Journal of Network and Systems Management (2020) 28:660–692
https://doi.org/10.1007/s10922-020-09524-1

Adaptive Telemetry for Software-Defined Mobile Networks

Luca Cominardi[1] · Sergio Gonzalez-Diaz[2] · Antonio de la Oliva[2] · Carlos J. Bernardos[2]

Received: 24 May 2019 / Revised: 19 December 2019 / Accepted: 5 March 2020 /
Published online: 19 March 2020
© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract
The forthcoming set of 5G standards will bring programmability and flexibility to levels never seen before. This has required introducing changes in the architecture of mobile networks, enabling different features such as the split of control and data planes, as required to support the rapid programming of heterogeneous data planes. Software Defined Networking (SDN) has emerged as a basic toolset for operators to manage their infrastructure, as it opens up the possibility of running a multitude of intelligent and advanced applications for network optimization purposes in a centralized network controller. However, the very basic nature that makes possible this efficient management and operation in a flexible way—the logical *centralization*—poses important challenges due to the lack of proper monitoring tools, suited for SDN-based architectures. In order to take timely and right decisions while operating a network, centralized intelligence applications need to be fed with a continuous stream of up-to-date network statistics. However, this is not feasible with current SDN solutions due to scalability and accuracy issues. This article first analyzes the monitoring issues in current SDN solutions and then proposes a telemetry framework for software defined mobile networks capable of adapting to the various 5G services. Finally, it presents an experimental validation that shows the benefits of the proposed solution at alleviating the load on the control and data planes, improving the reactiveness to network events, and providing better accuracy for network measurements.

Keywords 5G · Mobile · Network · SDN · OpenFlow · Monitoring · Telemetry · OAM · Experimental

✉ Luca Cominardi
luca.cominardi@adlinktech.com

Extended author information available on the last page of the article

1 Introduction

The fifth-generation of mobile networks demands high flexibility and adaptability from the infrastructure so as to allow operators to quickly provision and operate a wide spectrum of services with very distinct requirements [1]. This demands levels of programmability and flexibility not yet seen, triggering the adoption of architectures based on virtualization and separation of control and data planes. Software Defined Networking (SDN) [24], notably in the form of OpenFlow protocol [28], is one of the main approaches adopted by mobile operators towards achieving the desired flexibility in the network.

OpenFlow-based SDN architectures potentially enable optimal management of the network, as long as the applications operating it are timely provided with a rich set of statistics collected from the underlying infrastructure. These statistics have to be available at the network controller, which is a (logically) centralized entity. However, OpenFlow was originally conceived for campus and data center networks, therefore having a different set of requirements compared to mobile networks. Although the initially limited set of functionalities has been gradually extended to cover new protocols (e.g., Multiprotocol Label Switching—MPLS, Provider Backbone Bridges—PBB) and more sophisticated forwarding behaviors (e.g., packet encapsulation), monitoring support in OpenFlow is still limited compared to what is required in mobile networks [15, 18]. Considering the applicability to mobile networks, one of the areas where OpenFlow lags behind is monitoring and fault-detection [4].

To effectively and timely react to changes in the infrastructure and/or the service needs, operators need to put in place a set of constantly-active critical routines in their networks. These procedures are traditionally referred to as Operation, Administration, and Maintenance (OAM). Specifically, operation activities are undertaken to keep the network and its services up and running. Administration activities involve keeping track of resources in the network and how they are used. Maintenance activities are focused on facilitating repairs and upgrades in addition to corrective and preventive measures to make the managed network run more effectively. In the last decade, considerable effort was devoted to enrich existing transport technologies, such as MPLS by the Internet Engineering Task Force (IETF) and PBB by the Institute of Electrical and Electronics Engineers (IEEE), with a comprehensive set of OAM tools with the ultimate goal of providing a carrier grade packet-based network to operators. This effort eventually yielded to the release of two competing standards: MPLS-Transport Profile (MPLS-TP) and PBB-Traffic Engineering (PBB-TE). However, these protocols do not offer the necessary adaptability required in 5G networks because of the rigid implementation of the OAM functionalities.

OpenFlow, as currently defined, does not really support the rapid and scalable monitoring of resources. As a matter of fact, OpenFlow only permits the network controller to poll the switches for gathering simple statistics (e.g., number of packets, transmitted/received bytes, etc.) and requires any additional measurements to be directly implemented on top of the network controller. This is because of the contrasting design principles adopted by OAM and SDN:

- OAM defines *stateful* mechanisms that must be executed on the switch.
- OpenFlow defines a *stateless* forwarding model for the switch and delegates stateful logic to the controller.

As a consequence, realizing SDN-based monitoring presents significant challenges both in terms of scalability and accuracy in mobile networks. Indeed, the network controller needs to directly perform the necessary measurements on each of the numerous (up to tens of thousands) and distant (up to hundreds of kms) network nodes and links with the required precision and granularity (up to microseconds) in order to provide the necessary reliability (up to 99.9999%) to the very distinct services in 5G. To overcome those issues, current SDN-based monitoring approaches needs to be augmented with an automated communication process, namely *telemetry*, by which measurements and other data are: (i) generated and collected locally at network nodes subject to different service requirements, and (ii) transmitted to the controller for enabling an optimal network management.

The goal of this article is therefore to: (i) analyze the mismatch between current OAM tools and SDN solutions in mobile networks, and to (ii) propose a telemetry solution for software-defined mobile networks capable of adapting to the various service requirements expected in 5G. The remainder of the article is structured as follows: Sect. 2 provides an overview of SDN/OpenFlow, OAM and 5G networks for those readers not familiar with these concepts.[1] Afterward, Sect. 3 reports the analysis of the various issues in building an SDN-based monitoring in mobile networks. Next, Sect. 4 presents our solution which is experimentally evaluated in Sect. 5. Finally, Sect. 6 draws the conclusions.

2 Background and Overview

This section first briefly introduces the concept of SDN and explains how OpenFlow works. Then, it reports on the current state-of-the-art on SDN-based management and provides an overview of current OAM and OpenFlow protocols. Finally, it reports on 5G service requirements and 5G mobile network architecture, as defined by the 3GPP and the ITU-T, respectively.

2.1 SDN and OpenFlow in a Glimpse

Figure 1 shows a logical view of the SDN architecture, where the intelligence and control of SDN switches are centralized in SDN controllers. An SDN controller has the global view of the network and is capable of controlling, in a vendor-independent way, the network devices, namely SDN switches. These network devices are no longer required to implement and understand many different network protocol standards; instead, they can provide such functionality by accepting instructions from

[1] Section 2 could be skipped by the expert readers.

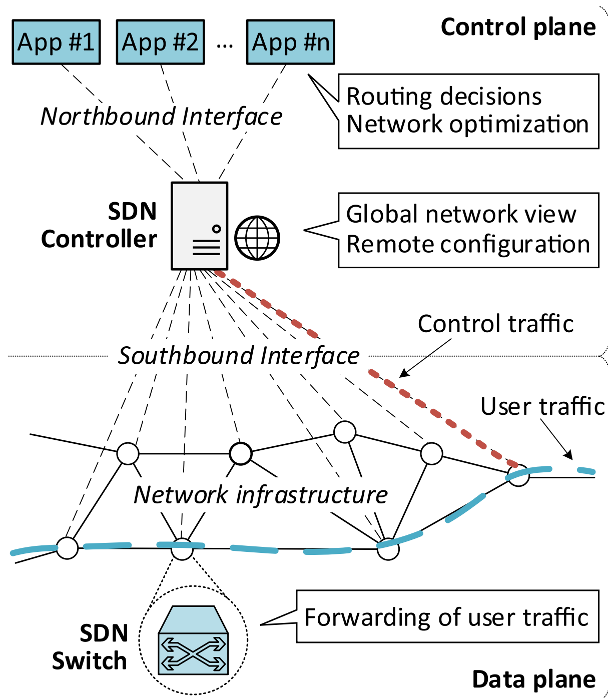


Fig. 1 SDN architecture

SDN controllers through the *southbound interface*. This yields to a significant saving in time and resources, as the network behavior can be easily controlled by programming it in the centralized controllers rather than using custom configurations in many different devices scattered across the network. The SDN controller is responsible for the maintenance of an abstract resource model of the underlying network which is then exposed to applications via the *northbound interface*, which is commonly implemented through Rest APIs.[2] Applications define the network behavior and may belong either to the network operator or to clients, the former usually having a broader scope and higher privileges than the latter.

The most widely adopted protocol for the southbound interface to control the network devices is OpenFlow,[3] which defines a *stateless* switch abstraction for packet forwarding. This means that the switch does not retain any information or state[4] about the received/transmitted packets. Therefore, the forwarding is solely based on the information contained in the packets and not on the past history. The main components of an OpenFlow-capable switch are:

[2] Note that there is no standard protocol defined for the northbound interface.

[3] This paper uses as a baseline the latest OpenFlow version, now at 1.5.1 [28].

[4] From here the term *stateless*.

- At least one physical and/or logical *port*.
- At least one *flow table* containing multiple *flow entries*.
- A *pipeline* that interconnects the ports and defines the interaction of matching packets with the flow tables.

OpenFlow also defines a communication protocol that gives access to the forwarding plane of a switch to an external controller over the network. Using this protocol, the controller can interact with the OpenFlow agent running on the switch to configure the *flow tables*, both proactively and reactively. Each *flow table* in the switch contains a set of *flow entries* that can be dynamically added, updated, or deleted by the controller. The main components of each *flow entry* are:

- *Matching fields* for matching against packet headers.
- *Instructions* allow to dynamically modify the *actions* applied to matching packets.

An example of action is the forwarding of the matching packets to a specific port. This is usually a physical port, but it may also be a logical or a reserved port. Logical ports are higher-level abstractions that may be defined in the switch using non-OpenFlow methods, e.g., link aggregation, tunnels, or loopback interfaces. Reserved ports may particularize generic forwarding actions like sending a message to the controller. This abstraction allows performing a *stateless* forwarding of packets.

2.2 Passive and Active Monitoring in a Glimpse

In order to take timely and right decisions while operating a network, centralized intelligence applications need to be fed with a continuous stream of up-to-date network statistics. Two approaches exist in the literature for retrieving the necessary data from the network: passive monitoring and active monitoring. Passive monitoring is based on the *observation* and sampling of the traffic flowing through the network. The network nodes (e.g., switches, routers) sample the incoming traffic, then they sent these samples to a central controller for being analyzed and infer the actual status of the network.

Traditional tools to implement passive monitoring for IP networks are sFlow [29], netFlow [10] and IPFIX [11]. Modern approaches overcoming the limitations of traditional tools are proposed by Sonchack et al. [31, 32], Tilmans et al. [33], and Gupta et al. [14]. These define various techniques for configuring and performing packet sampling, traffic mirroring and statistics aggregation in the network as well as the protocol for transmitting the samples to the central controller (i.e., the packet format) for further processing. By similarity, we can see the OpenFlow capability of sending packets from the data plane to the SDN controller as a building block for implementing passive monitoring in OpenFlow networks.

Examples of applications performing passive monitoring in OpenFlow are reported later in Sect. 2.3. On the other hand, active monitoring is based on the on-demand *generation* and transmission of probes over the network with the

goal of measuring specific metrics, e.g., bandwidth, latency, packet losses, etc. This kind of monitoring is commonly enabled by Operation, Administration, and Maintenance (OAM) protocols in operator networks. An overview of these protocols is provided later in Sect. 2.4.

By comparing passive and active monitoring, the former allows a simpler data plane since it does not require to maintain any state in the network. Indeed, in passive monitoring, all the information is stored and analyzed in the central controller. As a drawback, passive monitoring requires higher bandwidth on the control plane compared to active monitoring, which in contrast demands the data plane to store some state. Moreover, passive monitoring only allows detecting events that have already occurred in the network while active monitoring allows to pro-actively measure and predict potential problems in the network. A deeper analysis on this matter is presented later in Sect. 3.

2.3 SDN-based Management in a Glimpse

Many works in literature have shown the benefits of applying the SDN paradigm to network management, resulting in a better QoS for various applications and services. Particular attention has been given to the routing algorithms capable of selecting the best path for different types of services. In the works presented next, the routing decision is based on the traffic information passively collected by the switches as enabled, e.g., by OpenFlow. For instance Tomovic et al. [34], consider two classes of traffic: priority traffic, with strict bandwidth requirements, and best-effort traffic. The proposed frameworks calculate the optimal route as the shortest route having sufficient *bandwidth* available while minimizing the degradation of best-effort traffic. Egilmez et al. [12] propose instead an analytical framework for optimizing the forwarding of QoS-enabled streaming of scalable encoded videos. The optimal routes are calculated by solving a constrained shortest path problem where the *jitter* of the available paths is provided as input. Moreover, a variant of the proposed algorithm is proposed for interactive multimedia applications where the total *delay* is considered instead of the jitter. Tomovic et al. [35] analyze the suitability of different routing algorithms for performance-guaranteed traffic tunnels in backbone SDN networks subject to two constraints: *bandwidth* and path *delay*. The analysis considered both the computational time on the SDN controller and the bandwidth rejection ratio, which is commonly used as a performance indicator for QoS routing algorithms. Similarly, Guck et al. [13] provide a comprehensive evaluation framework and quantitative comparison of centralized QoS routing algorithms in SDN networks subject to *bandwidth*, *delay*, and *jitter* constraints. Finally, Bari et al. [3] propose PolicyCop, a vendor-agnostic QoS policy management framework for OpenFlow-based SDN. The framework provides an interface for specifying QoS-based Service Level Agreements (SLAs) for *bandwidth*, *latency*, and *reliability guarantees*, and enforces them using OpenFlow capabilities.

2.4 Current OAM Protocols in a Glimpse

The most widely-employed transport protocols nowadays are MPLS-TP and PBB-TE. The OAM tool-sets of those protocols (i.e., providing tools for active monitoring) are based on ITU-T Y.1731 [18] and IEEE 802.1ag [15] standards, respectively, and they present largely identical characteristics, being the former a superset of the latter. Both protocols define *Connectivity Fault Management* (CFM) mechanisms for path discovery, fault detection, fault notification, fault recovery, fault verification, and isolation. In addition, Y.1731 defines OAM functions for *Performance Monitoring*, such as frame loss, frame delay, and throughput measurements. These functions are typically implemented through the following set of protocols:

2.4.1 Continuity Check

Continuity Check protocol comprises the periodical transmission of heartbeat messages, namely *Continuity Check Messages* (*CCM*), to detect connectivity failures. These messages do not solicit a response and their transmission rate can be configured according to 7 standard values, spanning from 300 messages/s to 6 messages/h.[5] A sequence number can be optionally used to count CCM losses and detect any eventual link degradation. A burst of Continuity Check messages can be used for measuring one-way *bandwidth*, i.e., on asymmetric links. When the clock of the switches is synchronized, CCM messages can be timestamped and used for measuring the one-way *delay*.

2.4.2 Loopback

Loopback protocol is used for fault verification and isolation. Loopback messages are similar in concept to the *ping* tool. By sending Loopback messages to successive network nodes, an operator can determine the location of a fault as well as measure the two-way frame *delay* and *jitter* in a given network segment. Measuring the two-way delay with Loopback messages does not require the clocks of the switches to be synchronized. A burst of Loopback messages can be used for measuring two-way *bandwidth* on symmetric links.

2.4.3 Link Trace

Link Trace protocol is used for on-demand path discovery and verification between a pair of network nodes. *Linktrace Request* messages traverse hop-by-hop every node along the path between a source and a target node, the time-to-live (TTL) of each message is increased until it reaches the destination node (this is similar in concept to the *traceroute* tool). Each hop responds the request messages with an

[5] This paper focuses on the 3 highest transmission rates, which result in an inter-message interval of 3.3 ms, 10 ms, and 100 ms, respectively.

Linktrace Reply back to the originating node, thus allowing the operator to track the path followed by the initial message.

2.5 5G Network and Services in a Glimpse

5G services are key to understanding the changes being introduced in the new mobile network architectures. The 3GPP has defined a set of services with the corresponding requirements in [1, 2]. Those services are grouped into three categories, namely network slices: (i) enhanced Mobile Broadband—eMBB, (ii) Ultra-Reliable Low Latency Communications—URLLC, and (iii) Massive Internet of Things – MIoT. eMBB services are characterized by high *bandwidth* requirements, spanning from few Mbps to 1 Gbps per user, and by moderate-latency requirements, with the most stringent one being 2–4 ms for virtual meetings. Instead, URLLC services are characterized by low *latency* and high-reliability requirements. Tactile interaction and remote motion control for robots require a maximum end-to-end delay of 0.5–1 ms with a *jitter* of 100 µs. Moreover, URLLC defines a survival time[6] for the services, ranging from 10 to 100 ms, with a service availability up to 99.9999%. Finally, MIoT metrics relate more to the capability of the network system to handle millions of active connections generating sporadic traffic.

In order to provide these 5G services, the network architectures have to also go through some transformations as compared to current deployments. Figure 2 illustrates the 5G transport network reference architecture as recently proposed in [22]. The transport architecture comprises three segments: (i) access, (ii) aggregation, and (iii) core. The access comprises on average 6 antenna sites for each node M1 connected via a point-to-point link, and ~ 6 M1 nodes connected in a ring topology. Thus, each access ring connects a total of 36 antennas on average. Next, each aggregation ring comprises ~ 6 M2 nodes, each of which serves as the gateway to 4 access rings on average. Each aggregation ring is served by two M3 nodes for redundancy reasons, while each M3 node provides gateway capabilities to 2 aggregation rings. The mobile packet core network comprises two M4 nodes for each core ring and a variable number of other M4 nodes connected in a mesh fashion. The amount of M4 nodes highly depends on the physical deployment of the mobile network at country level. For the sake of example [8], reports 12 nodes M4 in the case of Germany.

3 Challenges in Actively Monitoring an OpenFlow Mobile Network at Scale

Monitoring support is very limited in OpenFlow. Current network controllers perform network monitoring by keeping track of the status of the OpenFlow ports by periodically collecting port statistics from the switches [12, 13, 34, 35]. These statistics provide information regarding the number of packets sent/received or dropped

[6] The survival time is the time that an application consuming a communication service may continue to operate without receiving any messages.

by the port, and whether the port is alive or not. Collecting those statistics involves a message exchange between the network controller and the switch that weights ~ 600 bytes for a single port.

According to the reference architecture shown in Fig. 2, there are $\sim 200,000$ ports for the network segment spanning from the antenna sites to the core ring. This results in a ~ 1 Gbps of monitoring traffic in case of polling the port statistics every second in those network segments. In the example cited in Sect. 2.5 for Germany [8], there are 12 of those segments, resulting in a total of ~ 12 Gbps. Nonetheless, this is only the bandwidth required to collect the port statistics which do not include any information related to the traffic flows configured in the network. To retrieve these statistics, the SDN controller needs to periodically poll the switches with a message exchange of ~ 500 bytes for each flow and switch. In case of having various flows configured in the network, the bandwidth required for monitoring can easily grow up to tenths of Gbps.

The periodic polling of statistics presents three main drawbacks in mobile networks: (i) a huge amount of bandwidth is required in the control infrastructure, (ii) the network controller should be able to process this huge amount of monitoring information in time, and (iii) the granularity offered to applications is bound to the polling interval, therefore an application would not be able to react quicker than the configured polling time. Additionally, current OpenFlow statistics do not provide the network controller with sufficient information for knowing the current status of the ports and links (e.g., delay, jitter, available vs advertised bandwidth, congestion level, etc.). For instance, a port may be considered alive but may not have link layer connectivity because of misconfiguration (e.g., wrong VLAN). Indeed, an OpenFlow port is considered alive if the carrier at the physical level is detected while no information is available on the status of the link itself. To overcome such limitations, a set of active measurements is required to enable a fine-grain view of the network status.

3.1 Active Monitoring with OpenFlow Switches

Active monitoring relies on the capability to inject test packets into the network, following them, measuring the relevant metrics, and store the results for future network optimization or data analytics. A key feature of being *active* is hence the capability of controlling the nature of the traffic generation, such as the timing, frequency, scheduling, packet sizes and types (e.g., to emulate various services), location, etc. This enables the emulation of distinct scenarios to check if Quality of Service (QoS) or Service Level Agreements (SLAs) are met. As a result, active monitoring permits to measure target metrics only when and where they are needed.

Because of the stateless forwarding performed by OpenFlow switches, as well as their incapability of generating and injecting any packets in the network, active measurements need to be entirely initiated and performed by the network controller. To that end, the logic of these measurements needs to be implemented as an application running on top of the controller. In the following, we analyze the challenges of implementing current OAM procedures as applications on the network controller.

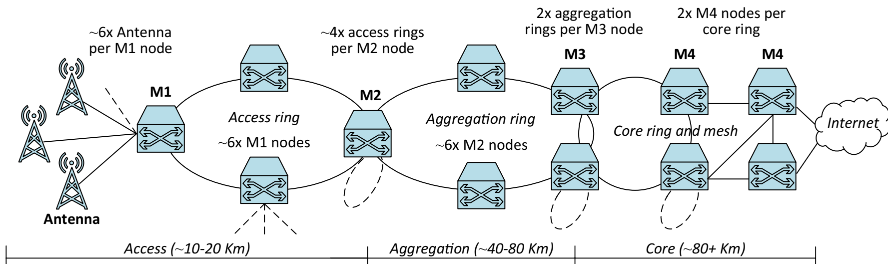


Fig. 2 Proposed reference architecture of a 5G transport network [22]

Particularly, we select Connectivity Check, Loopback, and Link Trace protocols as reference procedures because of their large deployment in today's operator networks.

3.1.1 Connectivity Check

Connectivity Check requires the switch to generate, timestamp, and send specific messages over the data plane to measure the unidirectional bandwidth and delay of a link (see Fig. 3a). However, OpenFlow does not provide any support for packet generation and injection. Therefore, the network controller needs to overcome such shortcoming and implement the CCM mechanisms as shown in Fig. 3a. Please note that the numbers (#) appearing in the following text refer to the distinct messages illustrated in Fig. 3. The controller first generates and timestamps a CCM message for each CCM-enabled switch port and then forwards it to the switch over the control plane (A.1). Next, the switch forwards the CCM message over the data plane (A.2). The receiving switch finally notifies the network controller of the successful CCM reception by forwarding on the control plane the frame received over the data plane (A.3).

In this way, the network controller supervises the connectivity status by keeping track of the CCM messages being sent and received. However, the measurement of the one-way delay is inaccurate because (i) the timestamping occurs before the packets are actually transmitted over the data plane, and (ii) the comparison between the transmission and reception time is done in the controller and not on the target switch. Moreover, the measurement of the one-way bandwidth is inaccurate because the control plane bandwidth becomes a limiting factor for the data plane bandwidth. Clearly, such an approach unnecessarily overloads the controller and the control plane. Indeed, adopting the most stringent connectivity checking configuration, that is to transmit a CCM message every 3.3 ms [18] on every port, generates a total of ~ 340 Gbps over the control plane from the antenna sites to M4 nodes (see Fig. 2), and several Tbps when considering 12 of those segments.

3.1.2 Loopback

Loopback requires the switch to generate, timestamp, and send specific messages over the data plane to measure the bidirectional bandwidth and delay of a link (see

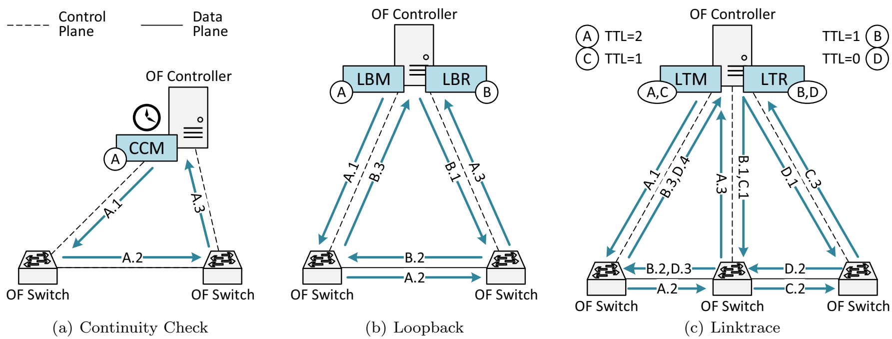


Fig. 3 Legacy OpenFlow-based operation of current OAM protocols

Fig. 3b). The network controller needs to create the Loopback message (LBM), timestamp it, and send it over the control plane to the switch (A.1), which in turn forwards the message on the data plane (A.2). This is then intercepted and sent back to the controller (A.3), thus triggering a timestamped Loopback Reply message (LBR). This is sent to the switch (B.1) and then forwarded on the data plane (B.2). The switch originating the initial message receives the Reply frame and forwards it back to the network controller (B.3). SDN-based implementation of the Loopback message suffers from the same limitations as the Connectivity Check, both in terms of accuracy and overload of the control plane.

3.1.3 Link Trace

Link Trace enables the hop-by-hop tracking of a certain path by sending a series of Link Trace Messages (LTM) with incremental Time To Live (TTL) values[7] as shown in Fig. 3c. The network controller generates an LTM message ($TTL = 2$) and sends it over the control plane to the switch (A.1). Such a message is then forwarded over the data plane (A.2) and back to the controller (A.3). Next, the controller generates a Link Trace Reply message (LTR) with a TTL value equal to $TTL(LTM) - 1$ as the response (B.1). Simultaneously, the controller decreases the TTL of the LTM message[8] and sends it back to the data plane (C.1). LTM and LTR messages are then forwarded on the data plane (B.2, C.2) and to the controller (B.3, C.3). Upon (C.3) reception, the controller generates an LTR (D.1) which is then transmitted on the data plane (D.2, D.3) and finally back to the controller (D.4). Link Trace potentially presents the same scalability issues[9] of Connectivity Check and Loopback. However, Link Trace is used for verifying the correct configuration of network paths and

[7] An LTM with a TTL equal to 0 is discarded by the network. An LTM message with $TTL = n$ serves at determining the nth hop.

[8] OpenFlow does not support CFM headers, thus the TTL cannot be decreased by the switch itself as e.g. in the IP protocol.

[9] Accuracy issues are not present because packets are not timestamped.

it is activated on-demand, presenting different scales of operation compared to CCM and LBM.

3.2 Towards a Stateful OpenFlow?

The several issues of OpenFlow in performing active monitoring can be traced down to two factors: (i) the incapability of OpenFlow to keep information on the forwarded traffic, and (ii) the impossibility of generating and injecting packets. In the recent years several works have been proposed, fostering the debate on *stateless* vs *stateful* OpenFlow. Bianchi et al. [5] propose OpenState, an OpenFlow-compatible abstraction to formally describe stateful processing of flows inside the switch itself. Such abstraction relies on eXtended Finite State Machines (FSM) and an API that can be implemented on the switch by largely reusing existing OpenFlow features. OpenState allows implementing reactive applications on the switches, such as port knocking which is commonly used for opening a port on a firewall. While this capability would be enough to keep track of incoming CCM messages for connectivity check, OpenState does not provide any support for packet generation on the switch.

Moshref et al. [27] propose FAST, which enables the controller to pre-install a state machine on the switch, thus allowing the switch to automatically record flow state transitions by matching incoming packets to installed filters. FAST defines an abstraction for state machines, a compiler for translating the state machines to the data plane API, and a data plane that includes a pipeline to support state machines with commodity switch components. Similarly to OpenState, FAST does not provide any support for packet generation.

Pontarelli et al. [30] propose FlowBlaze to fill two gaps of [5, 27]: (i) not defining a state access model that allows for both per-flow and global consistency, and (ii) not dealing with issues related to the integration of their proposed state machines in the machine model designed in [7] for fast programmable match-action processing in hardware. Although FlowBaze fills the above gaps, like OpenState and FAST, FlowBlaze still does not provide any support for packet generation.

Bifulco et al. [6] address such shortcoming by proposing an API that enables programmers to define in-switch packet generation operations, which include the specification of triggering conditions, packet's content and forwarding actions. The authors provide application examples for the delegation and implementation of ARP and ICMP handling from the controller to the switch. However, the proposed API can trigger the packet generation only in reaction to the reception of a packet at the switch and not as a reaction of some timed events. As a result, the proposed approach would be sufficient to generate Loopback Reply messages but not for the periodical generation of CCM messages.

Cascone et al. [9] propose SPIDER, a packet processing pipeline design for stateful SDN data plane that allows the implementation of failure recovery policies with fully programmable detection and rerouting mechanisms directly in the switches' fast-path. While SPIDER provides an OpenFlow-compatible way for fast-rerouting based on heartbeat messages (like CCM), other monitoring features are not considered. For instance, link degradation and delay can not be measured in SPIDER,

thus limiting the rerouting to hard-connectivity failures only, that is no messages are received. Therefore, rerouting based on the link delay or degradation is not possible. Moreover, SPIDER does not support the periodical reporting of the status of the links (e.g., quality) to the controller.

Summarizing, the works available in the literature propose different solutions for enabling stateful processing in the data plane mainly tailored to user traffic. However, active monitoring only requires stateful processing of the packets involved in some measurements (e.g., CCM). The amount of such traffic is expected to be negligible compared to user traffic. As a result, only a little portion of the traffic needs to be actively processed by the switch for telemetry purposes. For that reason, we advocate that extending OpenFlow processing from stateless to stateful processing for active monitoring purposes is not ideal. Indeed, such an approach would bring considerably large complexity in the switch fabric compared to the small amount of monitoring traffic that requires stateful processing. Therefore, a lighter solution is required in terms of switch complexity. Nowadays OpenFlow switches are commonly equipped with general-purpose processors[10] which mainly interact with the switch fabric only for configuration and management purposes. In our view, such processors could be leveraged as well to implement the stateful processing required for the telemetry procedures. In this way, OpenFlow performs the stateless processing of user traffic as usual while monitoring traffic is processed locally on the CPU switch. This allows staying compatible with the current OpenFlow solution as detailed in the next section.

4 Design of an Adaptive Telemetry System

This section presents the design of our adaptive telemetry system, namely ATS, for enabling stateful data plane processing tailored to active monitoring. Specifically, ATS aims at providing operators with a set of SDN-compliant tools for defining and configuring telemetry procedures on the switches. While the state-of-the-art solutions add extra features directly into OpenFlow protocol, we adopt a hybrid approach where the telemetry system interacts with the legacy OpenFlow pipeline, i.e., no extension is proposed to the current OpenFlow specifications. Therefore, such a hybrid approach does not envision any change on the switch backplane, which is the part internal to a switch implementing the OpenFlow pipeline and in charge of forwarding packets between ports. Figure 4 shows the ATS design which envisages three main components:

ATS application it runs on the controller and it is in charge of taking the decision of what, when, and where to measure. Since it runs on the controller, the application has a global view on the status of the network, and based on the active traffic flows, path configuration requests, and offered network services, the application decides what the parameters to be monitored in the underlying network are (e.g., delay, link

[10] For instance, the NoviSwitch 21100 is equipped with an Intel Core i7 and the Advantech ESP-9230 with an Intel Xeon.

quality, etc.), either periodically or on-demand. The active execution of those measurements is then delegated to the switches which follow the instructions received by the controller.

ATS plugin it runs on the controller and it is in charge of implementing the communication with the switches via a southbound interface. This interface exposes a RESTful API that provides a uniform and predefined set of operations to allow the network controller to dynamically configure the telemetry procedures on the switches and to receive notifications and alarms. Table 1 reports the Uniform Resource Identifier (URI) exposed by (i) the network controller via the ATS plugin and by (ii) the switch via the ATS agent (see next paragraph). Specifically, the configuration of the telemetry procedures is based on Finite State Machines (FSM), which are then executed locally on the switch. The API is exemplified in Sect. 4.1 while the FSM representation is further detailed in Sect. 4.2.

ATS agent it runs on the switches and it is in charge of: (i) locally executing the FSMs configured by the network controller and (ii) sending the appropriate notifications and alarms via the common API. As described in Sect. 2, in addition to physical ports, OpenFlow defines logical and reserved ports internal to the switch that can be used by external applications/components to interact with the OpenFlow pipeline. For instance, the reserved port CONTROLLER is used to send a packet from the switch to the network controller and vice versa. Similarly, the reserved port LOCAL enables components running on the switch to directly interact with the OpenFlow network. As a result, the ATS agent uses the LOCAL port to send/receive packets over/from the data plane through the standard OpenFlow pipeline.

4.1 Exemplary Scenario

In this section, we provide an example of ATS operation, i.e., measuring the one-way delay between two switches directly connected in an access ring.[11] The exemplary scenario, including the network topology and the high-level message flow, is reported in Fig. 4. Please note that the numbers (#) appearing in the following text refer to the distinct steps illustrated in the right-hand side of Fig. 4. The corresponding API of these steps is also highlighted in Table 1.

At some point in time, an ATS application running on the network controller decides that it needs to measure the one-way delay between the switch M1 and the switch M2. For instance, such a decision could be made in response to a path configuration request received by the network controller for time-sensitive traffic. Next, the ATS application selects the most appropriate measurement procedure (e.g., CCM with timestamp) and verifies that the ATS plugin and agents support the expected capabilities, e.g., by checking the supported ATS version retrieved by querying the plugin information via the */ats/* GET interface. Concurrently, the ATS application may verify that the same measurement procedure is not already running in the network by querying the ATS plugin via the */ats/report/* GET interface. In order to

[11] The clocks of two switches are assumed to be synchronized. See Sect. 5.4 for additional considerations on clock synchronization.

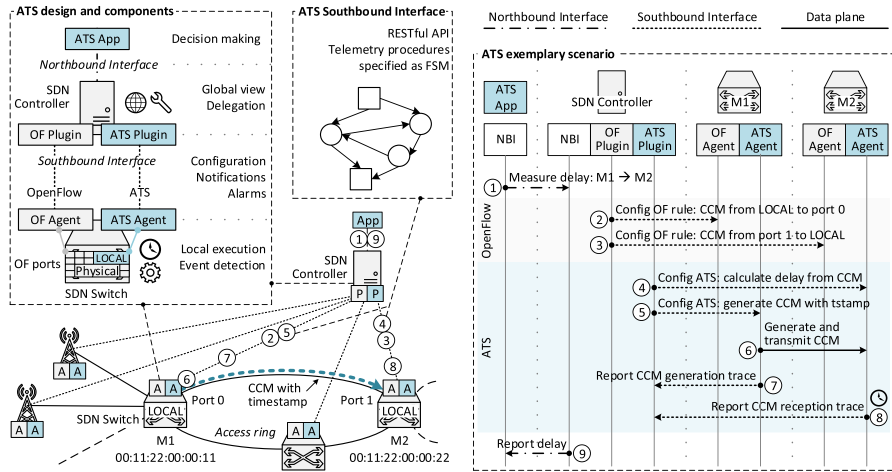


Fig. 4 Adaptive Telemetry System components and exemplary scenario with message flow

answer to the above queries, the ATS plugin may gather up-to-date information from the network switches via the */ats/* and */ats/fsm/* GET interfaces exposed by the ATS agents.

At this point, the ATS application asks the network controller to perform such measurement via the northbound interface (1), i.e., */ats/report/id* POST interface. In turn, the network controller configures the necessary OpenFlow rules for forwarding the traffic to and from the ATS agent (2), (3)[12] for such type of traffic. Let's assume that the port 0 of M1 switch is directly connected to the port 1 of M2 switch. In this case, the OpenFlow pipeline of M1 switch is instructed to forward the CCM messages generated by the ATS agent over the port LOCAL to the port 0. Similarly, the M2 switch is configured to forward the CCM messages received over the port 1 to the port LOCAL for being processed by the ATS agent.

Then, the ATS plugin configures on the switches the measurement procedures in the form of finite state machines (see Sect. 4.2 for further details). Specifically, the CCM reception and delay calculation are configured on the M2 switch (4) and the CCM generation on the M1 switch (5), both via the */ats/fsm/id* POST API exposed by the ATS agent. For instance, the M1 switch is configured to generate a total of 100 CCM messages with an interval of 10 ms between subsequent packets (6). After having transmitted all the CCM messages, the ATS agent is configured to report a trace of the generated messages (7). Similarly, the M2 switch is configured to compute the delay of each CCM message received and to report the trace of all computed values to the network controller once the last CCM is received (8). These reports are communicated from the ATS agents to the ATS plugin via the */ats/fsm/id/event* POST interface. Additionally, a timeout is configured for dealing with the case of the last CCM going lost. Finally, the network controller informs the

[12] These operations are standard OpenFlow operations and they not involve the ATS API.

Table 1 ATS RESTful API and mapping on the exemplary scenario

URI	Method	Description	#msg in Fig. 4
ATS plugin on the network controller			
/ats/	GET	Information about the ATS plugin	N.A., see Sect. 4.1
/ats/report/	GET	Reports available on the network controller	N.A., see Sect. 4.1
/ats/report/id	GET, POST PUT, DELETE	Read, create, update, delete a report on the network controller	(1),(9)
ATS agent on the switch			
/ats/	GET	Information about the ATS agent	N.A., see Sect. 4.1
/ats/fsm/	GET	Telemetry procedures available on the switch	N.A., see Sect. 4.1
/ats/fsm/id	GET, POST, PUT, DELETE	Read, create, update, delete a telemetry procedure on the switch	(4),(5)
/ats/fsm/id/event	POST	Send events to the telemetry procedure	(7),(8)

ATS application via the northbound interface on the measured delay (9) via the */ats/report/id* PUT interface

4.2 ATS Procedures Modeling

The previous paragraphs briefly introduced that the telemetry procedures are specified via finite state machines. This decision is based on our analysis of the standard OAM operations as defined in ITU-T Y.1341 [20] (ITU-T Y.1371 corollary standard), which is reported in the following.

ITU-T Y.1341 formally describes the OAM procedures as finite state machines by using the Specification and Description Language (SDL) [21]. SDL is a language targeted at the unambiguous description of the behavior of reactive and distributed systems. While SDL provides a rich set of functional blocks for behavior description, only a limited set is required for fully describing the behavior of the OAM procedures under consideration. Specifically, they can be described by exclusively using the following five SDL blocks:

1. *State* describes the status of the FSM that is currently waiting to execute a transition. Two special states define the *entry* and *exit* points of the FSM respectively.
2. *Task* defines a series of internal steps to be executed by the switch. Variable declaration and assignment, packet forging, and timer configuration are common tasks.
3. *Decision* is equivalent to an *if-then-else* statement. Local variables, header fields of the incoming packets, and timestamps are the usual comparison terms.
4. *Input* is the actual trigger of the transition and an event is its common representation. The events leveraged in the OAM operations are: (i) incoming packet, (ii) external signal, and (iii) timer expiration.
5. *Output* specifies a set of actions to be executed upon condition fulfillment or event reception. Packet transmission and signal firing are common outputs.

Figure 5a shows the SDL finite state machine (FSM) for describing the CCM generation procedure at the switch [20]. It is worth highlighting that this FSM implements the expected behavior of our exemplary scenario as illustrated in Fig. 4 and detailed in Sect. 4.1. While SDL can comprehensively describe the OAM protocols behavior, it only provides a basic model for describing the supported data types (e.g., integer, real, char, etc.). A more comprehensive data model is hence required by the network controller to unequivocally instruct the switch. Therefore, we propose a comprehensive data model for telemetry procedures based on the combination of IETF RFC 7223, IEEE 802.1Qcp [17], and Metro Ethernet Forum (MEF) documents [25, 26]. By combining and extending them, our proposal takes the form of a YANG model specifying the telemetry procedures and the *port* and *packet* data types for enabling the generation, transmission, and reception of packets, which are of paramount importance for the telemetry procedures.

With this data model, the next step is to design a generic FSM-representation for telemetry procedures that can be exchanged between the controller and the

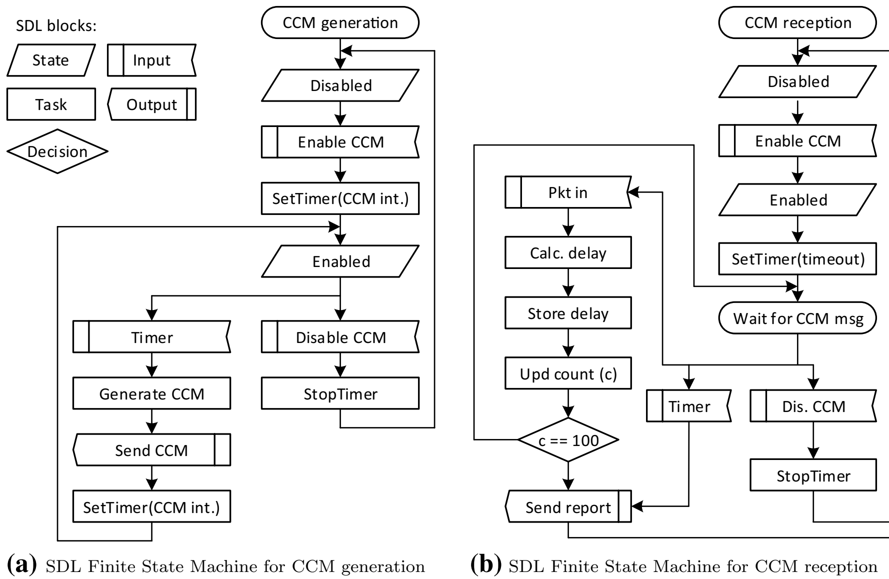
switches. To that end, we define the following basic concepts for ATS FSM representation: *state*, *transition*, *event*, and *datamodel*. Each state contains a set of transitions that define how the FSM reacts to events, which can be generated by the state machine itself or by external entities (e.g., packet reception). The data model defines how the data internal to the state machine is stored, read, and modified as well as its interpretation in conditional expressions. In the following we report the main ATS elements expressed as XML elements:

```
<xml version="ats">
<datamodel>
 <data id="port" type="port" expr="local"/>
 <data id="ccm" type="packet">
  <expr id="ether_dst">00:11:22:00:00:22</expr>
  <expr id="ether_src">00:11:22:00:00:11</expr>
  <expr id="ether_type">89:02</expr>
   ... additional header fields ...
  <expr id="sn">00:00:00:00</expr>
  <expr id="tstamp">00:00:00:00</expr>
 </data>
 <data id="report" type="list" expr="[]"/>
</datamodel>
<state id="disabled">
 <onexit>
   <send event="timer" delay="0.01s"/>
 </onexit>
 <transition event="enable" target="enabled"/>
</state>
<state id="enabled">
 <onexit event="timer">
  <assign target="ccm.sn" expr="ccm.sn + 1"/>
  <assign target="ccm.tstamp" expr="time.now"/>
  <assign target="report" expr="report + [ccm.
    tstamp]"/>
  <send target="port" type="packet" data="ccm"/>
  <if expr="ccm.sn < 100">
   <send event="timer" delay="0.01s"/>
  </if><else>
   <send target="ctrl" type="list" data="report"/
    >
   <send event="disable"/>
  </else>
 </onexit>
 <transition event="timer" target="enabled"/>
 <transition event="disable" target="disabled"/>
</state>
</xml>
```

Code 1 ATS code implementing the state machine for CCMgeneration illustrated in Fig. 5a.

<state>: this element holds the representation of a state and it can be used to express the SDL *State* block.

<data>, *<assign>*: the *<data>* element is used to declare and populate portions of the data model whilst the *<assign>* element is used to modify the data entries. These ATS elements combined can represent the SDL *Task* block.



(a) SDL Finite State Machine for CCM generation **(b)** SDL Finite State Machine for CCM reception

Fig. 5 SDL Finite State Machines [20] required for implementing the one-way delay measurement based on CCM as illustrated in the exemplary scenario of Fig. 4

<*if*>, <*elseif*>, <*else*>: allow describing conditional code execution and consequently the SDL *Decision* block. Conditional expressions are supported on local variables as well as on header fields and timestamps.

<*transition*>: defines the available transitions between states and the events that trigger them. The <*onexit*> and <*onenter*> elements are used to define whether the instructions need to be executed when leaving or entering a given state. Additionally, *Event I/O Processor* is used for emitting input and output events that result in state transition. To that end, a dedicated I/O processor is required to notify about incoming packets and trigger the transitions. This, along with *port* and *packet* data types, can be jointly used with the <*transition*> element to express the SDL *Input* block.

```
<xml version="ats">
<datamodel>
 <data id="port" type="port" expr="local"/>
 <data id="ccm" type="packet">
  <expr id="ether_dst">00:00:00:00:00:00</expr>
  <expr id="ether_src">00:00:00:00:00:00</expr>
  <expr id="ether_type">00:00</expr>
   ... additional header fields ...
  <expr id="sn">0</expr>
  <expr id="tstamp">0</expr>
 </data>
 <data id="report" type="list" expr="[]"/>
</datamodel>
<state id="disabled">
 <onexit>
  <send event="timeout" delay="1s"/>
 </onexit>
 <transition event="enable" target="enabled"/>
</state>
<state id="enabled">
 <onenter event="pkt_in" type="packet" data="ccm"
     >
  <if expr="ccm.ether_src == 00:11:22:00:00:11 &&
     ccm.ether_dst == 00:11:22:00:00:22">
  <assign target="report" expr="report + [time.
     now - ccm.tstamp]"/>
  <send target="port" type="packet" data="ccm"/>
  <if expr="ccm.sn == 100">
    <send target="ctrl" type="list" data="report"
     />
  </if>
  </if>
 </onenter>
 <onexit event="timeout">
  <send target="ctrl" type="list" data="report"/>
  <send event="disable"/>
 </onexit>
 <transition event="pkt_in" target="enabled"/>
 <transition event="disable" target="disabled"/>
 <transition event="timeout" target="disabled"/>
</state>
</xml>
```

Code 2 ATS code implementing the state machine for CCMreception illustrated in Fig. 5b.

<send>: this element is used to send events and data to external systems (e.g., to the network controller or to the data plane) and to raise events in the current system (e.g., raise a timer). This element can be used to express the SDL *Output* block and be leveraged, e.g., to fire an alarm from the switch to the network controller. Moreover, in conjunction with the *port* and *packet* data types, *<send>* can be used to transmit packets.

Code 1 and Code 2 show the ATS state machines implementing the one-way delay measurement described in Sect. 4.1. Particularly, Code 1 depicts the ATS state machine (flowchart shown in Fig. 5a) configured by the network controller on the M1 switch for generating CCM messages. This is the FSM sent by the network

controller to the M1 switch in step (5) of Fig. 4. Similarly, Code 2 presents the ATS state machine (flowchart shown in Fig. 5b) for computing the one delay based on received CCM messages on the M2 switch. This is the FSM sent by the network controller to the M2 switch in step (4) of Fig. 4. As it can be noticed, the *<data>* element in the data model allows defining custom packets by concatenating multiple *<expr>* elements which represent the header and the payload structure. This is useful to create and manipulate packets to be transmitted (e.g., increment the sequence number) and to decode the received packets according to a defined structure. These packets can be then transmitted or received over the LOCAL port defined in the data model. Additionally, the reserved port *ctrl* is provided by the ATS agent to allow communication to and from the network controller (e.g., to send the delay report). Moreover, the ATS agent provides the data type *time* to access the clock on the switch (i.e., current time available via *time.now*). Finally, the Event I/O Process provides the event *pkt_in* to trigger a state transition when a packet is received.

5 Experimental Evaluation

This section presents the experimental evaluation of ATS performance against legacy OpenFlow-based implementations. Figure 6 shows an overview of the testbed used, comprising two machines equipped with an Intel Xeon E5-2620 processor, 128GB of RAM, and running Ubuntu 18.04 Server. One machine is used as network controller while the other is used to emulate the topology of one access ring (see Fig. 2), which comprises one M2 node, six M1 nodes, and thirty-six antenna sites. The M1 nodes are assumed to be configured in a ring topology only at optical level. At logical level instead, they are connected point-to-point to their corresponding gateway (M2 node). This means that packets are enqueued only at gateway level, thus forming a logical tree topology (see Fig. 6), which comprises a total of 43 nodes and 84 ports. Each node is then implemented as an LXD[13] container, which runs inside the ATS agent and Open vSwitch[14] as OpenFlow agent implementation.

The legacy-SDN implementation of the OAM protocols (see Sect. 3.1) is based on the OpenFlow controller Ryu.[15] Regarding the ATS implementation, the ATS agent translates the XML-based finite state machine into a JavaScript representation. We then used SCION-CORE[16] as interpreter for these procedures, which are executed on nodejs,[17] a lightweight event-driven environment. PCAP[18] and nanotimer[19] nodejs modules are used as packet-event I/O processor and high-precision

[13] https://linuxcontainers.org/lxd/introduction/.

[14] https://www.openvswitch.org/.

[15] https://osrg.github.io/ryu/.

[16] https://github.com/jbeard4/SCION-CORE.

[17] https://nodejs.org/.

[18] https://github.com/node-pcap/node_pcap.

[19] https://github.com/Krb686/nanotimer.

timer, respectively. The results reported in the following are obtained by averaging 100 runs of each experiment.

5.1 Delay Measurement

The first objective is to evaluate the legacy-SDN and ATS accuracy in measuring the one-way and two-way link delay. To that end, we implemented the CCM and LBM mechanisms on both systems and compared them against the baseline measurement obtained with the *ping* tool. It is worth highlighting that the ping is a network layer mechanism (i.e., IP) that is not usually available on the traditional switches operating at the data-link layer (e.g., Ethernet). Nevertheless, such a tool is available on our testbed because the nodes of the network are implemented on LXD, which provides a full-fledged operating system environment. Moreover, we used tc,[20] which is a traffic control tool for Linux systems, to configure a fixed delay of 1 ms on the virtual links connecting the various switch instances.

The delay is then measured for an increasing number of simultaneously active ports and message generation interval. While the number of ports provides an estimation of how well the network controller scales with respect to the number of switches under its management, the message generation interval provides an estimation of how well the network controller scales with respect to the freshness of the measurements (e.g., the values are updated every 100 ms). To assess the first scalability aspect, we gradually activated a growing number of ports, starting from 1, with an incremental step of 4 ports until reaching 84 ports being simultaneously active. That is, we tested the systems under the scenarios of 1 active port, 4 ports, 8 ports, etc., until 84 ports. Regarding the second scalability aspect, we selected the three highest transmission rates defined in [18], which result in a generation intervals of 3.3 ms, 10 ms, 100 ms, respectively (see Sect. 2.4). Finally, we performed 100 runs for each of the scenarios obtained by combining the number of active ports and the message interval.

Figure 7a, b show the results for the one-way and two-way delay measurement, respectively. Noticeably, Fig. 7b shows that the round-trip delays measured via ATS and ping are comparable and they do not depend on the number of active ports nor on the message interval. This is also highlighted in Table 2 which reports the statistical characteristics of the delay measurements. More precisely, ping reports an average two-way delay of 2.10 ms which matches the value reported by our LBM implementation on ATS. For what concerns the one-way delay, we consider ping/2 as a baseline since our testbed is characterized by symmetric links. Moreover, since all the switches are running on the same physical machine, we can safely compute the one-way delay with CCM messages because all the containers share the same CPU clock (see Sect. 5.4 for additional considerations). According to these considerations, the one-way delay obtained with ping/2 is 1.05 ms. As expected, the CCM implementation on ATS reports an average one-way delay of 1.05 ms, matching the

[20] http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html.

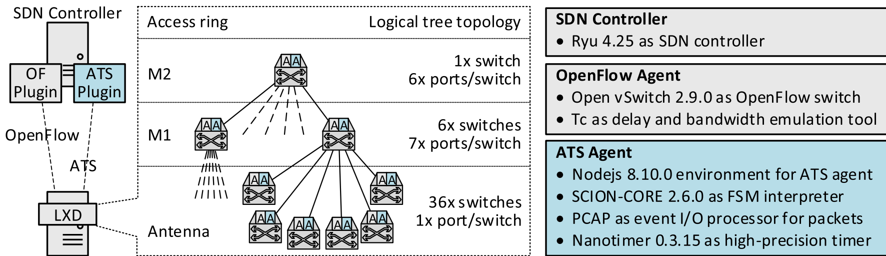
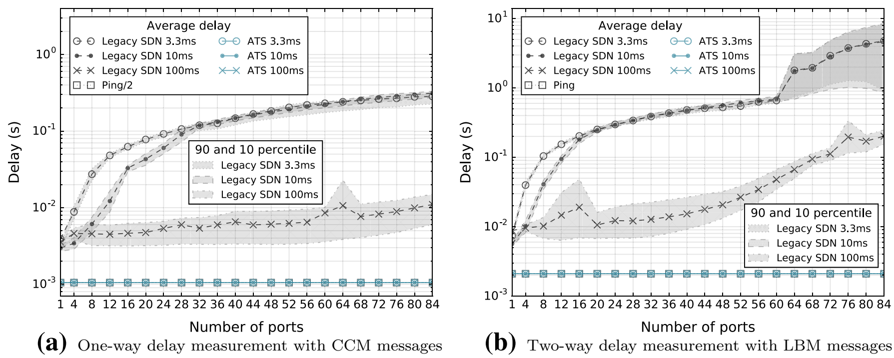


Fig. 6 Testbed overview and components

ping/2 value. Therefore, ping and our ATS implementation provide similar accuracy in measuring the one-way and two-way delay. It is worth highlighting that Fig. 7a, b do not show the confidence intervals for the ping and ATS data because they are not graphically appreciable.

Regarding the legacy-SDN solution, the measured delay depends on the number of active ports and on the configured message interval. More precisely, the delay measurement closest to ping and ATS occurs in case of 1 active port and a message interval of 10 ms for both CCM and LBM messages. In the case of CCM, the network controller reports an average of one-way delay of 2.96 ms. In the case of LBM instead, the network controller reports an average of two-way delay of 5.74 ms. In both cases, the reported value is $\sim$ 300% higher than the delay measured by ATS and ping because every message sent over the data plane requires two additional messages on the control plane. As it can be evinced in Fig. 7a, b, the delay measured by the network controller significantly increases with the number of ports and with smaller message generation intervals. The highest values for the one-way delay is obtained in case of 84 ports and a message interval of 10 ms (similar results are obtained in case of 3.3 ms). The average delay with CCM is 313.77 ms. Similarly, the highest values for the two-way delay are obtained in case of 84 ports and a message interval of 3.3 ms with an average measured delay of 4679.80 ms. It is clear that the delay measured by the controller is far from the reality, being several orders of magnitude larger than the reference value.

By comparing the above results with the performance requirements for low-latency and high-reliability scenarios defined by 3GPP [1], we can see that measuring the delay with legacy-SDN does not provide the necessary accuracy for critical services. For instance, 3GPP defines that *discrete automation* traffic requires a maximum end-to-end latency of 10 ms and a jitter of 100 µs. *Electricity distribution* instead requires an end-to-end latency of 5 ms and a jitter of 1 ms. Even in the most favorable case in legacy-SDN of measuring the one-way delay on 1 port at a time, the jitter on a single link is reported as 0.37 ms, which is above the maximum admissible value for *discrete automation*. Similarly, with 4 simultaneously active ports in legacy-SDN, the one-way measured delay is 4.61 ms with a jitter of 0.77 ms. This makes difficult to assess, i.e., whether the 5 ms end-to-end requirement is met for the *electricity distribution* service. On the contrary, with our implementation of ATS we can safely measure the delay with a very limited jitter (e.g., 30 µs in case of CCM



(a) One-way delay measurement with CCM messages **(b)** Two-way delay measurement with LBM messages

Fig. 7 Scalability of delay measurement with ping, legacy-SDN controller, and ATS

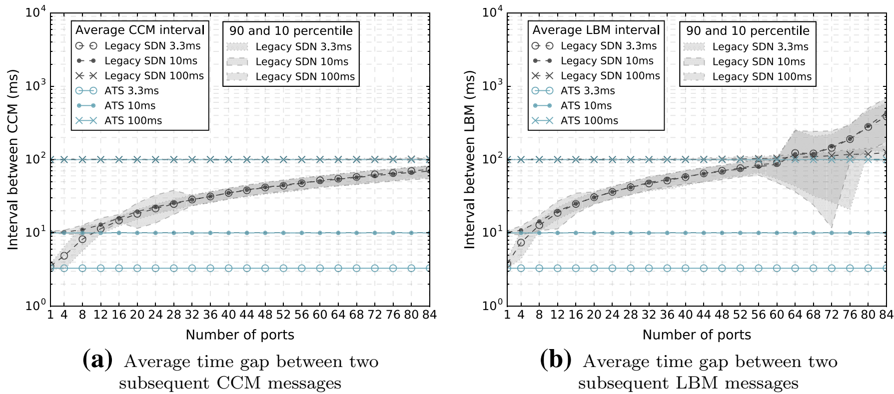
Table 2 Statistical characteristics of the delay (in ms) measured with ping, legacy-SDN controller, and ATS

Scenario	Solution	#Ports	Tx interval	Mean	5th pctl	95th pctl	Mode	Median	Std
One-way (CCM)	Ping/2	Any	Any	1.05	1.02	1.08	1.05	1.05	0.02
	ATS	Any	Any	1.05	1.01	1.10	1.05	1.05	0.03
	SDN	1	10	2.96	2.66	3.46	2.81	2.90	0.37
	SDN	84	10	313.77	260.98	531.49	298.43	295.40	88.23
Two-way (LBM)	Ping	Any	Any	2.10	2.04	2.16	2.10	2.10	0.04
	ATS	Any	Any	2.10	2.06	2.14	2.06	2.10	0.03
	SDN	1	10	5.74	5.10	6.36	5.44	5.48	0.43
	SDN	84	3.3	4679.80	667.61	8852.62	622.69	4806.67	2640.45

over one link). By reporting the measurements obtained by ATS, the network controller can hence safely decide whether a link is suitable for a given set of services, even the strictest ones requiring a maximum jitter of 100 µs.

5.2 Connectivity Status

The second objective is to evaluate how effectively the CCM and LBM messages can be used to detect the link status. Also, in this case, the evaluation is performed for an increasing number of simultaneously active ports (i.e., from 4 to 84 with a step of 4) and message generation interval (i.e., 3.3 ms, 10 ms, 100 ms). As it can be evinced from the previous evaluation, the overload suffered by the legacy-SDN controller produces an over-estimation of the link delay as a side effect. Since the network controller is not capable of generating and processing the CCM and LBM messages in time for all the ports, the network controller starts queuing the messages. This produces a gradual increment of the time gap between two subsequent messages, thus deviating from the configured interval. Figure 8a, b highlight the diverging trend for CCM and LBM, respectively, by depicting the time difference



(a) Average time gap between two subsequent CCM messages

(b) Average time gap between two subsequent LBM messages

Fig. 8 Scalability of message generation for legacy-SDN controller and ATS

measured between subsequent messages for an increasing number of ports and different message generation intervals. The results show that ATS is capable of generating the messages in compliance with the configured interval regardless of the protocol and the number of active ports as reported in Table 3.

Regarding the legacy-SDN solution, in case of 1 active port, the network controller generates CCM and LBM messages with an interval quite close to the target one as shown in Table 3. In the case of 84 ports instead, the average message interval significantly diverges from the target one. For instance, CCM messages are generated with an average interval of 73.85 ms for the 3.3 ms case, 71.13 ms for the 10 ms case, and 101.62 ms for the 100 ms case. LBM messages instead are generated with an average interval of 440.20 ms for the 3.3 ms case, 426.76 ms for the 10 ms case, and 124.54 ms for the 100 ms case. Additional statistical characteristics are reported in Table 3 for the extreme cases of CCM and LBM generated every 3.3 ms on 84 ports.

By comparing the above results with [1], we can see that supervising the connectivity status with legacy-SDN does not provide the necessary responsiveness for critical services. For instance, the *electricity distribution* the *process automation* services are characterized by a survival time of 10 ms and 100 ms, respectively. The survival time indicates the admissible maximum time for restoring the connectivity in case of a link failure or in case the delay requirement is no longer met (see Sect. 5.1). According to the CCM protocol [18], a connectivity failure is detected if no heartbeat messages are received within 3.5 times the configured interval (e.g., 11.55 ms in 3.3 ms case). Given the precision of ATS in periodically generating the messages, it is easy to detect whether no heartbeat messages are received before the timeout expiration. However, this is not the case for legacy-SDN since the increasing gap between two subsequent messages yields to a considerable amount of false-positive failure detection. For instance, in the extreme case of 84 ports and 3.3 ms target interval, CCM messages are generated every 73.85 ms, which is considerably higher than the timeout of 11.55 ms for detecting link failure. In our setup, this does not occur for the 100 ms case, thus allowing legacy-SDN to detect a connectivity

Table 3 Statistical characteristics of the message generation interval (in ms) with legacy-SDN controller and ATS

Solution	Message	#Ports	Target int	Mean	5th pctl	95th pctl	Mode	Median	Std
ATS	CCM/LBM	Any	3.3	3.30	3.27	3.33	3.30	3.30	0.16
	CCM/LBM	Any	10	10.00	9.97	10.03	10.00	10.00	0.12
	CCM/LBM	Any	100	100.00	99.96	100.04	100.00	100.00	0.24
SDN	CCM/LBM	1	3.3	3.52	3.13	3.96	3.47	3.50	0.88
	CCM/LBM	1	10	10.24	9.73	10.80	10.24	10.23	0.44
	CCM/LBM	1	100	100.33	99.70	101.01	100.46	100.31	2.14
	CCM	84	3.3	73.85	54.79	85.79	72.59	69.47	58.17
	LBM	84	3.3	440.20	48.94	976.95	449.40	395.59	341.98

failure within 350 ms. Still, this provides a measurement granularity of 100 ms which is not sufficient to comply with the survival time requirements (Table 4).

As it can be noticed, the survival time for the *electricity distribution* flow is 10 ms, which is smaller than the timeout of 11.55 ms when generating messages every 3.3 ms. Therefore, a smaller message interval is required to comply with that requirement. While this is easily achievable with ATS by simply updating the ATS state machine, it is undeniably harder in the legacy-SDN solution because of the scalability issues already appreciable with higher intervals. Moreover, the control plane delay may be taken into account to select the most appropriate message interval for the data plane. For example, in case of a control plane delay of 2 ms, a message interval of 1 ms would allow the network controller to receive a notification (e.g., link failure, maximum delay exceeded, etc.) within 5.5 ms, thus leaving 4.5 ms to restore the connectivity in case of a survival time of 10 ms. Finally, with this notification mechanism, ATS allows offloading the control plane by only transmitting information upon initial configuration and when certain conditions occur in the data plane. For instance, the message exchange for configuring the ATS procedure weights 7710 bytes while the notification weights 394 bytes, both including the protocols overhead: HTTP, TCP, IP, and Ethernet.

5.3 Bandwidth Measurement

The last objective is to evaluate the legacy-SDN and ATS accuracy in measuring the bandwidth available on a link. Similar to the previous cases, the evaluation is performed for an increasing number of simultaneously active ports (i.e., from 4 to 84 with a step of 4). In this case, messages are generated as fast as possible in order to saturate the available bandwidth. Figure 9a shows the control plane load and the bandwidth measured on the data plane by the network controller in case of legacy-SDN solution. Notably, the control plane load remains constant regardless of the protocol (i.e., CCM or LBM) and the number of active ports. This is because the network controller is capable of generating only a fixed amount of packets per

Table 4 Statistical characteristics of the bandwidth (in Mbps) measured with Iperf, legacy-SDN controller, and ATS with 1 active port

Solution	Message	#CPU	Mean	5th pctl	95th pctl	Mode	Median	Std
Iperf	UDP	1	1543	1487	1595	1526	1544	33
ATS	CCM	1	677	587	758	622	682	59
	CCM	2	1186	1032	1347	1210	1178	99
	CCM	4	2186	1946	2434	2168	2167	156
	CCM	8	3970	3614	4334	3955	3499	234
SDN	CCM	1	8.25	4.72	12.33	8.45	7.69	2.87

second. Particularly, the network controller generates an average of 708 packets per second, which results in an average control-plane load of 12.21 Mbps.

Because of the overhead introduced by the encapsulation of CCM and LBM messages in OpenFlow messages between the network controller and the switches, only an average of 8.25 Mbps is then forwarded on the data plane. Since the number of generated packets is constant, these are spread over all the active ports resulting in a measured bandwidth inversely proportional to the number of ports as shown in Fig. 9a. In the case of 84 ports, the average bandwidth measured per port is 0.146 Mbps with CCM, while it is 0.022 Mbps with LBM. As it can be noticed, the bandwidth values provided by CCM are higher than LBM because CCM involves the generation of fewer packets compared to LBM (see Fig. 3a, b). This is further highlighted in the case of LBM by the controller saturation starting with 64 ports.

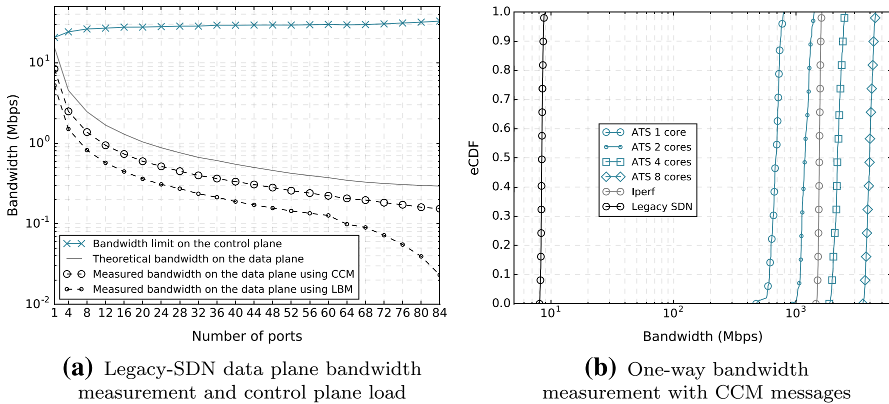
Moreover, we compare the obtained measurements with the maximum theoretical bandwidth measurable on the data plane by carrying the CCM and LBM messages as a payload over the TCP-based OpenFlow control channel. The achievable throughput for data transmitted over TCP is $\sim$ 75% of the available link bandwidth.[21] In our scenario, the control plane bandwidth is shared among all the connected switches. Figure 9a) shows that the experimental results follow the same trend as the theoretical value (gray line). This implies that even if the network controller is capable of generating messages at the desired rate, the bandwidth measurement would always be distorted by the TCP connection used in the OpenFlow control channel.

Figure 9b shows the one-way bandwidth measurement over one link performed with legacy-SDN and ATS. It is worth noticing that the virtual links in our testbed have no fixed speed, meaning that the available bandwidth is determined by how fast the physical machine can send a message from one virtual switch to another. To that end, we also measure the bandwidth with Iperf,[22] which is a software-based tool widely used for active bandwidth measurement.[23] The results obtained with Iperf are hence used as our comparison baseline. Consequently, we configured Iperf to generate UDP packets to be comparable with CCM where messages are not

[21] https://www.netcraftsmen.com/tcpip-performance-factors/.

[22] https://iperf.fr/.

[23] Iperf measurements are based on UDP or TCP sessions.



(a) Legacy-SDN data plane bandwidth measurement and control plane load

(b) One-way bandwidth measurement with CCM messages

Fig. 9 Bandwidth measurement with legacy-SDN controller, ATS, and Iperf

acknowledged. On our platform, Iperf is capable of generating an average of 1.543 Gbps, while ATS running on a single CPU core is capable of generating an average of 0.677 Gbps. By increasing the number of CPU cores simultaneously generating the messages, ATS linearly increases the measured bandwidth. This is because each CPU core is capable of generating an average of ~ 50 000 packets per second on our system.

By comparing the above results with the performance requirements for high data rate and traffic density scenarios defined by 3GPP [1], we can see that legacy-SDN is not capable of measuring whether there is enough bandwidth even for the least demanding service (15 Mbps of experienced data rate). On the contrary, ATS is capable of generating more than 1 Gbps with 2 CPU cores, which is the expected data rate experienced per user in indoor scenarios. Such measurements are expected to be performed on-demand upon a path configuration request to verify the fulfillment of the bandwidth SLAs. Finally, it is worth highlighting that our ATS implementation is based on JavaScript for prototyping reasons, while Iperf is written in C, a language that provides considerably higher performance. Even though we matched and surpassed the performance of Iperf in generating traffic, this came at the cost of using more CPU cores.

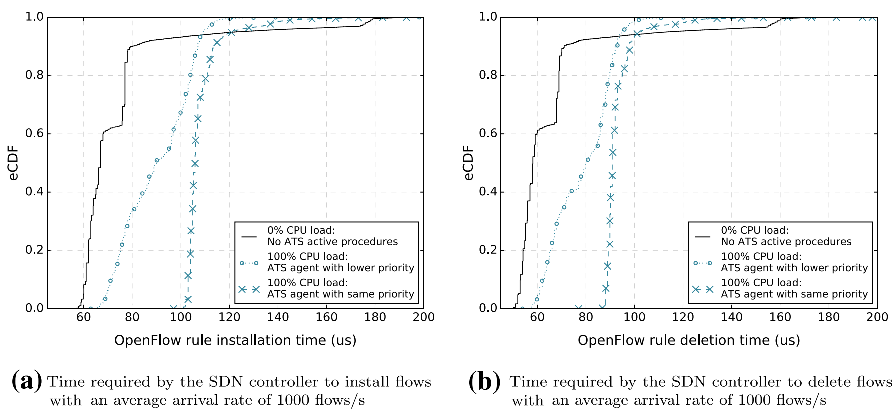
5.4 Implementation and Deployment Considerations

In addition to the comparative tests previously described, we performed some experiments to obtain deeper insight into ATS, especially in the CCM case. Special attention is paid to the CPU load and to the scalability with regard to the total number of ports. Particularly, we addressed the periodic generation of messages over multiple ports which is causing a computational outage on the legacy-SDN controller. To avoid such an issue in ATS, we opted for using a packet template stored in a template buffer associated with each port. Such an approach allows us to pre-load a template of the message on each port and to trigger its transmission every interval. Each transmission only requires the modification of few bytes in the buffer (e.g., sequence

number) thus reducing the total number of instructions to be executed. We tested our ATS implementation with a CCM interval of 3.3 ms on an emulated switch comprising 256 ports and we observed that the CPU load is: (i) mainly due to the interrupts generated by the high-precision timer, and (ii) almost independent of the number of active ports. As a result, our implementation is able to transmit a CCM message every 3.5 ms on each of the 256 ports whilst running on a single core.

A second additional test is performed to understand whether the CPU load generated by the ATS procedures introduces significant performance variations in the OpenFlow control plane, negatively affecting the network behavior. To that end, we analyzed the ATS impact on the time required by the OpenFlow agent to install or delete rules. The test puts under stress the system by running the ATS bandwidth measurement procedure (see Sect. 5.3) in all ports, resulting in a CPU load of 100%. Two distinct configurations are evaluated: (i) a lower processing priority is assigned to the ATS agent, and (ii) the same processing priority is assigned to the ATS agent and the OpenFlow agent. To stress even more the system, we assume a flow arrival of 1000 OpenFlow rules/s. Figure 10a, b illustrate the results and show that in the case of no active measurement, the OpenFlow agent process requires an average of 74.13 μs and 65.63 μs to install and delete an OpenFlow rule, respectively. In the case of running the CPU-intensive ATS procedures (e.g., bandwidth measurement), the switch respectively requires an average of 90.07 μs and 78.61 μs for respectively installing and deleting an OpenFlow rule in the case of the ATS process running at a lower priority. Likewise, the switch requires an average of 108.48 μs and 92.90 μs when the ATS process runs with the same priority as the OpenFlow agent.

At the light of these results, we can conclude that although the use of ATS (when performing CPU-intensive operations) impacts the performance of the OpenFlow control plane, this impact does not prevent the switch to install and delete rules within a delay of ∼ 108 μs. It is worth highlighting that while the use of the ATS process with the same priority (i.e., the worst case) introduces an additional delay



(a) Time required by the SDN controller to install flows with an average arrival rate of 1000 flows/s

(b) Time required by the SDN controller to delete flows with an average arrival rate of 1000 flows/s

Fig. 10 Time required by the SDN controller to install and delete flows on a switch subject to an average arrival rate of 1000 flows/s when the CPU is put under load by the ATS agent. Three scenarios are considered: (i) 0% CPU load, and 100% CPU load when the ATS agent process runs with (ii) a lower priority and (iii) same priority with regards to the OpenFlow process

of $\sim$ 34 μs, this does not represent a significant variation in the context of the OpenFlow control plane, which is based on TCP and operates at a longer timescale. To alleviate this problem, we therefore advise assigning a lower CPU priority to the ATS process, resulting in a higher reactiveness of the OpenFlow process and in a mitigation counter-measure against a potential ATS misconfiguration.

To conclude, time synchronization between all the network switches is required to measure i.e. the one-way delay and to have a common reference time for monitoring. In carrier grade networks there are two widely-adopted options for distributing the clock (a.k.a. frequency synchronization): IEEE 1588 [16] and Synchronous Ethernet [19]. The former defines a cost-effective packet-based clock distribution mechanism capable of providing a timestamp resolution of 8 ns with an accuracy 25 ns. The latter, instead, incorporates in the clock signal in the Ethernet physical layer, that is no ad-hoc messages for synchronization are sent, and it is capable of providing sub-nanosecond accuracy. While the former option still provides good accuracy for monitoring whilst being cheaper than the latter, it may occur that the clock distribution messages interfere with the network measurements and vice-versa.

Therefore, it is important to configure appropriate QoS policies on the switches so as to avoid the disruption of the clock distribution eventually caused by the network measurements. One possible solution is to assign a higher priority to the packets essential for clock synchronization and a lower priority to the packets required for network measurement. A comprehensive analysis of the available solutions for achieving clock synchronization over a packet-based network can be found in [23].

6 Conclusions

This article has identified a gap between current SDN solutions and carrier grade network requirements under OAM point of view. An analysis of widely-deployed OAM and SDN technologies has been hence performed showing that the stateless nature of OpenFlow poses significant scalability and accuracy problems in monitoring and managing the network. To overcome these issues, this paper proposes an Adaptive Telemetry System, namely ATS, to enable locally on the switches active measurements (e.g., delay, bandwidth, etc.) and their reporting (e.g., alarms). The design approach chosen for ATS showed to provide compatibility with standard OpenFlow switches and controllers. An Application Programming Interface (API) has been defined for enabling the remote configuration of telemetry procedures, which adopt a Finite State Machine (FSM) implementation. This enables the switches to locally execute the stateful procedures required for active monitoring.

Finally, an experimental evaluation has been presented, showing the benefits of ATS compared to legacy-SDN solutions. Particularly, ATS proved to bring significant benefits in terms of offloading the control plane (and network controller) and higher accuracy in the performed measurements, which comply with the performance requirements defined by 3GPP for 5G networks. To that end, the delay and bandwidth measurements obtained with ATS have proven to match the ones obtained with reference non-SDN tools, while providing higher flexibility in the

type of measurements that could be performed. Furthermore, ATS proved to be able to manage the periodical generation of messages over a large number of ports (up to 256) while running on a single CPU core. Finally, we provided some implementation insights on ATS and some deployment considerations regarding the process scheduler on the switch and the clock distribution in the network.

Acknowledgements This work has been partially funded by the H2020 Framework Programme Europe/Taiwan joint action 5G-DIVE Project (Grant No. 859881), by the H2020 Framework Programme EU 5G-Transformer Project (Grant No. 761586), and by the H2020 Framework Programme EU 5Growth Project (Grant No. 856709).

References

1. 3GPP: Service requirements for next generation new services and markets. TS 22.261, 3rd Generation Partnership Project (3GPP) (2018)
2. 3GPP: System Architecture for the 5G System. TS 23.501, 3rd Generation Partnership Project (3GPP) (2018)
3. Bari, M.F., Chowdhury, S.R., Ahmed, R., Boutaba, R.: Policycop: An autonomic qos policy enforcement framework for software defined networks. In: 2013 IEEE SDN for Future Networks and Services (SDN4FNS), pp. 1–7 (2013). https://doi.org/10.1109/SDN4FNS.2013.6702548
4. Bernardos, C.J., de la Oliva, A., Serrano, P., Banchs, A., Contreras, L.M., Jin, H., Zuniga, J.C.: An architecture for software defined wireless networking. IEEE Wirel. Commun. **21**(3), 52–61 (2014). https://doi.org/10.1109/MWC.2014.6845049
5. Bianchi, G., Bonola, M., Capone, A., Cascone, C.: Openstate: Programming platform-independent stateful openflow applications inside the switch. SIGCOMM Comput. Commun. Rev. **44**(2), 44–51 (2014). https://doi.org/10.1145/2602204.2602211
6. Bifulco, R., Boite, J., Bouet, M., Schneider, F.: Improving sdn with inspired switches. In: Proceedings of the Symposium on SDN Research, SOSR ’16, pp. 11:1–11:12. ACM, New York (2016). https://doi.org/10.1145/2890955.2890962
7. Bosshart, P., Gibb, G., Kim, H.S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., Horowitz, M.: Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM ’13, pp. 99–110. ACM, New York (2013). https://doi.org/10.1145/2486001.2486011
8. Bram, N., Mario, K., Sofie, V., Didier, C., Mario, P.: How can a mobile service provider reduce costs with software-defined networking? Int. J.Netw. Manag. **26**(1), 56–72 (2015). https://doi.org/10.1002/nem.1919
9. Cascone, C., Sanvito, D., Pollini, L., Capone, A., Sansò, B.: Fast failure detection and recovery in SDN with stateful data plane. Int. J. Netw. Manag. **27**(2), e1957 (2017). https://doi.org/10.1002/nem.1957
10. Claise, B.: Cisco Systems NetFlow Services Export Version 9. RFC 3954, Internet Engineering Task Force (IETF) (2004)
11. Claise, B., Trammell, B., Aitken, P.: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011, Internet Engineering Task Force (IETF) (2013)
12. Egilmez, H.E., Civanlar, S., Tekalp, A.M.: An optimization framework for qos-enabled adaptive video streaming over openflow networks. IEEE Trans. Multimed. **15**(3), 710–715 (2013). https://doi.org/10.1109/TMM.2012.2232645
13. Guck, J.W., Bemten, A.V., Reisslein, M., Kellerer, W.: Unicast qos routing algorithms for SDN: a comprehensive survey and performance evaluation. IEEE Commun. Surv. Tutor. **20**(1), 388–415 (2018). https://doi.org/10.1109/COMST.2017.2749760
14. Gupta, A., Harrison, R., Canini, M., Feamster, N., Rexford, J., Willinger, W.: Sonata: Query-driven streaming network telemetry. In: Proceedings of the 2018 Conference of the ACM Special

Interest Group on Data Communication, SIGCOMM'18, pp. 357–371. ACM, New York (2018). https://doi.org/10.1145/3230543.3230555

15. IEEE: Connectivity Fault Management. Standards for Local and metropolitan area networks 8021.ag. Piscataway, Institute of Electrical and Electronics Engineers (2007)
16. IEEE: Precision Clock Synchronization Protocol. Standard for Networked Measurement and Control Systems 1588. Piscataway, Institute of Electrical and Electronics Engineers (2008)
17. IEEE: Bridges and Bridged Networks Amendment: YANG Data Model. Standards for Local and metropolitan area networks 802.1Qcp. Piscataway, Institute of Electrical and Electronics Engineers (2017)
18. ITU-T: Operations, administration and maintenance (OAM) functions and mechanisms for Ethernet-based networks. Reccomendation G.8013/Y.1731. ITU Telecommunication Standardization Sector, Geneva (2015)
19. ITU-T: Timing characteristics of a synchronous Ethernet equipment slave clock. Reccomendation G.8262. ITU Telecommunication Standardization Sector, Geneva (2015)
20. ITU-T: Characteristics of Ethernet transport network equipment functional blocks. Reccomendation G.8021/Y.1341. ITU Telecommunication Standardization Sector, Geneva (2016)
21. ITU-T: Specification and Description Language—Comprehensive SDL-2010. Reccomendation Z.102. ITU Telecommunication Standardization Sector, Genava (2016)
22. ITU-T: Consideration on 5G transport network reference architecture and bandwidth requirements. Contribution 0462. ITU Telecommunication Standardization Sector, Study Group 15, Genava (2018)
23. Lévesque, M., Tipper, D.: A survey of clock synchronization over packet-switched networks. IEEE Commun. Surv. Tutor. **18**(4), 2926–2947 (2016). https://doi.org/10.1109/COMST.2016.2590438
24. McKeown, N.: Software-defined networking. INFOCOM Keynote Talk **17**(2), 30–32 (2009)
25. MEF: Service OAM Fault Management YANG Module. Specification MEF 38. Metro Ethernet Forum, Los Angeles (2012)
26. MEF: Service OAM Performance Monitoring YANG Module. Specification MEF 39. Metro Ethernet Forum, Los Angeles (2012)
27. Moshref, M., Bhargava, A., Gupta, A., Yu, M., Govindan, R.: Flow-level state transition as a new switch primitive for sdn. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, HotSDN '14, pp. 61–66. ACM, New York (2014). https://doi.org/10.1145/2620728.2620729
28. ONF: OpenFlow switch specification. Version 1.5.1, Open Networking Foundation (2015)
29. Phaal, P., Panchen, S., McKee, N.: InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176. Internet Engineering Task Force, Fremont (2001)
30. Pontarelli, S., Bifulco, R., Bonola, M., Cascone, C., Spaziani, M., Bruschi, V., Sanvito, D., Siracusano, G., Capone, A., Honda, M., Huici, F., Siracusano, G.: Flowblaze: stateful packet processing in hardware. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pp. 531–548. USENIX Association, Boston, MA (2019)
31. Sonchack, J., Aviv, A.J., Keller, E., Smith, J.M.: Turboflow: Information rich flow record generation on commodity switches. In: Proceedings of the Thirteenth EuroSys Conference, EuroSys'18, pp. 11:1–11:16. ACM, New York (2018). https://doi.org/10.1145/3190508.3190558
32. Sonchack, J., Michel, O., Aviv, A.J., Keller, E., Smith, J.M.: Scaling hardware accelerated network monitoring to concurrent and dynamic queries with flow. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18), pp. 823–835. USENIX Association, Boston, MA (2018)
33. Tilmans, O., Bühler, T., Poese, I., Vissicchio, S., Vanbever, L.: Stroboscope: Declarative network monitoring on a budget. In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pp. 467–482. USENIX Association, Renton, WA (2018)
34. Tomovic, S., Prasad, N., Radusinovic, I.: Sdn control framework for qos provisioning. In: 2014 22nd Telecommunications Forum Telfor (TELFOR), pp. 111–114 (2014). https://doi.org/10.1109/TELFOR.2014.7034369
35. Tomovic, S., Radusinovic, I., Prasad, N.: Performance comparison of qos routing algorithms applicable to large-scale sdn networks. In: IEEE EUROCON 2015-International Conference on Computer as a Tool (EUROCON), pp. 1–6 (2015). https://doi.org/10.1109/EUROCON.2015.7313698

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Luca Cominardi received his Bachelor's and Master's degrees in Computer Science at the University of Brescia, Italy, in 2010 and 2013, respectively. He received his Master's degree and Ph.D. in Telematics Engineering from the University Carlos III of Madrid (UC3M), Spain, in 2014 and 2019, respectively. Starting from 2019 he is a Senior Technologist at ADLINK Technology working on edge and fog computing as well as distributed systems. He is an active contributor to the Eclipse Edge Native Working Group and ETSI MEC Working Group, serving also as rapporteur in the latter. He has published more than 15 papers in international journals and issued 5 patents.

Sergio Gonzalez-Diaz received his Bachelor's and Master's degrees in telematics engineering from the University Carlos III of Madrid (UC3M) in 2015 and 2017 respectively. Currently he is working at UC3M, where he is also pursuing his Ph.D. in the same field, focusing his research on programmable networks and network virtualization, on which he has published several papers in international conferences and journals.

Antonio De La Oliva received his telecommunications engineering degree in 2004 and his Ph.D. in 2008 from the Universidad Carlos III Madrid (UC3M), Spain, where he has been an associate professor since then. He is an active contributor to IEEE 802 where he has served as Vice-Chair of IEEE 802.21b and Technical Editor of IEEE 802.21d. He has also served as a Guest Editor of IEEE Communications Magazine. He has published more than 30 papers on different networking areas.

Carlos J. Bernardos received the degree in telecommunication engineering and the Ph.D. degree in telematics from the University Carlos III of Madrid (UC3M), in 2003 and 2006, respectively. From 2003 to 2008, he was a Research and Teaching Assistant with UC3M, where he has been an Associate Professor, since 2008. He has published over 100 scientific papers in prestigious international journals and conferences. He is an active contributor to IETF since 2005, e.g. to AUTOCONF, MEXT, NETEXT, DMM, MULTIMOB, SDNRG and NFVRG working/research groups, being co-author of more than 30 contributions, 10 RFCs, has co-chaired the IETF P2PSIP WG, and currently co-chairs the IPWAVE WG and the Internet Area Directorate (INTDIR). He visited the Computer Laboratory of University of Cambridge in 2004 and the University of Coimbra in 2005. He has worked in several EU funded projects, being the technical manager of the FP7 MEDIEVAL and H2020 5G-Exchange projects, and the Project Coordinator of the 5G-TRANSFORMER and 5Growth projects. His current work focuses on virtualization applied to 5G networks.

Affiliations

Luca Cominardi[1] · Sergio Gonzalez-Diaz[2] · Antonio de la Oliva[2] · Carlos J. Bernardos[2]

Sergio Gonzalez-Diaz
serggonz@pa.uc3m.es

Antonio de la Oliva
aoliva@it.uc3m.es

Carlos J. Bernardos
cjbc@it.uc3m.es

[1] ADLINK Technology, Saint-Aubin, France

[2] University Carlos III of Madrid, Madrid, Spain