Chapter 20
*ubi*REST: A RESTful Service-Oriented Middleware for Ubiquitous Networking

Mauro Caporuscio, Marco Funaro, Carlo Ghezzi and Valérie Issarny

Abstract The computing and networking capabilities of today's wireless mobile devices allow for seamlessly-networked, ubiquitous services, which may be dynamically composed at run-time to accomplish complex tasks. This vision, however, remains challenged by the inherent mobility of such devices, which makes services highly volatile. These issues call for a service-oriented middleware that should (*i*) deal with the run-time growth of the application in terms of involved services (*flexibility*), (*ii*) accommodate heterogeneous and unforeseen services into the running application (*genericity*), and (*iii*) discover new services at run time and rearrange the application accordingly (*dynamism*). This chapter discusses the design and implementation of *ubi*REST, a service-oriented middleware that leverages REST principles to effectively enable the ubiquitous networking of Services. *ubi*REST specifically defines a layered communication middleware supporting RESTful Services while exploiting nowadays ubiquitous connectivity.

M. Caporuscio (✉) · M. Funaro · C. Ghezzi
Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza L. da Vinci 32, 20133 Milan, Italy
e-mail: mauro.caporuscio@polimi.it

M. Funaro
e-mail: funaro@elet.polimi.it

C. Ghezzi
e-mail: carlo.ghezzi@polimi.it

V. Issarny
INRIA Paris-Rocquencourt, Domaine de Voluceau, Le Chesnay 78153, France
e-mail: valerie.issarny@inria.fr

A. Bouguettaya et al. (eds.), *Advanced Web Services*,
DOI: 10.1007/978-1-4614-7535-4_20,

20.1 Introduction

With network connectivity being embedded in most computing devices, any device may seamlessly consume, but also provide, software applications over the network. Service-Oriented Computing (SOC) is a natural design abstraction to deal with ubiquitous networking environments [3]. Applications may conveniently be abstracted as autonomous loosely-coupled services, which may be composed to accomplish complex tasks. A service composition forms into a *network-based application*, which relies on the explicit distribution of services interacting by means of message passing. *Network-based applications* differ from *distributed applications* because the involved networked resources are independent and autonomous, rather than viewed as integral part of a conceptually monolithic system [47].

Issues related to the design/development of network-based systems have been largely discussed in literature, and several middleware solutions, providing different types of resource's abstraction (e.g., remote procedure, object, component, service), have been proposed to deal with them. However, such middleware solutions rely on the assumption that the underlying network is stable. Whereas, concerning ubiquitous networking, such assumption is no longer valid due to the intrinsic dynamism and resource mobility (both physical and logical) [42]. Indeed, ubiquitous applications emerge from the spontaneous aggregation of the resources available (within the environment) at a given time, and thus are characterized by a highly dynamic software architecture where both the resources that are part of the architecture and their interconnections may change dynamically, while applications are running. In these settings, two main problems must be faced: (*i*) achieving the ubiquitous networking environment on top of heterogeneous communication media, and (*ii*) providing a flexible architectural style, which allows for designing and developing applications resilient to such an extreme variability.

In ubiquitous networking environments applications run on devices (e.g., tablets and smartphones), which are usually interconnected through one or more heterogeneous wireless links, which are characterized by lower bandwidth, higher error rates, and frequent disconnections. Hence, key feature of ubiquitous networking environments is the diversity of radio links available on portable devices, which may be exploited towards ubiquitous connectivity. While computationally suitable for ubiquitous applications, such devices usually have serious issues with battery life when the computational burden grows. Thus, the middleware should be able to energetically optimize the communication through scheduling and handover across different radio links [41, 44]. This requires services to be network-agnostic [45], while the underlying middleware is in charge of exchanging messages over the network links that best matches Quality of Service (QoS) requirements [10], and further ensuring service continuity through vertical handover (handover between different protocols) [21]. In this setting, a primary requirement for supporting service-oriented middleware is to provide a comprehensive *networking abstraction* that allows applications to be unaware of the actual underlying networks while exploiting their diversities in terms of both functional and extra-functional properties.

As for the architectural layer, applications should support adaptive and evolutionary situation-aware behaviors. *Adaptation* refers to the ability to react to environmental changes to keep satisfying the requirements, whereas *evolution* refers to the ability of satisfying new or different requirements [6]. In order to be self-adaptable and easily evolvable, applications should exploit design models able to: (*i*) deal with the run-time growth of the application in terms of involved resources (*flexibility*), (*ii*) accommodate heterogeneous and unforeseen functionalities into the running application (*genericity*), and (*iii*) discover new functionalities at run time and rearrange the application accordingly (*dynamism*).

This chapter presents the *ubi*REST middleware, which enhances the *ubi*SOAP approach [9] by providing RESTful access to services. Specifically, *ubi*REST adopts the P-REST architectural style [7], a refinement of the REST style [14] that we have introduced to fulfill the aforementioned requirements, namely *flexibility*, *genericity*, and *dynamism*.

The chapter is organized as follows: Sect. 20.2 summarizes related work. Section 20.3 discusses the design rationale for *ubi*REST, whereas Sects. 20.4, 20.5, and 20.6 detail the core functionalities of *ubi*REST, namely *network-agnostic connectivity*, *ubi*REST *communication*, and *ubi*REST *programming model*, respectively. Section 20.7 presents an example showing how to develop a simple RESTfull ubiquitous application. Finally, Sect. 20.8 concludes the chapter, and sketches our perspectives for future work.

20.2 Related Work

Work related to *ubi*REST ranges different research areas from multi-radio networks integration to ubiquitous computing and service technologies.

*ubi*REST aims at providing a communication layer enabling RESTful services within ubiquitous networking environments. To effectively enable mobile RESTful services, *ubi*REST comprehensively exploits the ubiquitous networking environment by dealing with multi-radio networking on the mobile device. Concerning this issue, the Third Generation Partnership Project (3GPP) defines a standard layered architectures (decomposing into the network, control and service layers) enabling service-oriented applications in the B3G network [2]. In that direction, recent proposal that aims at interconnecting various networks at once, has been published by the ITU under the name of IMT-Advanced (also known as 4G) [22]. Main goal of IMT-Advanced is to achieve "Always Best Connected" property by embedding broadband in all types of consumer devices. Interactions among networks include horizontal (intra network) and vertical (inter network) handover for service continuity, and encompass complex functions such as billing and QoS. This de-facto eliminates the need for the user to know anything about the network (e.g., topology, radio). However, both systems require the network operator to deploy new entities within the network that allow the native infrastructures to work together. Contrary to this closed, network-controlled approach, *ubi*REST provides a set of abstractions

enabling clients to autonomously adapt to the available networks, and to benefit from networks characteristics. This requires neither to modify the network infrastructure nor to establish contracts with a predetermined network operator.

Concerning ubiquitous computing at large, the literature proposes different middleware classes, each addressing a specific issue: (*i*) Context-aware middleware [13] deal with leveraging context information to provide user-centric computation, (*ii*) Mobile computing middleware [34] aim at providing communication and coordination of distributed mobile-components, (*iii*) Adaptive middleware [35] enable software to adapt its structure and behavior dynamically in response to changes in its execution environment. However, each middleware provides an ad hoc approach, whereas standards-compliant solutions are still missing.

Moreover, many middleware proposals aim at supporting the development of ubiquitous applications through the provision of different abstractions—e.g., objects, components, and services. The Obje framework [12] is an object-oriented framework where networked devices appear to applications as objects that implement specific "meta-interfaces". Such "meta-interfaces" are further used by applications to exchange the behaviors needed to achieve compatibility at run time. The methods of such interfaces make use and return objects that themselves implement well-known interfaces. Hence, the loading of objects is made transparent to user applications, which simply see them as new implementations of already-known interfaces. The framework described in [20], addresses the distribution and deployment of components throughout the ubiquitous networking environment. It provides developers with an architecture description language to specify constrains on components, which can be considered at deploy time to find a distribution scheme satisfying all constraints. Service-Oriented Computing (SOC) provides natural design abstractions to deal with ubiquitous environments. Networked applications are abstracted as autonomous loosely-coupled services, which may be dynamically combined to accomplish complex tasks [3]. ReMMoC [15] and *ubi*SOAP [9] provide middleware functionalities supporting service provision over ubiquitous networks.

In particular, focusing on Service-Oriented Computing, the widespread adoption of WS technologies combined with mobile networking has led to investigating the definition of architectures dedicated to mobile Web services [19, 23]. Overall, existing efforts towards enabling mobile Web services platforms address the development of service-oriented applications on mobile, wireless devices that act mostly as Web service clients. However, todays device technologies enable mobile devices to act as Web services providers. To this extent, many optimizations for SOAP have been proposed to improve memory and CPU usage [49], as well as to improve the bandwidth requirement of SOAP communication [43, 51, 52].

The idea of exploiting RESTful principles beyond the Wed is not new, and some research projects have been investigating how to apply the REST architectural style to different fields—e.g., ubiquitous computing and web of things. For example, [26, 33] leverage RESTfulness in the context of Ambient Computing, whereas [16, 18] exploit REST principles to achieve the Internet of Things. However, these approaches rely on Web standards to achieve interoperation, and therefore they suffer from the Web's limitations, e.g., lack of mobility management, point-to-point communica-

tion, and client-server interaction style. To this end, the XWeb [37] project presents a web-oriented architecture relying on a new transport protocol, called XTP, which provides mechanisms for finding, browsing, and modifying information.

Furthermore, in [4] RESTfulness has been exploited to achieve scalability, by means of replication of resources, in the context of Web Services. In particular, the REactor (RESTful Actor) framework provides a RESTful Web service interface and a composable architecture which is capable of delivering scalability and high performance independently from the underlying deployment infrastructure. Moreover, due to its scalability and the flexibility, REST architectural style has been employed also for monitoring and controlling data- and computationally-intensive tasks, such as in the context of Grid [30].

However, even thought REST has been gaining wide popularity as well suited solution for a large class of problems, to the best of our knowledge, *ubi*REST is the first attempt to design and develop a resource-oriented middleware, which specifically supports the development of ubiquitous RESTful services by addressing all the above aspects together.

20.3 *ubi*REST Design Rationale

*ubi*REST has been conceived and designed to provide RESTful access to services over ubiquitous-networking environments. To this extent, *ubi*REST aims at effectively exploiting all the diverse network technologies at once to create an integrated multi-radio networking environment, hence offering *network-agnostic connectivity* to services. On the other hand, *ubi*REST aims to provide proper programming abstractions enabling service-oriented applications to adapt and evolve at run time.

Achieving the network-agnostic connectivity requires addressing a number of critical issues such as *network availability*, *user and application QoS requirements* and *vertical handover*. *Vertical handover* [50] is particularly important with respect to the *service continuity* requirement. Indeed, when a host changes its point of attachment (vertical handover between two networks), the IP address is modified accordingly in order to route packets to the new network. Hence, since the IP address is the base of any Internet transmission [36], all the ongoing connections break. Moreover, as devices can bind various networks at the same time, two interacting parties might communicate through multiple paths. Hence, choosing the best connection to serve a given interaction is a key issue to deal with in ubiquitous networks, as this significantly affects the QoS at large (e.g., availability, performance with respect to both resource consumption and response time, security) [5].

Pervasive applications are composed as composition of (heterogeneous) independent services, forming into a *network-based application*. Since mobility makes services available/unavailable suddenly, ubiquitous applications emerge de-facto from the spontaneous aggregation of the services available (within the environment) at a given time. As result, ubiquitous applications are characterized by a highly dynamic software architecture where both the services that are part of the archi-

tecture and their interconnections may change dynamically, while applications are running.

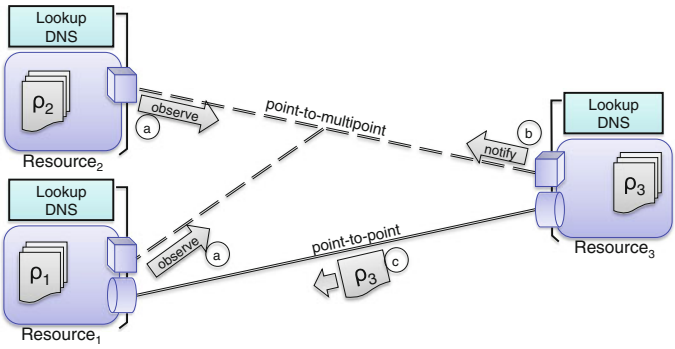

Fig. 20.1 P-REST architectural style

Issues related to the design/development of ubiquitous systems have been largely discussed in literature, and many middleware, providing different types of abstraction—e.g., objects [12], components [20], and services [9]—have been proposed to deal with them. Departing from these approaches, *ubi*REST tackles the problem by adhering to REST principles [14]: *addressability*, *statelessness*, *connectedness*, and *uniformity*. However, due to the inherent complexity of ubiquitous environments, the REST architectural style cannot be directly applied to them. Hence, we enhanced it by creating the P-REST (Pervasive REST) architectural style, which refines REST to specifically address ubiquitous networking environments, while keeping REST principles unaltered.

P-REST (see Fig. 20.1) promotes the use of *Resource* as first-class object that plays the role of "prosumer" [40], i.e., fulfilling both roles of producer and consumer. To support coordination among resources, P-REST extends REST with a set of new facilities: (*i*) a *Lookup* service enabling the run-time discovery of new resources, (*ii*) a distributed *Domain Name System* (DNS) service mapping resource URIs to actual location in case of mobility, and (*iii*) a coordination model based on the *Observer* pattern [25] allowing a resource to express its interest in a given resource and to be notified whenever changes occur in it.

Following the P-REST style, resources interact with each other by exchanging their representations. Referring to Fig. 20.1, both Resource_1 and Resource_2 observe Resource_3 (messages 1). When a change occurs in Resource_3, it notifies (message 2) the observer resources. Upon receipt of such notification, Resource_1 issues a request for the Resource_3 and obtains as a result the representation ρ_3 (message 3). Note that, while observe/notify interactions take place through the *point-to-multipoint* connector (represented as a cube), REST operations exploit *point-to-point* connector (represented as a cylinder). All the resources exploit both the *lookup* operation to

discover the needed resources, and the DNS service to translate URIs into physical addresses.

P-RESTful applications are built following the P-REST conceptual model [7], which defines: (1) a *environment* as a resource container providing infrastructural facilities (i.e., *lookup* and *observe/notify*); (2) a *resource* as a first-class object that, according to the REST *uniformity* [14], implements a fixed set of well-defined operations (i.e., `PUT`, `DELETE`, `POST`, `GET`, and `INSPECT`); (3) a semantics-aware *description* specifying both functional and non-functional properties of resources with respect to given ontologies.

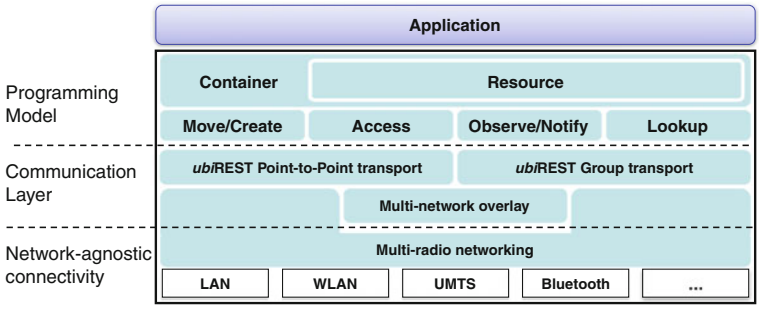

Fig. 20.2 *ubi*REST software architecture

Resources interact with each other by exchanging their *representation*s, which capture the current state of a resource. Furthermore, every Resource is bound to at least one *concrete URI* (CURI). P-REST enhances the concept of URI by introducing *abstract URI*s (AURI). An AURI is a URI identifying a group of resources. Indeed, a CURI allows for point-to-point communication, whereas an AURI allows for group communication. Resources can be used as building-blocks for composing complex functionalities. A *composition* is a resource that can, in turn, be used as a building-block by another composition. Resources involved in a composition are handled by means of a *composition logic*.

20.3.1 Run-Time Support

The *ubi*REST middleware provides the run-time support for the development of P-RESTful service-oriented applications by realizing P-REST at the infrastructure level, and providing developers with effective P-RESTful abstractions. Note that *ubi*REST cannot enforce REST principles at the application level, which is totally entrusted to the designer.

Referring to Fig. 20.2, the *ubi*REST architecture exploits a three-layer design where each layer deals with a specific issue.

Network-agnostic connectivity—Providing network-agnostic connectivity within ubiquitous networks relates to abstracting the rich and heterogeneous networking environment for reasoning about the networks characteristics and seamlessly manage them. To this extent, *ubi*REST goal is to support: (*i*) network abstractions to provide connectivity regardless of the actual underlying network technology, (*ii*) the selection of the best possible network matching the QoS needs expressed by the end user, and (*iii*) the unique identification and addressing of users applications within the networking environment irrespectively of their physical location.

Communication layer—To deal with the inherent instability of ubiquitous networking environments, *ubi*REST arranges devices in an Multi-network overlay built on top of *Network-agnostic connectivity* layer. Such an overlay is then exploited to provide two basic communication facilities, namely *point-to-point* and *group transport*. *Point-to-point* transport grants a given node direct access to a remote node, whereas *group* transport allows a given node to interact with many different nodes at the same time. Furthermore, the *ubi*REST communication layer provides facilities for managing code mobility [42].

Programming model—*ubi*REST provides the programming abstractions to implement P-RESTful applications by leveraging the functional programming features of the Scala language [48] and the Actor Model [1]. In particular, *ubi*REST defines two main abstractions and a set of operations to be performed on them. *Resource* represents the computation unit, whereas *Container* handles both the life-cycle and the provision of resources. The set of operations allowed on resources defines the message-based *ubi*REST *interaction protocol* and includes: (*i*) *Access*, which gathers the set of messages to access and manipulate resources, (*ii*) *Observe/Notify*, which allows resources to declare interest in a given resource and to be notified whenever changes occur, (*iii*) *Create*, which provides the mechanism for creating a new resource at a given location, and *Move*, which provides the mechanism to relocate an existing resource to a new location, and (*iv*) *Lookup*, which allows for discovering new resources on the basis of a given semantics-aware description.

*ubi*REST fulfills the set of requirements introduced above. *Flexibility* is achieved by exploiting the Actor Model, which in turn relies on the *ubi*REST *communication* to provide message-passing interaction among actors. *Genericity* arises from the uniformity principle exploited in conjunction with both code mobility and functional programming capabilities (e.g., high-order functions). *Dynamism* is provided by means of semantic lookup, uniformity and resource composition. The following sections clarify these aspects, and detail *network-agnostic connectivity* (Sect. 20.4), *ubi*REST *communication* (Sect. 20.5), and *ubi*REST *programming model* (Sect. 20.6), respectively.

20.4 Network-Agnostic Connectivity

In this section, for the sake of self-containment, we report an excerpt of the *network-agnostic connectivity* layer implementation [9].

The *network-agnostic connectivity* layer offers the core functionalities to effectively manage the underlying multi-radio environment through: (*i*) a network-agnostic addressing scheme together with (*ii*) QoS-aware network link selection and (*iii*) base unicast and multicast communication schemes.

Such Multi-Radio Networking (MRN) functionality are provided by means of two modules (see Fig. 20.3): (*i*) a Multi-Radio Networking Daemon (MRN-Daemon) that implements the provided features, and (*ii*) a Multi-Radio Networking API (MRN-Api) that allows for an easy and transparent access to the functionalities offered by MRN-Daemon. Furthermore, a *ubi*LET is any entity (e.g., application) that exploits the *network-agnostic connectivity* layer by accessing the functionalities provided by MRN-Daemon through MRN-Api.

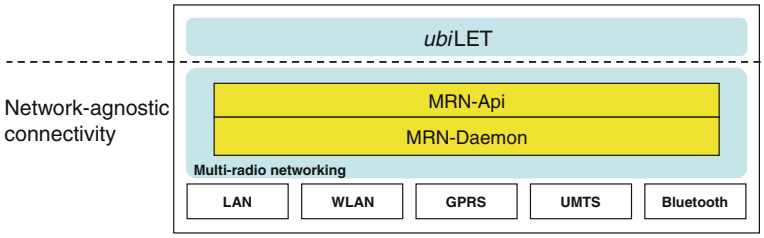

Fig. 20.3 Network-agnostic connectivity layer software architecture

In particular, MRN-Daemon is in charge of managing the entire communication between two devices through the underlying radio networks. It runs on each device and is accessible by many applications at the same time. It is also in charge of managing the *ubi*REST addressing scheme as well as its mapping to the actual set of IP addresses. On the other hand, MRN-Api is a component, embedded in the application, used to interact with the MRN-Daemon. It offers a set of high-level API allowing for an easy and transparent access to the services offered by he MRN-Daemon. Indeed, in order to communicate with each other, services deployed on *ubi*REST-enabled devices must use the functionalities provided by the MRN-Daemon through the MRN-Api.

Since *ubi*REST aims at running on resources-scarce platforms (e.g., PDA and mobile phones), which have limited CPU power, memory, and battery life, to best fit the resources available on the hosting device, *ubi*REST provides two different (but equivalent and fully compatible) implementations of the *network-agnostic connectivity* layer, namely *shared* and *embedded*.

A *shared network-agnostic connectivity* layer is implemented as a "shared" instance of MRN-Daemon, which is simultaneously accessed by multiple *ubi*LETs

(through the API provided by MRN-Api). Since all the *ubi*LETs access the same instance, possible conflicts can be solved in an automated way (i.e., two *ubi*LETs expressing conflictual QoS requirements over the interface activation).

On the other hand, an *embedded network-agnostic connectivity* layer is implemented by "embedding" the MRN-Daemon into the MRN-Api. In this case, each *ubi*LET accesses a different, and standalone, instance of MRN-Daemon. This solution is lighter than the *shared* one, and is obviously appropriate when there exists only one *ubi*LET per device. In fact, the *shared* MRN-Daemon interacts with MRN-Api by means of a TCP socket bound to the loopback interface. This requires for having a synchronized thread-pool managing the incoming concurrent requests. It thus implies both larger memory footprint and computational needs. However, *embedded* MRN-Daemons cannot communicate with each other and then, they cannot synchronize to solve possible conflicts.

Network-agnostic addressing—Devices embedding multiple network interfaces (e.g., WLAN and Bluetooth) may have multiple IP addresses, at least one for each active interface. Thus, in order to identify uniquely a given *ubi*LET in the network we associate it with a *Multi-Radio Networking Address* (MRN@). The MRN@ of a *ubi*LET instance is specifically the application's Unique ID, which maps into the actual set of IP addresses (precisely, $network\_ID \oplus IP$ addresses) bound to the device (at a given time) that runs the given instance. Referring to Fig. 20.4, the MRN@ associated to the *ubi*LET$_j$ running on Alice's device is:

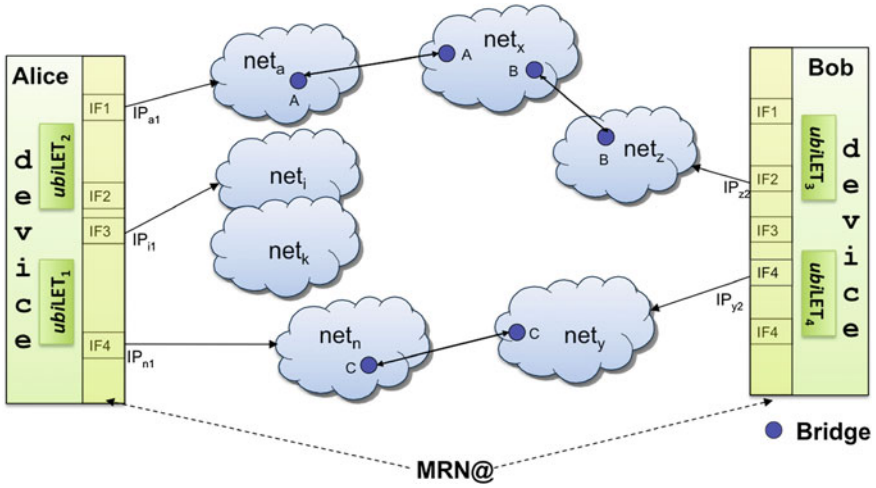

Fig. 20.4 Network-agnostic addressing over Multi-network overlay

$$MRN@_{ubiLET_j} \mapsto \{net_a \oplus IP_{a_1}, net_i \oplus IP_{i_1}, net_n \oplus IP_{n_1}\}$$

where $\forall j \in \{1, 2\}$, $MRN@_{ubiLET_j}$ is the ID of *ubi*LET$_j$ and, $\{net_a \oplus IP_{a_1}, net_i \oplus IP_{i_1}, net_n \oplus IP_{n_1}\}$ is the set of $network\_ID \oplus IP$ addresses denoting the actual location

of the device.[1] Then, upper layers shall use MRN@ as part of their addressing scheme (e.g., through WS-addressing in the case of Web services), which replaces the traditional IP-based addressing scheme. MRN@s are automatically generated and managed by multi-radio networking. Furthermore, multi-radio networking allows for performing a lookup operation that, starting from an MRN@, returns the set of IP addresses actually bound to it. The basic operations provided by network-agnostic connectivity are as follows. First, *Registration* allows the *ubi*LET to register within the network-agnostic connectivity layer and generates the MRN@ that uniquely identifies it. In particular, the *ubi*LET (i.e., user application) provides as input an identifier (locally unique), which is used to generate the MRN@ to be returned. Then, *Lookup* allows user applications to retrieve the actual set of IP addresses related to a given MRN@. If the resolution of MRN@ is not cached or needs to be updated, a request is multicasted to all the networks currently accessible and, if the device related to such MRN@ is reached, it will directly reply to the requester by supplying the actual set of IP addresses.

QoS-aware network link selection—Next to MRN@ addressing, it is crucial to activate and select the best possible networks (among those available) with respect to required QoS, which is defined as a set of pairs $< QoS_{attr}, QoS_{value} >$. Attributes are grouped in two subsets: (*i*) *quantitative* attributes that describe the performance provided by the networks—e.g., bitrate, packet loss transfer delay and signal strength—and allows for networks ranking, and (*ii*) *qualitative* attributes that describe those characteristics of the network that do not affect the network performance but should be considered—e.g., power consumption, price, coverage area. Departing from WS-oriented approaches (e.g., WS-Policy) that are "asymmetric" and they do not allow service consumers to specify their requirements, *ubi*REST strives to enable network QoS negotiation by trying to meet both provider and consumer requirements (expressed in terms of network QoS). To this extent, *ubi*REST provides two functionality, namely *interface activation* and *network selection*. *Interface activation* allows the user application to activate the best possible interfaces (among those available) with respect to the required QoS. In particular, the application submits its QoS requirement (a set of pairs $< QoS_{attribute}, QoS_{value} >$) to multi-radio networking, which in turn compares it with the QoS of each available interface (Network-side QoS and Context). In this case, since the interface is switched off, QoS refers to the theoretic values of a network interface declared by the manufacturer (e.g., GPRS maximum bitrate $=$ 171.2 Kb/s). If the interface satisfies the requirement posed by the application, within a given approximation expressed in percentage, it is activated. It is also possible to define priorities upon the various quantitative parameters, in order to specify if a given parameter is more important than the others. On the other hand, *network selection* is performed during the establishment of the communication and takes into account the QoS attributes required by the client application that is initiating the connection, as well as the networks active on the server listening for incoming connections, as given by the servers MRN@. If the client and

[1] For the sake of simplicity we refer to IP address, but it is actually implemented as IP address and port number, e.g., `128.131.10.1:90`.

the server share only one network that satisfies the requirements, it is used to carry on the interaction. On the other hand, when the two parties share more than one network, the selection algorithm selects the one that best meets the required QoS.

Multi-radio unicast and multicast—Once defined the MRN@ addressing scheme and the operations enabling the network link selection, the *network-agnostic connectivity* layer provides two base communication facilities: *synchronous unicast* and *asynchronous multicast*.[2] *ubi*REST *synchronous unicast* allows for messaging communication between two *ubi*LETs sharing at least one network. Specifically, it is provided by means of a logical stream channel that is used by the *ubi*LETs to read/write the packets belonging to the ongoing communication. Whereas, *ubi*REST *asynchronous multicast* allows for multicast messaging communication within a group of *ubi*LETs sharing at least one network. Specifically, it is provided by means of multicast packets that are sent to all members of a given group.

20.5 *ubi*REST Communication Layer

Providing communication within ubiquitous networks relates to comprehensively exploiting the rich, heterogeneous networking environment for message handling. In particular, *ubi*REST goal is to support: (1) *Mobility* so that active connections are maintained transparently to the application layer despite the mobility of nodes, as long as a network path exists, (2) efficient *messages routing* in multi-paths configurations (i.e., when multiple network paths exist between the consumer and the provider), (3) both *point-to-point* and *group* communications using the same abstractions (i.e., MRN@), and (4) *multi-network routing* so that access to resources in distant networks is enabled as long as there exists a path bridging the heterogeneous networks between the consumer and target resource provider.

To meet the above, *ubi*REST arranges devices in a multi-network overlay, a virtual network of nodes and logical links built on top of existing actual networks [11] and meant to augment the native network with new services. The *ubi*REST overlay network manages the logical links between nodes (i.e., resources) and enables message exchange. In particular, *ubi*REST embeds (*i*) the protocols that keep the overlay network connected when the topology of the underlying native network changes (e.g., as a consequence of mobility), and (*ii*) the routing algorithms that regulate the message flow between nodes according to the specific coordination model used, namely point-to-point communication and group communication. In pervasive environments, a key requirement for the overlay is the ability to self-organize itself into a flexible topology, as well as to maintain it. To this extent, *ubi*REST defines a custom transport layer that leverages *network-agnostic connectivity* and provides: (*i*) the multi-network overlay in charge of forwarding messages across independent networks, and (*ii*) two transports for point-to-point and group communication in ubiquitous networking.

[2] The interested reader is referred to [9] for further details.

Multi-network overlay—Thanks to the *ubi*REST *network-agnostic connectivity* layer, communication among nodes exploits the various network links that the nodes have in common by selecting the links that provide the required QoS. However, in some cases, it might also be desirable for nodes to be able to access resources that are hosted in *distant* networks to which the requesting node is not directly connected to (e.g., to provide continuity of service despite node mobility). For example, in Fig. 20.4, the device of Alice is connected to networks `a`, `i`, and `n`, through its various network interfaces. Clearly, the device can trivially access resources hosted in these networks. However, it cannot access resources hosted by Bob's device that is located in the distant networks `x`, `y`, and `z`. In fact, the *network-agnostic connectivity* layer does not provide neither an overlay IP network nor multi-network routing.

However, relying on the MRN@, together with both *unicast* and *multicast communication* schemes, *ubi*REST introduces an overlay network that is able to bridge heterogeneous networks, thus enhancing overall connectivity. In particular: (*i*) MRN@ addressing provides a two-layer identification scheme (i.e., $network\_ID \oplus IP$) allowing for uniquely identifying a device irrespectively of the network it belongs to, and (*ii*) *unicast* and *multicast communication* support allows for MRN@ management across the networks. Specifically, nodes that are connected to two (or more) different networks through their network interfaces can assume the role of *bridge* nodes. *Bridge* nodes quite literally "bridge" between two separate networks, relaying *ubi*REST *point-to-point* and *group* messages across those networks. Still, we assume that nodes will not access resources that would require the consecutive traversal of more than five wireless networks (see [17, 31] for a detailed analysis on wireless communication) in order to access them. Hence, still referring to Fig. 20.4, Alice has to route its request through an appropriate bridge node (i.e., bridges *A*, *B* and *C*, noting that each bridge node is displayed in each network it is part of).

Specifically, *bridges* are in charge of routing messages within the multi-network overlay by determining the best route to reach a distant network. To achieve these tasks, bridges nodes run an instance of OLSR [24] among each other, and exchange routing messages using the specific asynchronous multicast transport provided by the *network-agnostic connectivity* layer. Instead of concrete node addresses, however, bridges store as destinations the identifiers of the various present networks (i.e., *network_ID*) and as next hop the bridge that needs to be contacted next to eventually reach the target network. The, whenever a non-bridge node wants to access a resource outside one of the networks it is itself connected to, it simply routes the request to any bridge of choice that will then forward the request accordingly.

Point-to-point transport—The *ubi*REST *point-to-point transport* is a connection-oriented transport for supporting resource access. The *ubi*REST *point-to-point transport*: (*i*) leverages the *network-agnostic connectivity* layer to send and receive messages relying on the MRN@ addressing scheme, and (*ii*) delivers the message to the appropriate resource. When the CURI of the destination resource is specified (e.g., mrna://dd3ef7e3-5f50-3800-982d-62095c6e8075/cart), *ubi*REST selects *point-to-point* as transport layer and extracts the MRN@ from the CURI. Note that, when both the consumer and the provider simultaneously change the complete set

of IP addresses associated to their MRN@ (and no direct link exists) the session will break and the consumer needs to perform a resource discovery to find the same resource again and reestablish the communication.

Group transport—In ubiquitous networking environments, it is crucial to support point-to-multipoint interactions since it is central to advanced middleware services like dynamic discovery [53]. We thus introduce the *ubi*REST *group transport* over *multi-network overlay*, building upon the *asynchronous multicast* facilities provided by *network-agnostic connectivity* layer. Specifically, the *ubi*REST *group transport* is a connectionless transport for one-way communication between multiple peers in multi-network configurations. The *ubi*REST *group transport* interacts with the *network-agnostic connectivity* layer to send multicast messages based on an MRN@ identifying the group (i.e., AURI), and to deliver messages to the registered resources.

20.5.1 Code Mobility

Concerning the *genericity* requirement, *ubi*REST is able to accommodate heterogeneous and unforeseen functionalities into the running application. Unknown Java classes can be dynamically deployed in the overlay by leveraging code mobility [42]. *ubi*REST code mobility mechanism directly relies on the *ubi*REST communication facilities, then different coordination mechanisms impose different code mobility approaches. For point-to-point communication, *ubi*REST implements an end-to-end strategy that enables two *ubi*REST nodes to exchange executable code, whereas for group communication, *ubi*REST adopts a hop-by-hop strategy that, starting from the origin node, spreads the executable code towards multiple destinations.

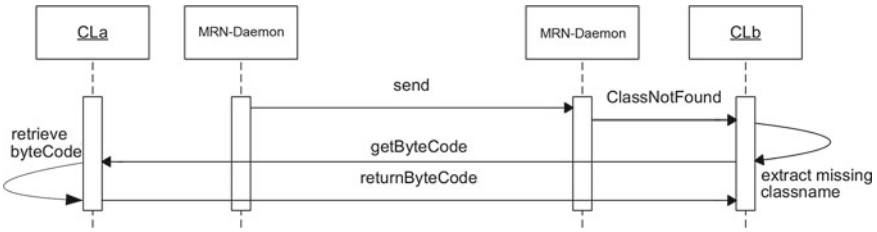

Fig. 20.5 Sequence diagram for point-to-point code mobility

Independently of the specific strategy, *ubi*REST implements an ad-hoc classloader hierarchy to cope with the "missing class" problem. In fact, when sending a message containing a Java object, such an object is serialized into a byte array and delivered towards the destination, which in turn deserializes the object before using it. However, if the object is unknown to the destination (i.e., the destination node does not hold the class bytecode for the received object), the object cannot be deserialized. To this extent, *ubi*REST implements a custom *classloader*, which is in charge of retrieving

the bytecode for the missing classes, and loading them at run time in the local JVM to allow for a correct deserialization. The JVM specification [32] allows for creating a tree-like hierarchy of classloaders to load classes from different sources. When a classloader in the hierarchy is asked to load a class, it asks its parent classloader to load it. If the parent classloader cannot find the class, the child classloader then tries to load it itself. If also the child classloader fails, an exception is thrown. *ubi*REST exploits such a feature by defining a custom *ubi*REST *classloader* as child of the standard Java Bootstrap classloader. When loading classes, the *ubi*REST *classloader* delegates its parent classloader (i.e., Bootstrap). If the Bootstrap classloader fails, then the bytecode is not available within the node and should be retrieved remotely. Thus, the *ubi*REST classloader contacts the origin *ubi*REST node asking for the missing bytecode. The origin side retrieves the bytecode from its classpath and sends it back to the requesting node. At this point, the *ubi*REST *classloader* holds the needed bytecode and can load the class and deserialize the incoming object. If other classes are missing, then such a procedure is iterated until the entire class closure is retrieved.

*ubi*REST implements an end-to-end strategy to achieve point-to-point code mobility among nodes. Referring to Fig. 20.5, let A be an *ubi*LET sending a message to an *ubi*LET B, and let CL_a, CL_b be the classloader of A and B, respectively. Whenever B receives a message containing an object of an unknown type from A, an Exception is thrown, and the control is passed to CL_b, which in turn asks for the missing class to CL_a. CL_a processes the request, encapsulates the needed bytecode into a message, and sends it back to B. Once the bytecode is available at CL_b, it can be loaded into the JVM. The whole procedure is recursively applied until the whole closure of the original class is available on B. The retrieved bytecode is now stored on B and made available for further instantiations.

As for the group communication, this solution is not applicable. In fact, *ubi*REST group communication relies on the underlying *Multi-radio multicast* where message sender and receiver are completely decoupled, and do not have any knowledge about each other. Moreover, applying the end-to-end strategy to group communication would flood the overlay network with requests for bytecode retrieval towards the origin node, which become overloaded. To prevent this problems, *ubi*REST *communication* adopts a hop-by-hop strategy, which spreads the bytecode across the Multi-network overlay towards all the destinations: *ubi*REST applies the end-to-end strategy at each *bridge* along the path between the origin node and each recipient.

20.6 The *ubi*REST Programming Model

As already introduced, P-REST defines systems that comply with the "network-based" paradigm, which rely on the explicit distribution of resources interacting by means of (asynchronous) message passing. Network-based applications can be easily modeled and developed as a set of interacting actors [29], a computational resource reacting to external stimuli (e.g., messages) by either (*i*) sending messages

to other actors, or (*ii*) creating new actors, or (*iii*) designating the behavior for the next stimulus.

The *ubi*REST programming model exploits the Scala [48] programming language, which (*i*) natively provides the Actor system, (*ii*) provides functional features (e.g., high-order functions), and (*iii*) is a JVM language, then allowing for Java libraries reuse (e.g., Multi-radio Networking), and for benefiting from JVM facilities (e.g., security manager for sandboxing). According to both the P-REST model and the *ubi*REST software architecture (Sect. 20.3), the *ubi*REST programming model revolves around the *resource* and *container* abstractions. A *resource* represents the computational unit, whereas *container* handles both the life-cycle and the provision of resources. The *ubi*REST programming model exploits the Scala Actor System [1] by benefiting from its intrinsic qualities: i.e., functional programming paradigm, event-based computation and shared-nothing concurrency, as well as Java interoperability. Hence, the set of *ubi*REST's abstractions is fully implemented in Scala and exploits the actor model.

Resource—The resource abstraction is directly mapped to a Scala actor. A `Resource` actor is defined as a Scala abstract class, which is further extended by any resource to be deployed within *ubi*REST. When extended and instantiated, a `Resource` object is initialized by specifying: (*i*) the CURI address, (*ii*) the set of operations available for the specific resource, and (*iii*) the resource's *Description* specifying the actual semantic concept implemented by a resource, defined as AURI.

According to the Scala Actor Model, `Resource` implements the `act()` method, which defines the resource's passive behavior, i.e., how the resource responds to external stimuli encoded as received messages. `act()` removes messages from its mailbox and, processes them accordingly. To prevent overriding, and then enforcing resources to conform to the REST uniformity principle, `act()` is defined as final, and accepts only messages defined by the P-REST uniform interface. Moreover, PUT, DELETE, and GET methods are declared as `final` and implement the well known semantics defined by HTTP. Further, *ubi*REST defines a new method INSPECT, which allows for retrieving meta-information about the resource (e.g., Description). Rather, the POST method is declared as `abstract` to allow developers to implement their own semantics. Furthermore, according to the Observer pattern defined by P-REST, a resource notifies the observers whenever its internal state changes. That is, when executing either a PUT, a DELETE or a POST operation, the resource actor exploits the underlying *ubi*REST *communication* to send a group message notifying the occurred changes.

According to P-REST, a resource plays a *prosumer* role, i.e., it is able to fulfill both roles of producer and consumer. In order to access external resources and consume their artifacts, a given resource sends request messages to the resources of interest and receives response messages. To this extent, *ubi*REST defines a `workflowEngine` function to be instantiated with the desired behavior by any `Resource` that wants to consume external resources. Indeed, the active behavior is specified by a `workflow` implementing the composition logic defined by P-REST. Specifically, *ubi*REST defines `workflowEngine` as a Scala higher-order function,

which takes a `workflow` as input and executes it:

$$\texttt{workflowEngine} : (\texttt{workflow} : Unit \Rightarrow Unit) \Rightarrow Unit$$

The definition of `workflowEngine` as a higher-order function provides *ubi*REST with the ability of accomplishing hot deployment of new active behaviors at run time. This feature, in turn, supports dynamic situation-aware evolution.

Furthermore, also *ubi*REST provides developers with a high-level domain specific data-flow language for coordinating resources, namely the PaCE (*ubi*REST Coordination languagE) [8]. Specifically, PaCE (*i*) allows developers to specify the active behavior (composition logic) of a composite resource in terms of the set of operations defined by the *ubi*REST programming model, and (*ii*) achieves both adaptation and evolution of compositions in terms of *resource addition*, *resource removal*, *resource substitution*, and *resource rewiring* [38].

Representation—Resources interact with each other by exchanging their representations. *ubi*REST provides a resource representation by serializing the resource instance into a byte array. All the fields specifying the internal state of a resource are serialized into the array. However, since a *ubi*REST resource is implemented as a actor, it is not directly serializable. In fact, actors inherit from threads, which are not serializable as well. To cope with this issue *ubi*REST exploits the *trait* mechanism provided by the Scala language. In Scala, traits are used to define object types by specifying the signature of the supported methods, similarly to interfaces in Java. However, unlike Java interfaces, traits can be partially implemented, i.e., it is possible to define default implementations for some methods. Thus, *ubi*REST defines a special Scala trait, which implements custom serialization/deserialization mechanisms through two methods, namely `writeExternal` and `readExternal`. Both methods are automatically invoked by the JVM when the object is serialized and deserialized, respectively.

The `writeExternal` method makes use of the Java reflection mechanism to (*i*) discover the names and the values of the attributes of a class extending `Resource`, (*ii*) filter out the attributes inherited by `Actor`,[3] and (*iii*) serialize the remaining attributes using standard serialization. On the other hand, when the JVM deserializes a `Resource`, it instantiates an empty object and invokes the `readExternal` method, which in turn reads serialized attributes from the input stream, and makes use of the reflection mechanism to properly assign values to resource's attributes. This mechanism allows for the automatic generation of resource representations. `Representation` stores the byte array generated by the `writeExternal` method.

It is important to note that, designers are entrusted with preserving information hiding in `Representation`, i.e., avoiding the serialization of internal state information. For example, given a resource abstracting an algorithm, it should not return

[3] Scala Actors are not serializable and do not contain information regarding the resource internal state.

the representation of the algorithm. Rather, it should return the representation of the resource abstracting the results computed by the algorithm.

Description—As introduced in Sect. 20.3, resource descriptions are semantics-aware and play a key role in *ubi*REST. In fact, since all resources implement the same interface, descriptions result to be the only discriminant. To this extent, *ubi*REST provides developers with a Resource Description Language (RDL) defined by means of a XML Schema (see Figs. 20.6, 20.7). In particular, a *Description* is composed of two entities: (*i*) the *functional* description, which describes the functionalities provided by the given resource, and (*ii*) the sURI and cURI attributes, which define the semantic concept implemented by the given resource and its concrete identifier (i.e., the actual resource URI), respectively. The *functional* description aims at specifying "what" capabilities are actually provided through the uniform interface. To this end, it describes, for each implemented operation, the semantic concept it refers to (i.e., *semanticRef*), the data expected as output, and the input parameters if required (e.g., POST and PUT operations require for an input parameter, whereas the others do not).

Container—*ubi*REST handles resources' life-cycle and provisioning through the `Container` actor, which is implemented as an *ubi*LET. Indeed, the `Container` stores references to the hosted resources into a *resource repository* built as a mapping

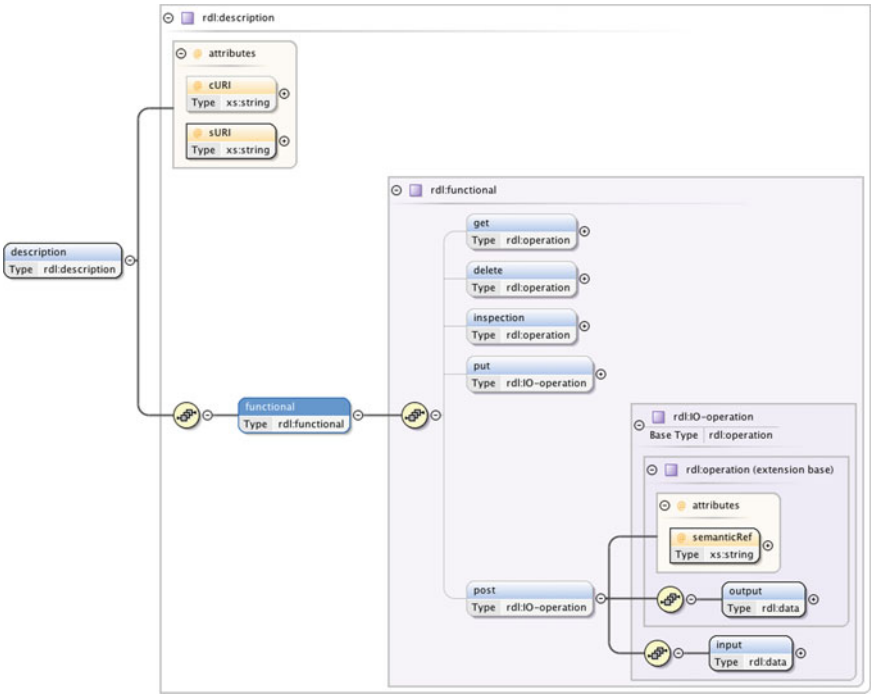

Fig. 20.6 Resource Description Language

from resources cURI to the respective `Resource` instance. Since a container is an active party in *ubi*REST, it also holds a cURI address, which is used to access a container's services. Hence, the container is in charge of handling three classes of incoming messages: (*i*) messages addressed to a specific resource hosted by the container, (*ii*) messages directly addressed to the container itself, and (*iii*) broadcast messages. In the first case, the container simply forwards the message payload to the right `Resource` actor. Messages addressed to the container are directly handled and processed. Finally, broadcast messages are received by the *ubi*REST node either as result of an active subscription within the Group-based communication submitted by a local resource (see Sect. 20.5), or as a *lookup* request. Notifications are delivered to subscribed resources, whereas lookup messages are processed by the container itself.

Concerning the outgoing messages issued by hosted resources, the container is in charge of forwarding such messages towards their destination by means of the proper communication protocol. `Lookup` messages are broadcast throughout the overlay; `Notify` messages are multicasted by means of the Group-based communication; `Observe` messages are encoded as subscriptions to a specific Group; the other messages are simply forwarded towards the final destination by exploiting the point-to-point communication facility.

As already pointed out, the container is in charge of managing resources' life-cycles and provision. In particular, a container creates and moves resources, provides support for resource lookup, as well as grants for resource access. While resource access is managed by the resource itself through its interface, creation, relocation and lookup operations are managed by containers.

```
<?xml version="1.0" encoding="UTF-8"?>
 <rdl:description aURI="presenter"
                  cURI="dd3ef7e3-5f50-3800-982d-62095c6e8075/Projector">
  <functional>
    <put semanticRef="display">
  <output xsi:type="rdl:simpleData"
      name="ack" type="bool" semanticRef="response"/>
      <input xsi:type="rdl:simpleData"
             name="PNG" type="bin" semanticRef="slide"/>
    </put>
  </functional>
 </rdl:description>
```

Fig. 20.7 A resource description example using RDL

To create a resource, the container must be provided with information concerning (*i*) the `Representation` of the resource to be created, and (*ii*) an optional cURI to be assigned to the resource. When creating a new resource, the container checks whether the cURI has been specified or not (if not a cURI is automatically generated), and extracts the `Resource` instance from the provided `Representation`. The newly created `Resource` is then deployed within the container and a new entry

is added to the *resource repository*. Finally, the new `Resource` is initialized and started.

When moving a resource r from a container C_A to a container C_B, *ubi*REST needs to coordinate the two containers in order to guarantee both the correct deployment of r within the container C_B, and the delivery of messages to the r's new location (to avoid packet loss). Specifically, C_A buffers all the incoming messages addressed to r. *ubi*REST performs the move operation in three steps: (*i*) C_A waits until r consumes all the messages already in its mailbox and reaches a quiescent state [28]; (*ii*) C_A generates a representation ρ for r; (*iii*) C_A invokes a `Create` operation on C_B by passing both ρ and the CURI of r; r is then created in C_B and kept quiescent. Once these steps are successfully accomplished, *ubi*REST updates the naming system with the new location of r, and activates it. Finally, C_A removes r from its *resource repository*, and forwards all the buffered messages towards C_B, where r is now able to consume old messages, as well as the new ones that are directly delivered to the new location.

Finally, a *lookup* operation is used to query the *ubi*REST overlay for resources of interest on the basis of their descriptions. In particular, *lookup* takes advantage of Scala functional features by allowing developers to specify their own lookup strategy as a `filter` function, which is used to filter out results to be returned to requesters:

$$\texttt{lookup}\colon (\texttt{filter}\colon RDL \Rightarrow Boolean, \texttt{d}\colon RDL) \Rightarrow \texttt{cURI}[\,]$$

Lookup is a high-order function that evaluates the function `filter` with all the RDL descriptions stored by the resource repository, and returns the list of CURI identifying those resources evaluated $true$. *ubi*REST provides a default implementation for `filter`, which exploits well known signature matching algorithm [39, 46]. Indeed, the `lookup` function matches a requested AURI, against the set of AURI implemented by the resources stored within the *resource repository*. Then, `filter` checks if provided and required AURIs, specified by means of ontology concepts, satisfy one of the following subsumption relationships: (*i*) the concepts are equivalent (*exact matching*), (*ii*) the provided concept subsumes the required one (*plugin matching*), (*iii*) the required concept subsumes the provided one (*subsume matching*), and (*iv*) there does not exist any subsumption relation between the two concepts (*fail*). If the result is not *fail*, then CURI of the matching resource(s) is returned to the requesting node.

20.7 *ubi*REST in Action: An Example

This section shows how *ubi*REST abstractions can be easily and intuitively exploited to develop a ubiquitous application, namely *Ubiquitous Slide Show* (USS).

To this extent, we introduce a simple scenario describing a USS use case: *Carl, a university professor, is going to give a talk at the conference room, and carries his laptop storing both the slides and related handouts. The conference room provides*

speakers with a smart-screen available on the local wireless network, whereas the audience is supposed to be equipped with devices (e.g., laptop, smartphone, tablet), which can be used for displaying either the slide currently projected on the screen or the related handouts. The audience and the speaker always refer to the same slide, and to the same page of the handouts. All the devices are supposed to have a ubi*REST instance running on them.*

USS conforms to the P-REST conceptual-model and specifies the following resources: `CurrentSlide` and `CurrentPage` represent the slide currently projected, and the corresponding handout page, respectively; `Remote` models the remote controller used by Carl to browse the slide show represented as an ordered list of slides; `PresReader` and `HoReader` visualize the slide show and the handout on the audience's devices, respectively; `Projector` handles the smart-screen of the conference room. The `Projector` resource is deployed within the smart-screen *ubi*REST container. `CurrentSlide`, `CurrentPage`, `Remote`, `PresReader` and `HoReader` are initially deployed on Carl's container, and made available to the devices in the audience which join the slide show.

Figure 20.8 shows a sequence diagram defining how such resources interact with each other to implement USS. `Remote` broadcasts a `Lookup` messages searching

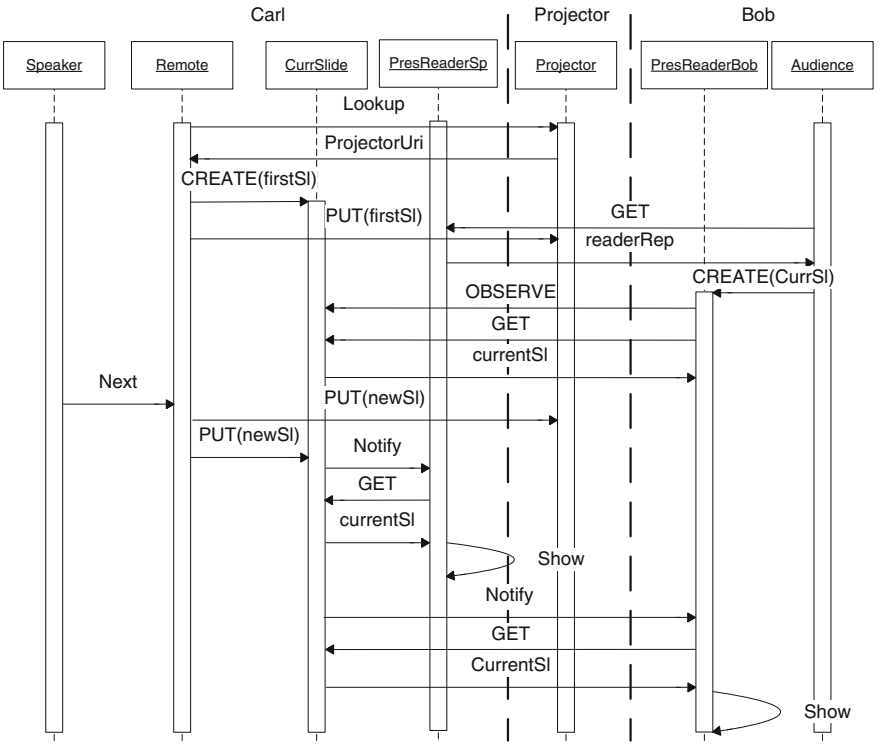

Fig. 20.8 Behavioral specification of the USS application

for a resource that implements the *presenter* concept, as defined by the `Projector` RDL description (see Fig. 20.7). As a result, it obtains the projector cURI that, in our example, matches the lookup request. Once the projector cURI's has been retrieved, Carl starts the slide show: `Remote` sends a `PUT` message, containing the representation of the first slide, to the projector, and then creates the `CurrentSlide` resource also initialized with the representation of the first slide.

On the other side, when a participant (say, Bob) enters the conference room, he uses the *ubi*REST *resource finder* built-in tool, which lists all the resources available within the overlay, to explore the environment and find the `PresReader` resource. Hence, selecting `PresReader` from the list, the *ubi*REST node issues a `GET` operation to retrieve a representation of `PresReader`, which, in turn, is used to create the `PresReaderBob` resource. Once this resource is created, it performs two actions: (1) it gets the state of `CurrentSlide` to initialize itself, and (2) it declares interest on observing the `CurrentSlide` resource (i.e., `OBSERVE` message).

When Carl needs to show the next slide of his presentation, he generates a `next` event that is handled by the `Remote`'s `workflowEngine` by performing a `PUT` operation on both `Projector` and `CurrSlide`. Modifying the `Projector` resource causes the projected slide to change, whereas modifying `CurrSlide` generates notifications towards all the resources that are observing `CurrSlide`. Then, `PresReaderBob` receives such a notification and retrieves the new `CurrentSlide` representation, which is then visualized on his devices.

This case study demonstrates that *Resource* is a natural abstraction for designing and implementing ubiquitous applications. Besides, they are simple and intuitive, as proven by the fact that USS consist of only 13 classes, among which the largest one required about 150 lines of code. However, evaluating the *ubi*REST against competitors is a hard task because of the lack of common test-beds. Hence, we are currently involving students in a Beta-Test phase, where a set of case studies will be developed by means of different middlewares.

20.8 Conclusion

Service-oriented computing appears as a promising paradigm for ubiquitous computing systems that shall seamlessly integrate the functionalities offered by networked resources, both mobile and stationary, both resource-rich and resource-constrained. In particular, the loose coupling of services makes the paradigm much appropriate for wireless, mobile environments that are highly dynamic. However, enabling service-oriented computing in ubiquitous networking environments raises two key challenges: (*i*) achieving the ubiquitous networking environment on top of multi-radio connectivity, and (*ii*) providing a flexible architectural style, which allows for designing and developing applications resilient to the extreme instability inherent to ubiquitous networking environments.

Exploiting multi-radio connectivity has led to the definition of various algorithms for optimizing the scheduling of communications over multiple radio interfaces,

e.g., [10, 27, 41]. Building on this effort, this paper has introduced a *network-agnostic connectivity* layer, which leverages multi-radio networking by means of a special addressing scheme for networked services, namely MRN@, a QoS-aware network selection mechanism and both *unicast* and *multicast* communication facilities. In particular, this layer is in charge of managing the low-level heterogeneity inherent to multi-radio networking environments, by allowing for the exploitation of different application-level communication protocols. Building upon these functionalities, the *ubi*REST *communication* layer implements two different transports, namely *ubi*REST *point-to-point* and *ubi*REST *group*, which leverage *network-agnostic connectivity* to enable the ubiquitous networking of RESTful services deployed on various devices—e.g., Tablets and smartphones—embedding multiple radio interfaces.

On the other hand, *ubi*REST strives to satisfies the *flexibility*, *genericity* and *dynamism* requirements by adhering to the P-REST principles and exploiting both functional programming and code mobility. Specifically, (*i*) *ubi*REST achieves *flexibility* by exploiting the Actor Model and relying on the *ubi*REST overlay network to provide message-passing interaction, (*ii*) *ubi*REST provides *genericity* through the exploitation of a uniform interface in conjunction with both code mobility and functional programming capabilities (i.e., high-order functions), and (*iii*) *ubi*REST provides *dynamism* by allowing resource composition.

Ongoing and future work is manyfold and proceed towards different lines of research. First of all, we are currently defining an high-level *composition language* allowing developers to specify their own resource compositions in an agile and asynchronous way. Further evolution of *ubi*REST is towards the satisfaction of extra-functional requirements. In particular, we want extend the Resource Description Language, and the *lookup* service as well, to consider extra-functional concerns (e.g., quality of service, security), and contextual information (e.g., physical location) while specifying and search for resources of interest, as well as when composing them. Concurrently, we aim at improving *ubi*REST performances in terms of network load by investigating different types of overlay networks (e.g., peer-to-peer and hybrid).

Acknowledgments This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227077-SMScom (http://www.erc-smscom.org), and by European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 257178 project CHOReOS—Large Scale Choreographies for the Future Internet—http://www.choreos.eu.

References

1. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)
2. Asprino, P., Fresa, A., Gaito, N., Longo, M.: A layered architecture to manage complex multimedia services. In: Proc. of 15th International Conference on Software Engineering and Knowledge Engineering (2003)

3. Bellur, U., Narendra, N.C.: Towards service orientation in pervasive computing systems. In: Proc. of the International Conference on Information Technology: Coding and Computing (2005)
4. Bonetta, D., Pautasso, C.: An architectural style for liquid web services. In: Proceedings of the Ninth Working IEEE/IFIP Conference on Software Architecture. Washington, DC, USA (2011)
5. Caporuscio, M., Charlet, D., Issarny, V., Navarra, A.: Energetic performance of service-oriented multi-radio networks: issues and perspectives. In: Proc. of the 6th international workshop on software and performance (2007)
6. Caporuscio, M., Funaro, M., Ghezzi, C.: Architectural issues of adaptive pervasive systems. In: G. Engels, C. Lewerentz, W. Schfer, A. Schrr, B. Westfechtel (eds.) Graph Transformations and Model-Driven Engineering, *Lecture Notes in Computer Science*, vol. 5765, pp. 492–511. Springer Berlin/Heidelberg (2010)
7. Caporuscio, M., Funaro, M., Ghezzi, C.: RESTful service architectures for pervasive networking environments. In: E. Wilde, C. Pautasso (eds.) REST: From Research to Practice, pp. 401–422. Springer New York (2011)
8. Caporuscio, M., Funaro, M., Ghezzi, C.: PaCE: A Data-Flow Coordination Language for Asynchronous Network-Based Applications. In: T. Gschwind, F. Paoli, V. Gruhn, M. Book (eds.) Software Composition, *Lecture Notes in Computer Science*, vol. 7306, pp. 51–67. Springer Berlin Heidelberg (2012)
9. Caporuscio, M., Raverdy, P.G., Issarny, V.: ubiSOAP: A service-oriented middleware for ubiquitous networking. IEEE Transactions on Services Computing **5**(1), 86–98 (2012)
10. Charlet, D., Issarny, V., Chibout, R.: Energy-efficient middleware-layer multi-radio networking: an assessment in the area of service discovery. Comput. Netw. **52**(1) (2008)
11. Doval, D., O'Mahony, D.: Overlay networks: A scalable alternative for P2P. IEEE Internet Computing **7**(4), 79–82 (2003)
12. Edwards, W.K., Newman, M.W., Sedivy, J.Z., Smith, T.F.: Experiences with recombinant computing: Exploring ad hoc interoperability in evolving digital networks. ACM Trans. Comput.-Hum. Interact. **16**, 3:1–3:44 (2009)
13. Ellebaek, K.K.: A survey of context-aware middleware. In: Proc. of the 25th conference on IASTED International Multi-Conference (2007)
14. Fielding, R.T.: REST: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)
15. Grace, P., Blair, G.S., Samuel, S.: A reflective framework for discovery and interaction in heterogeneous mobile environments. SIGMOBILE Mob. Comput. Commun. Rev. **9**, 2–14 (2005)
16. Guinard, D., Trifa, V., Wilde, E.: A resource oriented architecture for the Web of Things. In: Proceedings of Internet of Things (IOT). Japan (2010)
17. Gupta, P., Kumar, P.: The capacity of wireless networks. IEEE Transactions on information theory **46**(2) (2000)
18. Gupta, V., Goldman, R., Udupi, P.: A network architecture for the web of things. In: Proceedings of the Second International Workshop on Web of Things. New York, NY, USA (2011)
19. Hirsch, F., kemp, J., Ilkka, J.: Mobile Web Services: Architecture and Implementation. John Wiley & Sons (2006)
20. Hoareau, D., Mahéo, Y.: Middleware support for the deployment of ubiquitous software components. Personal Ubiquitous Comput. **12**, 167–178 (2008)
21. Huang, H., Cai, J.: Improving TCP performance during soft vertical handoff. In: Proc. of the 19th international conference on advanced information networking and applications (2005)
22. International Telecommunication Union (ITU): Global standard for International Mobile Telecommunications - IMT-Advanced. http://www.itu.int/
23. Issarny, V., Sacchetti, D., Tartanoglu, F., Sailhan, F., Chibout, R., Levy, N., Talamona, A.: Developing ambient intelligence systems: A solution based on web services. Automated Software Engg. **12**(1), 101–137 (2005)

24. Jacquet, P., Muhlethaler, P., Clausen, T., Laouiti, A., Qayyum, A., Viennot, L.: Optimized link state routing protocol for ad hoc networks. In: Proc. of the IEEE international multi topic conference: technology for the 21st century (2001)
25. Khare, R., Taylor, R.N.: Extending the representational state transfer (rest) architectural style for decentralized systems. In: Proceedings of the 26th International Conference on Software Engineering, pp. 428–437. Edinburg, UK (2004)
26. Kindberg, T., Barton, J.: A web-based nomadic computing system. Computer Networks **35**(4), 443–456 (2001)
27. Klasing, R., Kosowski, A., Navarra, A.: Cost minimisation in wireless networks with bounded and unbounded number of interfaces. Networks **53**(3), 266–275 (2009)
28. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Tran. Soft. Eng. **16**(11), 1293–1306 (1990)
29. Kuuskeri, J., Turto, T.: On actors and the rest. In: Web Engineering, *Lecture Notes in Computer Science*, vol. 6189, pp. 144–157. Springer Berlin/Heidelberg (2010)
30. Lelli, F., Pautasso, C.: Controlling and monitoring devices with REST. In: Proceedings of the 4th International Workshop on Distributed Cooperative Laboratories: "Instrumenting" the Grid (INGRID 2009). Italy (2009)
31. Li, J., Blake, C., De Couto, D.S.J., Lee, H.I., Morris, R.: Capacity of ad hoc wireless networks. In: Proc. of the 7th ACM international conference on mobile computing and networking (2001)
32. Lindholm, T., Yellin, F.: Java virtual machine specification. Addison-Wesley Longman Publishing Co., Inc. (1999)
33. Mancinelli, F.: Leveraging the web platform for ambient computing: An experience. IJACI **2**(4), 33–43 (2010)
34. Mascolo, C., Capra, L., Emmerich, W.: Middleware for mobile computing (a survey). In: Networking 2002 Tutorial Papers (2002)
35. McKinley, P., Sadjadi, S., Kasten, E., Cheng, B.: Composing adaptive software. Computer **37**(7), 56–64 (2004)
36. Network Working Group: RFC675 - Specification of Internet Transmission Control Program. http://www.ietf.org/rfc/rfc0675.txt (1974)
37. Olsen Jr., D.R., Jefferies, S., Nielsen, T., Moyes, W., Fredrickson, P.: Cross-modal interaction using XWeb. In: 13th annual ACM symposium on User interface software and technology, UIST '00, pp. 191–200 (2000)
38. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proceedings of the 20th international conference on Software engineering, pp. 177–186. IEEE Computer Society, Washington, DC, USA (1998)
39. Paolucci, M., Kawamura, T., Payne, T., Sycara, K.: Semantic matching of web services capabilities. In: First International Semantic Web Conference (2002)
40. Papadimitriou, D.: Future Internet - the Cross-ETP Vision Document. http://www.future-internet.eu/news/view/article/the-cross-etp-vision-document.html (2009). Ver. 1.0
41. Qureshi, A., Guttag, J.: Horde: separating network striping policy from mechanism. In: Proc. of the 3rd international conference on mobile systems, applications, and services (2005)
42. Roman, G.C., Picco, G.P., Murphy, A.L.: Software engineering for mobility: a roadmap. In: FOSE '00, pp. 241–258. ACM, New York, NY, USA (2000)
43. Sakr, S.: Xml compression techniques: A survey and comparison. J. Comput. Syst. Sci. **75**(5), 303–322 (2009)
44. Sorber, J., Banerjee, N., Corner, M.D., Rollins, S.: Turducken: hierarchical power management for mobile devices. In: Proc. of the 3rd international conference on mobile systems, applications, and services (2005)
45. Su, J., Scott, J., Hui, P., Crowcroft, J., de Lara, E., Diot, C., Goel, A., Lim, M., Upton, E.: Haggle: seamless networking for mobile applications. In: Proc. of the 9th international conference on ubiquitous computing (2007)
46. Sycara, K., Paolucci, M., Ankolekar, A., Srinivasan, N.: Automated discovery, interaction and composition of semantic web services. Journal of Web Semantics **1**(1), 27–46 (2003)

47. Tanenbaum, A.S., Van Renesse, R.: Distributed operating systems. ACM Comput. Surv. **17**, 419–470 (1985)
48. The Scala language. http://www.scala-lang.org/
49. van Engelen, R.A., Gallivan, K.: The gSOAP toolkit for web services and peer-to-peer computing networks. In: Proc. of the 2nd International Symposium on Cluster Computing and the Grid (2002)
50. Wang, H.J., Katz, R.H., Giese, J.: Policy-enabled handoffs across heterogeneous wireless networks. In: Proc. of the 2nd IEEE workshop on mobile computer systems and applications (1999)
51. Wolff, A., Michaelis, S., Schmutzler, J., Wietfeld, C.: Network-centric middleware for service oriented architectures across heterogeneous embedded systems. In: Proc. of the 11th International EDOC Conference Workshop (2007)
52. XML Protocol Working Group: SOAP message transmission optimization mechanism. http://www.w3.org/TR/soap12-mtom/
53. Zhu, F., Mutka, M.W., Ni, L.M.: Service discovery in pervasive computing environments. IEEE pervasive computing **4**(4) (2005)