available to switch between them.
- Realistic application models, including non-dummy data exchange, reactions of users on received data (e.g., liking, forwarding, etc.), and realistic traffic patterns.
- Very large simulation scenarios with up to thousands of nodes.
- Availability of data forwarding protocols for OppNets.

### *14.1.3 Opportunistic Networks and OMNeT++*

The OMNeT++ simulator and its development languages (the Network Topology Description (NED) language and C++) provide the building blocks to build the layers of functionality of nodes in networks that operate using different wired and wireless protocols. Such protocol implementations are usually made available in OMNeT++ as frameworks. The INET Framework (see Chap. 2) is one such framework that provides the implementations for the protocols associated with the Internet Protocol (IP) suite. It is referenced by the official distribution of OMNeT++ and is recommended to be used when simulating IP-based networks.

When considering support for OppNets, the INET Framework lacks some of the prominent protocols related to simulating nodes in OppNets. To compensate for this deficiency, a number of research efforts have been concentrated on extending the official frameworks and other third party frameworks for OMNeT++ to include protocols and mechanisms specifically for OppNets simulation. These extensions have stayed unofficial and some of the code developed have become obsolete due to the changes that have been done in the architecture of OMNeT++ and its official frameworks. In the following are some of the most relevant works related to opportunistic networks in OMNeT++.

The authors of [9] proposed a set of mechanisms and methodologies to simulate OppNets and Delay-Tolerant Networks (DTNs) in OMNeT++. Their main focus was on mobility models. Their work extended the mobility related components of the Mobility Framework (a discontinued framework for mobile ad hoc and sensor networks, partially integrated in INET) to simulate trace-based mobility and node interactions based on contact traces. The code developed by the authors includes a toolbox to generate mobility traces based on a given mobility model. The work has used a scenario focused on urban content distribution to evaluate the performance of the proposed mechanisms.

In [10], the work on [9] has been extended to evaluate OppNets with nodes which are equipped with multiple network interfaces (e.g., Bluetooth, IEEE 802.11, cellular, etc.). These interfaces are turned on and off when required to communicate depending on the context. The radios used in the multiple interfaces are realized

using the—now deprecated—MiXiM framework[1] (the successor of the Mobility Framework, which is also discontinued). The dissemination of data throughout the network is done using a simple data exchange protocol that exchanges the information when nodes actually meet. Mobility in this research work is based on traces obtained from the **Legion Studio** pedestrian simulator [14].

The work presented in [13] focuses on caching and relaying strategies to improve distribution of content in OppNets. The dissemination of data is based on a publish/subscribe model where data exchange is performed only for the requested data (i.e., subscribed) with unicast transmissions. The scenarios used for evaluating the strategies are twofold: an outdoor mobile user scenario with high mobility and a subway scenario with entering and leaving users.

The authors of [9, 10, 13] have made their implementation (called OppoNet) available at GitHub. This code was developed with OMNeT++ version 4.1 (2010).

The authors of [26] have built a model framework in OMNeT++ to simulate OppNets. This implementation is an extension of the OppoNet work of [9], additionally using the INETMANET framework. The focus of the work is on the forwarding layer of an opportunistic networking node, separating the operations into multiple subprocesses. The work introduces the ExOR and MORE protocols comparing the performance against Optimized Link State Routing (OLSR) used in MANET based nodes. The code is based on OMNeT++ version 4.2 (2013).

The authors of [25] have built models in OMNeT++ to evaluate the performance of physical layer protocols in OppNets. The work extends the discontinued framework to perform broadcast and changes the fading model to have symmetric signal receptions. The data dissemination protocol used in [25] selects a set of nodes to send data based on a priority which is computed using a metric (e.g., the closeness to a node). The source of this model (OppSim) is available at SourceForge. The code is based on OMNeT++ version 4.2 (2013).

Furthermore, some research papers used OMNeT++ to evaluate OppNets, e.g., [2, 15, 18]. They did not introduce new or dedicated frameworks, but made use of whatever framework or model already available. We discuss why OMNeT++ is a good choice for OppNets in the next section.

## 14.2 Why Use OMNeT++ for OppNets

OMNeT++ has a very sophisticated user interface, with many possibilities to visualize and inspect various scenarios. The latest version 5.4 also includes sophisticated 2D and 3D graphics support to visualize the simulation scenarios. OMNeT++ is also highly modular and clearly separates between node behavior and node parameters, which makes it very easy to run large parameter studies.

[1]The MiXiM project has been discontinued, and its contents have been merged into the INET Framework. New projects should be based on a recent version of INET instead of MiXiM.

**Table 14.1** High-level comparison between OppNets simulators

| Tool | Adyton | ONE | OMNeT++ | ns-3 |
|---|---|---|---|---|
| Platforms | Linux | Java (JDK 6+) | Win, Linux, Mac | Linux, Mac, FreeBSD |
| Programming language | C++ | Java | C++, NED | C++, Python |
| Parallelization | - | - | + | o |
| Documentation | + | + | ++ | + |
| Mailing lists and tutorials | o | ++ | ++ | + |
| User interface | - | + | ++ | - |
| Mobility models | o | + | ++ (OPS) | o |
| Link technologies | - | - | + (OPS) | + |
| OppNets data propagation | ++ | ++ | + (OPS) | o |
| User behavior models | - | - | + (OPS) | - |
| Traffic models | ++ | ++ | ++ (OPS) | ++ |
| Scalability | + | - | + | o |

OMNeT++ data including the described OPS framework
- no support, o partial support, + adequate support, ++ very good support

In an earlier work [3] we have compared the performance of four different simulation platforms for OppNets and we have found that OMNeT++ clearly outperforms the other three. The alternatives are:

- The Network Simulator 3 (ns-3)[2] is a discrete event simulator mainly designed for IP-based networks with an emphasis on the network layer (layer 3) and the upper layers of the protocol stack. ns-3 has been used several times for evaluating OppNets, but, in general, it does not offer a lot of models for them. It is also not possible to switch off the link layer, which makes the simulation runs very long and impractical for OppNets.
- The Opportunistic Network Environment (ONE)[3] [12] is a simulation tool designed specifically for OppNets. It is written in Java. ONE is designed specifically for OppNets and therefore offers a wide variety of models that continue to grow. Unfortunately, the ONE does not perform well for very large simulations, as we have also shown earlier [3].
- Adyton[4] [19] was released in 2015. It is written in C++ and is also dedicated to simulating OppNets scenarios. It has quite a large range of mobility models and data forwarding protocols, but unfortunately no graphical user interface. In terms of performance, Adyton is comparable to OMNeT++.

Table 14.1 compares the above listed simulation platforms in terms of the OppNets requirements, as identified in Sect. 14.1.2.

[2] https://www.nsnam.org.

[3] https://akeranen.github.io/the-one.

[4] https://npapanik.github.io/Adyton.

In the next sections, we will present our Opportunistic Protocol Simulator (OPS) framework and the related information for simulating OppNets. Section 14.9 provides a tutorial on how some use cases are configured to run simulations with the OPS framework and how results are interpreted. Section 14.10 is an outlook on future plans for the OPS framework.

## 14.3 The OPS Framework

OPS is a set of simulation models in OMNeT++ to simulate opportunistic networks. It has a modular architecture where different protocols relevant to opportunistic networks can be developed, plugged in, and configured to evaluate their performance. The models of OPS are grouped into protocol layers of a protocol stack. Figure 14.2 shows the general architecture of OPS and its individual protocol layers.

One special feature of OPS and its design is the separation of the *Application Layer* from the *Notification Generator*. The *Notification Generator* injects the data traffic into the network, while the *Application Layer* mimics more the behavior of an app on a user's smartphone, where the user can delete messages, like them, etc. The *User Behavior Model* simulates the user herself and her interactions with the app. More details about these three models are provided in Sect. 14.4.

The *Opportunistic Forwarding Layer* is where any forwarding protocol related to disseminating data in opportunistic networks is plugged in. In general, it handles the

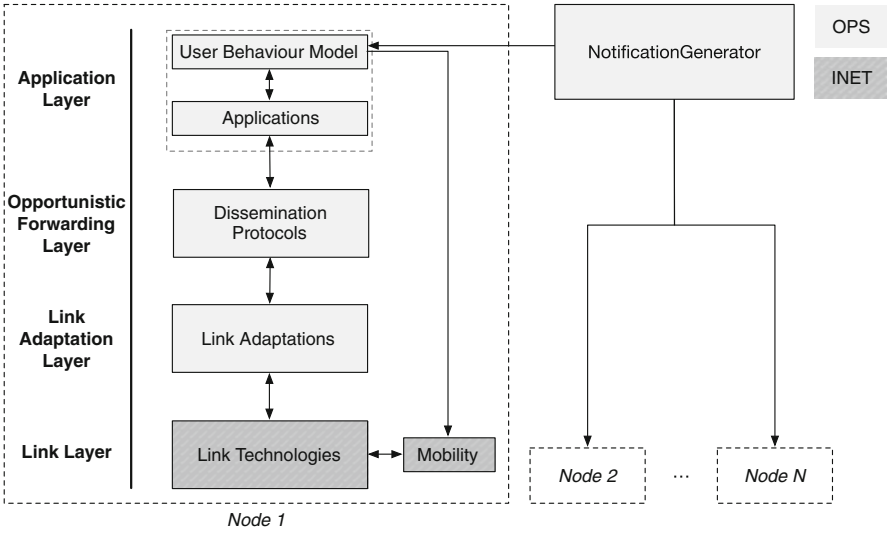



**Fig. 14.2** Generic OPS architecture, with its interactions with OMNeT++/INET

caching of data, the forwarding of tables or history, and the forwarding decisions. More details about the already implemented protocols and how to implement new ones are provided in Sect. 14.5.

The *Link Adaptation Layer* is a layer where mechanisms that convert messages from one form to another are deployed. An example for such a mechanism is the tunneling (and de-tunneling) of packets required when the OppNets is overlaid on nodes that are not part of the OppNets but are still required as intermediate nodes to carry packets. These nodes may only have the knowledge to forward other network protocols (not OppNets protocols). Therefore, when OppNets packets traverse these nodes, they are encapsulated by the underlying protocol. Currently, it is a place holder (*PassThru*) for later implementations.

The *Link Layer* is where any link access technology is implemented. In OppNets, it is highly important to be able to simulate large scenarios with hundreds or even thousands of nodes and very often, the link technology is abstracted and simplified as much as possible. We have implemented an INET-based module called `WirelessIfc`, which uses the Unit Disk Graph (UDG) for channel propagation. It can be parametrized with bandwidth, delays, and queuing strategies. Another possibility is to use other existing INET models to achieve more realistic channel propagation and link technology modeling. The interested reader should turn to the documentation and current status of the INET Framework (see Chap. 2).

*Mobility* of nodes is handled using the mobility interface provided by the INET Framework and any of its mobility models can also be used for OPS. Mobility is probably the most important factor when simulating OppNets and we provide some further discussion and some *do's* and *don'ts* in Sect. 14.7.

One contribution of OPS to the mobility models from INET is the implementation of reactive mobility models, i.e., mobility models which can be controlled by the *User Behavior Model*. More information is also provided in Sect. 14.7.

The interfaces between each layer of the protocol stack have a standard format which has to be adhered to by all protocol implementations. Figure 14.3 shows an example configuration of the protocol stack of a node in OPS using a configuration with the user behavior modeled application injecting data into the network.

The OPS simulator generates a log containing information on activities of each of the configured protocol layers during a simulation run. This information is parsed by a set of parsers made available in OPS to generate statistics. These parser output files are then used to plot the different graphs required to obtain the statistics. Currently, OPS provides the parsers to generate statistics for a number of metrics, described in greater detail in Sect. 14.8. These metrics are very specific to OppNets to evaluate how well the data is disseminated in a network. New parsers are required to be built or existing parsers must be extended to develop additional statistics.

The complete source code of OPS is available at Github.[5]

[5] OPS Github repository: https://github.com/ComNets-Bremen/OPS/.

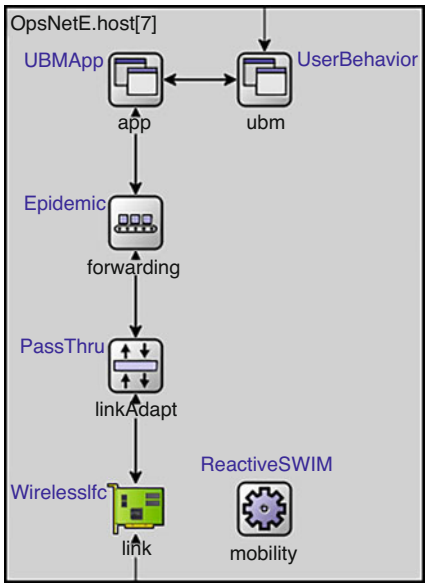



**Fig. 14.3** An example node configuration of an OPS node in OMNeT++

## 14.4 Application, User Behavior, and Notification Generator

The *Application Layer* consists of a set of models that emulates the use of applications when the influence of the users who use them is taken into consideration. All the models at this layer have to be configured to enable the user behavior influenced injection of data into a network, unlike the following layers of the protocols stack where different alternative models can be configured. The models of the application layer are described subsequently.

### *14.4.1 Notification Generator*

The *Notification Generator* is a global object (i.e., only one exists per simulation) that generates messages and delivers them to the `UserBehavior` model of each node to inject them into the network. Each message is identified with a set of keywords indicating the message type. While the keywords can be freely selected, there is one special keyword, called `emergency`, which signals to all layers that this is an emergency warning. It is mostly considered at the `UserBehavior` layer, where it leads to user reactions (e.g., running away). When nodes decide their own liking for any message, these keywords and their own preferences are taken into consideration.

The *Notification Generator* has several parameters:

- `notificationGenDistribution`: sets the distribution at which the messages are generated. (1) stays for constant intervals (no distribution), (2) for an exponential distribution, and (3) for uniform distribution. The mean value of all distributions is the value of the next parameter, `interEventTimeDuration`.
- `interEventTimeDuration`: the generation intervals for the messages.
- `radius`: the radius in meters from the epicenter of an emergency. People move out of this radius to get away from the emergency (e.g., a fire at a building). Used only for emergency messages.
- `locationsFilePath`, `eventsFilePath`: the paths to the files holding the locations where events occur and the events that occur at those locations. These files are created using the *ubm-data-gen.py* script in the *tools* folder. Check the sample configuration file used by this script.
- `logging`, `appPrefix`, `appNameGenPrefix`: parameter to control whether activity logging is enabled and the two prefixes used to name an application and the data. Leave the default in order to be able to use the provided parsers.
- `dataSizeInBytes`: payload size of the messages (events).

The *Notification Generator* always selects nodes randomly, providing those nodes with messages to be injected into the network.

### *14.4.2 User Behavior*

This model is tasked with handling the decisions related to comparing the keywords of a message and the preferences of a node to decide how a given message is classified by a user in terms of her/his fondness for that message. Once the decision is made, the message is flagged with the appropriate fondness value and delivered to the `BasicUBMApp`. As a result of the fondness assigned to a message, this model may decide to control the mobility of the user accordingly (e.g., start moving in the direction of a street performance in a city). Therefore, additionally, this model has a direct connection to the mobility model used through the `IReactiveMobility` interface. Details about the computation of fondness value are provided in [6].

The `UserBehavior` has several parameters:

- `avoidLocationTimer`: the time period that a location is avoided due to an emergency that occurred at that location.
- `react`: Boolean indicating whether a user reacts to an emergency event. The default value is `true`.
- `keywordsFilePath`, `eventsFilePath`: the paths to the files holding the keywords used as preferences by users and the events that contain these keywords. These files are created using the *ubm-data-gen.py* script in the *tools* folder. Check the sample configuration file used by this script.

- `reactToEventThreshold`: a real value between 0 and 1 to decide randomly whether there is a reaction to an event or not.
- `logging`, `appPrefix`, `appNameGenPrefix`: parameter to control whether activity logging is enabled and the two prefixes used to name an application and the data. Leave to the default in order to be able to use the provided parsers.

One special feature of the *User Behavior Model* is the so-called *Angry Bit Signal*. This is set for individual messages, which were received by this node, but received after the event had occurred (e.g., the concert was yesterday). This may happen with some data dissemination protocols, which do not use Time To Live (TTL) for messages (e.g., Keetchi, see Sect. 14.5) or when the TTL is set to the same value for all messages, irrespective of their contents. The *Angry Bit Signal* can be used as a quality of experience metric.

### *14.4.3 BasicUBMApp*

This module is a placeholder for more sophisticated applications. `BasicUBMApp` forwards all messages from the forwarding layer to `UserBehavior` and vice versa. This module only has parameters for setting up the logging paths (cf. Sect. 14.4.2 for `UserBehavior` and Sect. 14.4.1 for `NotificationGenerator`).

## 14.5 Forwarding Protocols

The generic operation of a forwarding protocol is separated into the following parts:

- *Caching of data*: Unlike in traditional networks, opportunistic networks are characterized by intermittent connections. Therefore, every node must employ a store-and-forward methodology when dealing with data.
- *Communicating with neighbors*: In opportunistic networks, the neighborhood (i.e., nodes in the vicinity with which a given node can communicate) changes constantly. Therefore, the forwarding protocol must employ mechanisms to pass-on data to maximize the propagation of this data in the network.

The current implementation of OPS includes several forwarding protocols, explained in the following subsections. In general, one can differentiate between destination-oriented or unicast protocols and destination-free or broadcast protocols [3]. The currently available forwarding protocols in OPS are capable of handling one or the other.

### 14.5.1 Epidemic Routing

*Epidemic Routing* is one of the first and best-known OppNets forwarding protocols [24]. It is inspired by epidemic dissemination of viruses and its principle says that when two nodes meet, they will exchange all their information. Technically speaking, when two nodes meet, one of them (usually the smaller Identifier (ID) or just by random access and backoff) sends a Summary Vector (SV) to the other, which lists all data items it currently holds. The second node will examine this list and will send back a list of its missing items. The first node will now send that missing data directly. The same is repeated for the same nodes vice versa. If there is enough time, both nodes will part with exactly the same data in their caches.

*Epidemic Routing* has the following parameters:

- *Re-synchronization period*: after two nodes have synced with each other, they wait for that period before they restart the procedure. This is done to avoid too often re-synchronizations.
- *TTL*: this is the maximum time-to-live for the packets after their creation. After that time, they get deleted from all caches around the network.
- *Cache size*: the size of the cache (in bytes) that holds data.
- *Maximum number of hops*: to limit the dissemination of packets, they can only travel a limited number of hops from their originator.

Packets in *Epidemic Routing* can be destination-oriented or destination-less. In fact, *Epidemic Routing* is a flooding protocol, which delivers all data to all nodes (and thus also to the single destination, if destination-oriented).

### 14.5.2 Spray and Wait

*Spray and Wait* [21] is a replication-based multi-copy routing protocol, which attempts to limit flooding. It consists of two phases: the *Spray* phase and the *Wait* phase. In the *Spray* phase, the source sprays the message to $L$ distinct relays. $L$ is effectively the number of copies of a message. The relays then forward the message to other $L$ distinct relays and so on. In the *Wait* phase, when the source and the relays are left with only one copy, they keep this final copy in their cache until they can directly deliver it to the destination. $L$ is a configurable parameter, where larger values result in an increased flooding of the network.

In *Binary Spray and Wait*, the source transmits $L/2$ copies to the first contact it comes across in the *Spray* phase. It keeps doing the same until it is left with only one copy which will be saved for direct delivery to the destination node, in case the source and destination ever meet. Every relay node in this protocol behaves in the same manner, transmitting $L/2$ copies of each message it has to the nodes it comes across and updates its $L$ for each message. The last copies of each message in every node are reserved for direct delivery to the destination.

When the nodes meet, they exchange their SV and request messages they do not have in their caches. The number of copies of each message is sent to the nodes along with the messages. The number of copies of each message in the network is limited by the protocol and no real flooding takes places as in Epidemic.

*Spray and Wait* has the following parameters:

- *Re-synchronization period*: after two nodes have synced with each other, they wait for that period before they restart the procedure. This is done to avoid too often re-synchronizations.
- *TTL*: this is the maximum time-to-live for the packets after their creation. After that time, they get deleted from all caches around the network.
- *Cache size*: the size of the cache (in bytes) that holds data.
- *Maximum number of hops*: to limit the dissemination of packets, they can only travel a limited number of hops from their originator.
- *L*: is the number of copies of data items to be created at the data sources.

### *14.5.3 Keetchi*

*Keetchi* [23] is an implementation of the Organic Data Dissemination (ODD) [5] concept. It is destination-less and it considers popularity of messages when forwarding data. It gives preferences to data, which are considered good or interesting by the users and it gives less priority or even drops messages, which are uninteresting. This protocol was the original reason why we started implementing the OPS framework. We needed a flexible environment, able to simulate individual users and their reactions to individual messages, which resulted in the *User Behavior Model* described in Sect. 14.4.

*Keetchi* is a machine-learning protocol. It assumes that the user evaluates each data item he/she receives by marking it with a "fondness" value. For example, if the messages are about jokes and are being disseminated to students on a campus, then the user can rate the joke upon reception with a number of stars, but can also ignore or even delete them. This reaction (simulated by the *User Behavior Model* from Sect. 14.4) is being translated to a numerical value and passed back to *Keetchi*. *Keetchi* itself has a list of all currently held data items (e.g., jokes in our example) and a *Goodness* value associated with each of them. Upon first reception of a data item, the *Goodness* value corresponds to the *Goodness* value for that data item of the sender (the last hop, not the original source of the joke). After a user reaction, this value gets combined in a Q-learning manner with the old *Goodness* value. The cache of *Keetchi* is sorted by these *Goodness* values. Additionally, items "age" in the cache by decreasing their *Goodness* values regularly. In this way, more preference is given to newly arriving items as well as most interesting data. If data items arrive and there is no space in the cache, the data with the least *Goodness* values get erased to accommodate the new ones.

The following parameters control the cache of *Keetchi*:

- *Aging interval*: how often the aging process is applied (in seconds).
- *Learning constant*: a parameter of the Q-learning approach, which controls whether more weight is given to the immediate reaction of the user or the network-wide *Goodness* value.

*Keetchi* also observes its communication context: it evaluates whether its neighborhood has changed or not. If the neighborhood is mostly new (as compared to the last time), *Keetchi* sends out a random data item from the upper part of its cache (the items with highest *Goodness* values) at regular intervals as a broadcast message to all neighbors. In case there is no significant change in the neighborhood, *Keetchi* starts sliding its sending window to less interesting data items and also slows down its sending interval. After some time without any change in the neighborhood, *Keetchi* stops sending data altogether to save energy.

The parameters which control the sending mechanism of *Keetchi* are:

- *Neighborhood Change Threshold*: how much percent difference is needed in the list of neighbors to focus again on the most important data items and start sending at the maximum sending frequency.
- *Backoff Timer Factor*: by how much to decrease the sending frequency in case there is not enough neighborhood change.

The impact of the individual parameters can be seen in [23].

*Keetchi* is a good option to compare against in case a large-scale, destinationless scenario is used, where data needs to reach many different parties and their locations/IDs are not known a priori. This is the case, for instance, for disaster relief operations or city-scale announcements about events, etc.

### *14.5.4 Deterministic Gossip Algorithm*

The two basic requirements of gossip algorithms are that the nodes are unaware of the whole topology and the nodes may initiate contact with only one of their neighbors at a time for data exchange. Deterministic Gossip Algorithm (DGA) [8] adds an additional requirement that nodes may choose only one new neighbor in every round of the neighbor discovery phase. All the nodes maintain a list of their established contacts and record the transaction of every data item sent or received by them.

Nodes receive the neighbor list at regular time intervals in the neighbor discovery phase. The neighbor list has all the neighbors that are in the wireless range of a node at a given time. When a node receives its neighbor list, it selects a random neighbor and checks through the established contacts list to find if the selected neighbor is an old contact. If the selected neighbor is an old contact, the node repeats this procedure through its neighbor list to identify new neighbors in its vicinity. When it finds a new

contact, the node sends the first data item in its cache to the new contact, records this transaction, and updates its established contact list.

All nodes send new data to their previously contacted nodes at regular intervals. Sending a new data item implies that the forwarding protocol selects an item from its cache that it has never sent to, nor received from, the selected old contact before. All these transactions are recorded by the sending and receiving nodes so that they do not repeat sending the same data.

The parameter that controls the sending mechanism of DGA is:

- *Contact neighbor interval*: it is the regular interval at which each node contacts all the previously contacted nodes (established contacts list) to send a new data item from its data cache.

### *14.5.5 Randomized Rumor Spreading*

In Randomized Rumor Spreading (RRS) [4], random data items are selected from a node's cache and sent out when neighbors are detected in the node's vicinity. The data transmission can be unicast or broadcast. In unicast, a node sends a random data item from its cache to one of its neighbors whereas in broadcast, the selected data item is received by all the neighbors of a node at the same time. Due to the random selection of data, it is highly likely that a node sends the same data item multiple times to the same neighbors and that all of the data items in the node's cache are not disseminated in the network.

RRS has the following parameter:

- *Cache size*: the size of the cache that holds data.

### *14.5.6 Further Protocols*

Additional protocols are continuously implemented and added to the OPS framework. *Prophet* and *Direct Delivery* are available as preliminary versions and planned to be included in the next OPS release.

*Prophet* [16] is a well-known and widely used protocol, also available for other OppNets simulators. It is strictly destination-oriented and many other protocols have been compared to it. *Direct Delivery* is probably the simplest OppNets forwarding protocol, where a node sends a data message only if it directly encounters its destination. *Direct Delivery* can be very well used in comparative simulation studies as the worst-case measure.

We encourage the reader to check the currently available forwarding protocols in our Github[6] repository.

[6]OPS Github repository: https://github.com/ComNets-Bremen/OPS/.

## 14.6 Link Layer

The link layer is one of the major challenges for OppNets. Here, the contradicting requirements by the OppNets protocol and the constraints of the used hardware, i.e., an end-user device like a smartphone, get together. This is where the forwarding protocol hands out the messages to be sent over some wireless channel to the neighbors.

On the one hand, most protocols for OppNets require flexible access to the link layer. In an ideal case, a continuous data stream can be sent to all devices in proximity using broadcast/multicast messages without the need of pairing the devices or exchanging a common secret. Furthermore, a neighbor discovery (*who is around me?*) handled by the link layer is an important feature.

On the other hand, encrypted and authenticated connections are preferred or even enforced by the major operating systems on mobile devices—namely, **iOS** and **Android**. Additionally, the systems limit background services to increase the device lifetime when running on battery power. These constraints could be overcome by rooting or jail-breaking the devices. Unfortunately, this drastically reduces the number of possible users as it is only an option for more experienced users. Additionally, the standards for direct data transmission between two devices differ between **iOS** and **Android**. Even for transferring a picture from one device to another requires in most cases an Internet connection.

Custom hardware or WSNs using, for example, IEEE 802.15.4, meet those requirements but are generally not carried by normal users. However, this kind of hardware could be used for a field test and a first evaluation of protocols.

Some existing implementations overcome these problems using various techniques. **FireChat** and the underlying framework MeshKit[7], for example, use a combination of Bluetooth and WiFi to establish the connection between two devices. However, this approach increases the energy consumption and thus decreases the battery lifetime drastically. Using Bluetooth Low Energy (BLE) beacons is another option. The disadvantage of the BLE technology is the very limited payload size of approximately 20 bytes.

In contrast to the real world, the constraints regarding the link layer in simulations are quite low. One can consider a very simplistic model which directly sends all data to all nodes without any physical layer constraints. On the other hand, it is also possible to perform highly sophisticated simulations using models close to the real-world implementations including radio propagation models, obstacles, different physical layers, etc. The choice depends on the requirements of the protocols, availability of models, available processing power, and simulation time.

A compromise between simplistic/fast and highly sophisticated/slow models is currently being evaluated by us. The idea is to replace the modeling of the data transmission (International Organization for Standardization (ISO)/Open Systems

[7] https://www.opengarden.com/.

Interconnection (OSI) link layer and below) by generic statistical models describing the behavior of the used protocols. Here, the main challenge is to integrate this kind of model into the OMNeT++ environment.

The link layer currently implemented in OPS (`KWirelessInterface`) belongs to the class of simple wireless interfaces. It uses the simplest radio propagation model, UDG, which assumes perfect transmission in a predefined radius and no transmission outside this radius. It does not simulate errors nor losses. It has the following main parameters:

- *Wireless range*: represents the communication range using a UDG model. It defaults to 30 m, which was experimentally identified in [20] for BLE.
- *Bandwidth*: The bandwidth of the wireless channel. It can be configured and defaults to 100 kbit/s, experimentally identified for BLE in [17].
- *Header size*: The protocol-related information assumed in addition to the payload for each packet. Here, a default value of 16 B is assumed.
- *Neighbor scanning period*: Determines how often a scan for neighboring nodes is performed. It defaults to 1 s.
- *Medium Access Control (MAC) address*: To distinguish the nodes, the MAC address is used and needs to be configured.

Currently, there is no simple option to use the more sophisticated models available, for example, in INET. The focus of OPS is to simulate opportunistic networks in a realistic scale and thus, large networks. In [3], we show that the used radio model has only a minor effect on the packet reception and delay, but a high influence on the simulation run time. Therefore, OPS uses the UDG model instead of more sophisticated models. However, one could use the sophisticated models offered by OMNeT++/INET by implementing a new adaptation layer (i.e., *Link Adaptation Layer*) between the OppNets protocol and the link layer.

## 14.7 Mobility Models

Mobility models control the movement of individual nodes in a simulation. There are three major types of models (cf. [3]), in particular:

- *Synthetic models*: These models use mathematical equations and other formal methods to compute the next location of a node. Many of these models are purely random, i.e., they select a random point or direction to follow and move the node with a predefined speed there. Such models are fast, scalable, but very unrealistic for simulating the mobility of real people.
- *Trace-based models*: These models rely on logs of real people and their mobility. Researchers have acquired many of these logs all over the world in the last two decades. Many of them are publicly available, such as the **CRAWDAD**[8]

[8] https://crawdad.org.

database. In simulations, the real mobility is simply replayed. Traces are very realistic, but slow and limited to the area and number of people who participated in the original logging campaign.

- *Hybrid models*: They combine both the above mentioned families by extracting some statistics or observations from real world traces and using them to parametrize and sophisticate synthetic models. Such models are still scalable, but achieve more realistic results than purely random models. Since they extract statistical data from real traces, the simulation of outliers, special cases, etc., is not possible.

### 14.7.1 Reactive Mobility Models

Additionally, in OPS we have introduced a new type of mobility model, i.e., *reactive mobility* models [6]. In traditional mobility models (e.g., all INET mobility models), the node is passive. The mobility model fully controls the movement of the node and simply overwrites its current position. In reactive models, the node can also signal to the mobility model that it wants to move to some particular location at some predefined time. As a special case, the node can also tell the mobility to run away, e.g., in a disaster scenario. This feature is very important and novel for simulating OppNets. If you refer back to Fig. 14.1, it becomes obvious that the dissemination of real messages (e.g., with a fire warning) will and should significantly impact the mobility of the receivers—they will run away. This will result in completely different network topologies, which were not possible to explore with passive mobility models.

To enable reactive mobility, we introduced the `IReactiveMobility` interface in OPS. It can be used to extend any non-trace-based mobility model to make it reactive. More details about how to implement such models can be found in the implementation of the reactive Small Worlds in Motion (SWIM) model [6]. Reactive mobility models need to be used together with our *User Behavior Model* (cf. Sect. 14.4).

### 14.7.2 Other Mobility Models in OPS

The use of both the user behavior and the reactive mobility models is optional in OPS. If they are not desired, any other INET mobility model can be used. However, not all of them are well suited for evaluating OppNets. Here, we give a short overview of available INET mobility models and whether they are a good choice for OppNets.

*Non-recommended Models* These include all random and deterministic models, i.e., random waypoint, random direction, rectangle mobility, circle mobility, tractor

mobility, Gauss Markov mobility, Mass mobility, Chiang mobility, and Const Speed mobility. These models do not represent the mobility of people or vehicles at all. They are, however, good for controlled experiments, i.e., where researchers would like to compare results from simulation with the real world.

*Recommended Models* These include trace based models, in particular BonnMotion [1]. BonnMotion is an external tool, which produces traces and has an INET counterpart, which uses these traces. BonnMotion is a very powerful tool, which can be used to either re-format real world traces for use in INET or can run various sophisticated mobility models and produce traces. However, it also includes random models (see above), which are nevertheless not recommended to use for OppNets. From the currently available models in BonnMotion (as of version 3.0.1), the map-based models, SLAW, SMOOTH, and TIMM are recommended and provide some realism. Some special scenarios can be simulated using Pursue (policemen chasing criminals), Nomadic, and RPGM (tourist groups roaming around). More information on these models are provided in the documentation of BonnMotion,[9] where also the original articles can be found.

Additionally, there are a few hybrid mobility models available, implemented directly for INET, in particular: SWIM [22], reactive and passive versions, and TRAILS [7]. These are all part of the OPS framework.

INET also includes some trace-based models for reading traces from ANSim.[10] However, this tool only includes random models and real-world traces, which we covered already with BonnMotion. Of course, ANSim can also be used for real traces. The same holds for NS2Mobility in INET—it replays files created by the Network Simulator 2.[11]

## 14.8 Metrics

OPS gathers all kinds of activity information (controlled by the `logging` parameter of the individual modules), which are later parsed with one of our **Python**-based parsers. Here, we offer a short overview for the different statistics that are computed by the parsers currently available in OPS.

- *Liked data receipts*: shows the amount of liked data (i.e., messages) received, compared to all the data received. Liked data are the data that were considered as being useful (interesting) by the users, i.e., they provoked a reaction with the maximum possible fondness. The statistics are computed per node as well as for the whole network.

[9] BonnMotion website: http://sys.cs.uos.de/bonnmotion/.

[10] http://www.ansim.info.

[11] https://ns2simulator.com.

- *Non-liked data receipts*: shows the amount of non-liked (i.e., non-interesting) data received, compared to all the data received. Non-liked are the data that were not classified as being useful at the beginning of a simulation. The statistics are computed per node as well as for the whole network.
- *Data delivery ratio*: shows the ratio of delivered data to all data that was generated.
- *Average delivery time*: shows how long (at average) it takes for data to be delivered to the intended recipient.
- *Average contact time*: shows the average time that nodes were in contact during a simulation.
- *Number of contacts*: shows the times that nodes were in contact during a simulation.
- *Cache activities*: shows the additions, removals, or updates performed on caches of every node in a network during a simulation.

As mentioned earlier, applications in OPS are able to operate as destination-oriented or destination-less data generators. For destination-oriented scenarios, statistics are gathered only for destinations. For destination-less scenarios, it is assumed that all nodes but the original sender is a destination and statistics are gathered accordingly.

It is also possible to parse the log file with other tools and scripts. The codes used in the log file and their meaning are described in the *LOG_ENCODINGS.md* file.

## 14.9 Tutorial

The OPS framework consists of a number of components which have to be built and configured to run simulations for identified scenarios. In this section, we provide a tutorial on using OPS in terms of use cases. The simulation of each use case is detailed in the next subsections: *use case*, *simulation setup*, and *results interpretation*.

OPS depends on a number of other software components for its successful operation. These components and the setup procedure are regularly updated when the OPS framework is extended with new functionality. The up-to-date setup procedure and the complete code are available at the OPS Github[12] repository.

Figure 14.4 shows the general procedure of running simulations and obtaining results which consist of five basic steps.

- *Step 1: Set up an ini file.* Once the use case and its unique parameters are identified, an *omnetpp.ini* file is configured with these parameters. There are a few sample *omnetpp.ini* files available with different configurations to use as examples. The parameters relevant to every model are listed in the *ned* file of the corresponding model together with the default parameters.

[12] OPS Github repository: https://github.com/ComNets-Bremen/OPS/.

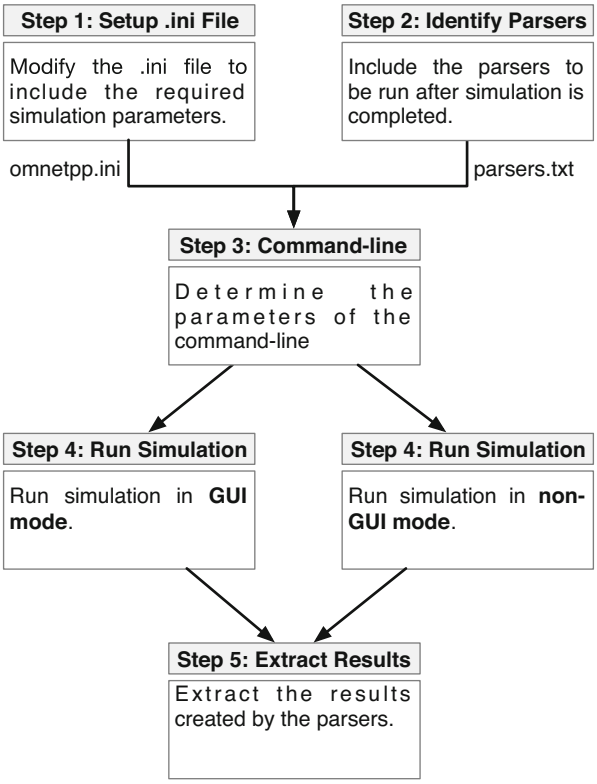



**Fig. 14.4** Procedure followed in running a simulation with the OPS framework

- *Step 2: Identify parsers.* The log that is created during *Step 4* contains the raw performance results of the simulated entities. To obtain statistical results, a set of parsers are run on the log files. These parsers, which are included in OPS, create scalar as well as vector statistics. They can be configured to run automatically after a simulation completes or they can be run separately on the generated log files. To run them automatically, the *parsers.txt* file is inserted with the required parser scripts.
- *Step 3: Configure the command-line.* The simulations are run from a command-line script, which automatically starts the OMNeT++ simulation and later, the configured parsers. The script has different parameters:
  - `-m cmdenv | qtenv`: non-graphical or graphical mode. In the graphical mode, the user can fully interact with the simulation from the OMNeT++ Graphical User Interface (GUI) environment.
  - `-c <ini file>`: path to the *ini* file created in *Step 1*.
  - `-p <parser file>`: path of the text file created in *Step 2* that includes the parsers to run automatically.

  - `-b <backup dir>`: after a simulation run, all files of the simulation are compressed and placed in a *tar* file. This parameter gives the path where this *tar* file is created.

- *Step 4: Run the simulation.* The simulation is run either in the GUI mode or non-GUI mode. In the GUI mode, the user has more control over the simulation run and can view the simulation graphically.
- *Step 5: Extract the results.* The results output by the parsers are held in a set of text files. These text results can be used to plot graphs using tools such as **Matlab** or **Python** scripts using the libraries **matplotlib** and **numpy**.

  The OPS Github repository contains a *README* with additional details.

### *14.9.1 Use Case 1: Are Messages Propagated Well?*

*Use Case* A group of students at a campus university is the basis of this use case. These students are a subset of the whole university population and they are part of an opportunistic network. They live at different locations at the university (i.e., hostels distributed around the campus). The students meet during their daily activities within the university at different times of the day (e.g., lecture rooms, study rooms, canteen, gym, coffee shop). They have common as well as different preferences and may meet individually or as groups at places of interest. When they meet, they exchange information between themselves. In this use case, we are interested in knowing how well the messages (events) are propagated in such a network.

*Simulation Setup* The group consists of 50 students (i.e., 50 nodes). The modeling of their mobility behavior is best represented by a mobility model that focuses on locations. The SWIM [22] mobility model in its reactive version is such a model where locations are classified as home locations, neighboring locations, and popular locations. When people are notified of events which are of interest (i.e., has the highest possible fondness or also referred to as liked data) to them, they may decide to attend those events. The campus is located in a 5 $km^2$ area. Table 14.2 shows the important parameters of the models used in the simulations.

Since we are interested in how well the messages propagate, we check the performance of the statistics *average delivery delay* and the *average delivery ratio* of messages (events). A further considered statistic is the *average contact time* of contacts made by the nodes.

*Interpreting the Results* Table 14.3 shows the collected statistics. The results show that the delivery of messages (events) to the nodes is close to 90% irrespective of whether the events were of interest to the students or not. This behavior is due to a number of factors. The two most influential factors are the behavior of the *Epidemic Routing* protocol and the sizes of the caches maintained at each node. In this use

**Table 14.2** Simulation parameters for use case 1

| Simulation control | |
|---|---|
| Nodes | 50 |
| Simulation duration | 7 days |
| User behavior model | |
| Number of locations | 20 locations |
| Mean event generation duration | Every 30 min |
| Forwarding model | |
| Forwarding protocol | Epidemic routing |
| Cache size | 5 MB |
| Mobility model | |
| Model | Reactive SWIM |
| Simulated area | 5 $km^2$ |
| Average speed | 1.2 m/s (walking speed) |
| Average pause time | Uniformly distributed between 20 min and 8 h |
| $\alpha$ | 0.5 |
| Number of locations | 20 |
| Wireless model | |
| Technology | BLE |
| Communication range | 30 m |
| Bandwidth | 100 kbit/s |

**Table 14.3** Simulation results for use case 1

| Statistic | Interested events (liked) | Un-interested events (non-liked) |
|---|---|---|
| Average delivery delay | 9.19 h | 9.32 h |
| Average delivery ratio | 88.35% | 89.36% |
| Average contact time | 2.61 h | |

case, the caches are purposely kept low (5 MB) to check how the protocol behaves. Since the protocol attempts to perform a complete synchronization of caches when two nodes meet, together with the higher contact duration (2.61 h), the nodes are still able to deliver a high percentage of the messages to all nodes.

But unfortunately, the *Epidemic Routing* protocol does not exploit the information about the interests of students (presented by the user behavior). Therefore, it is unable to provide a complete delivery of messages of interest to the students.

### *14.9.2 Use Case 2: Can WiFi Direct Help the Students?*

*Use Case* The students in the *Use Case 1* were using their BLE interfaces to communicate in the opportunistic network. The practical communication radius of

**Table 14.4** Simulation parameters for use case 2

| Wireless model | |
|---|---|
| Technology | WiFi direct |
| Communication range | 60 m |
| Bandwidth | 250 Mbit/s |

**Table 14.5** Simulation results for use case 2

| Statistic | Bluetooth low energy | WiFi direct |
|---|---|---|
| Average delivery ratio | 88.86% | 93.08% |
| Total packets sent | 55,879 | 57,781 |
| Average contact time | 2.61 h | 2.21 h |

BLE interfaces is 30 m as shown in [20]. But if the students use their WiFi Direct interfaces instead of BLE, would the propagation of information improve? We are therefore interested in knowing how the delivery of information changes in such an environment.

*Simulation Setup* The same simulation parameters are used as in the *Use Case 1* but with the wireless interface now configured to the parameters of WiFi Direct. Table 14.4 shows the changed parameters for the wireless communication. We will compute the same metrics as before for comparison.

*Interpreting the Results* Table 14.5 shows the collected statistics for both use cases for comparison, where we have taken the mean delivery rate and delay for all packets (liked and non-liked). The results show that the delivery rate has improved. This could be expected because of the higher bandwidth that WiFi Direct offers. But this has a down side. Limited cache sizes mean that messages are erased from caches to accommodate new messages. On the other hand, the availability of more bandwidth means the possibility of initiating more cache synchronization attempts. Therefore, due to these 2 actions, more and more packets are injected into the network as seen from the *Total Packets Sent*. It is also interesting to observe that the mean contact time has decreased with WiFi Direct, which is due to more but shorter contacts.

## 14.10 Ongoing and Future Works

In this chapter, we have explored the use of OMNeT++ for evaluating opportunistic networks. Beside identifying the main challenges and the most important simulation models, we also introduced our framework for opportunistic networks, OPS. OPS offers some new models and concepts, for example reactive mobility models and user behavior models. It is an ongoing effort and at this point we would like to encourage the interested reader to join the development process.

In the future, we will focus mainly on the following issues and goals:

- Abstract link layer: we will extend our current link layer abstraction to enable more pre-settings for various communication technologies with their specific properties.
- Data dissemination protocols: we are currently in the process of implementing *Direct Delivery*, *Optimal Routing*, *Prophet* [16], and *BubbleRap* [11], and we will select further data dissemination protocols for integration into OPS.
- Destination-oriented data: although the forwarding protocols are capable of handling data destined to specific nodes, the applications (specifically, the *Notification Generator*) do not support this feature. We intend to include this feature in the future where the destination is either selected randomly or to a configurable destination.
- Link adaptations for different link technologies: OPS is able to use different link technologies through the adaptations done at the *Link Adaptation Layer*. We intend to develop different *Link Adaptation Layer* modules for the link technologies that will be included in OPS.

**Acknowledgements** We would like to thank the students at the University of Bremen, who contributed over the years to create the OPS framework: Anas bin Muslim, Karima Khandaker, Mine Centinkaya, Kirishanth Chethuraja, and Jibin Pathapparambil John.

## References

1. Aschenbruck, N., Ernst, R., Gerhards-Padilla, E., Schwamborn, M.: Bonnmotion: a mobility scenario generation and analysis tool. In: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, SIMUTools '10, pp. 51:1–51:10. ICST, Brussels (2010)
2. Boldrini, C., Conti, M., Jacopini, J., Passarella, A.: Hibop: a history based routing protocol for opportunistic networks. In: 2007 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, pp. 1–12. IEEE, Piscataway (2007)
3. Dede, J., Förster, A., Hernández-Orallo, E., Herrera-Tapia, J., Kuladinithi, K., Kuppusamy, V., Manzoni, P., bin Muslim, A., Udugama, A., Vatandas, Z.: Simulating opportunistic networks: survey and future directions. IEEE Commun. Surv. Tutorials **20**(2), 1547–1573 (2018)
4. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 1–12. ACM, New York (1987)
5. Förster, A., Udugama, A., Görg, C., Kuladinithi, K., Timm-Giel, A., Cama-Pinto, A.: A novel data dissemination model for organic data flows. In: 7th EAI International Conference on Mobile Networks and Management (MONAMI), Santander. Springer, Cham (2015)
6. Förster, A., Muslim, A.B., Udugama, A.: Reactive user behavior and mobility models. In: Proceedings of the 4th OMNeT++ Community Summit. University of Bremen, Bremen (2017). http://arxiv.org/abs/1709.06395
7. Förster, A., Bin Muslim, A., Udugama, A.: Trails–a trace-based probabilistic mobility model. In: Proceedings of the 21st ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM). ACM, New York (2018)
8. Haeupler, B.: Simple, fast and deterministic gossip and rumor spreading. J. ACM **62**(6), 47 (2015)

9. Helgason, O.R., Jónsson, K.V.: Opportunistic networking in OMNeT++. In: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems and Workshops, Simutools 2008, Marseille, March 03–07. ICST, Brussels (2008)
10. Helgason, O.R., Kouyoumdjieva, S.T.: Enabling multiple controllable radios in OMNeT++ nodes. In: Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools) (2011)
11. Hui, P., Crowcroft, J., Yoneki, E.: Bubble rap: social-based forwarding in delay-tolerant networks. IEEE Trans. Mob. Comput. **10**(11), 1576–1589 (2011)
12. Keränen, A., Ott, J., Kärkkäinen, T.: The ONE simulator for DTN protocol evaluation. In: Proceedings of the 2nd International Conference on Simulation Tools and Techniques, Simutools '09, pp. 55:1–55:10. ICST, Brussels (2009)
13. Kouyoumdjieva, S.T., Chupisanyarote, S., Helgason, O.R., Karlsson, G.: Caching strategies in opportunistic networks. In: Proceedings of the IEEE International Symposium on World of Wireless, Mobile and Multimedia Networks, WoWMoM 2012, San Francisco, CA (2012)
14. Legion: Legion studio [online]. http://www.legion.com. Accessed 03 April 2017
15. Leontiadis, I., Mascolo, C.: Geopps: geographical opportunistic routing for vehicular networks. In: 2007 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, pp. 1–6. IEEE, Piscataway (2007). https://doi.org/10.1109/WOWMOM.2007.4351688
16. Lindgren, A., Doria, A., Davies, E.B., Grasic, S.: Probabilistic Routing Protocol for Intermittently Connected Networks. RFC 6693 (2012). https://doi.org/10.17487/rfc6693. https://rfc-editor.org/rfc/rfc6693.txt
17. Mikhaylov, K., Plevritakis, N., Tervonen, J.: Performance analysis and comparison of bluetooth low energy with IEEE 802.15.4 and simpliciTI. J. Sens. Actuator Netw. **2**(3), 589–613 (2013)
18. Musolesi, M., Hailes, S., Mascolo, C.: Adaptive routing for intermittently connected mobile ad hoc networks. In: Sixth IEEE International Symposium on a World of Wireless Mobile and Multimedia Networks, pp. 183–189. IEEE, Piscataway (2005)
19. Papanikos, N., Akestoridis, D.G., Papapetrou, E.: Adyton: a network simulator for opportunistic networks [online]. https://github.com/npapanik/Adyton (2015)
20. Sang, L., Kuppusamy, V., Förster, A., Udugama, A., Liu, J.: Validating contact times extracted from mobility traces. In: Puliafito, A., Bruneo, D., Distefano, S., Longo, F. (eds.) Ad-hoc, Mobile, and Wireless Networks, pp. 239–252. Springer International Publishing, Cham (2017)
21. Spyropoulos, T., Psounis, K., Raghavendra, C.S.: Spray and Wait: An Efficient Routing Scheme for Intermittently Connected Mobile Networks, pp. 252–259. ACM, New York (2005)
22. Stefa, J., Mei, A.: Swim: a simple model to generate small mobile worlds. In: Proceedings of IEEE INFOCOM Conference. IEEE, Piscataway (2009)
23. Udugama, A., Förster, A., Kuladinithi, K., Dede, J., Kuppusamy, V., Vatandas, Z.: My smartphone tattles: Considering popularity of messages in opportunistic data dissemination. MDPI Future Internet 2019, **11**(2), 29 (2019)
24. Vahdat, A., Becker, D.: Epidemic routing for partially connected ad hoc networks. Tech. Rep. Technical report number CS-200006, Duke University (2000). ftp://ftp.cs.duke.edu/dist/techreport/2000/2000-06.ps
25. Zhang, R., Chandran, A.R., Timmons, N., Morrison, J.: OppSim: a simulation framework for opportunistic networks based on MiXiM. In: Proceedings of the IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks, CAMAD 2014, Athens. IEEE, Piscataway (2014)
26. Zhao, Z., Mosler, B., Braun, T.: Performance evaluation of opportunistic routing protocols: a framework-based approach using OMNeT++. In: Proceedings of the 7th Latin American Networking Conference, pp. 28–35. ACM, Medellin (2012)

# Chapter 15
# openDSME: Reliable Time-Slotted Multi-Hop Communication for IEEE 802.15.4

**Florian Kauer, Maximilian Köstler, and Volker Turau**

## 15.1 Medium Access in Wireless Multi-Hop Networks

Industrial applications of wireless networks call for high reliability, even under the influence of high traffic load or external interference. This requirement is hard to fulfill with conventional radio technology like IEEE 802.15.4 [7] which is primarily used to build energy-efficient multi-hop networks to connect sensors and actuators. One of the main causes is the use of contention-based medium access such as Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). With CSMA/CA, the medium is sensed for an ongoing transmission and a transmission only takes place when no concurrent transmission is detected. Carrier sensing is, however, unreliable in multi-hop networks, especially in hidden node situations as depicted in Fig. 15.1 where a disturbing transmission cannot be sensed by another transmitter. This leads to collision of transmissions and thereby to packet loss. Several mechanisms, such as random backoff and retransmissions, are available to lower the probability of packet loss, but they cannot solve the fundamental problem. Also, slotted CSMA/CA is suggested to minimize overlapping transmissions, but the expectations have not been met [18]. The influence of the hidden node problem on the performance is analyzed, for instance, in [15]. A more in-depth analytical analysis that also takes the simultaneous retransmission problem into account is available in [14].

A different approach for medium access is Time Division Multiple Access (TDMA). For this, the time domain is divided into slots and it is negotiated beforehand which node shall be allowed to transmit when. If this is done correctly, no

F. Kauer (✉) · M. Köstler · V. Turau
Institute of Telematics, Hamburg University of Technology, Hamburg, Germany
e-mail: florian.kauer@tuhh.de; maximilian.koestler@tuhh.de; turau@tuhh.de



A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_15

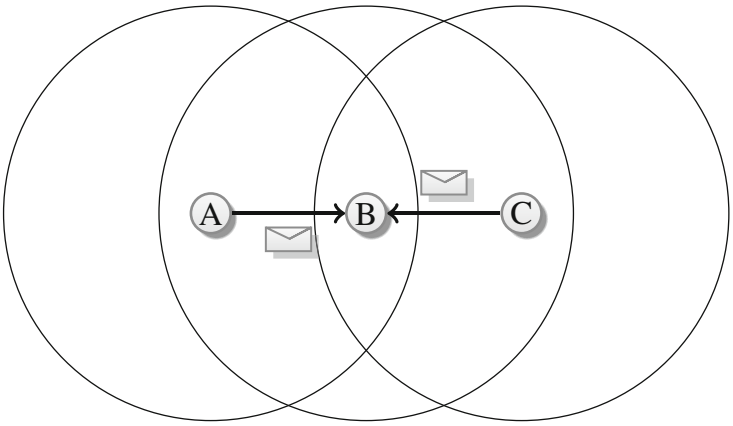



**Fig. 15.1** Hidden node problem

collisions between node transmissions are possible in the network. However, even if a properly constructed TDMA approach is safe against packet collisions within the same network, transmissions by other transmitters, e.g., IEEE 802.11 access points, could disturb the network. Therefore, it is also advised to utilize multiple frequency channels to sidestep external interferences, which are called Frequency Division Multiple Access (FDMA). Both approaches are the basis for two amendments of the IEEE 802.15.4 standard described in [6], namely, Time-Slotted Channel Hopping (TSCH) and the Deterministic and Synchronous Multi-Channel Extension (DSME), which were later integrated into the 2015 revision of the standard [7]. Starting from the 2011 standard edition [5], time synchronization via beacons was used for slotted CSMA/CA and the scheduling of time slots, but only for single-hop (star) topologies. It is also used in the Low Latency Deterministic Network (LLDN) approach of IEEE 802.15.4e not included in the 2015 revision. In contrast to these methods, TSCH and DSME can also be used in multi-hop networks. With this step, a variety of problems must be solved, which will be presented in Sect. 15.3. DSME already includes many procedures to build scalable multi-hop networks, while TSCH only standardizes a basic set of primitives that can be assembled to a functional data link layer. TSCH requires more standardization effort (ongoing under the label Internet Protocol Version 6 over the Time-Slotted Channel Hopping mode of IEEE 802.15.4e (6TiSCH)) to facilitate the features that DSME already provides, but could be used more flexibly. A comprehensive description of both approaches can be found in [3], while detailed performance evaluations of DSME are available in [2, 8]. In [12], DSME is analyzed using both OMNeT++ and the FIT/IoT-LAB testbed [1] with the implementation described in this chapter.

## 15.2 openDSME

To provide a research platform for experimenting with IEEE 802.15.4 DSME, openDSME was developed as the first open-source implementation of DSME.[1] It is implemented in C++ and follows a multi-platform approach. It can be integrated in OMNeT++ as the primary simulation and development environment of openDSME, but also in a host operating system such as **Contiki** [4] or **CometOS** [17]. This makes it possible to execute the same scenarios in the simulator as well as on wireless hardware. Since it is very cost- and time-intensive to deploy a testbed, using OMNeT++ is very useful for achieving short development cycles, convenient debugging, and easy testing of new ideas without the need of specialized hardware. Furthermore, the experiments can be accurately reproduced.

With this multi-platform approach, the same software that has already proven to provide the anticipated functionality in the simulator can be used with wireless hardware and the testing can then focus on the platform-specific peculiarities. Porting is easily possible as long as the platform offers a basic task-scheduling service and interfaces to an IEEE 802.15.4 transceiver (real or simulated). The Medium Access Control (MAC) is implemented in software, including the generation of acknowledgments. On the upside, this allows for a very flexible implementation. Especially under consideration of inefficient hardware MAC layers [19], this is a large bonus and furthermore allows for easy adaptation to other platforms. On the downside, for hardware with little computational power, timing issues become very relevant. DSME poses real time requirements for delays such as the maximum wait time for an Acknowledgment (ACK). For OMNeT++, however, this is not an issue, because the simulated time is decoupled from the wall clock time.

For OMNeT++, openDSME is a data link layer that communicates with the network layer and the physical layer via messages, while the INET Framework (see Chap. 2) is used to provide the upper and lower layers as shown in Fig. 15.2. The implementation can transparently replace the existing IEEE 802.15.4 data link layer of INET without any special handling by the other layers.

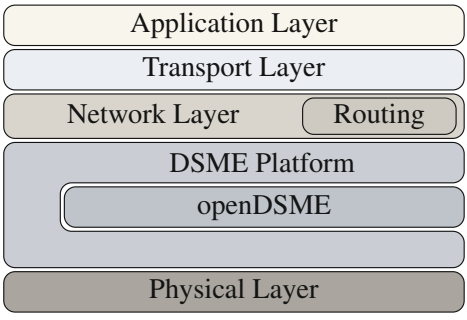



**Fig. 15.2** Integration of openDSME in the network stack

[1] Github repository of openDSME: https://github.com/opendsme.

## 15.3 Deterministic and Synchronous Multi-Channel Extension

This section explains the basics of DSME. The complexity of multi-hop TDMA is considerably higher than for CSMA/CA due to the need for archiving distributed consensus about the schedule. For example, while two nodes have to coordinate their transmissions, they have to prevent other nodes in the neighborhood from transmitting at the same time and frequency. Still, a reuse in other parts of the network is possible to increase the throughput. The following list gives a short overview of the features that are required to build a scalable TDMA-based data link layer. The remainder of the section shortly explains how the most important features are implemented in IEEE 802.15.4 DSME.

- Distributed synchronization of time with an accuracy in the range of milliseconds (cf. Sect. 15.3.1).
- Flexible network formation to allow for joining and leaving of nodes.
- Slot allocation and deallocation (cf. Sect. 15.3.2):
  - On-demand and parallel to normal network operation.
  - Decentralized to avoid the overhead of a centralized slot assignment.
  - Avoidance of slot collisions while maintaining spatial reuse.
  - Adaption to changing channel conditions.
- Mitigation of external interferences.
- Flexible scheduling to adapt to a changing topology or fluctuating traffic (cf. Sect. 15.3.3).

In DSME, time is split into slots of equal length. Figure 15.3 depicts the DSME slot structure. The slot length can be modified with the Superframe Order (SO) parameter. 16 time slots together form a superframe. The first one is reserved for beacons that are used for time synchronization. It is followed by the Contention Access Period (CAP) of 8 slots. In this phase, messages can be exchanged with conventional CSMA/CA. The advantage is that no previous slot assignment has to take place, but on the other hand, packet collisions are possible in the CAP. Therefore, it is primarily used for management traffic, i.e., traffic to prepare the

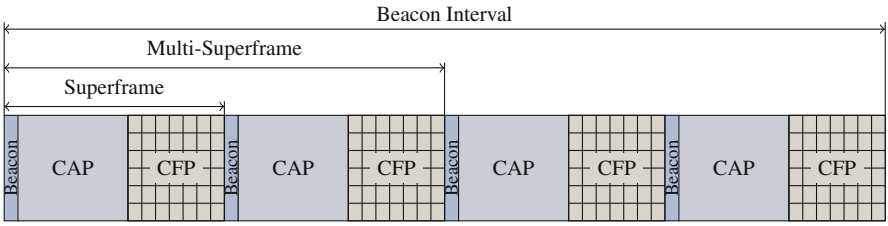



**Fig. 15.3** Basic structure of IEEE 802.15.4 DSME

communication in the Contention-Free Period (CFP). In the CFP phase, Guaranteed Time Slots (GTSs) are assigned to at most one transmitter and receiver pair. This assignment is permanent until the slot is deallocated again or an error causes a timeout.

In principle, only seven distinct GTSs could be assigned within a neighborhood to avoid concurrent transmissions. Since this might be too few, assigning parallel GTSs on different frequency channels is possible. Furthermore, this approach allows avoiding frequency channels with external disturbances, called channel adaption. A second option available in DSME is channel hopping, where the used frequency channel rotates over the available channels in a given channel hopping sequence.

Since this might still result in too few distinct GTSs, especially in dense networks, multiple superframes can be combined to the so-called multi-superframe. The only difference now is that the GTS in one superframe can be assigned to a different transceiver/receiver pair than the one of another superframe. The number of superframes per multi-superframe is always a power of two and can be specified by the Multi-Superframe Order (MO) parameter. The multi-superframes also allow a special CAP reduction mode. If this mode is activated, only the CAP in the first superframe of a multi-superframe is active, and the other CAPs can be filled with GTSs. This is mainly useful to increase the throughput because more GTSs are available per time. The disadvantage is a lower amount of time that can be used for management traffic. Depending on the network this could lead to problems such as colliding management traffic [10] but is usually advised if the network is not of very high density and high throughput is important.

Finally, there is the beacon interval that determines how many distinct beacon slots are available. It is adjusted by the Beacon Order (BO) parameter and is important for a functional time synchronization and network association.

### *15.3.1 Beacons for Time Synchronization and Network Association*

An essential requirement for TDMA is a common notion of time in the network. For this, special messages, the beacons, are sent out at regular intervals at exactly predefined points in time that indicate the beginning of a superframe. All other time slots are relative to the beginning of the message so that all nodes have a common notion of time. On hardware, it is of high importance to accurately determine when the message reception has started and this is usually done by a timestamping mechanism provided by the transceiver. This approach is different from TSCH, where the time synchronization is performed at every message exchange using the acknowledgment.

The beacons are also used for a second purpose, namely, the network association process. In a real-world scenario, it is not necessarily known beforehand which nodes are part of the network and how the network should be configured. Therefore,

a network association procedure is required. For this, one node, usually the one with a connection to another network (the gateway node), is specified as the Personal Area Network (PAN) coordinator. The beacon sent out by the PAN coordinator contains all relevant information so that other nodes can join the network.

Thus, to join the network, a previously unassociated node starts to scan for beacons. This can take place on a single, predefined frequency channel or successively on all available channels. Once the node receives a beacon, it synchronizes its own clock to this beacon. Now the node will wait for the next CAP and send out an association message that will eventually be answered by the coordinator. Later beacons are required for regular realignment in order to compensate clock drift. This interval could be increased by regulating the clock frequency, too, as described in [11].

One transceiver cannot reach all nodes directly in larger multi-hop networks. Multiple nodes must hence send out beacons. For this, a node can become a coordinator itself. There is always only one PAN coordinator in the network, but there might be multiple coordinators. If the network is not very dense and the beacon interval is long enough, i.e., enough distinct beacon slots are available, it might be feasible that every node takes on the role of a coordinator. Otherwise, a more complex selection algorithm is necessary.

Once a node decides to become a coordinator, it selects a beacon slot and sends out a beacon allocation notification to inform the neighbors about the decision. To avoid that this node selects an already allocated beacon slot, allocated beacon slots are recorded in the beacon bitmap, and furthermore, a beacon collision notification can be sent to signal that a beacon allocation should be withdrawn.

While a node might receive beacons from multiple transceivers, it should only resynchronize with the beacons sent out by its synchronization parent. Otherwise, loops of nodes could emerge that do not synchronize to the rest of the network and thus drift out of the common notion of time.

### *15.3.2 GTS Management*

After a node is associated and synchronized, it can communicate with other nodes in the network. The openDSME implementation will automatically decide if an upper layer message should be sent in the CAP or the CFP. Broadcast messages will be sent in the CAP, while messages for a single node will trigger a GTS allocation to enable communication in the CFP. In the beginning, the network layer will usually receive and send broadcast messages in the CAP to build a routing table. Once routing paths are known, application layer messages can be directed to single nodes.

Before communication can take place in the CFP, a GTS has to be reserved. This is solved with a three-phase slot allocation handshake between the two involved nodes of the later GTS communication. The second and third messages are sent

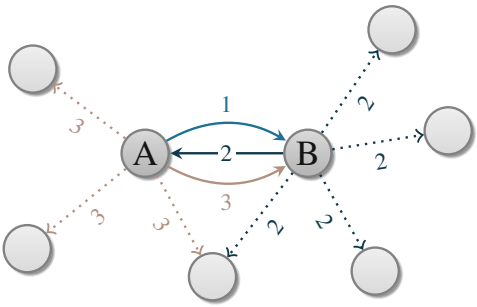



**Fig. 15.4** GTS allocation handshake

as link-layer broadcasts and will therefore also be received by all nodes in the neighborhood. The procedure, as depicted in Fig. 15.4, consists of the following three steps:

1. The initiator of the handshake (A) sends a DSME GTS request to the other node (B) indicating its preferred slot and all other free slots.
2. Node B selects a matching slot that is free for both nodes and responds with a DSME GTS response. Since it is sent as a broadcast, all nodes in this neighborhood of B are informed about the decision.
3. Node A again sends back a message, the DSME GTS Notify, and thus also informs the neighborhood of A.

If the handshake was conducted successfully, A and B insert the allocated slot in their Allocation Counter Table (ACT) and all nodes in the neighborhood of A and B mark the allocated slot in their Slot Allocation Bitmap (SAB) as in use to avoid that it is reused in later allocations. Unfortunately, errors could occur in this phase, especially if messages are not received properly. To mitigate the problem, neighbors can send a duplicated allocated notification (as a DSME GTS request) in case they detect a conflict. Further details about this problem can be found in [10].

#### 15.3.2.1 Slot Expiration

As long as a GTS is in use, it does not expire and does not have to be reallocated. However, unused slots or slots with a lot of external interferences will eventually expire. For this, the receiver maintains an idle counter in the ACT that counts how many times in a row no message was received in a GTS. Furthermore, the sender maintains a counter that is incremented if no acknowledgment was received for a transmission. If either of these counters exceeds the value of `macDsmeGtsExpirationTime`, the associated node will start a deallocation that matches the GTS allocation handshake, but is flagged as deallocation.

### *15.3.3 Scheduling*

An important aspect that determines the performance of the network is scheduling, that is, which slots and how many of them should be assigned between which nodes. A trivial approach would be to allocate a slot in both directions for every pair of neighbors. This would, however, be a waste of resources because most pairs will only exchange few or no traffic. Therefore, it is advised to take the routing of messages into account and assign slots to links based on the traffic volume. For example, in a data-collection scenario, a lot of traffic will emerge near the PAN coordinator, so the neighbors of the PAN coordinator benefit from having multiple slots towards the PAN coordinator. The scheduling is, however, not specified in IEEE 802.15.4 DSME itself. However, it is still part of the openDSME implementation to provide a seamless integration and is described in Sect. 15.4.1 in detail.

## 15.4 Implementation

We now present the implementation of openDSME in more detail to enable an efficient usage as well as to give a guide for future extensions. The architecture of openDSME is introduced by explaining how a message is passed through the stack. Important auxiliary modules are presented and finally the concept of the platform abstraction in openDSME is introduced.

### *15.4.1 The openDSME Stack*

Figure 15.5 shows the structure of the most important modules. It can be coarsely separated in the `DSMELayer` that is the actual implementation of IEEE 802.15.4 DSME and the `DSMEAdaptionLayer` that provides helpers that are not specified in the standard, but are important for a seamless integration in a network stack. These layers are interconnected by the standardized Medium Access Control Common Part Sublayer (MCPS) and Medium Access Control Sublayer Management Entity (MLME) Service Access Points (SAPs). The communication with the upper and lower layers is done by means of the platform-specific `DSMEPlatform` as presented in Sect. 15.4.3.

If the network layer wants to send a message, it forwards this message to the `MessageHelper` that decides if any actions have to be performed before the message can be sent. If the node is not even associated, a scan will be issued via the `ScanHelper` and after a successful synchronization, an association will take place via the `AssociationHelper`.

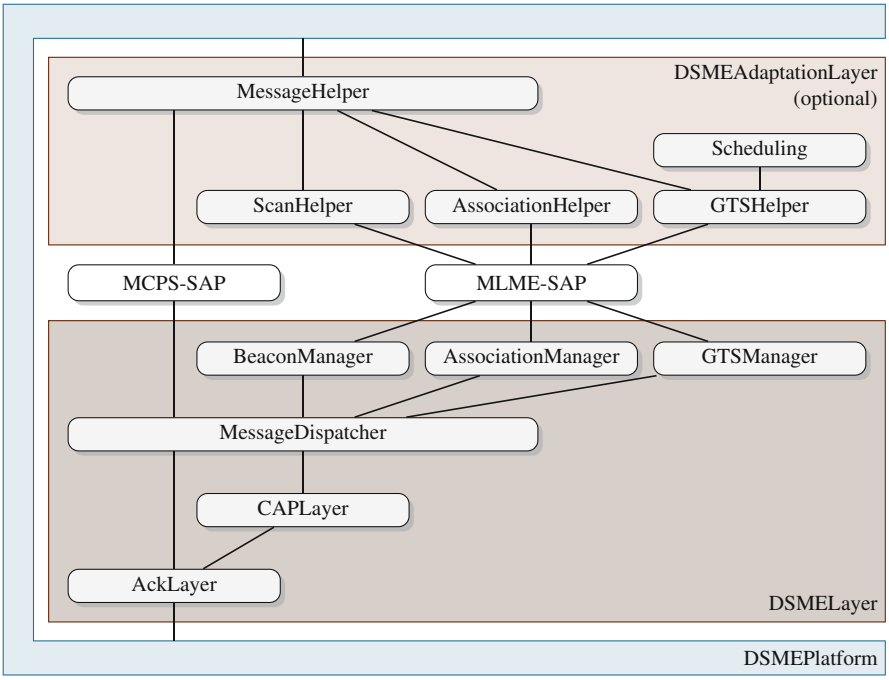



**Fig. 15.5** Software structure of the openDSME implementation

The `GTSHelper` is informed about the incoming message and will consult the scheduling to find out if an allocation or deallocation is needed. A slot is automatically allocated if a message arrives towards a node to which no GTS is currently allocated. The current implementation does not try to optimize the delay and thus randomly selects a slot out of the range of available slots. The number of incoming messages during a multi-superframe is counted and this measure is used to control the number of required GTS. An important goal at this point is to achieve a stable system, so it must be avoided that high network jitter leads to ongoing allocation and deallocation of slots. This would congest the CAP with severe consequences. Up to now two approaches are implemented, a Proportional, Integral, and Derivative (PID) controller and a moving average filter with hysteresis. Still, this part leaves many questions unanswered and opens challenging opportunities for future research.

The `MessageHelper` then forwards the data messages via the MCPS interface to the `MessageDispatcher`. It also specifies if the message shall be sent in the CAP or the CFP. Currently, all link-layer broadcasts of data packets are sent in the CAP and all other data packets are sent in the CFP. The `MessageDispatcher` is responsible for maintaining the DSME time structure and to send the correct messages at the given point in time and the corresponding frequency channel.

Messages to be sent in the CAP are forwarded to the `CAPLayer` that is responsible for the CSMA/CA mechanism of the CAP. As soon as a message shall be sent, it is forwarded to the `AckLayer` that will send the message to the physical layer and also manages the reception of the acknowledgment, if requested.

The `AckLayer` is also responsible for receiving messages from the physical layer and sending out acknowledgments. The message is then forwarded to the `MessageDispatcher`. Data messages will be sent to the upper layer via the MCPS interface and the `MessageHelper`. DSME specific packets, such as beacons, association messages, or GTS commands, are dispatched to the respective manager modules.

### *15.4.2 Auxiliary Modules*

In openDSME, several auxiliary modules are included that are not part of the stack. The most important ones are now presented subsequently.

The `NeighborQueue` is responsible for maintaining a list of the neighbors as well as to queue the messages to be sent. The neighbor list itself is implemented as red–black tree to allow for an efficient retrieval by the MAC address. Instead of using the map implementation included in the C++ Standard Template Library (STL), an own implementation is provided to allow the usage of openDSME on hardware platforms without default STL support such as Atmel microcontrollers. A neighbor is added to this list when the first message is issued by an upper layer to be sent to this neighbor. The queue itself is implemented as a set of linked lists, one for every neighbor and one for maintaining a set of free slots. This allows to operate on a fixed amount of message slots and thus avoids the need of dynamic memory. Still, it provides the oldest message pending for a given neighbor in $O(1)$ (after getting the neighbor iterator in $O(logn)$, where $n$ is the number of neighbors). This is especially important for DSME, because the message to be sent in a slot does not only depend on the age, but also on the neighbor that listens in a given slot.

The `DSMEAllocationCounterTable` stores the currently allocated slots and is also implemented as a red–black tree. This allows for efficient access to the state of every slot without the need to store the state of the idle slots explicitly. This would waste a lot of memory, because, especially when using a large MO, usually a fraction of the slots per multi-superframe are actually active for a given node. Related is the `DSMESlotAllocationBitmap` that stores the slots occupied by all neighbors and the `BeaconBitmap` that stores the occupied beacon slots in the neighborhood.

The operation of the modules in the stack itself, such as the `GTSManager`, is specified in the form of finite state machines. For this, the `DSMEBufferedMultiFSM` class is provided that allows to spawn multiple parallel transactions (for example, if multiple GTS transactions run in parallel) and can buffer pending events. Other modules such as the `AckLayer` do not require parallel transactions and thus only use a subset of the provided functionality.

The `TimerMultiplexer` multiplexes single timer to the different timers that are used in the `DSMEEventDispatcher` to dispatch the various events required for the operation of openDSME. This includes the slot event that fires at every slot (in use) and a pre-slot event that is fired shortly before the slot event to prepare the transceiver (e.g., tune it to the correct channel). Furthermore, a Carrier Sense Multiple Access (CSMA) timer is used for controlling the backoffs in the CAP and an ACK timer is used to notify a lost acknowledgment. All these events are derived from a single 62.5 kHz timer that has to be provided by the platform. This frequency corresponds to the duration of a IEEE 802.15.4 symbol duration of 16 µs and all other relevant times can be specified as multiples of the symbol duration.

### *15.4.3 openDSME Platform Abstraction*

A platform has to specify the following three interfaces (marked with a leading `I`). For OMNeT++ these are already provided, but for porting openDSME to a new (simulation or hardware) platform, these are most relevant.

The `IDSMERadio` interface provides the access to the (simulated or actual) transceiver. This includes setting the channel or starting a Clear Channel Assessment (CCA), but also the handover of messages in both directions. Because DSME requires an accurate timing, especially for the beacons, the transmission of messages is a two phase process where the message is first *prepared*, i.e., written to the transceiver, and at the correct point in time, a *sendNow* instruction is issued.

The second interface is the `IDSMEMessage` that is a container for the message content. This approach allows the platform to use its own representation of messages. For hardware, one usually wants to implement a fixed pool of fixed sized messages to avoid the use of dynamic memory, while for OMNeT++, it is possible to wrap an INET packet so there is no need to repack the message content of upper layers.

All other relevant interfaces are provided by `IDSMEPlatform`. This includes, for example, an interface to the timer used by the `TimerMultiplexer` or the generation of random numbers.

In `inet-dsme`, the `DSMEPlatform` provides most of the platform-specific code that also includes the interface to the upper layers and the initialization. From the point of view of OMNeT++, this class is a monolithic module that implements the `inet::IMACProtocol` interface. Since it is implemented as a single OMNeT++ module the message exchange between the different parts of openDSME is not traceable as OMNeT++ messages as it is the case, for instance, in [13], but it enables a much easier transition to hardware implementations.

## 15.5 Tutorial

The following tutorial explains the usage of openDSME using a multi-hop data-collection scenario. The example demonstrates how to compare the performance of CSMA/CA and DSME. In the given example, 18 nodes are statically arranged in concentric circles around a sink node. This topology resembles an exemplary application of a wireless multi-hop network in a solar tower power plant [16] and is challenging due to the high number of hidden node situations. Every node, except the sink, sends data packets with exponentially distributed sending intervals with mean 500 ms. For the network layer, Greedy Perimeter Stateless Routing (GPSR) is used, to provide a maximum scalability also for larger networks [9]. This tutorial demonstrates how to specify a scenario, how to run it, and, finally, how to analyze the obtained results.

In order to have a go at the discussed application, OMNeT++ 5.3 and the openDSME framework source code are required. Installation instructions are available online.[2] After the installation, the scenario is set up with the configuration file shown in Listing 15.1. It can also be found at *inet-dsme/simulations/example.ini*.

**Listing 15.1** The *example.ini* configuration file used in this tutorial

```
[General]
network = Net802154

# Speed up
**.radioMedium.rangeFilter = "interferenceRange"
**.host[*].wlan[*].radio.*.result-recording-modes = -histogram,-vector

# Mobility
**.numHosts = 19
**.host[*].mobilityType = "StaticConcentricMobility"

# Traffic generator
**.host[*].trafficgen.packetLength = ${packetLength = 75B}
**.host[*].trafficgen.sendInterval = exponential(0.5s)
**.host[*].trafficgen.startTime = 30s
**.host[*].trafficgen.warmUpDuration = 190s
**.host[*].trafficgen.coolDownDuration = 15s
**.host[*].trafficgen.continueSendingDummyPackets = true
**.host[*].trafficgen.destAddresses = "host[0](modulepath)"

**.host[0].trafficgen.numPackets = 0
**.host[1..].trafficgen.numPackets = 100

# Link-Layer
[Config CSMA]
**.host[*].wlan[*].macType = "Ieee802154NarrowbandMac"
**.host[*].wlan[*].mac.queueLength = 30
**.host[*].wlan[*].mac.macMaxFrameRetries = 7 # max. value for IEEE 802.15.4

[Config DSME]
**.host[*].wlan[*].macType = "DSME"
**.host[0].wlan[*].mac.isPANCoordinator = true
```

[2] openDSME Github repository: https://github.com/openDSME/inet-dsme.

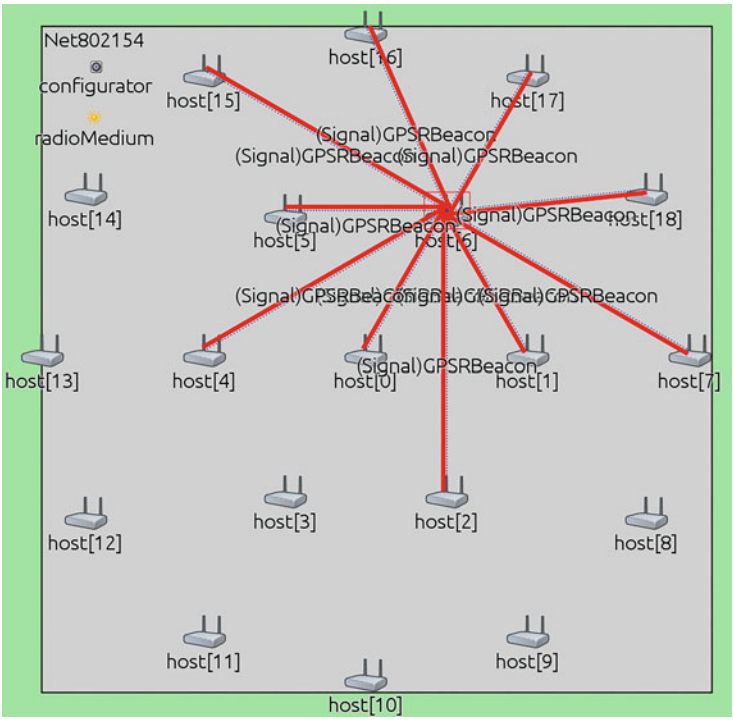



**Fig. 15.6** Static topology of concentric circles

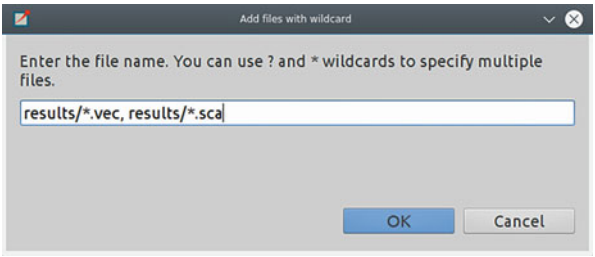



**Fig. 15.7** Adding wildcards for result files

Now the simulation can be started in OMNeT++ **Qtenv** that visualizes the topology as shown in Fig. 15.6. When running the simulation, we see how messages are exchanged, in the figure, for example, a GPSR beacon.

Increase the simulation speed by switching to the express mode. After a warm-up phase, every node sends 100 messages and then the simulation stops automatically after a short cool-down phase. When both configurations (for `CSMA` and `DSME`) were executed successfully, the collected data can be analyzed by creating a new analysis file. To do that, right-click on the `inet-dsme` project and select *New*→*Analysis File (anf)*. Afterwards, use *Add Wildcard...* as shown in Fig. 15.7.

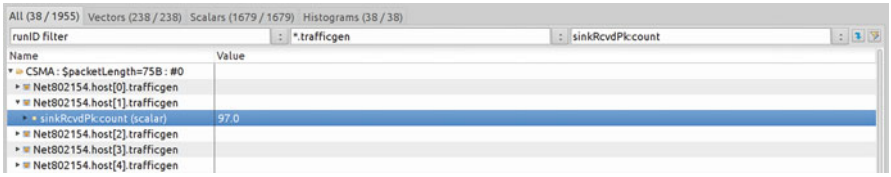



**Fig. 15.8** Filter result file for `sinkRcvdPk:count`

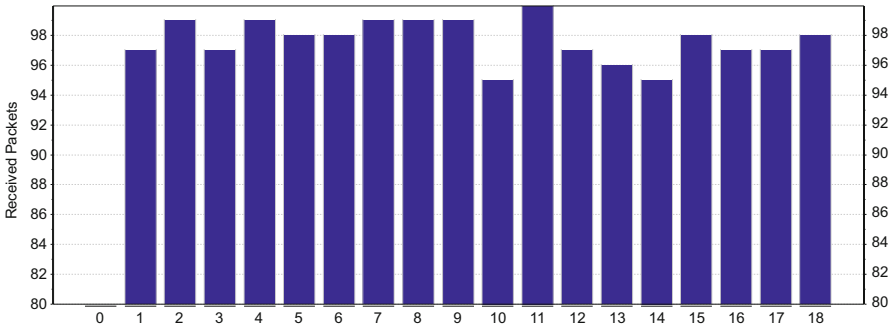



**Fig. 15.9** Number of received packets from every node for `CSMA`

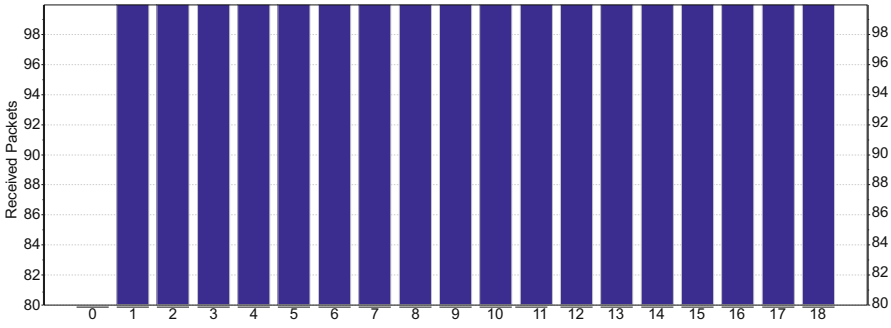



**Fig. 15.10** Number of received packets from every node for `DSME`

Since we are interested in the reliability of the message transmission, we filter for the `sinkRcvdPk:count` scalar in the `trafficgen` module as shown in Fig. 15.8. It gives the number of packets successfully received from the given node. The value will be close to 100 since this is the total amount of packets sent by every node. Plotting this value for `CSMA` and `DSME` via the context menu of the entries and choosing the range 80–100 for the $y$-axis will yield Figs. 15.9 and 15.10.

While some packets are lost in the CSMA/CA scenario, no packets are lost for DSME, demonstrating the ability of DSME to avoid collisions. Of course, when lowering the `**.host[*].trafficgen.sendInterval` parameter (line 14 in Listing 15.1) or increasing the network size with the `**.numHosts` parameter

(line 9), the amount of traffic will increase and eventually the DSME nodes will no longer be able to process all messages and queue drops will occur.

The reader is invited to test the influence of the parameters on the respective packet delivery ratios. It is, for instance, insightful to conduct a parameter study to plot the packet delivery ratio over the send interval for different setups of the DSME superframe structure as it is done in [12]. Also, the development of different scheduling techniques[3] and their impact on communication reliability and timeliness will lead to interesting research to improve the applicability of DSME.

## 15.6 Conclusion and Future Work

In this chapter, openDSME, an open-source implementation of IEEE 802.15.4 DSME, was presented. By using DSME, packet collisions can be avoided, leading to a higher reliability suitable for industrial applications. The presented implementation supports the OMNeT++ simulator and can be used for extensive simulative evaluations, but can be equally used for deploying real-world sensor networks, for example, by using the **Contiki** operating system.

Promising directions for future research include algorithms to assign time slots to nodes, for instance, in a way that minimizes the End-to-End (E2E) delay. Also, an intelligent channel adaption that explicitly takes external interferences into account would be beneficial to improve the resilience in a congested frequency spectrum.

**Acknowledgements** The authors would like to thank everyone who has contributed to the development of openDSME, starting with Tobias Lübkert for the first functional OMNeT++ DSME implementation, Sandrina Backhauß (now Köstler) for mastering the complex data structures, Axel Neuser for the **Contiki** port, and Florian Meyer for the channel hopping and CAP reduction functionality.

## References

1. Adjih, C., Baccelli, E., Fleury, E., Harter, G., Mitton, N., Noel, T., Pissard-Gibollet, R., Saint-Marcel, F., Schreiner, G., Vandaele, J., Watteyne, T.: FIT IoT-LAB: a large scale open experimental IoT testbed. In: IEEE 2nd World Forum on Internet of Things (WF-IoT). IEEE, Piscataway (2015). https://doi.org/10.1109/WF-IoT.2015.7389098
2. Alderisi, G., Patti, G., Mirabella, O., Bello, L.L.: Simulative assessments of the IEEE 802.15.4e DSME and TSCH in realistic process automation scenarios. In: Proceedings of the 13th International Conference on Industrial Informatics (INDIN), pp. 948–955. IEEE, Piscataway (2015). https://doi.org/10.1109/INDIN.2015.7281863

[3]For adding new scheduling techniques, a generic interface and some examples are provided in the *dsmeAdaptionLayer/scheduling* folder of openDSME.

3. De Guglielmo, D., Brienza, S., Anastasi, G.: IEEE 802.15.4e: a survey. Comput. Commun. **88**, 1–24 (2016). https://doi.org/10.1016/j.comcom.2016.05.004
4. Dunkels, A., Grönvall, B., Voigt, T.: Contiki—a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of the 29th International Conference on Local Computer Networks, pp. 455–462. IEEE Computer Society, Los Alamitos (2004). https://doi.org/10.1109/LCN.2004.38
5. IEEE Standards Association: IEEE standard for local and metropolitan area networks–part 15.4: low-rate wireless personal area networks (LR-WPANs). IEEE Std 802.15.4–2011–Revision of IEEE Std. 802.15.4$^{TM}$-2006. The Institute of Electrical and Electronics Engineers, Inc., Piscataway (2011). https://doi.org/10.1109/IEEESTD.2011.6012487
6. IEEE Standards Association: IEEE standard for local and metropolitan area networks–part 15.4: low-rate wireless personal area networks (LR-WPANs) amendment 1: MAC sublayer. IEEE Std 802.15.4e-2012–Amendment to IEEE Std 802.15.4$^{TM}$-2011. The Institute of Electrical and Electronics Engineers, Inc., Piscataway (2012). https://doi.org/10.1109/IEEESTD.2012.6185525
7. IEEE Standards Association: IEEE standard for low-rate wireless networks. IEEE Std 802.15.4-2015–Revision of IEEE Std. 802.15.4$^{TM}$-2011. The Institute of Electrical and Electronics Engineers, Inc., Piscataway (2016). https://doi.org/10.1109/IEEESTD.2016.7460875
8. Jeong, W.C., Lee, J.: Performance evaluation of IEEE 802.15.4e DSME MAC protocol for wireless sensor networks. In: Proceedings of the 1st IEEE Workshop on Enabling Technologies for Smartphone and Internet of Things (ETSIoT), pp. 7–12. IEEE, Piscataway (2012). https://doi.org/10.1109/ETSIoT.2012.6311258
9. Karp, B., Kung, H.T.: GPSR: greedy perimeter stateless routing for wireless networks. In: Proceedings of the 6th Annual International Conference on Mobile Computing and Networking, MobiCom '00, pp. 243–254. ACM, New York (2000). https://doi.org/10.1145/345910.345953
10. Kauer, F., Köstler, M., Lübkert, T., Turau, V.: Formal analysis and verification of the IEEE 802.15.4 DSME slot allocation. In: Proceedings of the 19th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM). ACM, New York (2016). https://doi.org/10.1145/2988287.2989148
11. Kauer, F., Kallias, E., Turau, V.: A dual-radio approach for reliable emergency signaling in critical infrastructure assets with large wireless networks. Int. J. Crit. Infrastruct. Prot. **21**, 33–46 (2018). https://doi.org/10.1016/j.ijcip.2018.02.002
12. Kauer, F., Köstler, M., Turau, V.: Reliable wireless multi-hop networks with decentralized slot management: an analysis of IEEE 802.15.4 DSME (2018). http://arxiv.org/abs/1806.10521. Preprint
13. Kirsche, M., Schnurbusch, M.: A new IEEE 802.15.4 simulation model for OMNeT++/INET. In: Proceedings of the 1st OMNeT++ Community Summit (2014). http://arxiv.org/abs/1409.1177
14. Meier, F., Turau, V.: An analytical model for fast and verifiable assessment of large scale wireless mesh networks. In: Proceedings of the 11th International Conference on the Design of Reliable Communication Networks (DRCN). IEEE, Piscataway (2015). https://doi.org/10.1109/DRCN.2015.7149011
15. Pešović, U., Mohorko, J., Benkič, K., Čučej, Ž.: Effect of hidden nodes in IEEE 802.15.4/ZigBee wireless sensor networks. In: Proceedings of the 17th Telecommunications Forum (TELFOR), pp. 161–164 (2009)
16. Pfahl, A., Randt, M., Meier, F., Zaschke, M., Geurts, C.P.W., Buselmeier, M.: A holistic approach for low cost heliostat fields. In: Proceedings of the 20th International Conference on Concentrated Solar Power and Chemical Energy Technologies (SolarPACES). Peking, China (2014). https://doi.org/10.1016/j.egypro.2015.03.021
17. Unterschütz, S., Weigel, A., Turau, V.: Cross-platform protocol development based on OMNeT++. In: Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS), pp. 278–282. ICST, Brussels (2012). https://dl.acm.org/citation.cfm?id=2263063

18. Wang, F., Li, D., Zhao, Y.: Analysis and compare of slotted and unslotted CSMA in IEEE 802.15.4. In: Proceedings of the 5th International Conference on Wireless Communications, Networking and Mobile Computing, pp. 1–5. IEEE, Piscataway (2009). https://doi.org/10.1109/WICOM.2009.5303580
19. Weigel, A., Turau, V.: Hardware-assisted IEEE 802.15.4 transmissions and why to avoid them. In: Proceedings of the 8th International Conference on Internet and Distributed Computing Systems (IDCS 2015), pp. 223–234. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-23237-9_20

# Index





A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5