Chapter 18
RESTful Service Architectures for Pervasive Networking Environments

Mauro Caporuscio, Marco Funaro, and Carlo Ghezzi

Abstract Computing facilities are an essential part of the fabric of our society, and an ever-increasing number of computing devices is deployed within the environment in which we live. The vision of pervasive computing is becoming real. To exploit the opportunities offered by pervasiveness, we need to revisit the classic software development methods to meet new requirements: (1) pervasive applications should be able to dynamically configure themselves, also benefiting from third-party functionalities discovered at run time and (2) pervasive applications should be aware of, and resilient to, environmental changes. In this chapter we focus on the software architecture, with the goal of facilitating both the development and the run-time adaptation of pervasive applications. More specifically we investigate the adoption of the REST architectural style to deal with pervasive environment issues. Indeed, we believe that, although REST has been introduced by observing and analyzing the structure of the Internet, its field of applicability is not restricted to it. The chapter also illustrates a proof-of-concept example, and then discusses the advantages of choosing REST over other styles in pervasive environments.

Introduction

The Internet evolution is moving fast from "sharing" to "co-creating". The clear distinction between content producer and consumer roles, which characterized the Internet so far, is blurring towards a generic "prosumer" role that acts indistinguishably as both producer and consumer (Papadimitriou 2009). Hence, a "prosumer" is any active participant in a business, information, or social computing process. When prosumers are integrated with the computational environment and available anytime and anywhere, they are generically denoted as "things". Likewise, the term

M. Caporuscio (✉)
Politecnico di Milano, Piazza Leonardo, Da Vinci 32, 20133 Milano, Italy
e-mail: caporuscio@elet.polimi.it

E. Wilde and C. Pautasso (eds.), *REST: From Research to Practice*,
DOI 10.1007/978-1-4419-8303-9_18,

Internet of Things is also often used. Due to the multitude of possible different "things" available within the environment, applications require knowledge and cognitive intelligence in order to discover, recognize, and process such a huge amount of heterogeneous information. "Things" provide services to other "things". Furthermore, it is possible to retrieve them, interact with them and change their state, and compose them to build composite "things", thus creating an *Internet of Services*.

The above concepts of prosumer, internet of things and internet of services underlie the Future Internet vision (Papadimitriou 2009), which in turn rests on the future communication and computational infrastructure. We will be virtually connected through heterogeneous means, with invisible computing devices pervading the environments (Saha and Mukherjee 2003). Such environments, referred to as *pervasive networking environments*, will be composed as spontaneous aggregation of heterogeneous and independent devices, which do not rely on predetermined networking infrastructures.

In pervasive networking environments applications emerge from compositions of the resources (the "things") available in the environment at a given time. Indeed, pervasive applications 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. As an example, because of mobility, new things may become available dynamically, while others may suddenly disappear.

In order to face the extreme flexibility that characterizes pervasive environments, applications must support adaptive and evolutionary situation-aware behaviors. *Adaptation* refers to the ability to self-react to environmental changes to keep satisfying the requirements, whereas *evolution* refers to the ability of satisfying new or different requirements.

In order to be self-adaptable and easily evolvable, applications should exploit design models that support *loose coupling*, *flexibility*, *genericity*, and *dynamicity*. Different architectural styles and coordination mechanisms have been proposed to deal with, and reason about, distributed applications. For example, the *procedural* style, where stateless components interact via remote procedure call, or the *object oriented* style, where stateful components interact via messages. Or the *service-oriented* style, where functional or object-oriented components are not directly bound, but rather the binding may be achieved dynamically after a discovery procedure.

This chapter exploits the REpresentational State Transfer (REST) style to achieve adaptation and evolution in the context of pervasive networking environment. REST has demonstrated to be a well-suited design abstraction to deal with flexibility, genericity and dynamism (Fielding 2000), which are inherent properties of the Internet. In fact, since networked software applications are conveniently abstracted as autonomous loosely-coupled resources, they can be dynamically discovered and accessed at run time (e.g., by means of search engines), as well as combined on-the-fly to accomplish complex tasks (e.g., mashups).

The standards available for the WEB support quite effective technologies targeting the Internet domain. However, supporting WEB resource access in pervasive

networking environments is still challenging. In fact, actual WEB standards essentially rely on stability assumptions associated with distributed systems and do not take into account the issues introduced by mobility (Roman et al. 2000) and, more generally, situational change, which instead permeates pervasive applications. In this case, the network structure is no longer stable and resources may come and go (physical mobility), as well as resources may move among devices (logical mobility). To comply with these constraints this chapter promotes the adoption of the REST architectural style as a design model.

The remainder of the chapter is organized as follows. "Background" describes background information on design models and software adaptation. "Why REST?" discusses why we should adopt a REST approach to address software adaptation and evolution in pervasive environments. "REST for Pervasive Systems" introduces P-REST, a meta-model for pervasive REST-oriented applications. "P-RESTful Self-adaptive Systems" illustrates how to design an adaptive and evolvable system according to P-REST. "P-REST at Work: The EXPO2015 Scenario" validates the proposed approach through a case study. "Conclusion and Final Remarks" concludes the paper and delineates future work.

Background

Research has been focusing for more than a decade on adaptive and distributed systems. Such systems have been investigated from many points of view and at different levels of abstraction. Particular attention has been devoted to architectural aspects, i.e., how to architect distributed systems to make them amenable to changes (Cheng et al. 2009). In this area, research has been mainly following two trends. On the one hand, it focused on the properties to be met by software architectures to enable applications to adapt to run-time changes. On the other, research focused on high-level models of architecture that can be kept alive at run time to support adaptation.

Since our work builds on top of both the research lines, in this section we will give a brief review of the main architectural styles emerged during the past decade and then we will go through the work on run-time models.

Architectural Styles

Many different architectural styles[1] have been proposed to deal with, and reason about, distributed systems. They can be classified according to several dimensions: (1) the type of *coupling* imposed by the model on entities; (2) the degree of *flexibility*, that is the ability of the specific model to deal with the run-time growth

[1] We also use the terms *architectural model* and *design model* interchangeably throughout the paper.

Table 18.1 Distributed design models dimensions

	Coupling	Flexibility	Genericity	Dynamism
RPC	Tight	✗	✗	✗
OO	Tight	✗	✗	✗[2]
SOA	Loose	✓	✗	✓[2]
REST	Loose	✓	✓	✗[2]

of the application in terms of involved components; (3) the degree of *genericity*, that is the ability to accommodate heterogeneous and unforeseen functionalities into the running application; (4) the kind of *dynamism*, that is the possibility to rearrange the application in terms of binding, as well as adding new functionality discovered at run time.

Table 18.1 classifies the main architectural models in terms of their characteristics with respect to the pervasive networking issues.

The oldest design model for distributed architectures is based on functional distributed components that are accessed in a synchronous way through *Remote Procedure Call* (RPC). This supports a client–server style, where: (1) client and server are tightly coupled, (2) adding/removing functions strongly affects the behavior of the overall network-based system, (3) function signatures are strict, and (4) binding between entities is generally statically defined and cannot vary (new functions cannot be discovered at run time).

Object Oriented architectures support distributed objects, and provide higher-level abstractions by grouping functions (methods) that manipulate the same object and encapsulating (and hiding) state information. The type of interaction among objects, however, is synchronous, as for the previous case. In summary: (1) interacting objects are still tightly-coupled in a client–server fashion, (2) adding/removing entities as the system is running is hard to support, (3) interfaces are specified via strict method signatures, and (4) once a reference to a remote object is set, normally it does not change at run time, and there is no predefined way of making objects discoverable (i.e., supporting this feature requires for additional ad-hoc effort).

Service Oriented Architecture (SOA) is a further step from the previous two cases because networked entities are abstracted as autonomous software services that can be accessed without knowledge of their underlying technologies. In addition, SOA opens the way to dynamic binding through dynamic discovery. In summary: (1) services are independent and loosely-coupled entities, (2) services can be easily added/removed and accessed, irrespective of their base technology, (3) service access is regulated by means of well-defined interfaces, and (4) binding between services can in principle be dynamically established at run time (although in existing SOA application this is not common practice), and new entities may be discovered and bound dynamically.

[2]This feature is conceptually feasible, although several existing instantiations of the architectural style do not support it.

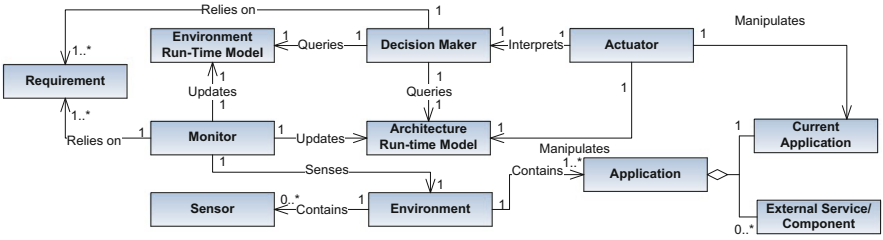

Fig. 18.1 Conceptual model for self-adaptive systems

REpresentational State Transfer (REST) differs from all the previous models in the way distributed entities are accessed and in the way their semantics is captured. REST entities are abstracted as autonomous and univocally addressable *resources*, which have a uniform interface consisting of few well-defined operations. In all previous cases, entities have different and rich interfaces, through which designers capture the semantic differences of the various entities. In REST, all entities have the same interface. Semantic information is attached separately to the identification mechanism that allows entities to be accessed. In addition, interaction with REST entities is stateless. In summary: (1) resources are independent and loosely-coupled entities, (2) resources can be easily added, removed and accessed, irrespective of their underlying technology, (3) resource access is regulated by means of a uniform interface, and (4) binding between resources is dynamically established at run time even though, in general, there is no standard way to discover and access them. However, this might be achieved by means of additional support.

Model-centric Software Adaptation

As we mentioned earlier, once an architecture is built, following some specific style, it is useful to keep a model of the architecture alive at run time to support dynamic adaptation. This section briefly elaborates on this important concept. The pivotal role played by architectural run-time models was initially recognized by Oreizy et al. (1998). In our previous work we explored this idea in two different directions in Epifani et al. (2009) and Caporuscio et al. (2010). The former paper discusses how the model can be updated as a consequence of changes observed in the environment and how this change may drive self-adaptations. The focus is on changes of the non-functional requirements of the application (performance and reliability). The latter introduces and motivates a conceptual-model (shown in Fig. 18.1) that identifies the building blocks of self-adaptive systems dealing with both *adaptation* and *evolution*. In this approach, the model kept alive at run time is composed of two sub-models, which describe the application and the environment, respectively – i.e., *Architecture Run-time Model* and the *Environment Run-time Model*.

In case of evolution, Requirements change and the Decision Maker (which in this case most likely requires human intervention) leverages the Architecture Run-time

Model to reason about the current state of the application and to devise a new abstract architecture that meets the new Requirements. The Actuator is in charge of translating the solution into an architecture and keeping the Architecture Run-time Model synchronized with the new Architecture. Adopting a suitable architectural style for describing the Architecture Run-time Model eases the decision maker's reasoning process (i.e., rules and constraints are well-known and predefined) and provides the actuator with a clear set of actions (i.e., actions are narrowed by the style's constraints) that can be performed. This also guarantees that newly devised solutions are compliant with the change by construction.

As for adaptation, an application must be aware of the environment it is working in. This is modeled by the Environment entity, which contains the applications running in an environment. An Application is described as an aggregation of the description of its architecture and of the external services or components it interacts with. The conceptual model includes the Sensors that abstract the physical context. The Decision Maker accesses the Environment Run-time Model and the Architecture Run-time Model to decide about the possible adaptive changes that need to be made to the architecture in response to changes in the environment. As opposed to evolution, adaptation is mostly achieved in a self-managed manner by the Actuator.

Why REST?

The exploitation of the REST architectural style in the context of pervasive systems is still challenging, and literature so far has been focusing mainly on interaction protocols. For example, Romero et al. (2010) exploit REST to enable interoperability among mobile devices within a pervasive environment.

On the other hand, we are interested on investigating the issues related with the design and development of RESTful applications able to evolve and adapt at run time. To this extent, this section discusses how the design of self-adaptive applications benefits from the REST principles.

The original REST architectural style (Fielding 2000) defines two main architectural entities (see Fig. 18.2): the *User Agent* that initiates a request and becomes the ultimate recipient of the response, and the *Origin Server* that holds the data of interest and responds to user agent requests. REST defines also two optional entities, namely *Proxy* and *Gateway*, which provide interface encapsulation, client-side and server-side respectively. The data of interest, held by origin servers, are referred to as *Resources* and denote any information that can be named. That is, any resource is bound to a unique *Uniform Resource Identifier* (URI) that identifies the particular resource involved in an interaction between entities. Referring to Fig. 18.2, when *User Agent* issues a request for the resource identified as R_b to *Origin Server*$_2$, it obtains as a result a *Representation* of the resource (i.e., ρ_b). Specifically, a *Representation* is not the resource itself, but captures the current state of the resource in a format matching a standard data type.

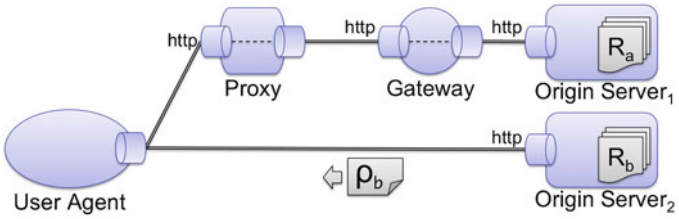

Fig. 18.2 REST architectural style

The concept of a *Resource* plays a pivotal role in the REST architectural style. In fact, it can be seen as a model of any object in the world (i.e., "things") with a clear semantics that cannot change over its lifetime. An application built according to the REST style is typically made of a set of interacting resources. An application built according to the REST architectural style is said to be "RESTful" if it does respect the four basic principles introduced by Fielding (2000) and then elaborated by Richardson and Ruby (2007): *Addressability, Statelessness, Connectedness, Uniformity*. These principles, along with the design model they induce on the application, seem to naturally apply to pervasive environments.

Addressability requires resources to have at least one URI. This RESTful applications to be found and consumed, as well as their constituent resources to be accessed and manipulated. The possibility to retrieve and use constituent resources enables prosumers to opportunistically reuse parts of a RESTful application in ways the original designer has not foreseen (Edwards et al. 2009).

The *statelessness* principle makes REST very appealing to pervasive systems. It establishes that the *state of the interaction* between a user and a RESTful application must always reside on the user side.

Since the state of the interaction is kept by the user, computations can be suspended and resumed (without losing data) at any point between the successful completion of an operation and the beginning of the next one. Indeed, using two different but equivalent resources,[3] will produce the same results. This is important in a pervasive environment since a computation, hindered by the departure of a resource, can, in principle, be resumed whenever an equivalent resource is available. Other advantages – for non-ephemeral resources – are contents "cacheability" and the possibility of load balancing through resource cloning. Hence, statelessness enhances (1) decoupling of interacting resources, (2) flexibility of the model, since it allows for easily rearranging the application at run time and, (3) scalability, by exploiting resource caching and replication. The price to pay derives from the need for an increased network capacity because the whole state of the interaction must be transferred at each request.

[3]We define two resources as equivalent iff they have the same behavior and adopt the same encoding for their representations.

The *connectedness* principle, which refers to the possibility of linking resources to one another, is the backbone of RESTful applications. It was initially introduced by Fielding in his thesis (Fielding 2000) as the "Hypermedia As The Engine Of Application State" (HATEOAS) principle. It allows for establishing dynamic and lightweight workflows such that: (1) clients are not forced to follow the whole workflow – i.e., they can stop at any time – and, (2) workflows can be entered at any time by any client provided with the proper link.

Furthermore, the state can be passed to a resource by means of the URI where it can be retrieved. In this way such a state is retrieved only if (when) needed, according to a lazy evaluation scheme.

Uniformity means that every resource must understand the core operations and must comply with their definition.

Thus, there will be no interface problems among resources. Since operations have always the same name and semantics, the genericity of the model is improved. Clearly the problem is not completely solved because data semantics and encoding must still be negotiated. It could be argued that reliance on data encoding and semantics increases the coupling between resources. However, REST eliminates the need for negotiating also the name and semantics of operations, as it happens for instance in SOA (Vinoski 2008).

Different from SOA, where service semantics is defined by means of the operations it exposes, the semantics of a resource is identified by its name. Indeed, the URI defines which semantic entity the resource models. However, as we will discuss later, this is good practice intended to ease comprehension for human beings, and cannot be applied to generic RESTful applications.

REST for Pervasive Systems

REST technologies rely on (1) the stability of the underlying communication environment and (2) tightly-coupled synchronous interaction protocols only. Pervasive environments, instead, require to (1) cope with an ever-changing communication infrastructure because devices join and leave the environment dynamically (Roman et al. 2000) and (2) to support loosely-coupled asynchronous coordination mechanisms (Huang and Garcia-Molina 2001).

This section investigates how the REST architectural style should be modified to cope with pervasive environments, and introduces the Pervasive-REST (P-REST) design model. Indeed, to make REST pervasive we need to adapt the different levels of abstraction, namely the *architecture*, the *coordination* model, and the *infrastructure*.

As we observed, in pervasive environments and, more generally, in systems envisioned for the Future Internet the role of "prosumer" will be central. Furthermore, such a prosumer role might be played by any "thing" within the environment. Hence, we foresee the necessity of departing from usual REST description of the world, made in terms of *user agents* that consume *resources* from *origin servers*

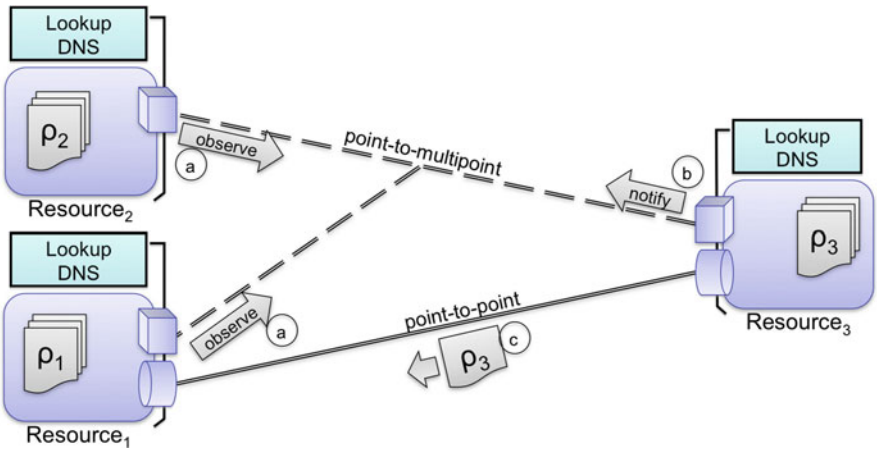

Fig. 18.3 P-REST architectural style

(see "Why REST?"). Rather, the P-REST architectural style promotes the use of *Resource* as first-class object that fulfills all roles. This means that, at the architectural level, we remove the distinction among actors, and thus we model entities in the environment as resources, which can act both as clients and servers.

To support coordination among resources, we extend the traditional request/response REST mechanism through primitives that must be supported by an underlying middleware layer. First, we assume that a *Lookup* service is provided, which enables the discovery of new resources at run time. This is needed because resources may join and leave the system dynamically. Once the resource is found, REST operations may be used to interact with it in a point-to-point fashion. The *Lookup* service can be implemented in several ways [e.g., using semantic information (Mokhtar et al. 2006), leveraging standard protocols (Romero et al. 2010)]. However, we do not rely on any specific implementation since we are focusing on the study of the design model.

The *Lookup* service yields the URI of the retrieved resource. Since resource locations may change as a result of both logical mobility (e.g., the migration of a resource from a device to another) and physical mobility (e.g., resources temporarily or permanently exiting the environment), a service is needed to maintain the maps between resource URIs and their actual location. Such service plays the role of a distributed Domain Name System (DNS) (Network Working Group 2003).

In addition, we adopt a coordination style based on the Observer pattern, as advocated in the Asynchronous-REST (A-REST) proposal described by Khare and Taylor (2004). This allows a resource to express its interest in state changes occurring in another resource by issuing an *Observe* operation. The observed resource can then *Notify* the observers when a change occurs. In this case, coordination is achieved via messages exchanged among resources.

Figure 18.3 summarizes the modification we made to the REST style. Resources directly interact with each other to exchange their representations (denoted by ρ in

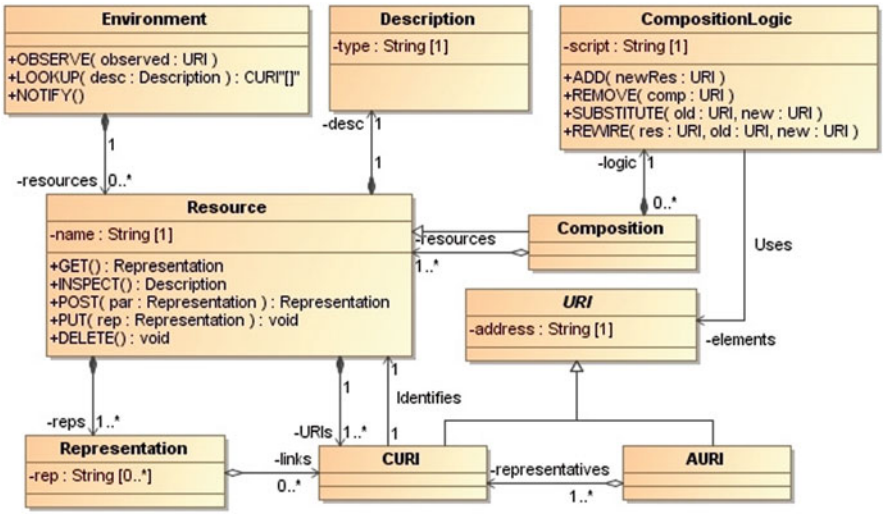

Fig. 18.4 P-REST meta-model

the figure). Referring to Fig. 18.3, both Resource_1 and Resource_2 observe Resource_3 (messages ⓐ). When a change occurs in Resource_3, it notifies (message ⓑ) the observer resources. Once received such a notification, Resource_1 issues a request for the Resource_3 and obtains as a result the representation ρ_3 (message ⓒ). 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-REST Meta-model

Along with the P-REST architectural style introduced above, we also define a P-REST meta-model (depicted in Fig. 18.4) describing the pervasive environment, the resources within the environment, and the relations among resources that define a pervasive application.

The `Environment` entity defines the whole distributed and pervasive environment as a resource container, which provides infrastructural facilities. In particular, it provides three operations that can be invoked by a resource: (1) `OBSERVE`, which declares its interest in the changes occurring in a resource identified by a given URI, (2) `NOTIFY`, which allows the observed resource to notify observers

about the occurred changes, and (3) `LOOKUP`, which implements the distributed lookup service. These operations are the straightforward implementations of both the A-REST principle and of the lookup service, respectively.

Since `Resource` is a unifying first-class object, the P-REST meta-model describes every software artifact within the environment as a Resource. According to the *Uniformity* principle (see "Why REST?"), each resource implements a set of well-defined operations, namely `PUT`, `DELETE`, `POST`, `GET`, and `INSPECT`. The `PUT` operation updates the resource's internal state according to the new representation passed as parameter. The `DELETE` operation results in the deletion of the resource. The `POST` operation may be seen as a remote invocation of a function, which takes the representation enclosed in the request as input. The actual action performed by `POST` is determined by the resource providing it and depends on both the input representation and the resource's internal state. The semantics of the `POST` operation is different for different resources. This differs from the other operations whose semantics is always the same for every resource. Even if the semantics of `POST` is not defined by the architectural style, it is still constrained. Indeed, it can have only one semantics per-resource, and thus, overloading is not allowed. The `GET` implements a read-only operation that returns a representation of the resource. The `INSPECT` operation allows for retrieving meta-information about the resource.

REST operations can be *safe* and/or *idempotent*. An operation is considered *safe* if it does not generate side-effects on the internal state, whereas it is *idempotent* if the side-effects of $N > 0$ identical requests is the same as for a single request. `GET` and `INSPECT` operations are both idempotent and safe, `PUT` and `DELETE` operations are not safe but they are idempotent, whereas for the `POST` operation nothing is guaranteed for its behavior.

The REST architectural style does not provide any means to describe the semantics of resources, which is rather embedded in the URIs of resources or delegated to natural language descriptions. Instead, P-REST assumes that every resource is provided with meta-information concerning both its static and dynamic properties. As an example, for a resource representing a theater, the semantic description includes the total number of seats (a static property) as well as the number of free seats (a dynamic property). Indeed, P-REST promotes resource's semantics as first-class concern by explicitly introducing the `Description` entity. Specifically, Description describes both functional and non-functional properties of a resource, possibly relying on a common ontology that captures the knowledge shared by the entire pervasive environment (Berners-Lee et al. 2001). Description can also define which operations, among the available ones, are allowed or not – e.g., `DELETE` could be forbidden on a specific resource. Moreover, Description entities are also used to achieve dynamism (see Table 18.1). In fact, Descriptions support the implementation of the lookup service by exploiting efficient algorithms for distributed semantic discovery (e.g., Mokhtar et al. 2006), thus enabling de facto run-time resource discovery. As introduced above, Descriptions are retrieved via the `INSPECT` operation. Referring to the HTTP uniform interface that underlies

REST, `INSPECT` operation encapsulates both `HEAD` and `OPTION` operations and goes further by providing also the functional and non-functional specification of the target resource.

At run time, resources have their own internal state, which should be kept private and not directly accessible by other resources. The `Representation` entity overcomes such an issue by exposing a specific rendering of its internal state rather then the state itself. Hence, a Representation is a complete snapshot of the internal state, which is made available for third-party use. Every resource is associated with at least one representation, and multiple representations might be available for a given resource. This is particularly useful when dealing with heterogeneous environments in which several different data encodings are needed. A resource's representation can be retrieved by means of the `GET` operation, which can also implement a negotiation algorithm to understand which is the most suitable representation to return.

As introduced in "Why REST?", *addressability* states that every resource must be identified by means of an URI. Hence, in P-REST, every Resource is bound to at least one `Concrete URI` (CURI), and multiple CURI can refer to the same resource. Resources without any CURI are forbidden, as well as CURIs referencing multiple resources. However, P-REST enhances the concept of URI by introducing the `Abstract URI` (AURI) entity. Specifically, an AURI is a URI that identifies a group of resources. Such groups are formed by imposing constraints on resource descriptions (e.g., all the resources implementing the same functionality). The scheme used to build AURIs is completely compatible with the one used for CURI, thus they can be used interchangeably. Moreover AURIs are typically created at run time by exploiting the LOOKUP operation to find resources that must be grouped. This addition to the standard concept of URI is meant to support a wider range of communication paradigm. Indeed, using CURI allows for establishing point-to-point communication while using AURI allows for multicast communication. The latter can be useful, for instance, to retrieve the values of an entire class of sensors (e.g., humidity sensors scattered in a vineyard).

Resources can be used as building-blocks for composing complex functionalities. A `Composition` is still a resource that can, in turn, be used as a building-block by another composition. REST naturally allows for two types of compositions: *mashup* and *work-flow*. A *mashup* is a resource implemented by exploiting the functionalities provided by third-party resources. In this case, an interested client always interacts with the mashup, which in turn decomposes client's requests into sub-requests and routes them to the remote resources. Responses are then aggregated within the mashup and the result is finally returned to the client. On the other hand, a composition built as *work-flow* directly leverages the HATEOAS principle. In this case, an interested client starts interacting with the main resource and then proceeds by interacting with the resources that are discovered/created step-by-step as result of each single interaction.

Resources involved in a composition are handled by a `Composition Logic`, which is in charge of gathering resources together and, if they were not designed to interact with each other, of satisfying possible incompatibilities (e.g., handling

the encoding mismatches between representations provided and expected by component resources).The composition logic is executed by a composition engine, which implements the classic architectural adaptation policies, namely *component addition*, *removal*, *substitution*, and *rewiring* (we will discuss later how such operations work). In the case of mashups, the composition logic describes how the mashup's operation are implemented; that is, how they are wired to component resources' operations. Indeed, the composition logic is the direct consequence of the exploitation of REST principles: (1) the composition is defined in terms of explicit relations between resources (i.e., connectedness), (2) resources involved in the composition are explicitly identified by means of resource identifiers (i.e., addressability) and, (3) operations on resources are expressed in terms of their interface (i.e., uniformity).

According to REST terminology, an application built following the P-REST design model is said to be P-RESTful.

P-REST Run-time Support

Traditional distributed systems differ from pervasive systems in terms of the assumptions on the underlying networking infrastructure. In particular, in pervasive systems (1) the network stability assumption is no longer guaranteed (i.e., network topology and routing strategies change over time) and (2) devices hosting resources are mobile and may have scarce processing power. Indeed, computing devices can come and go and, as a result, the network topology can change in response to either a node's arrival/departure or performance needs. Due to this new networking scenario, in order to make P-RESTful applications effective, we need to abandon the usual networking infrastructure exploited by REST. To cope with these issues, and to offer programming abstractions suitable for the rapid and efficient development of P-RESTful applications, we introduce the PRIME (P-Rest run-tIME) middleware.

Referring to Fig. 18.5, the PRIME middleware presents a layered software architecture where each layer, spanning from *transport* to *programming abstraction*, deals with specific concerns.

Transport layer: The pervasive environment, and its inherent instability calls for the adoption of a communication system resilient to structural changes (e.g, node arrival and departure). To this extent, PRIME arranges the nodes (i.e., devices) involved in the pervasive environment in a cooperative overlay network built on top of low-level wireless communication technologies (e.g., Bluetooth, Wi-Fi, Wi-Fi Direct, and UMTS). That is, each device makes use of the overlay network and, at the same time, cooperates in it by actively participating to the distributed packet routing. The transport layer is network-agnostic and does note rely on any specific technology. Indeed, it can be used on top of any IP-based network.

Coordination layer: Relying on the transport layer, PRIME provides two basic coordination mechanisms, namely point-to-point and point-to-multipoint.

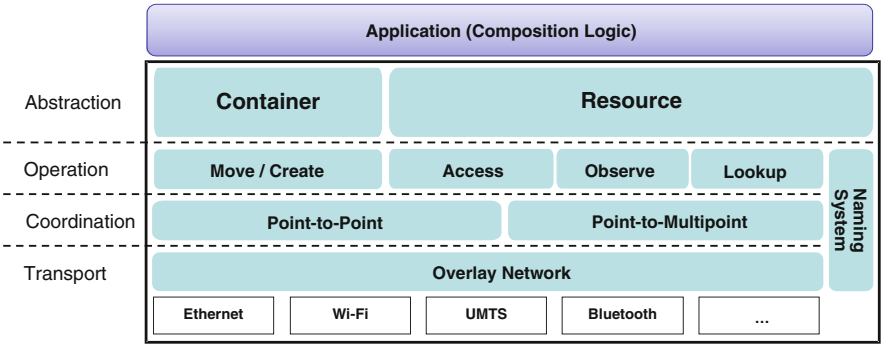

Fig. 18.5 Layered representation of PRIME

Point-to-point communication grants a given node direct access to a remote node, whereas *point-to-multipoint* communication allows a given node to interact with many different nodes at the same time.

Operation layer: The operation layer specifically deals with the concepts defined by the P-REST meta-model. In particular, it is in charge of providing the set of actions that can be performed on resources. *Access* gathers the set of operations needed to access and manipulate a resource – i.e., the set of standard REST operations provided by resources in Fig. 18.4. *Access* operations exploit the coordination layer to achieve point-to-point request-response interactions. `OBSERVE` allows resources to declare interest in a given resource, while `NOTIFY` benefits from point-to-multipoint communication and allows observed resources to advertise all the observers about occurred changes. `LOOKUP` allows for searching for new resources based on the description fed to it. The operation layer provides also `CREATE` and `MOVE` operations. While `CREATE` provides the mechanism for creating a new resource at a given location, `MOVE` provides the mechanism to migrate an existing resource between locations. Resource migration is useful when dealing with load balancing –by relocating the resource to an outperforming host–, device mobility –by relocating the resource to a more stable host[4]–, or energy management –by relocating a resource from a host with low battery to a lost with full battery. All the operations make use of a DNS whose task is keeping URIs consistent despite resources mobility. To this extent, the naming system shall be able to resolve URIs into physical addresses without letting resource mobility hinder such mechanism.

Abstraction layer: On top of the operation layer, PRIME provides the set of facilities and programming abstractions needed to implement P-RESTful applications. In REST, resources are held by Web servers, which handle both their life-cycle and provision. PRIME offers the same abstraction by means of *containers*. That is,

[4]Clearly, this scenario requires for additional mechanisms able to foresee whether the device leaves the environment.

each device within the pervasive environment hosts one container that, just like a Web server, handles both the life-cycle and the provision of its resources. However, unlikely Web servers, containers provide the primitives for both creating resources and migrating resources among containers (i.e., `MOVE`, `CREATE`). Their behavior, however, can be customized in order to achieve specific behaviors. For example, the `CREATE` operation can be made aware about the current load of the local container and actually allocate a resource in another similar container. As a final remark, the physical address provided by the DNS for a specific URI actually is the container's one. Indeed, a container receives all its contained resources' requests and dispatches them to the right resource based on the cURI.

Using the programming abstractions provided by the *Abstraction layer*, a P-RESTful application is then built as a resource that relies on other resources to meet its requirements. Specifically, the interactions between resources is specified by means of a composition language, which allows for composing and managing sets of resources. PRIME offers primitives to modify the composition logic at run tim, thus enabling architectural reconfiguration (i.e., `ADD`, `REMOVE`, `SUBSTITUTE` and `REWIRE`). We will account for these operations in "P-RESTful Self-adaptive Systems."

The PRIME APIs exploit a functional programming paradigm, which naturally achieves resource composition as the sequential application of functions. Functions are bound each other by accepting and producing immutable data structures. Immutable data structures map to resources representation, and functions are the operations exposed by resources. Through a functional language, resource compositions amounts to wiring the output of a function (i.e., operation) to the input (i.e., resource representation) of the next function. Such a functional composition can also be applied to functions that are, in turn, implemented as compositions. Thus the handling of arbitrarily complex compositions is easy and intuitive.

It is worth to note that, the abstractions provided by PRIME recall the ones introduced by CREST (Erenkrantz et al. 2007). The difference between the two approaches lies in the fact that PRIME provides such operations as infrastructural facilities, whereas in CREST resource mobility is promoted to a design principle. For such a reason we keep containers and their operations outside the P-REST meta-model. Indeed, a designer who wants to instantiate the P-REST meta-model should not be concerned with problems related to the deployment and distribution of the application.

P-RESTful Self-adaptive Systems

We argue that self-adaptive applications for pervasive systems may benefit from the adoption of the P-REST design model. To prove this, we show how the conceptual model for self-adaptive systems (Background) can be implemented by means of the P-REST meta-model (REST for Pervasive Systems), and show how the mechanisms provided by PRIME make P-RESTful application effective.

Both the conceptual model (Fig. 18.2) and the P-REST meta-model (Fig. 18.4) contain an *environment* entity. While in the conceptual model the environment is populated by generic software artifacts, in P-REST all the entities contained in the environment are modeled as a resource.

As shown in Fig. 18.2, the conceptual model revolves around the *architecture run-time model* and the *environment run-time model*. In P-REST, the architecture of the application is rendered by means of the set of resources it is composed of and the *composition logic* that orchestrates them. The type of composition used (i.e., workflow or mashup) depends on the specific functional requirements of the application. The environment run-time model is a composition of resources defined as a mashup. The corresponding composition logic is in charge of realizing the mashup by querying component resources and aggregating the results of such queries. Thus, this composition logic plays the role of the *monitor*.

Here we are not concerned with investigating how a *decision maker* might exploit the run-time models to adapt/evolve the system. Rather we want to show which mechanisms, enabled by P-REST, can be leveraged by the *actuator* to modify the running system according to decision maker's instructions. As reported by Oreizy et al. (1998), an actuator operating at the software architecture level should support two types of change: one affecting the components, namely *addition*, *removal* and *substitution*, and one affecting the connectors, namely *rewiring*.

The problem of dynamically deploying and/or removing a component from an assembly has been repeatedly tackled in literature (Kramer and Magee 1990; Vandewoude et al. 2007). Such solutions are often computationally heavy and require expensive coordination mechanisms. Moreover, preserving the whole distributed state is often very difficult since the internal state of a component is not always directly accessible. To make the problem easier, several architectural styles have been introduced. According to P-REST, adding a new resource is trivial and requires two simple steps: (1) deploy the new resource within the environment, and (2) make it visible by disseminating its URI. Once these steps are performed, the resource is immediately able to receive and process incoming requests.

On the other hand, removing a component can in general cause the loss of some part of the distributed state. P-REST, instead, works around this problem because of the stateless nature of the interactions. That is, the removed resource carries away only its internal state, thus the ongoing computations it is involved in are not jeopardized.

Substituting a component with another one cannot be simply accomplished by composing removal and addition operations. Specifically, the issue here concerns how to properly initialize the substituting component with the internal state of the substituted one. Indeed, due to information hiding it is not always possible (and not even advisable) to directly access the internal state of a component. Clearly the component can always expose part of its internal state but there is no guarantee about the completeness of the information provided. On the contrary, P-REST imposes that a resource's representation is a possible rendering of its internal state, which is always retrievable by exploiting the `GET` operation, eventhough the resource is

embedded within a composition. Thus, leveraging the interaction's statelessness and the properties of a resource's representation, a P-REST resource can be substituted almost seamlessly.

As pointed out by the P-REST meta-model (see Fig. 18.2), every composition holds a composition logic describing it. Architectural run-time adaptation can be achieved by modifying the composition logic. Hence, the Composition Logic, which undertakes the run-time change, offers a specific `substitute` operation that is aware of the composing resources and of the status of requests in the composition. In particular, the semantics of the `substitute` operation is provided by means of its pseudo-code, where we leverage the PRIME container abstraction:

```
1  void substitute(cURI oldr, cURI newr) {
2      Container c = DNS.resolve(oldr)
3      c.bufferRequests(oldr);
4      c.waitForFinish(oldr);
5      Representation temp = send(GET, oldr);
6      send(PUT(temp), newr);
7      this.components.substitute(oldr,newr);
8      List<Messages> reqs = c.getPendingReqs(oldr);
9      for (Message m: reqs)
10         send(m, newr);
11 }
```

The first step of the operation retrieves the reference to the container of the old resource (i.e., the resource to be substituted). As we have already highlighted, the physical address of a container coincides with the physical address of all the resources contained in it. Thus, the `resolve` operation provided by the DNS can be exploited to retrieve, given a cURI, the physical address of the container managing the resource identified by cURI. Once retrieved, such a reference is used to access the operations offered by the container for monitoring and regulating the activities of the contained resources (i.e., their life-cycle). Line 3 instructs the container holding the old resource to buffer all the incoming requests directed to `oldr` while the substitution is taking place. As a next step, the `substitute` operation executes a blocking operation to wait for `oldr` to finish processing all the requests that are still ongoing (line 4). Now the internal state of `oldr` can be retrieved safely through its uniform interface (line 5) and used to initialize the new resource (`newr`) using a `PUT` operation (line 6). As stated above, a composition logic knows all its composing resources (through their cURIs), and we are assuming their cURIs to be stored in an instance of a data type called `components`. The instruction on line 7 substitutes the old cURI with the new one, so that the latter will always be used from now on instead of the former. Lines 8–10 retrieve the buffer of blocked requests addressed to `oldr`, and let `newr` consume them. It is important to remark that since the state of the new resource is overwritten by the `substitute` routine, it is good practice to create the new resource from scratch in order to avoid unpredictable side-effects. Indeed, if the newly inserted resource is used concurrently by other

compositions, overwriting its state can be harmful. The complementary argument applies to the substituted resource. It is not deleted because it might be concurrently used by other compositions.

As for rewiring components, due to the stateless nature of the interactions, changing the URIs within the Composition Logic is sufficient for accomplishing the task. Referring to the meta-model in Fig. 18.2, the signature of the rewire operation is:

```
REWIRE (cURI res, cURI old , cURI new)
```

Its semantics is such that all the occurrences of the `old` cURI in the resource `res` will be substituted with the `new` cURI. In a mashup composition `res` is always the mashup itself because it is the only resource actually managed by the composition logic. In a workflow composition, it is important to specify `res` because it is possible that the scope of the rewiring is not extended to the whole composition, but it must be applied only to a specific point in the workflow. Unlike the `substitute` operation the state of the old resource is *not* transferred to the new one.

P-REST at Work: The EXPO2015 Scenario

In this section we describe a small case study, which is inspired by the 2015 Milan Universal Exposition (Expo2015). We envision a city-wide pervasive environment where people, equipped with mobile devices embedding networking facilities (e.g., PDAs, smart-phones), are interconnected with each other to share information and functionalities. Any attendee may be a prosumer, acting as either participant or organizer of unexpected events.

Specifically, suppose that Carl wants to organize and promote his own BarCamp. A BarCamp[5] is an ad-hoc and spontaneous event with discussions and demos where participants, who are the main actors of the event, interact with each other sharing knowledge about a specific topic. To bootstrap his BarCamp, Carl has to (1) choose the topic and advertise the event in order to gather potential participants, (2) find and reserve a free pavilion, and (3) deploy the needed software infrastructure to handle participants' registrations.

Hereafter we address the functional design of the BarCamp application, starting from the identification of the involved resources and their relationships. Figure 18.6 sketches a simplified design of the application where some details are omitted for simplicity. We represent the Environment as an enclosing container for the resources instead of representing it as an explicit box and, as a consequence, drawing a containment relation from every other entities towards it. Also, representations and descriptions do not appear in the diagram since they are not relevant to our purpose.

[5] http://www.barcamp.org/.

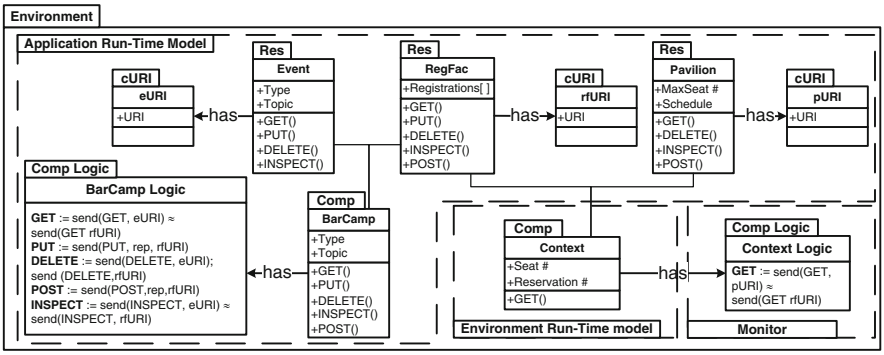

Fig. 18.6 Resource diagram of the BarCamp application

The cornerstone element of the BarCamp application described in Fig. 18.6 is the BarCamp resource, which is designed as a composition of (1) Event, which carries information about the event and (2) RegFac, which gathers attendee registrations to the event.[6] The associated composition logic, namely BarCamp Logic, defines the behavior of the operations exposed by the composite resource. The `GET` operation is computed by retrieving the current representation from both Event and RegFac and joining them (join operations are denoted by the $\oplus$ symbol). The actual result will be returned as a representation containing information about the event along with the registrations gathered so far. The `PUT` operation is directly mapped to the `PUT` operation provided by RegFac. The `DELETE` operation deletes the composite resource by invoking `DELETE` on Event and then on RegFac. The `POST` operation directly maps to the `POST` operation provided by RegFac. In this case, the specific semantics of POST is to create a new registration in the RegFac. The `INSPECT` operation is computed by inspecting both Event and RegFac and joining the results. The Context resource carries environmental data. It exposes only the `GET` operation that is computed by the ContextLogic by joining the number of available seats in the Pavilion and the number of registrations submitted to RegFac.

The application design, shown in Fig. 18.6, is a static description of the application and does not take into account deployment concerns, which in turn should be specified by means of different notation [e.g., UML Deployment Diagrams (Object Management Group 2010)]. Hence, we assume that resources created by Carl, namely BarCamp, BarCamp Logic, Event, Context and Context Logic will be deployed on his PDA. On the other hand, Pavilion and RegFac resources are hosted by the corresponding pavilion's infrastructural server.

According to "Background", in order to address self-adaptation the application should implement the concepts defined by the conceptual model in Fig. 18.1. The

[6]We assume that the software implementing the registration facility is provided by the Exposition Center's infrastructure as a downloadable package to foster the organization of spontaneous events.

application is implemented by BarCamp and its constituent resources (i.e., Event, RegFac and Pavilion). Hence, the architecture run-time model (dashed area in Fig. 18.6) is represented by the BarCamp composition, its constituent resources (i.e., Event, RegFac, Pavilion), and the BarCamp Logic that orchestrates the composition. The whole run-time model represents the current semantics of the application and will be the hinge of the adaptation activities.

The Context resource maps straightforwardly to environment run-time model concept, while the context logic plays the role of monitor since it is in charge of aggregating data and feeding them to the context resource (i.e., environment run-time model). The case study presented here does not use neither sensors nor external services/components. Moreover, since we are not interested in investigating solutions for automated decision-making and actuation, we assume a human-in-the-loop solution for both the Decision Maker and the Actuator roles.

Let us assume that, once advertised, the BarCamp event is very successful and the number of requests exceeds the maximum number of available seats. Carl monitors the ongoing situation by querying the context resource, and decides to adapt the application to the changing context – i.e., relocate the Barcamp to a larger pavilion. The software support for the BarCamp must adapt accordingly. The Exhibition Center's policy forbids the use of a pavilion's machinery to organizers unless they have a valid reservation for it. Since Carl is going to cancel his reservation for the first pavilion, he must substitute the original RegFac resource, which encapsulates the state of the first pavilion (i.e., the registrations gathered so far), with a new one encapsulating the new pavilion's state. Contextually, Carl does not want to restart the registration process from scratch.

To accomplish the substitution, Carl must (1) `substitute` the old RegFac resource, in both BarCamp and Context compositions, with the new one, and (2) `rewire` the old Pavilion resource with the new one in the Context composition. Note that, Pavilion is rewired, instead of being substituted, because we need to preserve the internal state of RegFac (i.e., the registrations). On the other hand, since Pavilion is a read-only resource that gathers information about the facility, it does not have an internal state to be transferred from the old instance to the new one. Thus, Carl creates the new RegFac resource and passes its URI as a parameter to the `substitute` operations exposed by BarCampLogic and ContextLogic, along with the URI of the old RegFac resource. Hence, Carl retrieves the cURI of the new Pavilion resource and uses it as a parameter for the `rewire` operation of the ContextLogic. In this way substitution takes place automatically.

Conclusion and Final Remarks

In this chapter we have addressed the problem of designing applications operating in evolving pervasive environments. Such applications are required to support adaptive and evolutionary situation-aware behaviors, to deal with changes occurring in the run-time environment. Changes are mainly the result of the dynamic appearance/disappearance of functionalities and the interaction with the physical context.

We presented our model-centric conceptual model, which identifies the building blocks of self-adaptive systems dealing with both adaptation and evolution. We advocated the benefits of the REST architectural style in pervasive settings (due to its loose coupling, flexibility and dynamism) and proposed Pervasive-REST (P-REST), a REST-oriented approach for designing pervasive applications. P-REST is a meta-model that can be instantiated to design applications that follow the P-REST principles. Moreover, to support the development of P-RESTful application we introduced PRIME, a distributed middleware and a development framework that both realizes the pervasive networking environment and offers programming abstractions for implementing P-REST.

Furthermore, we have shown how to render the entities of the conceptual model using the P-REST meta-model, and presented a case study for which we designed an application exploiting P-REST. Such a case study can be implemented by exploiting any of the architectural styles discussed in "Background". However, the adoption of P-REST reduces the effort of providing at design time the mechanisms needed for adaptation purposes. Indeed, by exploiting P-REST, the application can be managed at run time without the need for the designer to foresee possible adaptation issues at design time. In particular, the basic mechanisms we took advantage of are:

1. Retrieving the internal state of a component,
2. Initializing the internal state of a component,
3. "Unboxing" a composition to access one of its composing elements,
4. Run-time rebinding of components within the composition logic.

To support the same set of adaptation mechanisms within an application designed according to a traditional SOA paradigm, the designer should foresee several special cases at design time. First, the designer should figure out how to grant direct access to information embedded in a composition. Indeed, a service composition is provided (and consumed) through a set of interfaces and most of the business logic is hidden behind those interfaces. Thus, referring to the case study presented in "P-REST at Work: The EXPO2015 Scenario," an ad-hoc interface should be provided for exposing only the information regarding the registration facility. The same applies for granting access to information regarding the pavilion needed to trigger the adaptation. Finally, one more ad-hoc interface must be designed to allow for initializing the new registration facility with the old state. Moreover, dynamic binding is not directly provided by SOA, but requires for additional ad-hoc support.

We have shown instead that the adoption of P-REST allows adaptation to be carried out in a seamless way, without any special preventive actions by the designer since all the needed functionalities are imposed by the architectural style.

Acknowledgements This research has been Funded by the European Commission, Programme IDEAS-ERC, Project 227077-SMScom (SMSCom 2008).

References

Ben Mokhtar, S., Kaul, A., Georgantas, N., Issarny, V.: Efficient semantic service discovery in pervasive computing environments. Middleware 2006, pp. 240–259 (2006)

Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American (2001)

Caporuscio, M., Funaro, M., Ghezzi, C.: Architectural issues of adaptive pervasive systems. In: G. Engels, C. Lewerentz, W. Schäfer, A. Schïurr, B. Westfechtel (eds.) Graph Transformations and Model Driven Enginering – Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday, *Lecture Notes in Computer Science*, vol. 5765, pp. 500–520. Springer (2010)

Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.): Software Engineering for Self-Adaptive Systems, *Lecture Notes in Computer Science*, vol. 5525. Springer, Berlin, Heidelberg, New York (2009)

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**(1), 1–44 (2009). DOI http://doi.acm.org/10.1145/1502800.1502803

Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: ICSE '09, pp. 111–121. IEEE Computer Society, Washington, DC, USA (2009). DOI http://dx.doi.org/10.1109/ICSE.2009.5070513

Erenkrantz, J.R., Gorlick, M., Suryanarayana, G., Taylor, R.N.: From representations to computations: the evolution of web architectures. In: ESEC-FSE '07, pp. 255–264 (2007)

Fielding, R.T.: REST: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)

Huang, Y., Garcia-Molina, H.: Publish/subscribe in a mobile enviroment. In: Proceedings of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access, pp. 27–34 (2001)

Khare, R., Taylor, R.N.: Extending the representational state transfer (rest) architectural style for decentralized systems. In: ICSE '04, pp. 428–437. IEEE Computer Society, Washington, DC, USA (2004)

Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Tran. Soft. Eng. **16**(11), 1293–1306 (1990). DOI http://dx.doi.org/10.1109/32.60317

Network Working Group: Role of the Domain Name System (DNS) (2003). RFC3467

Object Management Group: Unified Modeling Langiage Specification (2010). Version 2.3

Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE '98, 1998.

Papadimitriou, D.: Future internet – the cross-etp vision document. http://www.future-internet.eu (2009). Version 1.0

Richardson, L., Ruby, S.: Restful web services. O'Reilly (2007)

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). DOI http://doi.acm.org/10.1145/336512.336567

Romero, D., Rouvoy, R., Seinturier, L., Carton, P.: Service discovery in ubiquitous feedback control loops. In: DAIS, pp. 112–125 (2010)

Saha, D., Mukherjee, A.: Pervasive computing: A paradigm for the 21st century. Computer **36**(3), 25–31 (2003). DOI http://doi.ieeecomputersociety.org/10.1109/MC.2003.1185214

SMSCom: Self Managing Situated Computing. http://www.erc-smscom.org/ (2008)

Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Trans. Softw. Eng. **33**(12), 856–868 (2007). DOI http://dx.doi.org/10.1109/TSE.2007.70733

Vinoski, S.: Demystifying restful data coupling. IEEE Internet Computing **12**(2), 87–90 (2008)