Politecnico di Milano
*Dipartimento di Elettronica e Informazione*
Dottorato di Ricerca in Ingegneria dell'Informazione

# A Reference Model and a Run-time Support for Pervasive Adaptive Systems

Doctoral Dissertation of:
**Marco Funaro – 738566**

Advisor:
**Prof. Carlo Ghezzi**
Tutor:
**Prof. Gianpaolo Cugola**
Supervisor of the Doctoral Program:
**Prof. Carlo Fiorini**

2011 – XXIV

Politecnico di Milano
*Dipartimento di Elettronica e Informazione*
Piazza Leonardo da Vinci, 32 I-20133 — Milano

# Acknowledgements

Comincio questi ringraziamenti con lo scusarmi con tutti quelli che, inevitabilmente, tralascerò per colpevole dimenticanza.

I primi ringraziamenti da fare sono sicuramente quelli "istituzionali". Perciò ringrazio il mio relatore, Carlo Ghezzi per avermi guidato in questi tre anni. Ringrazio, inoltre, Mauro Caporuscio per avermi aiutato durante l'intero dottorato e per aver avuto la pazienza di lavorare a stretto contatto con me. Una speciale menzione merita Alessandro Campi per aver speso molto del suo tempo per convincermi a intraprendere il dottorato, prima, e per supervisionare la mia ricerca minore, dopo. Per completare il quadro, è d'obbligo citare coloro i quali hanno avuto fiducia in me dandomi la possibilità di fare un'esperienza divertente ed educativa come quella di insegnare: Alessandro Campi (di nuovo!), Gianpaolo Cugola e Paola Spoletini.

Altri ringraziamenti quasi istituzionali vanno alla mia famiglia (mamma, papà, Stefano e i nonni) che mi ha sempre supportato in questi tre anni.

Il percorso di dottorato, oltre ad avermi fatto crescere professionalmente, mi ha anche permesso di incontrare molte persone che sono diventate importanti da aggiungere a chi, invece, c'è sempre stato. Cominciando da questi ultimi, è doveroso ringraziare i miei amici di sempre, che ormai sono *habitué* delle mie tesi: Claudio (anche in assenza!), Lara (che mi ha sempre trattenuto dal diventare troppo ingegnere) e Ale (l'unico non ingegnere al mondo che mi capisce davvero). Altri amici che devono essere citati sono i miei compagni di viaggio giapponesi (ovviamente non solo per questo) Alex *Gloom* Frignani e (Gian)Andrea Uggetti. Ultima tra i non Politecnici è Martina Berardi che negli ultimi mesi ha portato un po' di freschezza nella routine di tutti i giorni.

Passando a chi, invece, ho incontrato in questi ulteriori tre anni di Politecnico, non si può non partire dai componenti storici dell'ufficio follia: dando precedenza all'unica donna, Liliana *La Marchesa* Pasquale, proseguendo, poi, in ordine alfabetico con Luca Cavallaro e le sue continue minacce di morte (se non peggio), Daniel J. Dubois e la sua incredibile (nel bene e nel male) coerenza, Alessandro Margara e la sua missione di portare a noi poveri stolti la Parola di Steve ed Andrea Mocci che amiamo tutti ricordare in versione maestrina alla lavagna che ci catechizza tutti sull'uso del principio di sostituibilità della Liskov. Oso arrogarmi il diritto di assegnare un seggio onorario a Diego Perez che è diventato uno di noi. Tutte loro, oltre ad essere semplici colleghi, sono diventati cari amici con cui ho condiviso tante esperienze (più o meno gradevoli) anche al di là del lavoro. In futuro ci sarà sempre qualcosa che ci legherà l'uno all'altro e che non potremo mai dimenticare: lo *stupidiario*. Per

quanto riguarda coloro i quali che non hanno avuto il privilegio di far parte dell'ufficio follia nella sua epoca d'oro un ringraziamento particolare va a Gigi Drago per i quasi dieci anni di mutua sopportazione. Altri da citare per aver in qualche modo condiviso e reso più piacevole il mio soggiorno al Politecnico sono, in ordine sparso, Marcello Bersani, Nicola Calcavecchia, Santo Lombardo, Antonio Filieri, Mario Sangiorgio, Amir Molzam Sharifloo (e la sua malàttia), Leandro Sales, Alberto Marinelli, Alessandro Sivieri, Guido Salvaneschi, Giordano Tamburrelli, Matteo Miraz, Danilo Ardagna e Sam Guinea. Da aggiungere all'elenco c'è anche Martina Maggio che è stata accolta spesso tra di noi nonostante il suo grave atto di apostasia, come si fa a preferire l'automatica all'informatica? Una menzione a sé stante la merita sicuramente Achille Frigeri e la sua rubrica del Papa della settimana che ha tanto dilettato me ed Andrea Mocci per molte pause caffè. Per concludere in bellezza, mi resta solo da ringraziare Paola Spoletini, che è passata da "quella pazza dell'esercitatrice di Informatica Teorica" ad essere praticamente una di famiglia. D'altra parte, se posso dire che, dopo dieci anni, Milano è come una casa per me, è anche, e soprattutto, per merito suo.

# 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: (*i*) pervasive applications should be able to dynamically configure themselves, also benefiting from third-party functionalities discovered at run time and, (*ii*) pervasive applications should be aware of, and resilient to, environmental changes. In this thesis 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 advocate for 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. Following this belief, we created a new architectural style, called P-REST, that is derived by REST by taking into account the inherent instability of pervasive environments. We also provided the P-REST users with (*i*)a methodology to design P-RESTful applications, (*ii*) a run-time support called PRIME that natively provides the P-REST abstractions, and (*iii*) a coordination language called P*a*CE meant to discover and orchestrate functionalities available in the pervasive environment. Furthermore, we devised a case study to show how the run-time support and the language can be used to develop complex applications and to assess the performance and scalability of PRIME.

# Riassunto

Le infrastrutture informatiche sono, al giorno d'oggi, una parte fondamentale del nostro tessuto sociale e un numero sempre crescente di dispositivi viene continuamente attivato nell'ambiente che ci circonda. La visione del *pervasive computing* sta diventando realtà. Per riuscire a sfruttare fino in fondo le opportunità dell'ambiente pervasivo, si è sentita la necessità di rivisitare le metodologie classiche di sviluppo software alla luce delle mutate condizioni ambientali. Infatti le applicazioni pervasive dovrebbero poter autoconfigurarsi dinamicamente, anche traendo beneficio da funzionalità di terze parti scoperte a tempo di esecuzione e, inoltre, dovrebbero avere coscienza dei cambiamenti nell'ambiente e continuare a funzionare correttamente nonostante questi ultimi. In questa tesi vogliamo affrontare l'ambiente descritto sopra da un punto di vista architetturale con l'obiettivo di facilitare sia lo sviluppo di software pervasivo che l'adattamento a tempo di esecuzione necessario per far fronte ai cambiamenti nell'ambiente. Andando più nello specifico, in questa tesi, si vuol proporre l'adozione dello stile architetturale REST per far fronte alle sfide poste dall'ambiente pervasivo. Infatti in questo lavoro si sostiene che, nonostante REST sia stato introdotto sulla base dell'osservazione e dell'analisi della struttura di internet, il suo campo applicativo non sia ristretto ad internet stessa. Seguendo questa linea di pensiero, si è creato un nuovo stile architetturale chiamato P-REST derivato da REST tenendo in considerazione l'intrinseca instabilità degli ambienti pervasivi. Inoltre, il nuovo stile è stato corredato da una metodologia progettuale *ad hoc*, un supporto per l'esecuzione chiamato PRIME che fornisca nativamente le astrazioni computazionali usate da P-REST e, ultimo ma non ultimo, un linguaggio di coordinamento chiamato P*a*CE concepito per scoprire e orchestrare le funzionalità disponibili nell'ambiente pervasivo. Per mostrare l'efficacia dell'intera metodologia, dimostrare l'utilizzo degli strumenti messi a disposizione e valutare le prestazioni di PRIME è stato condotto lo studio di un caso abbastanza complesso da essere rappresentativo per le applicazioni pervasive.

# Contents

# List of Figures

# List of Tables

# Part I.

# 1. 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 [1]. 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". Such "things" can be the most disparate devices (e.g., desktop PCs, laptops, tablets, appliances, sensors, actuators, etc.). This situation is often referred to as *Internet of Things*. 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.

The above concepts underlie the Future Internet vision [1], 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 [2]. 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 "things" available in the environment at a given time. Such applications, sometimes also called *open-world* applications [3], are characterized by a highly dynamic software architecture: both the components that are part of the architecture and their interconnections may change dynamically, while applications are running. New components may, in fact, be created by component providers and made available dynamically. Components may then be discovered, deployed, and composed at run time, removing pre-existing bindings to other components. Applications are often highly distributed, i.e., components are deployed and run on different computational units. In many cases, the components that constitute an application are also operated and run by decentralized and autonomous entities.

Dynamic architectures of the kind we described above are created to

support the adaptive and evolutionary situation-aware behaviors that characterize pervasive systems. Sometimes it may be useful to distinguish between *adaptation* and *evolution*. Adaptation refers to the actions taken at run time and affecting the architectural level, to react to the changing environment in which these systems operate. In fact, changes in the physical context may often require the software architecture to also change. As an example, a certain facility used by the application may become unaccessible as a new physical environment is entered during execution and a new facility may, instead, become visible. Or a certain facility's behavior may be changed unexpectedly by the owner of the service and the change may be incompatible with its use from other parts of the application. Evolution instead refers to changes that are the consequence of requirements changes. For example, a 3-D interface becomes available and must be used instead of the traditional interface previously used. In general, long-lived pervasive systems require that applications should follow some strategies to

- detect the relevant changes in the situation in which they operate, such as the physical environment (or even the changing requirements), and
- react by self-organizing themselves and adapting their behavior in response to such changes.

Adaptive pervasive systems raise many challenges to software engineering as highlighted by Cheng et al. in [4]. Indeed, such systems stress the known methods, techniques, and best practices to their extreme and introduce new difficult problems for which new solutions are needed. The notions of variability and adaptation must permeate all phases, from requirements to design and validation, and even run time. Indeed, the sharp traditional separation between *development time* and *run time* becomes blurring. Traditionally, changes are handled off-line, during the maintenance phase. In the new setting, they must be also handled autonomously at run time, as the application is running and providing service. To achieve that, software systems must be able to reason about themselves and their state as they operate, through adequate reflective features available at run time. They must be able to monitor the environment, compare the data they gather against an expected model, and detect possible situational changes. Whenever a deviation is found, an adaptation step must be performed, which modifies the software architecture and, as a consequence, the running application. For example, the adaptation step might simply perform a new deployment and re-bindings to different components, or re-binding to different external services. In

other cases, the adaptation strategies may be more complex. To lighten the burden of dealing with all the problems presented above at run time, it is of critical importance bringing at run time and keep alive the models usually confined in the software design phase [5].

In the context just described, this thesis is concerned with finding the most suitable architectural abstractions for the pervasive scenario and provide a set of tools to assist developers from design time to run time. The original contribution of this thesis are the following:

- a conceptual model to guide the development of adaptive pervasive systems;
- a novel architectural style called P-REST, built upon the REST style, that fits the scenario described above;
- the PRIME middleware designed and implemented to fully support P-REST
- the P*a*CE coordination language embedded in PRIME to orchestrate functionalities in pervasive environments.

## 1.1. Thesis structure

This thesis is structured as follows: Chapter 2 introduces to the general scenario of adaptive systems and set the tone of the discussion for the thesis; Chapter 3 explains the limitations of REST when dealing with pervasive scenarios and how we overcame such limitations using the P-REST style. Chapter 4 describes PRIME, the middleware we designed and implemented as a run-time support for P-REST. Chapter 5 presents the P*a*CE coordination language that we have devised to ease the development of complex behavior in PRIME. Finally, Chapter 6 concludes the thesis and outlines future research lines.

# 2. Architectural issues of adaptive pervasive systems

In this chapter we want to:

1. describe the general context of adaptive and evolvable systems,
2. narrow the focus to address the pervasive scenario, and
3. characterize the contribution of this thesis with respect to this general scenario.

Thus the chapter is divided into two Sections, Section 2.1 will address the first two points, while Section 2.2 will be concerned with the third one.

## 2.1. Design models for adaptive systems

Research on dynamically adaptable and evolvable software systems became very active in past decades. Already in 1995 Mary Shaw in [6] advocated for the need of departing from the standard object-oriented design paradigm for certain classes of problems where the control of a process was the main concern. According to her, classic software design tends to compute the desired output on the basis of the inputs only. This approach fits smoothly most problems but, whenever the problem must also take into account unpredictable variables that can change at run time, the classic approach fails. Therefore, she proposed a design model borrowed by the control theory. To take into account unpredictable variables, she advocated for the introduction of a *feedback loop* aimed at controlling the computing process and make it adaptable with respect to the surrounding environment. The feedback loop is in charge of continuously monitoring the output variables in order to modify input variables to adjust the behavior of the computing process. Building on the experience with the new design model, the author identifies the cases the use of the new design model is convenient:

- when the task involves continuing action, behavior, or state

- when the software is embedded; that is, it controls a physical system
- when uncontrolled computation does not suffice, usually because of external perturbations or imprecise knowledge of the external state

In pervasive environments, like the ones we have described in the previous chapter, the external perturbations and the imprecise knowledge of the environment are not only expected, but even likely. For these reasons the community felt the necessity to rigorously investigate the problem of controlling software. To do so, the need of a sharp distinction between the computation that must be observed and controlled, and the software that implements the controller soon arose. This separation of concerns, according to Müller et al. in [7], allows for giving to the control loop the same dignity of the controlled software along the whole development process. It means that, not only the controlled software must be properly designed, documented, developed, tested and maintained, but also the control loop must be involved in the same activities. Besides, applying the classic software engineering techniques to a modular control logic helps in making it reusable, analyzable, composable, and cost-effective as highlighted by Cheng et al. in [8].

Once recognized the need for a principled approach to software adaptation, the research community strove for identifying the conceptual entities involved in adaptation and their relationships. A very high-level conceptual model was presented by Dobson et al. in [9] where they presented the model in Figure 2.1. Although the model is tailored to the problem of network communication, the 4 main phases (i.e., Collect, Analyze, Decide, Act) are valid for every self-adaptive system. For each phase several approaches to the problems that must be coped with are shown. Still, this model is very high-level, hence, it is not suitable for a rigorous guidance during the design phase.

Moving towards a more concrete model, we must mention the one introduced along with the proposal of *autonomic computing*. In the early 2000s, Kephart and Chess promoted the vision of autonomic computing [10], which focuses on a new generation of software systems that can manage themselves to achieve their goals in a changing environment. A system can be defined as autonomic if it enjoys the following properties:

**Self-Configuration** : an autonomic system should be able to (re)configure itself according to a high-level goal. Such goal is just specified declaratively, the system should figure out *how* the goal can be reached.

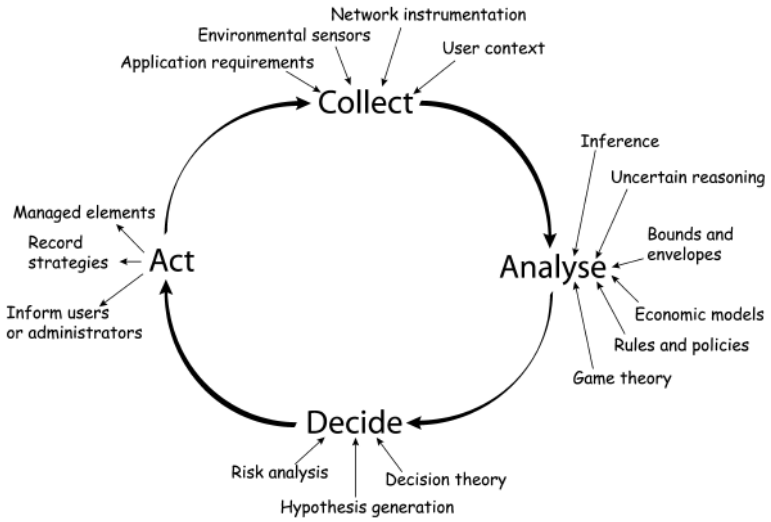


Fig. 1. Autonomic control loop.

Figure 2.1.: high-level control loop for self-adaptive systems

**Self-Optimization** : an autonomic system should always take the opportunities to optimize itself *proactively* in order to improve efficiency.

**Self-Healing** : an autonomic system should *react* to its failures by detecting (or even predicting) them and taking actions to restore its functionalities

**Self-Protection** : an autonomic system should strive for protecting itself from malicious external attacks as well as from unwise changes coming from its user.

Such properties are often referred to as *self-** properties.

Always according to [10], an autonomic system is built starting from a collection of autonomic elements that interact with each other by providing and consuming services. Kephart and Chess also propose the high-level software architecture that such autonomic elements must comply with. In practice, they advocate for the installation of a feedback control loop whose main components are depicted in Figure 2.2. There, the status of the *Managed Element* is read by the *Monitor* through the *Sensors*. The *Analyzer* recognizes possible deviations from the overall goals that should be solved by the *Planner*. Then, the *Executor* exploits *Actuators* to enact changes suggested by the planner. All the components just described take advantage of the *Knowledge* gathered by the *Autonomic*

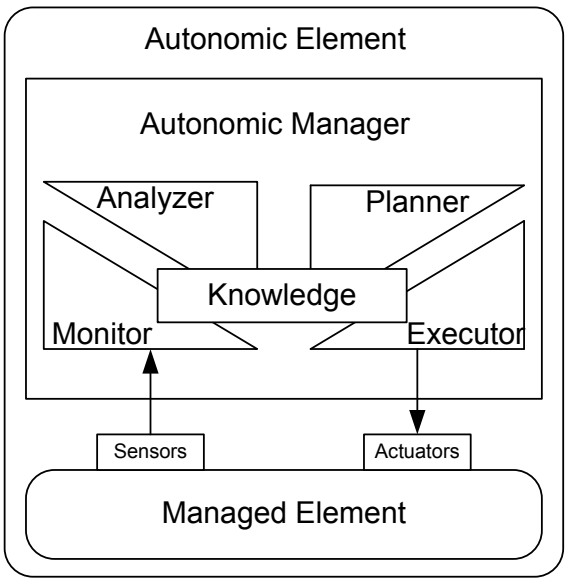


Figure 2.2.: MAPE-K loop

*Element* about itself and its operating environment. The feedback loop of Figure 2.2 has been named MAPE-K loop from the initials of its main entities, that is, Monitor, Analyzer, Planner, Executor, and Knowledge. After the publication of the autonomic manifesto, a lot of work has been carried out to deeply investigate every component in Figure 2.2. A wide coverage of this research is out of the scope of this thesis, see [11] for a complete survey. Instead, here, we want to narrow the scope and focus on the software architecture standpoint. Indeed, The MAPE-K cycle is very high-level and general because it is designed to model every possible autonomic system is every possible domains. It is clear that, by narrowing the application domain, more precise architectural models can be built, and a more precise model leads to an increased guidance in the application design and development.

As an example of design model for autonomic systems in a specific domain, we can refer to the model introduced by Bucchiarone et al. in [12] that captures the life-cycle of Service Oriented autonomic systems (see Figure 2.3). The rightmost circle represents the (simplified) traditional development process that starts with requirements elicitation, goes through the system construction and arrives to the deployment and to the run-time management. The leftmost circle, on the other hand, represents the action needed for adaptation. Thus, it starts with the recognition of the need for adaptation, followed by the search for a suit-

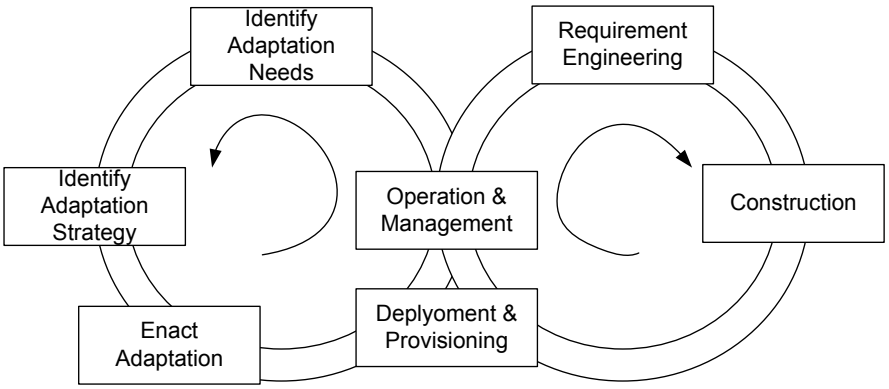


Figure 2.3.: The life-cycle of adaptable service oriented autonomic systems

able adaptation strategy. Once such strategy is found, it is enacted and the system undergoes to a new deployment phase and it resumes its standard behavior (by reentering the rightmost cycle).

Following the example in [12], here we want to give a specialized and refined version of the MAPE-K cycle for the pervasive domain. To this extent, in [13] we presented the conceptual model depicted in Figure 2.4 to describe the fundamental concepts and properties that characterize software architectures for adaptive pervasive systems. Notice that the model is just a reference, hence it can result in very different implementations, even partial where it fits.

The conceptual model is aimed at handling and controlling the *Application* entity. The latter is built using several software artifacts, which can be owned by both the application developer and third parties. The part owned by the application developer is called *Controlled Application*, while the third-party software is represented by the *External Service/-Component* entity. Such distinction is important because the *Actuator* can only operate on the controlled application because the third-party software is, by definition, out of its control. The controlled application and the external services/components are immersed in the *Environment*. The latter also contains *Sensors* to enhance the application with contextual data.

A key role in the control of the application is played by the *Requirement* entity, which defines the initial input steering the construction of the application assembly, as well as the desired run-time behavior. Indeed, it defines the set of properties that the application must satisfy at run time. Requirements are also important to precisely define the dif-

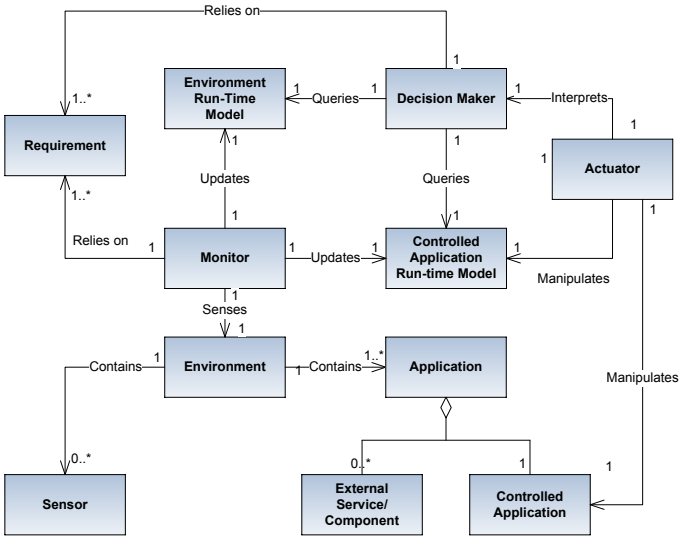


Figure 2.4.: Reference Architectural Model for Self-Adaptive, Pervasive Systems

ference between evolution and adaptation: evolution refers to the ability of reconfiguring the controlled application at run time as a response to a change in the requirements, whereas adaptation refers to the ability to modify the controlled application so that it can keep satisfying the requirements in spite of changes within the execution environment. This twofold role of requirements demands for (*i*) a *Decision Maker*, that assembles an abstract description of the application complying with the requirements, and (*ii*) a *Monitor*, that is in charge of collecting data about the application's run-time behavior to detect (and predict where it is possible) possible violations of the requirements.

The decision maker is the entity that is in charge of enforcing requirements. It should be able of (*i*)creating an application complying with the requirements from scratch and (*ii*) of adapting and evolving the application at run time. The decision maker generates abstract descriptions of the solutions it devises. The descriptions are abstract in the sense that they don't deal with implementation details and are technology-agnostic. To devise such description, the decision maker needs to know the current situation both of the controlled application and of the surrounding environment. These two pieces of information are captured by the *Application Run-time Model* and by the *Environment Run-time Model* entities, respectively. The former captures the run-time state of the application,

whereas the latter is concerned with the surrounding physical context and with all the third-party software offered in the environment. The use of two models are needed because dealing with the real application and the real environment is often intractable in non-trivial cases. As a final remark on the decision maker, it can also be not completely automated and human-in-the-loop solutions are allowed by the reference model.

The abstract descriptions produced by the decision maker are fed to the *Actuator*. The last is introduced to decouple the decision maker by the lower-level implementation details. On one hand, it executes the command issued by the decision maker in order to create the application (in the application synthesis phase) or manipulate the controlled application (in case of adaptation or evolution). On the other hand, it is in charge of keeping the controlled application synchronized with the controlled application run-time model. Thus, the decision maker can carry out its role without directly interacting with the controlled application. The indirection allows for generating two equivalent applications, implemented by means of two different technologies, starting from the same abstract description. We only need to provide two different and technology-specific actuators.

To continuously provide the decision maker with fresh information about the environment, the conceptual model prescribe the presence of a *Monitor*. Its final aim is updating the two run-time models with information gathered from the real world. It feeds the environment run-time model with both data gathered from external services/components that are not part of controlled application and sensor data that provide relevant information about the physical environment. Whereas the data coming from the controlled application and from its external services/-components are fed to the controlled application run-time model.

## 2.2. Architectural styles

In the previous section we have given a general overview of the problems that arise when coping with software adaptation. Now we want to narrow the focus to correctly frame our work. Specifically, referring to the model in Figure 2.4, here we want to focus on the constraints that we can impose on the design of the application (that are, then, reflected into the application run-time model) to make the work easier for the actuator. Indeed, constraining the application design decreases the number of free variables that must be taken into account, on one hand, by the decision maker to analyze and solve problems and, on the other hand, by the

actuator to manipulate the running application.

Clearly, the constraints that are commonly imposed to the design of an application must be well engineered in order to effectively benefit the run-time adaptability. In literature a set of constraints is commonly called *architectural style*. According to [14], "An architectural style defines a vocabulary of components and connector types, and set of constraints on how they can be combined." By focusing on architectural styles, it is possible to focus on adaptation from an abstract and high-level standpoint, which may enable systematic and even formal reasoning.

Let us first observe that in the general case, if no specific constraints are assumed on an architecture, a run-time change that requires dynamic updates of components or connectors may require suspension of (parts of) an application to achieve some desirable level of consistency. Managing suspensions can be very complex. This problem has been faced elsewhere in the literature [15, 16]. In these works applications are represented as graphs with components as vertices and connectors as edges. Both works are focused on finding algorithms to safely substitute a component without jeopardizing ongoing computations. The solutions, in most cases, force the suspension of the most part of the components.

Taylor et al. in [17] scrutinized several architectural styles used by state-of-art software systems according to the following criteria:

- How and how much the system's behavior can be changed;
- How long the system's evolution takes to be effective;
- How the state of the system is changed when that system evolves;
- In which environment the system is executing;

Examples of examined styles and corresponding systems are: the C2 style [18, 19, 20] supported by the ArchStudio tool suite; the publish-subscribe style [21], implemented for example by Siena; the REST style [22] used for web browsing; the CREST style [23] adopted by AJAX and other JavaScript-based technologies.

The classification given in [17] is general and addresses generic adaptive systems. Here we are concerned with adaptive systems in pervasive environments, thus we propose a classification along the following dimensions:

(*i*) the type of *coupling* imposed by the model on entities;

(*ii*) the degree of *flexibility*, that is, the ability of the specific model to deal with the run-time growth of the application in terms of involved components;

(*iii*) the degree of *genericity*, that is the ability to accommodate heterogeneous and unforeseen functionalities into the running application;

(*iv*) 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 2.1 classifies the main architectural models in terms of their characteristics with respect to the pervasive networking issues.

Table 2.1.: Distributed design models dimensions

| | Coupling | Flexibility | Genericity | Dynamism |
|---|---|---|---|---|
| RPC | tight | ✗ | ✗ | ✗ |
| OO | tight | ✗ | ✗ | ✗[2] |
| C2 | loose | ✓ | ✗ | ✓ |
| SOA | loose | ✓ | ✗ | ✓[2] |
| REST | loose | ✓ | ✓ | ✗[2] |

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

**Object Oriented** architectures support distributed objects, and provide higher-level abstractions by grouping data and the functions meant to manipulate them and state encapsulation. The type of interaction among objects, however, is synchronous. In summary: (*i*) interacting objects are still tightly-coupled in a client-server fashion, (*ii*) adding/removing entities while the system is running is hard to support, (*iii*) interfaces are specified via strict method signatures, and (*iv*) 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). Two two representative implementations

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

are the CORBA (Common Object Request Broker Architecture) specification [24] that allows for remote object invocation and manipulation through a distributed broker, and JINI [25] that added to the remote object manipulation a partial syntactic object discovery facility. In both cases adaptation is not allowed by the run-time support.

**C2** style, introduced in [18, 19, 20], models a system as a set of components and connectors. Each component has one top connector and one bottom connector. A connector can accommodate an arbitrary number of top and bottom components. Components can issue requests towards their top connector (hence each component is aware of the components connected to its top connector) and generate notification towards the bottom component. Connectors broadcast all the requests coming from the bottom components relaying them to the top components. The opposite happens for notifications. Regarding our classification: (*i*) components are loosely coupled because connectors prevent direct communication among components; (*ii*) components can be added removed at run time almost seamlessly by just linking and unlinking them to the connectors of the running system; (*iii*) components must be aware of the interface of the upper components while they are completely independent from the lower components; (*iv*) connectors effectively decouple components so modifying the system architecture at run time is quite easy and intuitive. The principles of the C2 style are used by ArchStudio [26] to realize architecture-based software adaptation. ArchStudio is a complete suite of tools that takes care of both the design and the run-time support for adaptive software. the suite allows for adding and removing components in a C2 system at run time by directly using the run-time representation of the software architecture.

**Service Oriented Architecture (SOA)** a style in which 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: (*i*) services are independent and loosely-coupled entities, (*ii*) services can be easily added/removed and accessed, irrespective of their base technology, (*iii*) service access is regulated by means of well-defined interfaces, and (*iv*) 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. We can name two run-time supports that relies on service orientation. The first one, ReMMoC [27], provides SOA abstractions for pervasive systems and its main concern is overcoming heterogeneity of communication protocols and coordination mechanisms in mobile environments. The second one is the *ubi*SOAP [28] middleware that makes the standard SOA interactions seamless with respect to the underlying communication technology.

**REpresentational State Transfer (REST)** , introduced by Fielding in his doctoral thesis [22], 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 and 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, *interactions* among REST entities are stateless. In summary: (*i*) resources are independent and loosely-coupled entities, (*ii*) resources can be easily added, removed and accessed, irrespective of their underlying technology, (*iii*) resource access is regulated by means of a uniform interface, and (*iv*) 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. For the REST style we present four works. In [29], Web principles and technologies have been applied to Ambient Computing. In particular, the proposed framework addresses interoperability issues by using the Web Platform as a common ground to build systems that follow the Ambient Computing principles. The same line of research has been followed by the Cooltown project [30], which extends web principles to devices and services situated in the physical world. However, both approaches leverage Web standards to achieve interoperability, then they are also bound by the web's limitations, e.g., lack of mobility management, point-to-point communication only, and client-server style interactions. To this end, the XWeb [31] project presents a web-oriented architecture relying on a new transport protocol, called XTP, that provides mechanisms for finding, browsing, and modifying information. A different line of research has been followed by Bonetta

and Pautasso in [32] where they propose the adoption of REST to design and implement *liquid* web services, that is, services able to both scale up and scale out. Scaling up refers to the ability of exploiting new hardware resources deployed in the original hosting machine while scaling out refers to the exploitation of hardware resource made available on different hosting machines.

The genericity of the interface coupled with the statelessness of the interactions make REST an interesting candidate for implementing adaptive systems in pervasive environments. The next chapter is completely devoted to deepen the understanding of this intuition and to make it more precise and systematic.

# 3. REST for pervasive systems

In the previous chapter we reviewed the architectural styles that best lend themselves to run-time evolution and adaptation. In this chapter we want to explain why we picked REST among the others and which modifications are needed to adapt REST to the pervasive environment. This work was already presented in [33].

## 3.1. 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. [34] exploit REST to enable interoperability among mobile devices within a pervasive environment.

Unlike [34], we do not use REST principles to cope with heterogeneity, rather we want to investigate how the REST design model can be used to build 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.

Another similar research line is also pursued in the field of Web of Thing [35, 36]. They try to enable "things" to be accessible and programmable via a RESTful interface. While they want to impose as-is REST on networks of embedded sensors and actuators, we want to modify REST to make it suitable to pervasive environments.

The original REST architectural style [22] defines two main architectural entities (see Figure 3.1): 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 Figure 3.1, when a user agent issues a request for the resource identified as $R_b$ to $OriginServer_2$, it obtains as a result a *Representation* of the

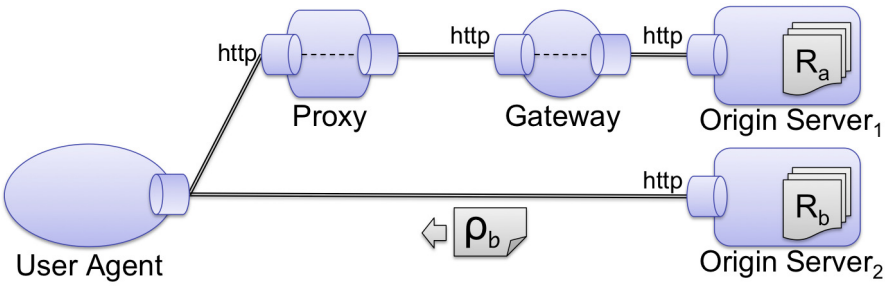


Figure 3.1.: REST Architectural Style

resource (i.e., $\rho_b$). A *Representation* is not the resource itself, but captures the current state of the resource in a format matching a standard data type. Moreover, from the moment it is created on, a representation is no longer bound to the evolution of the resource. It means that further updates of the resource are not reflected on the representation and vice versa.

The concept of *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 [22] and then elaborated by Richardson and Ruby [37]: *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 allows 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 [38].

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[1], will produce the

[1]We define two resources as equivalent iff they have the same behavior and adopt

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 (*i*) decoupling of interacting resources, (*ii*) flexibility of the model, since it allows for easily rearranging the application at run time and, (*iii*) 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 back and forth at each request.

The *connectedness* principle, which refers to the need of linking resources to one another, is the backbone of RESTful applications. It was initially introduced by Fielding in his thesis as the "Hypermedia As The Engine Of Application State" (HATEOAS) principle. Links among resources induce a lightweight and dynamic work-flow 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 URI.

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 when (if) 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 has been 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 (See [39] for further discussions on the topic).

Differently 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, is it a good practice for URIs to be self-explanatory in order to ease human comprehension for human beings. However, such practice is not strictly prescribed by the REST architectural style.

---

the same encoding for their representations.

## 3.2. REST for pervasive systems

REST technologies rely on (*i*) the stability of the underlying communication environment and (*ii*) on a tightly-coupled synchronous interaction protocol (i.e., HTTP). Pervasive environments, instead, require to (*i*) cope with an ever-changing communication infrastructure because devices join and leave the environment dynamically [40] and (*ii*) to support loosely-coupled asynchronous coordination mechanisms [41].

This section investigates how the REST architectural style should be modified to fit pervasive environments, and introduces the Pervasive-REST (P-REST) design model. To this end, we need to make changes at three levels of abstraction, namely the *architectural*, the *coordination*, and the *infrastructural* levels.

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 (see Section 3.1). 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 user agents and origin servers. so a resource can play both active and passive roles at the same time.

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 [42], leveraging standard protocols [34]). 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's' URIs and their actual location. Such service plays the role of a distributed Domain Name System (DNS) [43].

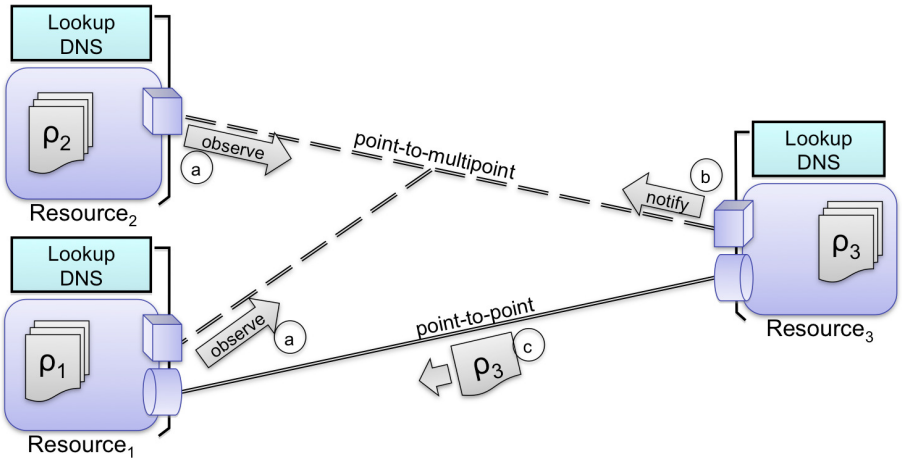


Figure 3.2.: P-REST Architectural Style

In addition, we adopt a coordination style based on the Observer pattern, as advocated in the Asynchronous-REST (A-REST) proposal described by [44]. 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 3.2 summarizes the modification we made to the REST style. Resources directly interact with each other to exchange their representations (denoted by $\rho$ in the Figure). Always in Figure 3.2, both $\text{Resource}_1$ and $\text{Resource}_2$ observe $\text{Resource}_3$ (messages ⓐ). When a change occurs in $\text{Resource}_3$, it notifies (message ⓑ) the observer resources. Once received such a notification, $\text{Resource}_1$ issues a request for the $\text{Resource}_3$ and obtains as a result the representation $\rho_3$ (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.

### 3.2.1. P-REST meta-model

Along with the P-REST architectural style introduced above, we also define a P-REST meta-model (depicted in Figure 3.3) describing the pervasive environment, the entities within the environment, and the relations among entities 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: (*i*) `OBSERVE`, which declares the interest of a resource in the changes occurring in a different resource identified by a given URI, (*ii*) `NOTIFY`, which allows the observed resource to notify its observers about the occurred changes, and (*iii*) `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 metamodel describes every software artifact within the environment as a Resource. According to the *Uniformity* principle (see Section 3.1), each resource implements a set of well-defined operations, namely `PUT`, `DELETE`, `POST`, and `GET`. 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 current state of the resource. Notice that resources are not required to always have all operations enabled (e.g., a resource that models the seats available in a cinema cannot be deleted by its users).

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 executing an operation $N > 1$ times yields the same effect of executing the same operation once. `GET` is idempotent and safe, `PUT` and `DELETE` 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. Indeed, P-REST promotes resource's semantics as first-class concern by explicitly intro-

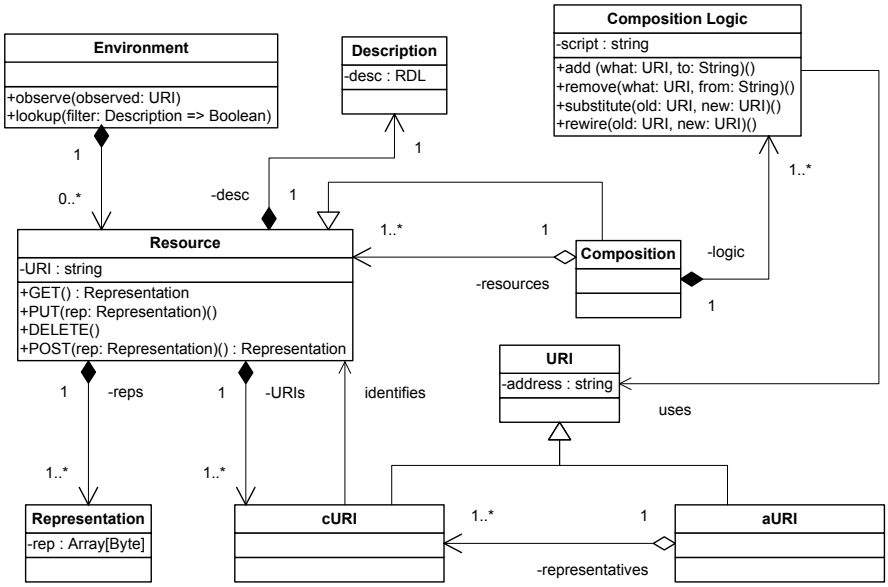


Figure 3.3.: P-REST meta-model

ducing the `Description` entity. A description contains 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 [45]. 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 2.1). In fact, Descriptions support the implementation of the lookup service by exploiting efficient algorithms for distributed semantic discovery (e.g., [42]), thus enabling de facto runtime resource discovery. Referring to the HTTP uniform interface that underlies REST, a description contains not only the information usually retrieved through the `HEAD` and `OPTION` operations but is also goes further by providing 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. To provide access to the internal state, `Representation` entity is introduced. A representation is a specific rendering of a resource internal state. 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 Section 3.1, *addressability* principle 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 point-to-point 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. Following [46], a `Composition` of resources is still a resource that can, in turn, be used as a building-block by another composition. Clearly a `Composition` must expose the REST uniform interface and preserve its semantics, i.e., an idempotent call on the composite resource cannot result on non-idempotent calls on the component resources.

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: (*i*) the composition is defined in terms of explicit relations between resources (i.e., connectedness), (*ii*) resources involved in the composition are explicitly identified by means of resource identifiers (i.e., addressability) and, (*iii*) 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.

## 3.3. 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 (Section 2.1) can be implemented by means of the P-REST meta-model (Section 3.2), and show how the mechanisms provided by PRIME make P-RESTful application effective.

Both the conceptual model (Figure 2.4) and the P-REST meta-model (Figure 3.3) 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 Figure 2.4, 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 re-

sults 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. [47], an actuator operating at the software architecture level should support two types of change: one affecting the components, namely component *addition*,component *removal* and component *substitution*, and one affecting the connectors, namely connector *rewiring*.

The problem of dynamically deploying and/or removing a component from an assembly has been repeatedly tackled in literature [15][16]. 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: (*i*) deploy the new resource within the environment, and (*ii*) 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, even though 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 Figure 3.3), 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.

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 disruptive. 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 Figure 3.3, the signature of the rewire operation is:

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

Its semantics is such that the `old` cURI is substituted with the `new` cURI. Unlike the `substitute` operation the state of the old resource is *not* transferred to the new one.

It is important to remark that the execution of the composition script must be suspended in a safe state before the adaptation actions can be performed.

# Part II.

# 4. PRIME

The PRIME (P-Rest run-tIME) middleware provides the run-time support for the development of P-RESTful applications[1]. As depicted in Figure 4.1, the PRIME architecture exploits a two-layer design where each layer deals with a specific issue. Specifically, (*i*) PRIME *communication layer* implements an overlay network providing both point-to-point and point-to-multipoint transports among nodes, and (*ii*) PRIME *programming model* provides programmers with the proper abstractions and operations to implement P-RESTful applications.

**Communication layer** – To deal with the inherent instability of pervasive environments, PRIME arranges devices in an overlay network built on top of low-level wireless communication technologies (e.g., Bluetooth, Wi-Fi, 3g, HSDPA, etc.). Such an overlay is then exploited to provide two basic communication facilities, namely *point-to-point* and *point-to-multipoint*. *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. Furthermore, the PRIME communication layer provides facilities for managing both physical and logical mobility [40].

**Programming model** – PRIME provides the programming abstractions to implement P-RESTful applications by leveraging the functional features of the Scala programming language [48] and the Actor Model [49]. PRIME defines two main abstractions and a set of operations to be performed on them. *Resource* represents the computational unit, whereas *Container* handles both the life-cycle and the provision of resources. The set of operations allowed on resources defines the message-based PRIME *interaction protocol* and includes:

1. *Access* operations allows for accessing and manipulating resources,
2. *Observe/Notify* operation allows resources to declare interest in a given resource and to be notified whenever changes occur,
3. *Create* operation provides the mechanism for creating a new resource at a given location and *Move* provides the mechanism to

[1] PRIME is available at `http://code.google.com/p/prime-middleware/`, under the GNU/GPLv3 license.

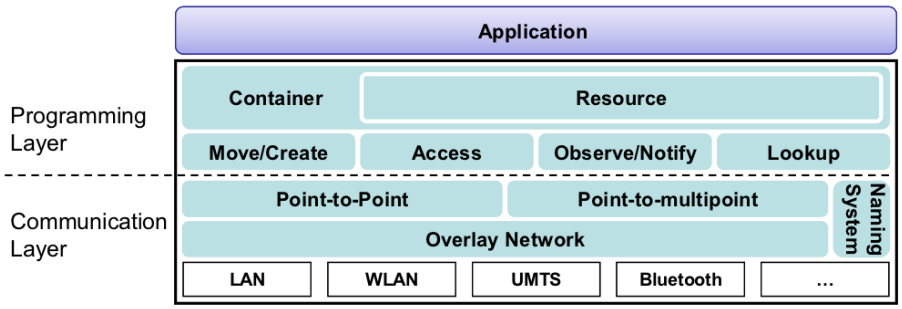


Figure 4.1.: PRIME software architecture

relocate an existing resource to a new location,

4. *Lookup* operation allows for discovering new resources on the basis of a given description.

PRIME meets the set of requirements introduced in Section 2.2:

1. *flexibility* is achieved by exploiting the Actor Model, which, in turn, relies on the PRIME overlay network to provide message-passing interaction among actors,
2. *genericity* arises from the uniformity principle exploited in conjunction with both code mobility and functional programming capabilities (e.g., high-order functions)
3. *dynamicity* is provided by means of semantic lookup, uniformity and resource composition.

Next Sections clarify these aspects and provide implementation details about the *communication layer* (Section 4.1) and the *programming model* (Section 4.2), respectively.

## 4.1. The PRIME communication layer

PRIME arranges devices in an overlay network, a virtual network of *nodes* and *logical links* built on top of an existing actual network [50]. Indeed, overlay networks implement a set of network services that are not available in the native network. Hence, in PRIME the overlay network implements the mechanisms for the management of the *logical links* between

*nodes* and for the message exchange. In particular, PRIME embeds (*i*) the protocols that keep the overlay network connected when the topology of the underlying native network changes as a consequence of mobility, and (*ii*) the routing algorithms that regulates the message flow between nodes with reference to the specific coordination model used.

Due to the high dynamism inherent to the pervasive networking environment, the main requirement for the overlay network is the ability to self-organize into a flexible topology, as well as to maintain it. To this end, PRIME leverages the REDS framework [51]. Specifically, REDS arranges nodes in an overlay to provide both content-based publish/subscribe and point-to-point communication. REDS nodes can be either *thin nodes* or *broker nodes*. *Thin nodes* are used by REDS users to send and receive messages, whereas the *broker nodes* are responsible for implementing the desired routing strategy and managing the routing tables.

Topology construction and maintenance are key issues when dealing with overlay networks. To this extent, different strategies might be exploited depending on the nature of the "relationship" between nodes, namely *physical* or *logical*. *Physical* relationships are those emerging from the physical environment and the underlying native network, such as in the case of unstructured networks, where links between nodes are established by considering physical context – e.g., radio interferences and energy consumption [52]. On the other hand, *logical* relationships are implied by the application domain, such as in the case of peer-to-peer networks, where links between nodes are established by considering the high-level business logic – e.g., file sharing and content-based routing [53]. PRIME exploits a synergic strategy in order to optimize overlay networks with respect to both *physical* and *logical* nodes relationships [54].

To meet the PRIME requirements, we added two class of functionalities to REDS, namely the bootstrap support and the code mobility facilities. In the remainder of this section we will address both.

When a PRIME node bootstraps, it is associated with both a thin node and a broker node, where the thin node represents the gateway for incoming and outgoing messages – referred to as Gateway Thin Node (GWTN from now on). However, to effectively enter the overlay network, the PRIME node needs to discover an *access point* (i.e., a broker node that is already part of the overlay) and connect to it. To this end, PRIME resorts to UPnP (Universal Plug and Play) [55], which provides a standard protocol to publish, discover and consume services in a subnet. Specifically, every PRIME node advertises itself and, when a discovery request is received, it returns its own IP address, along with contextual information related to it. According to [54], this includes: (*i*) available energy and physical location, (*ii*) mobility profile (e.g., a desktop ma-

chine, a laptop, a mobile phone, etc.) since the more it is stable the more a node is qualified to play the *access point* role, and (*iii*) computational resources since powerful nodes can better accomplish routing tasks. When the new PRIME node searches the network for access points, it receives as result a set of records describing the broker nodes eligible for such a role. Thus, the requesting node evaluates the obtained results, and connects to the broker node that best fits for the *access point* role. The overall overlay infrastructure is then built by iterating such a procedure for every newly started PRIME node. Furthermore, such mechanism based on UPnP is also exploited to cope with network instability. Due to device mobility, a PRIME node might become isolated from the rest of the overlay. Thus, the algorithm above is restarted to allow the node to discover a new *access point*, and re-join the overlay.

It is worth to note that, in order to guarantee the correct message exchange between PRIME nodes, when the overlay topology changes the routing tables should be updated accordingly. To this extent, PRIME provides a specific Domain Naming System, as stated by P-REST (see Section 3.2), which maintains the mapping between PRIME nodes and their actual location within the overlay. REDS provides developers with mechanisms to define their own routing strategies, as well as to build and update routing tables. Since REDS identifies both broker and thin nodes with a unique name, which is used to route packets, PRIME encodes routing tables as maps, where the key represents the final destination (thin node name), and value is the next hop in the overlay towards the destination. Indeed, sending a packet to a symbolic name of a node causes the packet to be delivered, hop by hop, to the right host in the network. Further, to effectively account for mobility issues, PRIME defines a specific set of rules achieving the correct update on the routing tables. First, in case of multiple registrations of the same identifier, the last registration overrides the previous one. Second, PRIME always grants unique names for new nodes. Hence, whenever a node is moved, it is registered at the new location with the old name. Such a new registration is then propagated throughout the whole overlay network, updating defacto the naming system with the new location for the node. Moreover, since PRIME implements point-to-multipoint communication by leveraging the native REDS publish/subscribe protocol, subscription tables are updated according to the publish/subscribe semantics [51].

Referring to the *genericity* requirement, PRIME should be able to accommodate heterogeneous and unforeseen functionalities into the running application. It means the PRIME must enable all its nodes to handle resources whose bytecode appeared somewhere in the system at run time. To this extent, the PRIME *communication layer* leverages code

mobility [40] and enables PRIME to dispatch messages whose bytecode is known by the source node only. Specifically, PRIME adopts two different approaches to code mobility, according to the specific coordination model used. Concerning point-to-point communication, PRIME implements an end-to-end strategy that enables two PRIME nodes to exchange bytecode. Whereas, concerning point-to-multipoint communication, PRIME adopts a hop-by-hop strategy that, starting from the origin node, spreads the executable code towards multiple destinations.

Independently of the specific strategy, PRIME 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. There, the object must be deserialized before it can be used. However, if the object is unknown to the destination (i.e., the destination node has not the bytecode for the received object), the object cannot be deserialized. To this extent, PRIME implements a custom *classloader* which is in charge of retrieving the bytecode for the missing classes and load them at run time in the local JVM to allow a correct deserialization. The JVM specification allows for creating a tree-like hierarchy of classloaders to load classes from different sources. When a classloader in the hierarchy is asked for loading a class, it will, as a first step, ask its parent classloader. If the parent classloader cannot find the class, the child classloader tries to load it itself. If also the child classloader fails, a `ClassNotFoundException` is thrown. PRIME exploits such a feature by defining a custom PRIME classloader as child of the standard Java Bootstrap classloader. When loading classes the PRIME classloader delegates to its parent classloader. If the Bootstrap classloader fails, then the bytecode is not available within the local node and should be retrieved remotely. Thus, the PRIME classloader contacts the origin PRIME node asking for the missing bytecode. The origin side retrieves the bytecode from its classpath and sends it back to the requesting node. Now, the local PRIME *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.

As introduced above, PRIME implements an end-to-end strategy to achieve point-to-point code mobility among nodes. Referring to Figure 4.2, let $A$ be a PRIME node sending a message to a PRIME node $B$, and let $\text{CL}_a$, $\text{CL}_b$ be the PRIME classloader of $A$ and $B$, respectively. Whenever $B$ receives a packet, it asks $\text{CL}_b$ for loading the needed class to deserialize the packet. $\text{CL}_b$ asks its parent classloader for loading the needed class. Clearly, if bytecode for the objects in the packet is not available, the bootstrap classloader will throw a `ClassNotFoundException`,

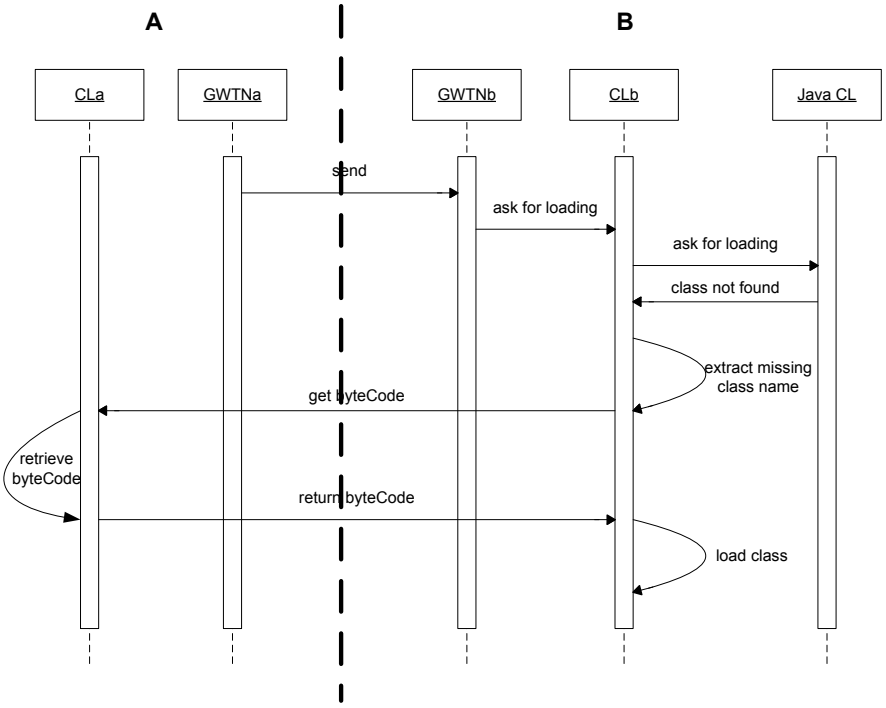


Figure 4.2.: Sequence diagram for point-to-point code mobility

and the control is passed to $\text{CL}_b$. $\text{CL}_b$ extracts the name of the needed class, and asks for it to $\text{CL}_a$. $\text{CL}_a$ processes the request, encapsulates the needed bytecode into a message, and sends it back to $B$. Once the bytecode is available at $\text{CL}_b$, 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 point-to-multipoint communication, this solution is not applicable. In fact, as discussed above, PRIME point-to-multipoint communication relies on the REDS content-based publish/subscribe native protocol. Because of the adoption of such coordination model, message sender and receiver are completely decoupled, and do not have any knowledge about each other. Moreover, applying the end-to-end strategy to point-to-multipoint communication would flood the overlay network and the source node would be overwhelmed by requests for bytecode retrieval. To this extent, PRIME adopts a hop-by-hop strategy, which spreads the bytecode across the overlay towards all the destinations. Since REDS exploits content based routing, when a message is published to a broker, the broker deserializes the message, matches its con-

tent against the stored subscriptions, and forwards the message to the next hop in the overlay towards the matching subscribers. If the message contains unknown bytecode, such a deserialization will fail due to the "missing class" problem. Then, PRIME applies the end-to-end strategy described above at each hop along the path between the publisher and each subscriber. It is worth to note that such a solution is optima since REDS implements a *subscription forwarding* strategy over a tree-based topology [51]. This guarantees messages to be routed through the minimum spanning-tree connecting the publisher to all the subscribers.

## 4.2. The PRIME programming model

Following the P-REST conceptual-model discussed in Section 3.2, the PRIME programming model revolves around two abstractions: (*i*) *resource*, which represents the computational unit, and (*ii*) *container*, which handles both the life-cycle and the provision of resources. Indeed, the PRIME programming model exploits the Scala Actor System [49]. An actor system is a set of actors the interact with each other only through immutable messages, every other interaction is forbidden. Every actor has its own *mailbox* where incoming messages are queued. Messages are dequeued sequentially and processed one by one according to the specific semantics of an actor. Such processing results in at least one of the following events:

1. changes in the internal state of the actor,
2. generation of new messages,
3. creation of new actors.

If a message does not generate at least one pf the previous reactions, it does not affect the computation.

by benefiting from its intrinsic qualities: i.e., functional programming features, event-based computation and concurrency, as well as Java interoperability. Hence, the set of PRIME's abstractions is fully implemented in Scala and exploits the actor model, whereas the *communication layer* is implemented in Java and integrated as external stand-alone library.

Furthermore, PRIME nodes interact with each other by means of message passing, by delegating the effective message delivery to the underlying communication layer. This allows PRIME developers to implement a pervasive application as a set of independent and autonomus network-based actors, which interact through message-passing, irrespectively of

their actual locations (i.e., either local or remote). Indeed, from the developer's perspective, a pervasive application is abstracted as a composition of interacting resources, according to the P-REST conceptual-model (see Figure 3.3).

### 4.2.1. Resources in PRIME

Following the above discussion, the *Resource* abstraction is directly mapped to a Scala actor. In particular, a `Resource` actor is defined as a Scala abstract class that must be extended by any resource to be deployed within PRIME. According to the P-REST conceptual-model, 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 (defined as a set of constants) and, (*iii*) the resource's *Description* that allows the resource to be found by interested parties.

According to the Scala Actor Model, `Resource` implements the `act()` method, which is the method where actors process messages. In this specific case, the `act` method defines a resource's passive behavior. PRIME `Resource` declares the `act()` method as `final` to prevent overriding and then enforcing resources to conform to the REST uniformity principle: `act()` accepts only messages defined by the P-REST uniform interface (see Section 3.2), and handles them by invoking the corresponding methods. Moreover, PUT, DELETE, and GET methods are declared as `final` and implement the well known semantics defined by REST. Whereas, the POST method is declared `abstract` to allow developers to implement their own semantics. Furthermore, according to the Observer pattern defined by P-REST (see Section 3.2), a resource notifies the observers whenever its internal state changes. To this end, when executing PUT, DELETE or POST operations, the resource actor exploits the underlying PRIME *communication layer* to send a point-to-multipoint message notifying the occurred changes.

Still 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 them, a given resource sends request messages to the resources of interest and receives response messages. To this extent, PRIME defines a `workflowEngine` classes to be instantiated by any `Resource` that wants to consume external resources, by exhibiting active behaviors. Indeed, the active behavior is specified by a P*a*CE script (we will introduce P*a*CE in Chapter 5) implementing the composition logic defined by P-REST (see Figure 3.3).

As stated by P-REST, resources interact with each other by exchanging their representations. PRIME provides resource's representation by

serializing the resource instance into a byte array. Indeed, all the fields specifying the internal state of a resource are serialized into the array. However, since a PRIME resource is implemented as a Scala `Actor` class, it is not directly serializable. In fact, a Scala actor maps straightforwardly to Java `Thread` class, which is not serializable as well. To cope with this issue PRIME 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, PRIME defines a special Scala trait, which extends the Java `Externalizable` interface and 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.

On one hand, `writeExternal` 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`[2], 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 resources representation. Indeed, a `Representation` stores the byte array generated by the `writeExternal` method.

### 4.2.2. Containers in PRIME

PRIME handles resource's life-cycle and provisioning through the `Container` actor, which is implemented as a Scala singleton object. This forces every PRIME node to have one and only one container handling the hosted resources. Indeed, the `Container` object stores references to the hosted resources into a *resource repository* built as a map whose keys are the resources' CURIs and values are the corresponding resource instance.

Since a container is an active party in PRIME, it also owns a CURI address, which is used to access 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*) messages without an

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

explicit recipient. 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, messages without an explicit recipient are received by the PRIME node either as result of an active subscription within the publish/subscribe system submitted by a local resource (see Section 4.1), 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. Specifically, `Lookup` messages are broadcast throughout the overlay; `Notify` messages are published by means of the publish/subscribe communication; `Observe` messages are encoded as subscriptions; 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-cycle and provision. In particular, a container creates and moves resources, provides support for resources lookup, as well as grants resource access. While resource access is managed by the resource itself through its interface, creation, relocation and lookup operations are managed by containers.

To create a resource, the container must be provided with information concerning the `Representation` of the resource to be created, and 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 from a container $C_A$ to a container $C_B$, PRIME needs to coordinate the two containers in order to guarantee both the correct deployment of the resource within the container $C_B$, and the delivery of messages to the resource new location (to avoid packet loss). Specifically, $C_A$ locks the resource and buffers all the incoming messages addressed to it. Once the resource is locked, PRIME performs the move operation in three steps: (*i*) $C_A$ waits for the locked resource to consume all the messages in its mailbox; (*ii*) $C_A$ generates a `Representation` for the locked resource; (*iii*) $C_A$ invokes a `Create` operation on $C_B$ by passing both `Representation` and CURI of the `Resource` to be moved; the new resource is created in $C_B$ and kept locked. Once these steps are successfully accomplished, PRIME updates the naming system (for both

point-to-point and point-to-multipoint communication) with the new location for the moved resource and unlock the resource in $C_B$. Finally, $C_A$ removes the resource from its resource repository, and forwards all the buffered messages towards $C_B$. The resource is now able to consume old messages, as well as the new ones that are directly delivered to the new location.

Finally, *lookup* operation is used to query the PRIME 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:

```
lookup:
   (Filter: Description => Boolean, d:Description) => cURI[]
```

`lookup` is a high-order function that, given the `filter` function and the description `d` as parameters, returns the list of CURI matching the provided description. As we have already pointed out in 3.2, the results of the `lookup` operation represent an AURI.

`Lookup` is also used to implement the *resource finder* tool. Such tool issues a lookup request a filter functions the matches every resource to gather the references to all the resources in the network and present them to the user. The latter can decide to install resources on its device by issuing a `GET` on the remote resource and then using the retrieved `Representation` as input for a local `CREATE`.

To complete the description of the `Container` we need to explain how it communicates with the underlying layer described in Section 4.1. To cope with messages flowing from `Container` to REDS, we created a `CommunicationGateway` class that makes the communication functionalities available through a unique and simple interface. `CommunicationGateway` accepts messages coming from the `Container` and prepares them to be processed by REDS. Apart from standard REDS primitives, `CommunicationGateway` also exposes a primitive for sending broadcast messages (the broadcast is implemented by subscribing, by default, every node in the REDS overlay to the "broadcast" topic). Instead, for messages flowing in the opposite direction (from REDS to `Container`), we implemented the `Receiver` class that gathers messages coming from the point-to-point and publish-subscribe facilities and puts them in the `Container`'s mailbox.

As a summary, we present in Figure 4.3 the class diagram for the classes composing the PRIME programming model.

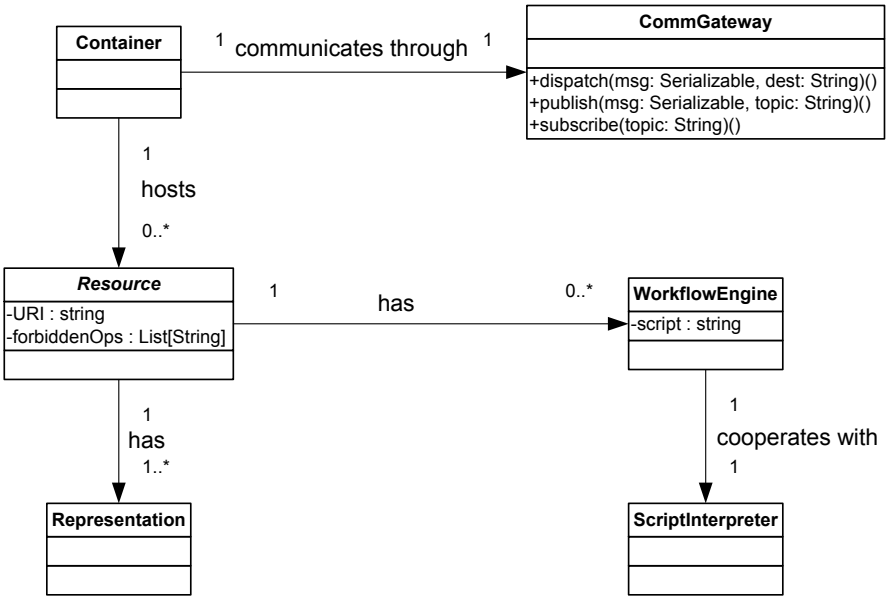


Figure 4.3.: Class diagram for the PRIME programming layer

### 4.2.3. Messages in PRIME

The PRIME programming model is based on the message passing concurrency paradigm. Hence, all of the operations available for `Container`s and `Resource`s are encoded as messages. The latter ones are organized as a class hierarchy.

Referring to Figure 4.4[3], a PRIME message (`PrimeMsg`) is defined as a class implementing the Java `Serializable` interface, and declares two attributes, namely `Destination` and `Payload`. `Payload` is an abstract class defining the `id` attribute only. Since the PRIME communication layer supports asynchronous communication only, as dictated by the actor model, `id` is used to bind requests with the corresponding responses. Further, PRIME defines six types of `Payload`.

`Observe` message is used by a resource to declare its interest in changes occurring in remote resources. The `uri` attribute refers to the cURI identifying the resource of interest. Such a message is handled by the PRIME communication layer as a REDS subscription, where the `uri` attribute is the discriminant used to filter events. On the other side, every resource notifies occurred changes by generating a `Notify` message. A

[3]Note that the syntactical conventions used in figure are the Scala ones (e.g., generics are denoted with square brackets, rather that with angular brackets as in Java).

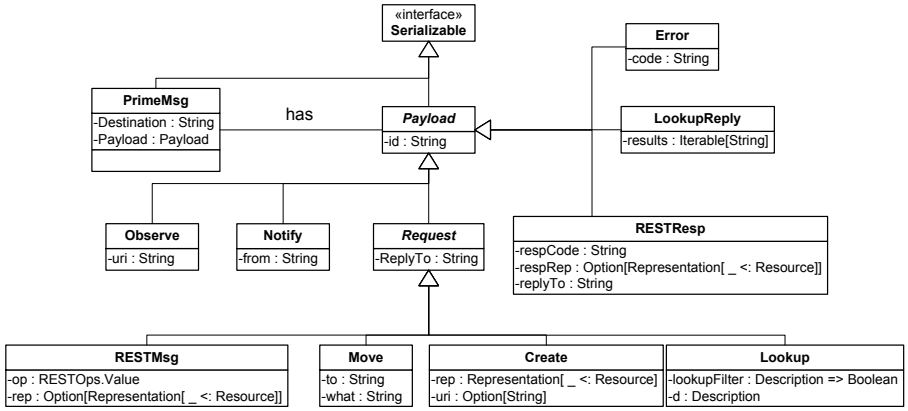


Figure 4.4.: The message hierarchy defined by PRIME interaction protocol

`Notify` message has a `from` field identifying the origin resource CURI. Note that, the `Notify` message does not carry any information about the changes happened in the observed resource. Rather, any observer can decide whether it needs to get an update or not. The `Notify` message is handled by the PRIME communication layer as a REDS notification, which is matched against active subscriptions and delivered to recipients.

The `Request` abstract class represents generic requests exchanged among resources, and defines a `replyTo` field containing the CURI of the origin resource. Indeed, a `Request` can be a `RESTMsg`, a `Move` a `Create`, or `Lookup`. `RESTMsg` encapsulates REST operations (i.e., `GET`, `POST`, `PUT`, and `DELETE`) to be performed on the destination resource. Specifically, the `op` attribute is defined as an enumeration identifying the specific operation to be performed, whereas `rep` carries a resource representation if needed. `Move` message defines two attributes: `to` contains the CURI of the destination location (i.e., a container's address), and `what` contains the CURI of the resource to be moved. `Create` message defines two attributes: `rep` contains the representation used to instantiate the new resource at the given location (container), and `uri` (which is optional) contains the CURI to assign to the new resource. This set of messages is handled by the PRIME communication layer through the point-to-point facility. Finally, the `Lookup` message is used to query the PRIME nodes in the environment to discover resources of interests. As discussed above the `Lookup` message carries the `lookupFilter` and the `Description` to be matched. The above request messages can generate three types of responses: (*i*) `RESTResp` is used to reply to the `RESTMsg` by

possibly carrying a response representation (`respRep`), depending on the specific operation invoked; (*ii*) `LookupReply` is used to reply to `Lookup` request by carrying the results obtained by applying `lookupFilter`; (*iii*) `Error` is used to acknowledge the requester when a REST operation fails. Response messages are dispatched by the PRIME communication layer through the point-to-point facility.

# 5. PaCE

By building the PRIME middleware we, in practice, put in place a distributed actor system built on top of a publish-subscribe system (REDS). Actors are a very good abstraction for dealing with distributed and asynchronous environments. Indeed, they are perfect to implement simple, passive `Resource`s that are reactive elements that must only process incoming messages and answer the sender with the result of a computation. The same argument does not apply for active resources (i.e., the ones with a `CompositionLogic`). The latter ones must send requests and assemble the responses to coordinate several resources. Indeed, the request-response semantics of messages in P-REST poorly fits the actor model. To show how much this issue is relevant we will make use of an example. Let a `Composition Logic` $A$ issue a `GET` request to a `Resource` $B$, if we were using a remote procedure call coordination mechanism the pseudo-code would be:

```
result =
  send(PrimeMessage(URI_B, RESTMsg(GET, None, URI_A, getId)))
```

The call would be synchronous and the execution would suspend until the arrival of the response. The result carried by the response would be put in the result variable and the execution could continue. But the actor model is all about asynchronous communication and lock-free concurrency. To achieve the same behavior we need to adopt a work-around that dramatically diminishes the readability of the code, increases the probability of introducing bugs and increments the development time. In particular, the issue of the request and the handling of the response take place in two separate places. Such solution is sketched in the following:

```
case 1 =>
 send(PrimeMessage(URI_B, RESTMsg(GET, None, URI_A, getId)))
case 2 => . . .
case 3 => . . .
case RESTResp(respCode, rep, replyTo, id) if(id == getID) =>
 process response
case 5 => . . .
```

The `PrimeMessage` is issued towards $B$ in a case branch and the response is handled in a different branch. Such branches can even be in different actors and it would make the problem worse. Moreover, actors do not block, thus the actor that issued the request can process other messages while waiting for the response, even messages of the same type of the response. Therefore, a mechanism to bind responses to their requests must be put in place. We use unique identifiers (the `getId` variable in the example) to solve the last problem.

In summary we have to face a trade-off, on one side we have the advantage of asynchronous message passing and of the shared-nothing concurrency approach of the actor model, on the opposite we have to deal with an increased effort in writing and developing applications in this scenario. To cope with this problem we decided to raise the level of abstraction and design and implement a domain specific language to compose `Resource`s in PRIME that we called P*a*CE (a loose acronym for Prime Composition languagE). P*a*CE, as every coordination language, should be able to coordinate the different tasks that must be carried out to achieve a final goal. Such tasks in P*a*CE can be either third-party `Resource`s queried through PRIME or local functions written in another language (in our case Scala or Java) that can be used in P*a*CE scripts. Besides, we also provided the language interpreter with the possibility to modify programs at run time. Actually, the approach can be used for every actor system but we implemented a prototype development environment tailored for our work.

## 5.1. Background

The growth in complexity and heterogeneity of software systems imposed the need of raising the level of abstraction in order to make the software development process as rigorous as possible. One of the earliest ideas was proposed by Gelernter and Carriero [56]. They advocated that programming consists of

**computational part:** the *tasks* that must be executed to achieve the final goal

**coordination part:** the order in which tasks must be executed to achieve the final goal.

The composition language we envision is a coordination one. The core of the language is concerned with using `URI`s to retrieve `Resource`'s `Representation`s and manipulate and combine them. In practice, an application written using such language, will hinge on the *data* (`Representation`s)

exchanged among composing `Resources`. In the past, a lot of research has been conducted on programming languages that promoted the *data-flows* among instruction to first-class citizens. They are known as *data-flow languages*. The introduction of this alternative architecture was mainly motivated by the inherent unsuitability of the von Neumann's architecture to the massive parallelism due to its global program counter and its shared memory that rapidly become bottlenecks in parallel programs [57]. To this extent, data-flow approach proposed a completely different computational model suitable to parallelism. In the data-flow computational model, a program is represented by a directed graph where nodes represent instruction, while arcs represent the data dependencies between instructions and are unbounded FIFO queues. When all the arcs entering a node (the *firing set* for that node) have data on them, the node becomes a *fireable node*. The instruction represented by a fireable node is actually executed at any time after the node became fireable. The result of the execution is the removal of the first element from every entering arc and the production of a result that must be place on at least one of the outgoing arcs. Then, the node stops executing as long as it is again non-fireable.

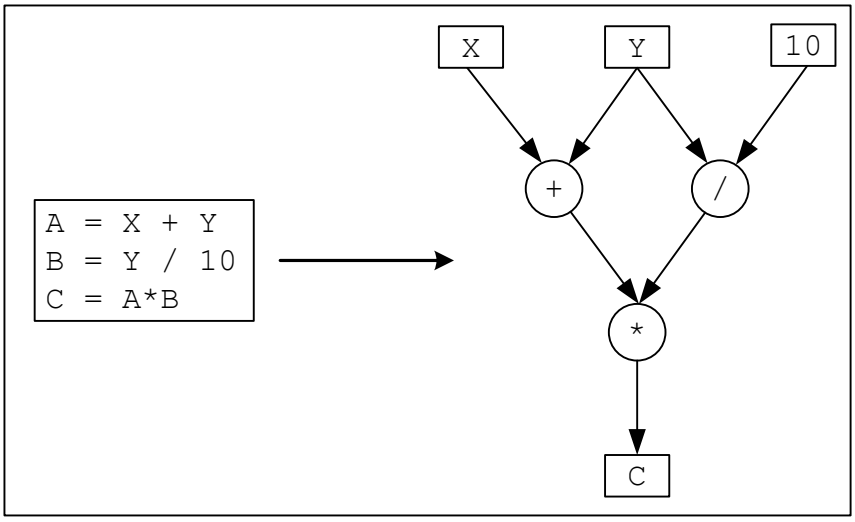


Figure 5.1.: A simple program and its translation into the equivalent data-flow graph

In Figure 5.1 is shown the translation of a simple program (on the left-hand side) into the equivalent data-flow graph (on the right-hand side). Input data — two variables (`X` and `Y`) and a constant (`10`) — are represented at the top of the graph. At the beginning, two nodes are fireable, namely the addition and the division nodes. They can be executed concurrently and their result will be placed on their outgoing arcs. When both have finished, the multiplication node becomes fireable and can produce the result `C`. Under the assumption of having a true parallel hardware, the advantage of using data-flow languages to exploit parallelism is clear. Let alone the speedup, the real gain is the implicit parallelization of the program. In von Neumann architectures, the programmer must explicitly parallelize programs by manually identifying cases where parallelism can be exploited and then by using memory-locking techniques. Rather, in the data-flow architecture, parallelism is implicitly and automatically exploited whenever it is allowed by data dependency. Even from this small example we can notice two important issues. The first one is that instructions don't have an execution order anymore. Indeed, they are executed whenever it is possible. Moreover, all the operators are purely functional since no side effect is allowed. Data are never modified but they are read from the incoming arcs and new data are generated as result and placed on the output arcs. The functional operators along with the ordered queues are enough to guarantee a deterministic behavior for the model, that is, for a given set of inputs, a program always produces the same set of outputs (see [58, 59, 60] for further details).

In [61] Johnston et al. review the state-of-art data-flow languages and build a list of features that are more or less common to all the data-flow languages. Such list comprises:

1. data dependencies equivalent to scheduling,
2. freedom from side-effects,
3. single assignment of variables,
4. an unusual notation for iterations due to features 2 and 3,
5. lack of history sensitivity in procedures.

To allow the data dependencies to induce scheduling (1) the features 2 and 3 are fundamental. Indeed, if variables can either be reassigned during execution or being modified at run time by other routines, the scheduling decided by the compiler at compile time can be invalidated at run time. Moreover, because of the single-assignment rule, the order of

statements is, in general, not relevant. Single assignment conflicts with the imperative style in loops (4). Indeed the classic increment of the loop variable is forbidden. To cope with this problem, loops are implemented through special keywords such as `next` to calculate the value of the loop variable at the following iteration (the Lucid language [62] is the first example where such solution has been adopted). Last, but not least, usually data-flow languages take from the functional languages the lack of history sensitivity for procedures (5) because in a language without deterministic order of execution, histories cannot be univocally built by a programmer. Therefore, operations must have a functional flavor and the result of an invocation must rely only on the input parameters and not on previous invocations.

It is clear that exploiting a data-flow approach to compose and coordinate software components is very appealing especially in distributed environment for two different reasons:

1. the data focus allows for a more natural approach to modeling compositions since, if aided with a visual support, it can also be used and understood by non technical people;

2. the developer must not identify the tasks that can be parallelized since the execution model allows for automatically parallelization.

Pautasso and Alonso exploited the first benefit by proposing the JOpera visual composition language [63] and run-time support [64]. The JOpera language models both data-flow and control-flow dependencies among the tasks in the composition and the development environment is in charge of keeping the two perspectives consistent. The approach does not exploit the data-flow model to implicitly achieve parallelization, rather, the latter is achieved through imperative constructs inserted in the control-flow perspective. In [63] Pautasso and Alonso point out that the data-flow perspective is not enough to model every process because it ignores indirect dependencies (e.g., tasks communicating through databases or configuration files) or because there are dependencies that are not data-related like a compensation handler. In the language we are going to propose from the next section on, the first issue is addressed by adopting a purely functional approach where no side-effects are allowed and thus no indirect dependencies can be introduces, while the compensation handler issue is out of our research scope.

Regarding the second benefit — the implicit parallelization —, a complete general-purpose coordination language has not been proposed. Rather, researchers focused on raising the level of abstraction by proposing languages where nodes in the data-flow graph are functions written in differ-

ent languages. For example Bernini and Mosconi [65] proposed a visual data-flow language called VIPERS where the node in the data-flow graph are Tcl fragments. Also textual approaches exist, like GLU [66] that embeds C fragments in the LUCID [62] data-flow language. These solutions exploit the implicit parallelism and delegate to other languages the whole computation. As a final remark, they are mostly focused on exploiting parallel computers for scientific computations and are not designed to fit distributed environments.

## 5.2. PaCE: Syntax and Semantics

As we have widely discussed previously in this chapter, P*a*CE adopts the data-flow execution model to make parallelism implicit, and thus less problematic, for the application developer. To achieve this goal we followed the state of art and we opted for a purely functional language with a single-assignment policy. Moreover, to keep P*a*CE light and simple we made it dynamically typed, thus `Representation`s' type are only checked at run time.

Last but not least, we decided to design P*a*CE as an interpreted language and not as a compiled one because, as we will see later, we want to be able to modify P*a*CE programs at run time.

### 5.2.1. Syntax

P*a*CE's syntax reminds that of an imperative syntax where every instruction is an assignment except for control structures and for the output mechanism. Since we are in a functional environment, functions can just produce a result starting from their inputs without changing the surrounding environment. So, the only effects produced by the application of a function are contained in the returned result that must be immediately stored in a variable, otherwise the execution of the function has no effects and it results completely useless. As we mentioned before, the only operations that can be used in a non-functional fashion are the output ones. The `OUTPUT` non-terminal allows for using the `write` primitive to print on the standard output and for using ad-hoc functions provided by the user.

As we have already pointed out, P*a*CE and its run-time support, even if can be used as a general-purpose framework, are tailored to P-REST. Therefore, all of the P-REST operations are directly embedded in the language. Thus, they can be found in Figure 5.2, where the generative EBNF for P*a*CE is shown. Though, they are not treated all in the same way.

The P-REST access operations are all derived from the `OP` non-terminal. `OPONEPAR` are invoked with one mandatory parameter – i.e., the list of `URI`s identifying the target resources –, whereas `OPTWOPAR` gets the `URI`s list, and an additional parameter containing a representation. Return values are lists of representations, and depend on the specific operation used: `GET` returns the representation of the target resource; `PUT`, `DELETE` and `POST` return a representation of the *status code* (e.g, "ERROR", "OK", and "NORESPONSE"). All of these operations are designed to work on lists of `URI`s to mimic the AURI mechanism.

Moving on to the other operations introduced by P-REST, two of them (`CREATE` and `LOOKUP`) are treated separately just for intrinsic differences in the syntactical structure, while the `OBSERVE` operation is treated in a completely different way. The `LOOKUP` operation is invoked with only one parameter, that is, the identifier of the function used to filter the `Resource`s, and the value returned is a (possibly empty) list of `URI`s. The `CREATE` operation is invoked by always providing the `URI` of the `Container` where the new `Resource` must be created and the `Representation` that must be used as a basis for the creation. A third parameter can be optionally provided if the caller wants to impose a specific `URI` for the newly-created resource. The return value is the `URI` of the new `Resource` even if the optional parameter is set. The `OBSERVE` operation, as we have already pointed out in 4.2, introduces an event-driven communication model. It is reflected in the syntactic form adopted in P*a*CE. It closely resembles a control structure since it has a body that is enclosed by parentheses and introduced by a keyword. In this case the keyword is `observe` and is followed by a list of `URI`s that must be monitored. The instructions contained in the body of the `observe` structure are executed whenever an event coming from the observed resources arrives.

It is also possible to issue `GET` and `PUT` on two special `URI`s: `stdin` and `stdout`, respectively. The operation `a = GET(stdin)` reads from the standard input a string and store it in the variable `a`. As opposite, the instruction `PUT(stdin, URI)` writes the value denoted by `URI` on the `stdout`.

Call to external functions can be generated by the `ASSGNM` nonterminal. Two kind of functions can be invoked:

Side-effect-free functions that have a functional flavor, return a value and are used to manipulate P*a*CE variables (e.g., to translate the data from one encoding to another) and

State-manipulation functions (SMFUN) that are used to manipulate the internal state of the composite resource and return no value.

The last elements left to examine are the control structures. Some problem could arise from the iteration structure. Indeed, as already mentioned, the single assignment prevents the implementation of standard loops where the loop variable is incremented at every cycle. To cope with this problem we used a higher-level control structure of the form:

**while** `var in` $\texttt{b}_1$ `to` $\texttt{b}_2$ `{ . . . }`

where the value of variable `var` ranges between $\texttt{b}_1$ and $\texttt{b}_2$. In this way we don't need to explicitly update `var` thus the single assignment is not a problem anymore. Two more control structures are provided, the infinite loop (i.e., **while** `(true) {...}`) and the classic conditional structure (i.e., **if** `(cond) {...}` **else** `{...}`).

### 5.2.2. Semantics

Here we want to give an overview on the semantics of P*a*CE. It is important to understand that we are not concerned with defining the operational semantics for the P-REST operations, rather we are concerned with specifying the behavior of P*a*CE as a coordination language.

P*a*CE is inspired by data-flow languages and, thus, it adopts the data-flow execution model. Though, it is implemented on top of an imperative framework and with an imperative syntax and semantics. The most prominent problem is parallelizing, where possible, the sequential code. To this extent, we decided to not completely abandon the sequential execution but to make asynchronous the execution of each instruction. Thus, instructions are still evaluated in order and one by one but the execution does not wait for an instruction to finish before continuing. Since the execution of an instruction is asynchronous the execution does not stop as long as data for executing new instructions are available. Each operation is invoked and returns immediately by yielding a *future* variable. A future variable is a variable that will be eventually filled with the result of a computation. When the value of the variable is accessed, if it has already been filled, the value is normally used, otherwise the computation suspends waiting for the variable to be filled.

This approach, in principle, allows P*a*CE scripts to be executed according to the data-flow execution model, but, practically, the execution of parallel operations cannot be started at the same time but they are executed as soon as the interpreter reaches them. As a consequence, by paying the little penalty of a non-perfect parallelism, the information about

the order of instructions is available at run time for `ScriptInterpreter`. This is of fundamental importance for dealing with I/O and control structures. Indeed, in classic data-flow languages, the order of instructions is completely lost at run time, therefore the input operations are very tricky. Let us use the following example to better explain:

```
a = read()
c = op1(a)
b = read()
```

It is easy to notice that the two reads are independent, thus, according to the data-flow execution model, they can be executed in any order. As a consequence, it is impossible to bind the result coming from a `read` to the right variable because there is no information about the execution order. What happens is explained rigorously through the Petri net in Figure 5.3. The problem lies in that the user can insert both `a` and `b` without any constraint and they can be non-deterministically read by the first or the second `read`

To cope with the problem we rely on the instruction order and on the mutual exclusion for the access to the standard input. In this way, the first `read` is executed and it locks the standard input. When the second `read` should be executed it is suspended as long as the first `read` receives its input. If we hadn't any information about the instruction order, finding a deterministic order among the `read`s would have been impossible. In Figure 5.4 is presented the Petri net that models the correct behavior. In this case the two `read`s are ordered and the place labeled as `stdin lock` ensures the mutually exclusion for the access to the standard input. Moving on to the output mechanism, the same problem does not arise for the `write`. Indeed, it can take place whenever the variable that must be written becomes available. Even if an external function is used instead of the built-in `write`, there is no problem because they should only be a way to use an output means different from the standard one.

So far we have explained how the normal execution flow is calculated in a P*a*CE script, now we must explain how this normal flow can be altered by control structures. We are referring to the loop (i.e., `while`) and to the conditional expression (i.e., `if-else`). Even if it is quite similar, the `observe` structure will be treated separately. Let us start with the conditional structure. As soon as the conditional expression becomes evaluable, it is evaluated and the proper branch is executed. In practice, the execution flow is calculated according to the rules we have explained before with the only addition that every instruction in

both branches of the conditional structure implicitly depends on the evaluation of the conditional structure. In Figure 5.5 a Petri net is used to precisely specify the behavior of the `if-else` control structure. Notice that potential dependencies of the body from previous instructions are not reported in the Petri net for the sake of clarity.

The looping support, instead, is slightly more complicated. Indeed, the concept of loop does not fit well the data-flow paradigm but, as already recognized by Ackerman in [67], having loops, instead of the equivalent tail recursion, is fundamental for the adoption of the paradigm. The problem of loops in data-flow languages is that variables cannot be updated in different iterations. Some common computation, like the following for the factorial of 5, cannot take place:

```
int res = 1
for ( int i = 1; i <= 5; i++){
  res *= i
}
```

In P*a*CE we decided, as a design decision, to not address this problem. Indeed, we conceived P*a*CE as coordination language, therefore no computation should take place in P*a*CE scripts since it should take place in remote services or in external functions. Putting aside this design considerations, we want to explain how the execution flow is calculated in P*a*CE in case of loops. We wanted to keep the `while` structure as close as possible to the imperative flavor. To do so we forced every iteration to happen in isolation, that is, all the instructions in an iteration must be completed before the next iteration can begin and all the variables allocated in one iteration are deallocated at the end of the iteration itself to avoid violating the single-assignment rule. It can be thought as if the end of the body of a loop is a synchronization point for all the instructions executed in the body. Clearly, if any instruction in the loop body has a dependency on data from outside the loop, it must be satisfied before the first iteration can take place. In Figure 5.6 is presented the Petri net that models this approach. To keep the Petri net simple and focus only on the execution flow for loops, we assume here that no dependency is in place neither (*i*)among instructions within the body nor (*ii*) among instructions in the body and previous instructions. The `synch` place forces the net to wait for the completion of all the instructions in the body before allowing the `while` condition to be evaluated again. In case of infinite loop the execution can continue as soon as all the operations in a given iteration finish because there is no condition to be evaluated.

The introduction of the `observe` operation is an attempt to allow event-based programming in P*a*CE. So far we have assumed that the

only coordination mechanism exploited by P*a*CE is the request-response one. In distributed systems computations on a network node can be triggered by *events* happening in another node. That is why we inserted the `observe` operation in the P-REST architectural style (see 3.2). In P*a*CE the `observe` operation accepts as parameter a list of containing the `URI`s of the resources that must be observed. As soon as the input parameter is available, the underlying `OBSERVE` primitive of PRIME is invoked. From this point on, every event that arrives from the observed resources is put on a queue. Them, the queue is emptied and the events are consumed one by one by executing the body of the `observe` structure is executed according the execution model we have outlined before. Notice that an `observe` block is executed in isolation in a separated thread to avoid to lock the entire computation in an infinite loop. Once again, we use a Petri net for specifying the behavior of the control structure under examination (Figure 5.7). In this case, the event generator produces tokens (events) that the `observe` must consume one by one. The place labeled as *event arrived* models the incoming queue.

As for the external functions, side-effect-free and state-manipulation functions must be treated separately. The former ones do not introduce any problem because they are perfectly compliant with the data-flow paradigm.The latter ones (SMFUNs) can conflict with the `POST` operation since it can modify the internal state of the composition. Besides, concurrent SMFUNs can potentially modify the same data. To this extent, on one hand, P*a*CE provides mutual exclusion mechanisms to avoid the simultaneous access to the state of the composite resource. On the other hand, `POST` and SMFUNs should not be used contemporary to avoid unforeseen (and unpredictable) behaviors of the P*a*CE scripts.

## 5.3. PaCE: Interpreter

In the previous section we have outlined the execution model we devised for P*a*CE. In this section we want to explain how the P*a*CE interpreter can execute the scripts and map them to PRIME. The `ScriptInterpreter` class is instantiated by the `WorkFlowEngine` class in PRIME. The `WorkFlowEngine` class acts, then, as a gateway for messages both generated by and addressed to the `ScriptInterpreter`.

First of all, the interpreter is written in Scala and takes advantage of the Scala parser combinator [68] library. The latter implements a backtracking parsing algorithm so it is not very optimized but it was not a main concern for our prototype. Each production of the EBNF is recognized and translated to obtain the desired semantics. The in-

terpretation procedure is a recursive one and uses an auxiliary symbol table that contains all the bindings between the variable names and their values. The data structure we have chosen to implement the symbol table is a slightly modified map. Indeed, it, not only supports the usual operations for maps (random access through keys), but it also maintains the insertion order of the keys. Anyway, the map accepts keys of type `String` (the variable names) and values of type `Future[Any]`[1]. In the following we will use the same outline we adopted for presenting P*a*CE's semantics. Therefore, we will start explaining how the asynchronicity is achieved for standard instructions, then we will focus on the I/O system and, eventually, we will address the control structures.

Before going into more details, a consideration is in order. Although we said that the operations in P*a*CE are executed on lists of parameters, here, for the sake of clarity, we assume that operations are applied on lists with just one element. After all, the generalization is straightforward. An operation on a list of $n$ parameters is equivalent to $n$ parallel operations where every operation accept one parameter of the list. The results, the result of the operation on a list of parameters can be built by juxtaposing the results of the parallel operations.

Since P*a*CE is of functional inspiration, the instructions are all assignments where to a left-hand side (LHS) is assigned a right-hand side (RHS). The LHS is the name of the variable that will contain the value produced by the RHS. Thus, it must be inserted in the symbol table as a key and bound to its value. The way used to obtain the value is key here. In RHS there can be either a P-REST operation (except `OBSERVE`), an external function call, or a constant. Let us examine the first case before. For the P-REST operations to be executed, PRIME must be exploited. To this extent, a worker thread is spawned and it is charged with handling the P-REST request, we called such thread `RequestHandler`. The first action the `RequestHandler` takes is substituting all the variable names in RHS with the corresponding values. Such values can be found in the symbol table. As we have already pointed out, the values in the symbol table are `Future` variables so the actual value might not be there when needed. Thus the `RequestHandler` must suspend and wait for all the needed variables to become available. If the request were handled by the main parsing thread and not by the `RequestHandler`, the execution of the P*a*CE script would have been suspended, hence preventing the following instructions to be executed even if all the data dependencies were satisfied. Once all the variables needed to issue the P-REST request be-

[1]The type `Any` in Scala is the super-type of every other type, including primitive types

come available, the `PrimeMessage` is built and a unique `id`[2] is generated and attached to the `PrimeMessage`. The latter is forwarded to the hosting `WorkFlowEngine` and, in turn, to the hosting `Container`. Now the problem is properly binding the eventual response to the right request, hence to the right LHS. To do so, a receiving actor (implemented by the `RecActor` class) is created and bound to the `id` of the request. Such binding is stored in the `WorkFlowEngine` so that the response will be forwarded to the right `RecActor`. Every `RecActor` is initialized with an output buffer where the response will be eventually placed. Now everything is in place for retrieving the response and binding it to the request. The last problem is updating the symbol table. Obviously, the key that must be added is the LHS. The corresponding value should be a `Future` variable wrapping the response. To obtain such variable we used the Scala *future block*. A future block is a block of code that returns a value. At run time, when a future block is reached, a `Future` variable is immediately returned and the code block is concurrently executed in another thread. When the block is completed, the returned value is automatically inserted in the `Future` variable that was returned in the first place. Going back to the problem at hand, the value that must be inserted in the symbol table is the `Future` variable returned by a future block that waits for the corresponding `RecActor` to put the response on the output buffer.

It is important to notice that, actually, `RecActor` is an abstract class with two specialized subclasses: `GreedyRecActor` and `TimedRecActor`. The latter is used for retrieving responses for the `lookup` operation while the former serves all the other operations. The difference is that in the case of `lookup`, the issuer must accumulate the results coming from an unknown number of `Container`s and present them together, whereas in all the other cases only one response is expected and it is immediately put on the buffer. For this reason, in case of `lookup`, there is no way to know when the `RecActor` can stop and return the results. Thus the `TimedRecActor` is initialized with a timeout. Once the latter expires the results are put on the output buffer and further responses are discarded.

As for the invocation of external functions, the run-time support invokes them in a future block, so the RHS to be put in the symbol table is immediately returned without going through the `RequestHandler`. Nonetheless, the external functions can share among them a state. This departs from the functional paradigm used to invoke the remote P-REST operations. To prevent possible conflicts they are executed in isolation exploiting the information about the execution order and adopting a so-

[2]The `id` is unique for the single `WorkFlowEngine`

lution similar to the one adopted for the aforementioned `read` problem.

Moving on to the I/O mechanism. to enforce the behavior described in Figure5.4 we just needed to entrust the reading mechanism to a specific actor called `InputActor` that wraps the access to the standard input and enqueues all the `read` requests to guarantee the mutual exclusion.

As for control structures, the `if-else` structure is handled by using the equivalent control structure in Scala. The implicit dependency on the condition evaluation is ensured because the values needed for the evaluation must be looked up in the symbol table. The look-up procedure suspends if any value is still not available. Such suspension takes place before the parsing of any branch has begun.

As for the `while` structure, the same mechanism used for the `if-else` is applied to impose the dependency on the condition evaluation. In this case we also need to guarantee the isolation of each iteration. To do so, we exploit the particular properties of the symbol table. Indeed, the latter can be accessed both as a map and as a list. At the beginning of every iteration, we store the value of the highest index in the symbol table. At the end of every iteration we try accessing all the variables with an index higher than the one we stored. All these variables have been allocated in the loop body. In this way synchronization is achieved. Moreover, all the variables allocated in an iteration are removed from the symbol table because they cannot be reassigned in the following iteration due to the single-assignment rule.

To complete the picture, only the `observe` operation is left to cover. It exploits the same mechanism of the `while` structure to execute iterations in isolation (while it does not need the initial synchronization that is both needed for the `while` and `if-else` structures). The difference is that, since the triggering notifications can come from any of the observed resources, most likely the `observe` body will need the `URI` of the resource generating the specific event. To this end, the `ScriptInterpreter` makes available a special variable called `obsURI` containing such `URI`. Notice that every `observe` block has its own `obsURI` variable that cannot be accessed from outside the block. As already mentioned, the body of the `observe` is executed in a separate thread by a third subclass of the `RecActor` class. It is called `ObserveRecActor` and it is initialized with the body of the `observe` structure and, whenever an event arrives, it triggers the parsing and the execution of the body.

### 5.3.1. Support for external functions

External functions are of fundamental importance for P*a*CE since they carry out all the computation that is not related to the P-REST op-

erations. In order to be used, external functions must be made available to the interpreter at run time. To this extent, the developer of a P-RESTful application can define all the functions he needs by adding them to a class that extends the `ExternalFunctions` abstract class. The latter has all the machinery needed to parse, validate and make available all the functions defined in its subclasses. To achieve its goal, the `ExternalFunctions` class makes heavy use of the Java reflection. Every function is parsed, its name extracted, its return type validated and its body made invokable by the PaCE interpreter. The name of the function is added to a special symbol table called `funSymTable` contained in the `ExternalFunctions` class. It is a hashmap that maps the names to the body of the functions. Hence, its type is `HashMap[String, Method]`.

The `ExternalFunctions` class exposes a method to invoke the functions it wraps. This method has following signature:

```
invoke (methName: String, pars: Any*): Future[Option[Any]]
```

The `invoke` method accepts the name of the functions that must be invoked and a list (possibly empty) of parameters[3]. The method executes the function denoted by `methName` in a future block, so that the invocation is, again, asynchronous. Clearly, the invoked function can have no returning type so the complete return type of the `invoke` function is `Future[Option[Any]]`. Besides, the `invoke` method prevents conflicts among SMFUNs by executing them serially. In this way the asynchronous execution of the PaCE script is not hindered and the safety of the execution is guaranteed.

As a final remark, it is important to point out that the same solution used for external functions can be applied to the case of external variables. The external variables are added to the symbol table when the `ScriptInterpreter` is initialized along with the `container` variable that is available by default and contains the `URI` of the local `Container`.

## 5.4. Run-time Adaptation in PaCE

As we mentioned in 3.3, one of the desiderata for P-RESTful application is to be adaptable and evolvable at run time. Always in 3.3, we reported that it is commonly accepted that an application must support 4 operations on its component and connectors to achieve adaptability and evolvability. Such operations are:

[3] the notation `Any*` is the Scala notation for the Java varargs.

- Component addition
- Component removal
- Component substitution
- Connector rewiring

As a first step, we need to understand what these operations mean in P*a*CE. Component addition and removal refer to the possibility to add or remove a `URI` from a specific list. Connector rewiring means changing the binding between the name and the corresponding value for a variable containing a `URI`. Component substitution is like a connector rewiring except that the state of the older resource must be transferred to the new one.

For the `ScriptInterpreter` to be able to execute one of these operations, it must suspend the execution of the script to avoid inconsistencies since the symbol table must be manipulated. All the requests for reconfiguration are evaluated asynchronously, that is they are queued and, whenever the `ScriptInterpreter` is in a safe state, they are executed. The interpretation procedure reaches a safe state whenever it recursively tries to evaluate a new non-terminal because the symbol table is not accessed by any other thread. Therefore, the `ScriptInterpreter` accesses the list of reconfiguration actions and carries them out one by one.

The component addition and removal are easily addressed. They are both handled in their dedicated functions whose signature is:

```
def add(what: String, to: String)
def remove(what: String, from: String)
```

Firs of all, the `what` parameter is examined. It can either be a variable name (if so, it must be present in the symbol table) or directly an `URI`. In the former case the actual value must be retrieved from the symbol table, in the latter the variable can be used as is. Then, the list denoted by the name in `to` (`from`, respectively) is extracted from the symbol table. Now, the new `URI` must be added (or removed if it exists) from the list just retrieved. As a last step, the symbol table must be updated and addition (removal) is finished.

As for the connector rewiring the signature of the handling function is:

```
def rewire(what: String, with: String)
```

In this case the `what` parameter is always a variable name, so it must be looked up in the symbol table. The `with` parameter, instead, is always a `URI`. The symbol table must be updated by discarding the old binding for the `what` variable in favor of the new binding with the `with` variable.

The component substitution is handled by its own function with the following signature:

```
def substitute(what: String, with: String)
```

The only difference with respect to the `rewire` function is that, not only the the symbol table is updated, but also a `GET` is issued towards the old `URI` and the response is used to issue a `PUT` towards the new `URI`. In this way the state of the old resource is transferred to the new one.

A final remark is in order for the case of adaptations that involve a variable containing the `URI`s of an observe. Indeed the variable of the `observe` structure is used only once, that is, at the beginning when the subscription towards the remote resources are generated. Thus, every adaptation operation must be examined carefully:

**add:** in this case the intended semantics is adding a new resource to the pool of the already observed resources. Thus, whenever an add is requested, a new subscription must be made in order to start following also the new resource. Furthermore, to make the script catch immediately up with the change, the body of the `observe` is executed by using the added `URI` as `obsURI`;

**remove:** the intended semantics is the opposite of the add operation. It means that one resource must not be followed anymore. Thus the corresponding subscription in the publish/subscribe system must be removed;

**rewire:** the expected behavior is dropping the old resource pool in favor of a new one. This operation is implemented by executing the `remove` of every `URI` in the old list followed by the `add` of every `URI` in the new list. The series of `add` operations impose the execution of the `observe` body with every new variable used as `obsURI`;

**substitute:** this operation, if applied to the variable containing the observed resources does not affect the behavior of the `observe` block.

Notice that if the variable containing the list of observed resources appear in other places in the script, for those cases are applied the standard rules for adaptation described above.

```
%
% ID          = ([A-Za-z] | [0-9])+
% INTEGER     = [1-9]([0-9])*
% URI         = ID
% OP          = ‘get’ | ‘put’ | ‘delete’ | ‘post’
% STATEMENTS  = STATEMENT*
% BLOCK       = ‘{’ STATEMENTS ‘}’
% STATEMENT   = LOOP | BLOCK | ASSGNM | OBSERVE | CREATE | OUTPUT
%             | INFLOOP | IF | WRITE | LOOKUP
% OBSERVE     = ‘observe(’ URI ‘)’ BLOCK
% CREATE      = ID ‘= create(’ URI ‘,’ ID ‘)’
%             | ID ‘= create(’ URI ‘,’ ID ‘,’ URI ‘)’
% LOOKUP      = ID ‘=’ ‘lookup(’ ID‘)’
% ASSGNM      = ID ‘=’ OP ‘(’ URI (‘,’ ID)? ‘)’
%             | ID ‘=’ ID ‘(’ (STRING)? (‘,’ STRING)* ‘)’
%             | ID ‘= read()’
% OUTPUT      = ID ‘(’ (STRING)? (‘,’ STRING)* ‘)’
%             | ‘write(’ID‘)’
%             | ‘write(’STRING‘)’
% LOOP        = ‘while’ ID ‘in’ INTEGER ‘to’ INTEGER BLOCK
% INFLOOP     = ‘while (true)’ BLOCK
% IF          = ‘if (’BOOLEXP‘)’ BLOCK ‘else’ BLOCK
% BOOLEXP     = BOOLEXP
%             ( ‘&&’ | ‘||’ | ‘<’ | ‘>’| ‘<=’ | ‘>=’ | ‘==’ )
%             BOOLEXP
%             | ID | INTEGER | STRING |‘!’ BOOLEXP
%             | ‘(’ BOOLEXP ‘)’| ‘true’| ‘false’

ID          = [A-Za-z]([A-Za-z] | [0-9])*
INTEGER     = [1-9]([0-9])*
STRING      = "([A-Za-z] | [0-9])*"
URI         = ID+ | STRING+
OP          = OPTWOPAR ‘(’ URI ‘,’ ID ‘)’
            | OPONENOPAR ‘(’ URI ‘)’
OPTWOPAR    = ‘post’ | ‘put’
OPONEPAR    = ‘get’ | ‘delete’
BLOCK       = ‘’ STATEMENT ‘’
STATEMENT   = LOOP | ASSGNM | OBSERVE | CREATE | OUTPUT
            | INFLOOP | IF | LOOKUP
            | STATEMENT+
OBSERVE     = ‘observe(’ URI ‘)’ BLOCK
CREATE      = ID ‘=’ ‘create(’ URI ‘,’ ID ‘)’
            | ID ‘=’ ‘create(’ URI ‘,’ ID ‘,’ URI ‘)’
LOOKUP      = ID ‘=’ ‘lookup(’ ID ‘)’
ASSGNM      = ID ‘=’ OP
            | ID ‘=’ ID ‘(’ (STRING)? (‘,’ STRING)* ‘)’
    | ID ‘(’ (STRING)? (‘,’ STRING)* ‘)’
            | ID ‘=’ ‘read()’

OUTPUT      = ID ‘(’ (STRING)? (‘,’ STRING)* ‘)’
            | ‘write(’ ‘stdout’ ‘,’ ID ‘)’
            | ‘write(’ ‘stdout’ ‘,’ STRING ‘)’
LOOP        = ‘while’ ID ‘in’ ID ‘to’ ID BLOCK
INFLOOP     = ‘while’ ‘(’ ‘true’ ‘)’ BLOCK
IF          = ‘if’ ‘(’ BOOLEXP ‘)’ BLOCK ‘else’ BLOCK
BOOLEXP     = BOOLEXP
```

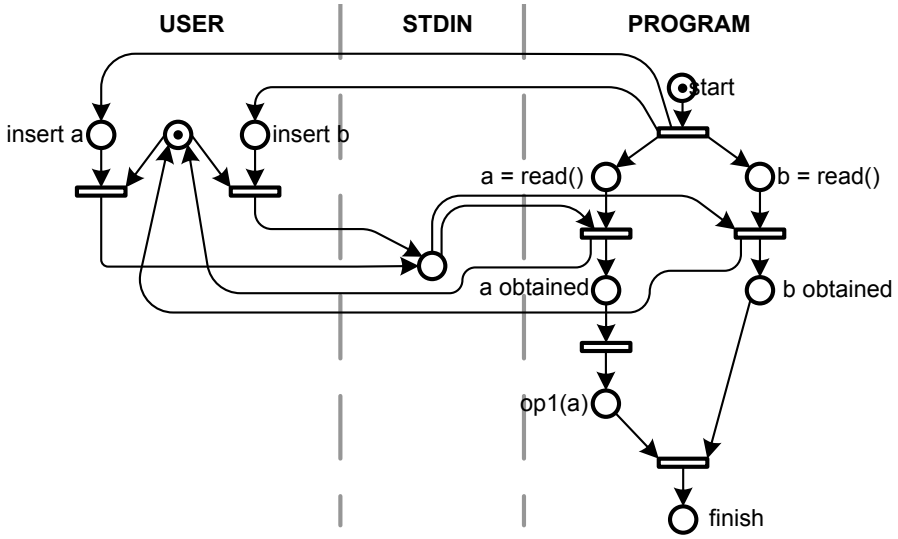


Figure 5.3.: The behavior of two independent `read`s with a pure data-flow execution model

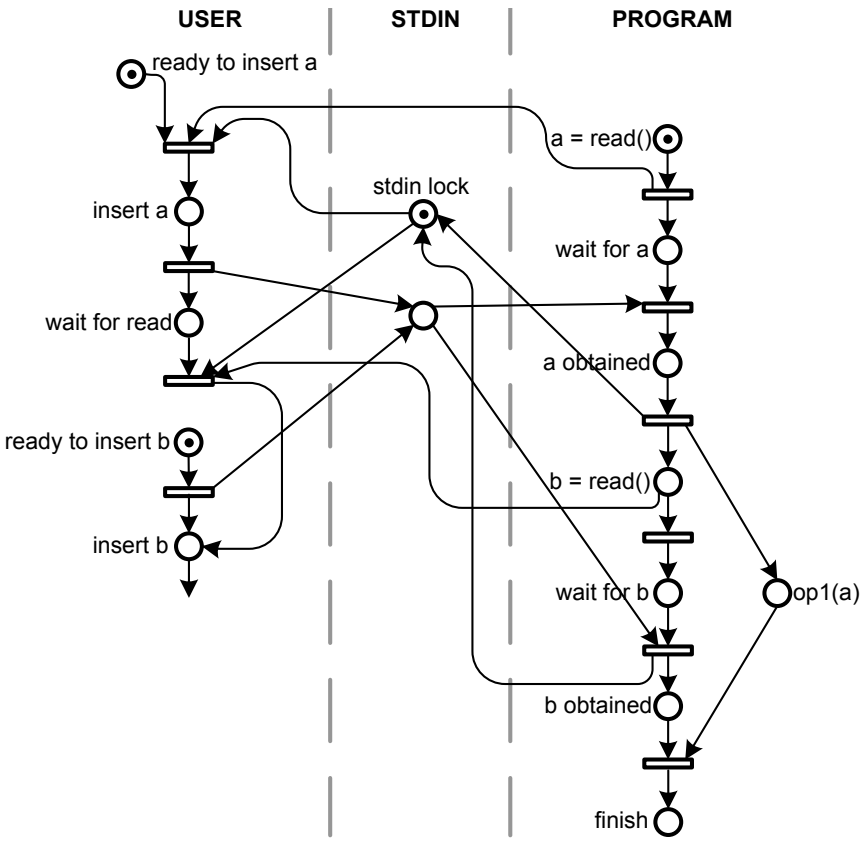


Figure 5.4.: The behavior of two independent `read`s in P*a*CE execution model

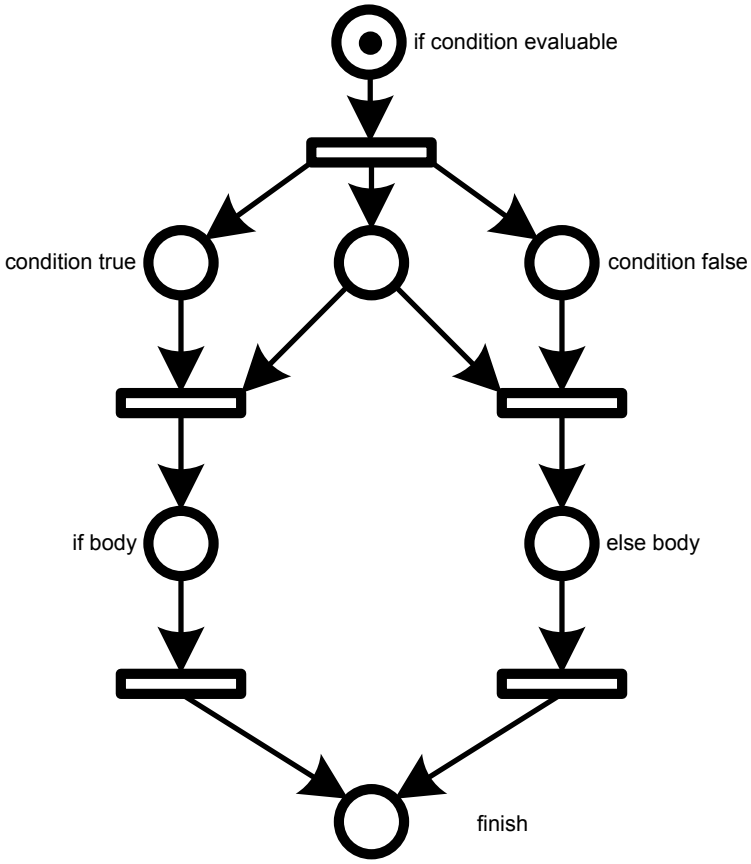


Figure 5.5.: The behavior of the `if` control structure

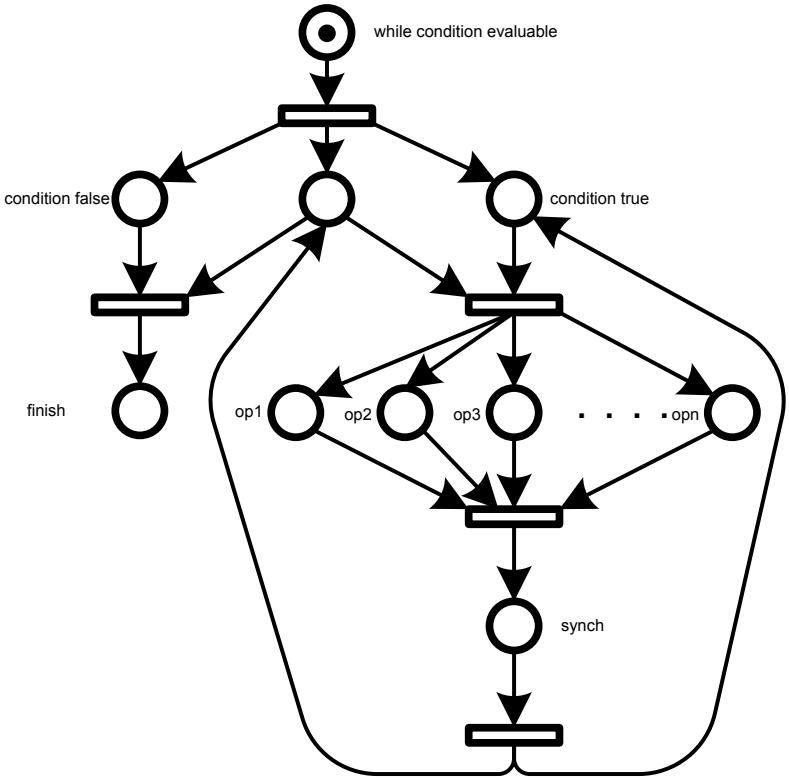


Figure 5.6.: The behavior of the `while` control structure

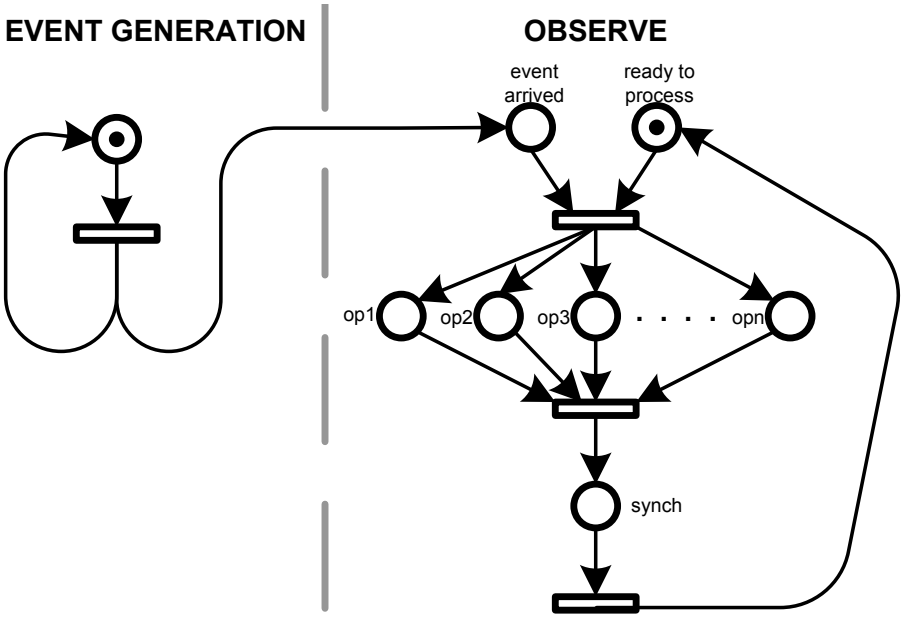


Figure 5.7.: The behavior of the `observe` control structure

# 6. PRIME in action

To assess the effectiveness of PRIME, this Chapter describes a use case scenario and its implementation, to show how PRIME capabilities are exploited to develop a pervasive application.

## 6.1. The Pervasive Slideshow scenario: description

Alice, a university full professor, is going to give a seminar about her recent research activity. Alice enters the conference room carrying her laptop, where she stores both the slides she is going to present and the related handout. 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, PDA), which can be used for displaying either the slide currently projected on the screen or the handout. We want the audience to be on pace with the presentation. It means that the audience and the speaker must refer to the same slide and to the same page of the handout. As a default, the audience will be following the slides of the presentation. Should someone in the audience switch from the slides to the handout, he should be able to do so at run time without reinstalling or restarting the presentation software. Note that, all the devices mentioned above – i.e., Alice's laptop, smart-screen and audience's devices – are supposed to have a PRIME instance deployed and running.

## 6.2. The Pervasive Slideshow scenario: implementation

The *Pervasive Slide Show* application conforms to the P-REST conceptual-model. The resources involved are the following:

**Presentation** and **Handout** are the resources that actually own the data, that is all the slides of the presentation and all the pages of the handout. The presentation and the handout are modeled as lists of

images that Alice can browse by issuing the `forward` or `backward` commands. These interactions are handled through a PaCE script deployed on their `WorkflowEngine`. Whenever Alice wants to move through the presentation (or the handout), the `Presentation` (`Handout`) updates the pointer to the current slide accordingly and propagate the change to whole environment. Notice that the updates on `Presentation` are also propagated to the projector, while the updates on the `Handout` are not.

**CurrentSlide** and **CurrentPage** resources encapsulate the image corresponding to the current slide or page. They are meant to be observed by all the components of the audience to spread the updates coming from `Presentation` and `Handout` resources. They also serve as a decoupling mechanism between the `Presentation` and `Handout` resource, on one hand, and the audience, on the other. If such mechanism were not in place, according to the statelessness principle of REST, the whole state of `Presentation` and `Handout` should be transferred to the audience at every update, by introducing an unnecessary traffic overhead on the network. Both `CurrentSlide` and `CurrentPage` are passive resources where only the `GET` and `PUT` operations are enabled. The `GET` is used by the audience, while the `PUT` operation are used by `Presentation` and `Handout`, respectively, to manage updates.

**Projector** resource is deployed on the smart-screen and is a purely passive resource with the only `PUT` operation enabled. Whenever a `PUT` is issued towards the `Projector` resource, the enclosed representation is interpreted as an image and rendered on the screen.

**Reader** resource is installed at run time on the audience's devices and is used to render the slideshow or the handout on such devices. It is configured to follow, by default, the slideshow by observing `CurrentSlide`. It is also possible, for the audience, to toggle between the slideshow and the handout. It is a purely active resource, thus no operation is enabled.

So far, we gave the big picture and we presented all the actors involved. In the following, we will go in more details and present and explain the PaCE scripts used.

We start with the script for `Presentation` and `Handout`. Since `Presentation` and `Handout` are very similar, in Figure 6.1 we present the script for `Presentation` only. The first instruction is a `lookup` needed to retrieve the `URI` of a `Projector` resource. The `projSearch` parameter is an external function that is designed to recognize the `Projector` resource. Then, the script does not wait for the `lookup` to yield its results, and immediately enters an infinite loop. The latter handles the interactions with the speaker. A `read` is issued to decide whether the slideshow must

```
proj = lookup(projSearch)
while (true){
  cmd = read()
  if (cmd == ‘‘fwd’’){
    rep = getNextSlide()
    PUT(currSlide, rep)
    PUT(projector, rep)
  }
  if (cmd == ‘‘bwd’’){
    rep = getPreviousSlide()
    PUT(currSlide, rep)
    PUT(projector, rep)
  }
}
```

Figure 6.1.: PaCE script for `Presentation` resource

move forward or backward. If a `forward` command is inserted by the user, the script resorts to the `getNextSlide` external functions to obtain the representation of the next slide. It is an example in which external functions have a shared state, indeed, the external function extracts the binary representation of the next slide and updates the pointer to the current slide. Now, if the lookup has already yielded its results (if not the script just suspends), the script issues a `PUT` towards both `Projector` and `CurrentSlide`[1]. Two considerations are in order here. Firstly `currSlide`, the `URI` of `CurrentSlide`, is provided as an external variable since the owner of the presentation also owns `CurrentSlide` and knows its `URI`. The second consideration, instead, concerns the `lookup`. The latter can return a list of `URI`. In this case we decided to project the presentation on every projecting device found in the overlay, so the `PUT` is issued towards an AURI, that is, all the results coming out of `lookup`. If we wanted to filter the `lookup` result, we should have used an external function of apply the filter. As a final remark, the `Handout` script is almost identical. The only difference is that the handout must not be projected, therefore the `PUT` is issued towards `CurrentPage` only.

Moving on, we examine the script for the `Reader` resource (see Figure 6.2). It uses two external variables, namely the `URI`s of `CurrentSlide` and `CurrentPage`. Such `URI`s can be directly embedded in the script because `Reader` is developed and deployed by the same developer of

[1] The case of a backward command is exactly the same, except from invoking the `getPreviousSlide` function instead of the `getNextSlide` function

```
observe(slURI){
  slide = GET(obsURI)
  view(slide)
}
while(true){
  cmd = read()
  if(cmd == ‘‘ho’’)
    sub(obsURI, hoURI)
  if(cmd == ‘‘pres’’)
    sub(obsURI, presURI)
}
```

Figure 6.2.: PaCE script for `Reader` resource

`CurrentSlide` and `CurrentPage`. The script starts with an `observe`. As stated in Chapter 5, the body of the `observe` is entrusted to an independent thread, which is executed every time the observed resources send out a notification. In this specific case, every time an event is generated by `CurrentSlide`, the script issues a `GET` to retrieve the representation of the new slide. When the representation arrives it is displayed through the `view` external function. The latter is allowed for not returning any value because it is an output function. The second part of the script is an infinite loop that handles the interaction with a user in the audience. Indeed, the loop allows for toggling between the slideshow and the handout. The toggling operation is implemented through the `rew` external function. It is just a wrapper for the `rewire` operation exposed by the `ScriptInterpreter` described in 5.4. As a consequence, the binding for the `obsURI` variable is changed and, since the change involves an observed `URI`, a new evaluation of the body of the `observe` is forced. The final result is that now the `Reader` is showing the handout and not the slideshow anymore.

Figure 6.3 gives a comprehensive picture of slideshow case. For the sake of clarity, we left out the resources concerning the handout. The `Presentation`'s `workflowEngine` broadcasts a `Lookup` message searching for a resource able to render the slideshow. As a result, it obtains the `Projector`'s CURI that, in our example, matches the lookup request. Once obtained the `Projector`'s CURI, Alice can start the slideshow. To this end, `Presentation` sends a `PUT` message, containing the representation of the first slide, to `Projector`, and then creates the `CurrentSlide` resource also initialized with the representation of the first slide.

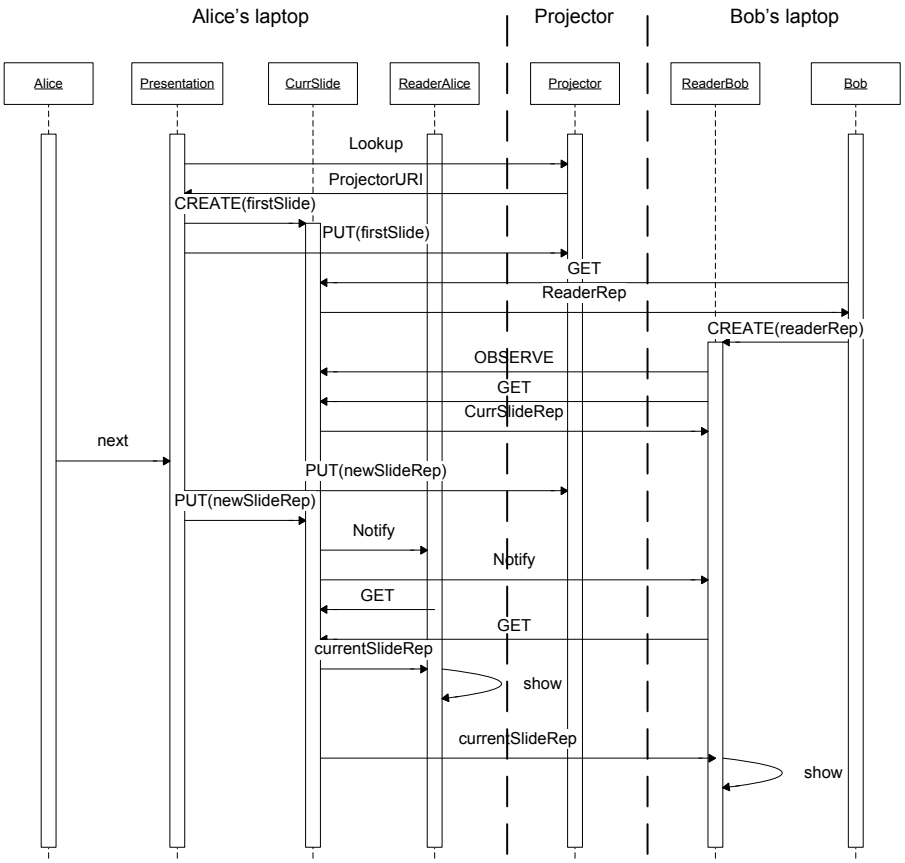


Figure 6.3.: Sequence diagram for our scenario

When entering the conference room, Bob uses the PRIME resource finder built-in tool, which lists all the resources available within the overlay network, to explore the environment and find the `Reader` resource. Hence, selecting `Reader` from the list, the PRIME node issues a `GET` operation to retrieve a representation of `Reader`, which, in turn, is used to create the `ReaderBob` resource. Once this resource is created it performs two actions: (*i*) it gets the state of `CurrentSlide` to initialize itself, and (*ii*) it declares interest on observing the `CurrentSlide` resource (i.e., `OBSERVE` message).

When Alice needs to show the next slide of her presentation, she issues a `forward` command that is handled by the `Presentation`'s `workflowEngine` by performing a `PUT` operation on both `Projector` and `CurrSlide`. Modifying the `Projector` resource causes the projected slide to change, while

modifying `CurrSlide` generates notifications towards all the resources that are observing `CurrSlide`. Then, `ReaderBob` receives such a notification and retrieves the new `CurrentSlide` representation, which is then visualized on his devices.

## 6.3. Evaluation

This section describes the quantitative analysis carried out to assess the PRIME scalability. We set up two experiments: the first one to assess the scalability of PRIME with respect to the pervasive slideshow scenario described in Section 6.1, the second one to assess the scalability of the code mobility facilities we implemented in REDS.

### 6.3.1. prime scalability experiments

The analysis aims at demonstrating that PRIME performance, expressed in terms of memory footprint and CPU load, is compatible with the characteristics of handheld devices available nowadays on the market – i.e., memory: 1GB RAM, CPU: Dual-core 1.2GHz. To this extent, we set up a virtual machine, configured according to the above parameters, and run two different experiments: the first one aims at evaluating how a PRIME instance handles a high number of concurrent requests,the second one at evaluating how PRIME instance, handling a large number of resources, impacts device performance.

Figure 6.4 presents the performance of Alice's device when faced with requests from 100 remote nodes. Specifically, plotted data has been gathered by profiling the PRIME node hosted on Alice's laptop. *Region 1* (from 0 to 13 seconds) shows the computational burden induced by the deployment of the PRIME `Container`, as well as of the `Presentation` and `CurrSlide` resources. The memory footprint is stable around 3.0%-4.0%, which is mostly due to the need of loading images in memory. The CPU load is very low because all the resources are idle. *Region 2* (from 13 to 117 seconds) corresponds to the creation of the nodes, and the installations of the `PresReader`s. The memory footprint presents a fluctuation due to garbage collection and reaches a maximum of 10%. The CPU load, on average, is lower than 10%. In *Region 3* (from 117 to 142 seconds), both memory footprint and CPU Load are stable around 6% and 5% respectively, due to absence of activity. In *Region 4* (from 143 to the end) the memory footprint increases to 6.5% and the CPU Load has a spike reaching about 95%, as consequence of the remote updates. The sensible decrease of memory footprint, around 1.5%, at second 156 is due to the JVM garbage collector. This suggests the adoption of an

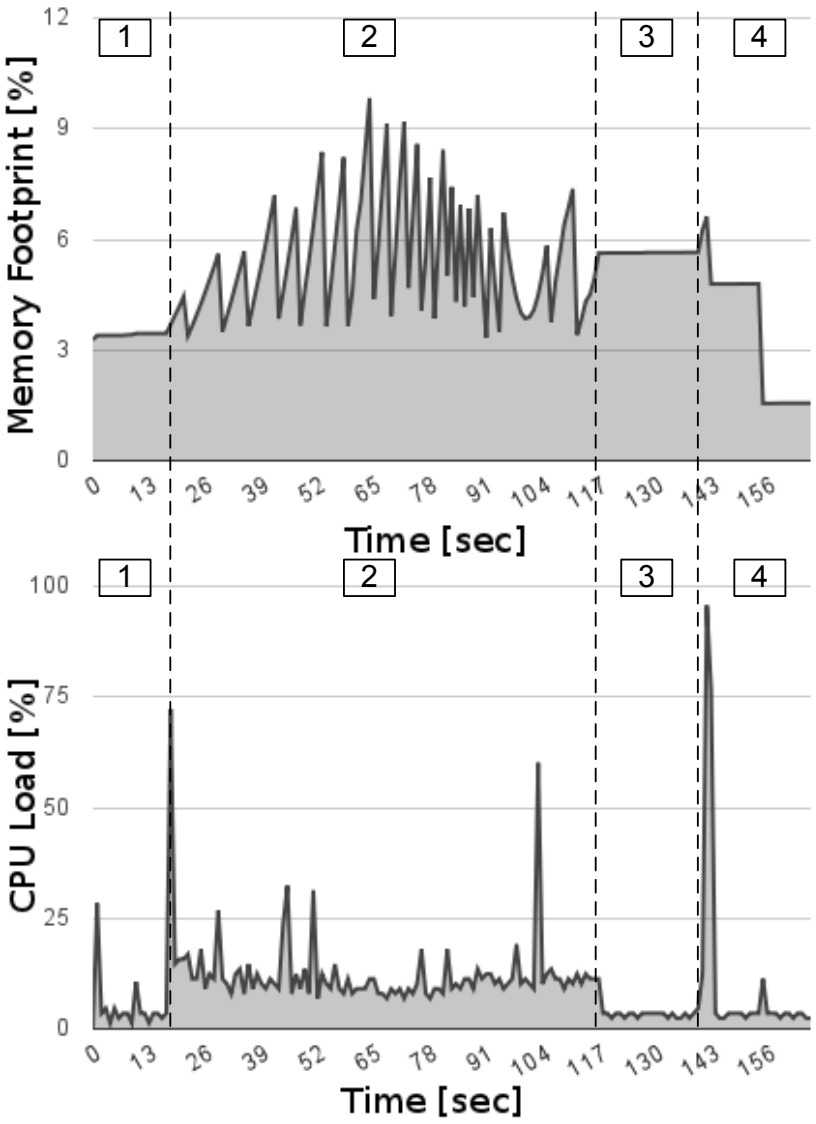


Figure 6.4.: Concurrent requests

aggressive garbage collection policy. Indeed, an increased rate of the garbage collector invocations sensibly decreases the memory footprint of PRIME, at the expense of a slight increase in CPU load.

Figure 6.5 presents the results obtained in terms of memory footprint and CPU load required by a container to handle 100 resources. Specifically, plotted data has been gathered by profiling the PRIME node con-

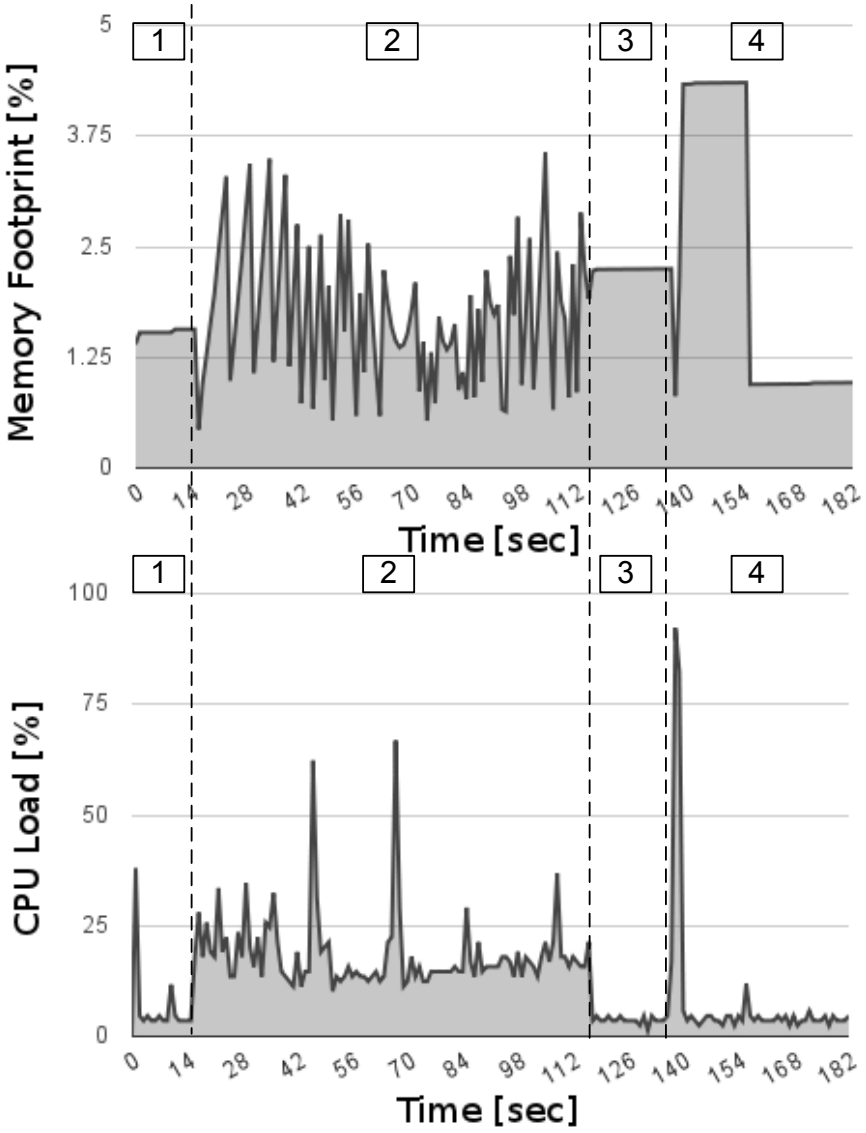


Figure 6.5.: Resource management

taining the 100 attendees at once during the execution of the *Pervasive Slide Show*. Note that, the `Presentation` resource is deployed at a different node. Looking at Figure 6.5, we can notice that the trend of gathered data follows the same pattern observed in the first experiment. Moreover, the memory footprint never exceeds the 4.5%, and CPU Load is, on average, around 13%.

On the basis of such analysis, we can then conclude that PRIME is compatible with the characteristics of handheld devices available nowadays on the market.

### 6.3.2. prime code mobility experiments

The experiment we want to show here is aimed at assessing the performance of the code mobility facility we implemented in REDS. In particular, we are interested in evaluating the overhead introduced by moving bytecode among nodes. To do so we built several overlays with a growing number of nodes and a tree topology, we subscribed all the node except the root to the "broadcast" topic, and, then, we measured the times that took for a packet published by the root on the "broadcast" topic to be delivered to all the nodes in the tree. The packet contained a class weighting about 991Kb that, at the startup, was only deployed on the root node. We sent the packet twice, so that the first time the missing class must be transferred to all the receiving nodes, while the second time the class is already available and only the packet itself must be relayed. To give more relevance to the experiments we iterated each of them 5 times and the values reported here are average values.

The first topology we tried is a linear one. We started with a topology with only two nodes (a sender and a receiver) and we arrived to a linear topology with ten node (one sender and nine receiver).

In Figure 6.6 results are reported. The upper diagram depicts the transmission time for the packet in our experiment. The dashed line refers to the time needed to transfer the packet with the missing class from the root, where it is published, to the only leaf of the linear tree. It means that the time needed to transfer 9 times the missing bytecode is close to 3 seconds. While the transfer penalty is not negligible, it is important to notice that, once the class is disseminated in the overlay, no more penalty is introduced. Indeed, the solid line refers to the time needed to deliver the same message for the second time, when the missing class is already present in the whole overlay. In the lower diagram is depicted the percentage overhead imposed by the need of transferring the missing bytecode. It is the percentage of the total transfer time used to move bytecode. In this case, the overhead is very high and steadily over the 90%.

Moving to the next run of the experiment, we used a complete binary tree with varying depth. We started with a depth of 2 (hence with 3 nodes) and the we added a full level for every new run of the experiment up to a depth of 5. Thus the sequence is 3,7,15,31. Figure 6.7 shows how the changed topology directly impacts the difference between the

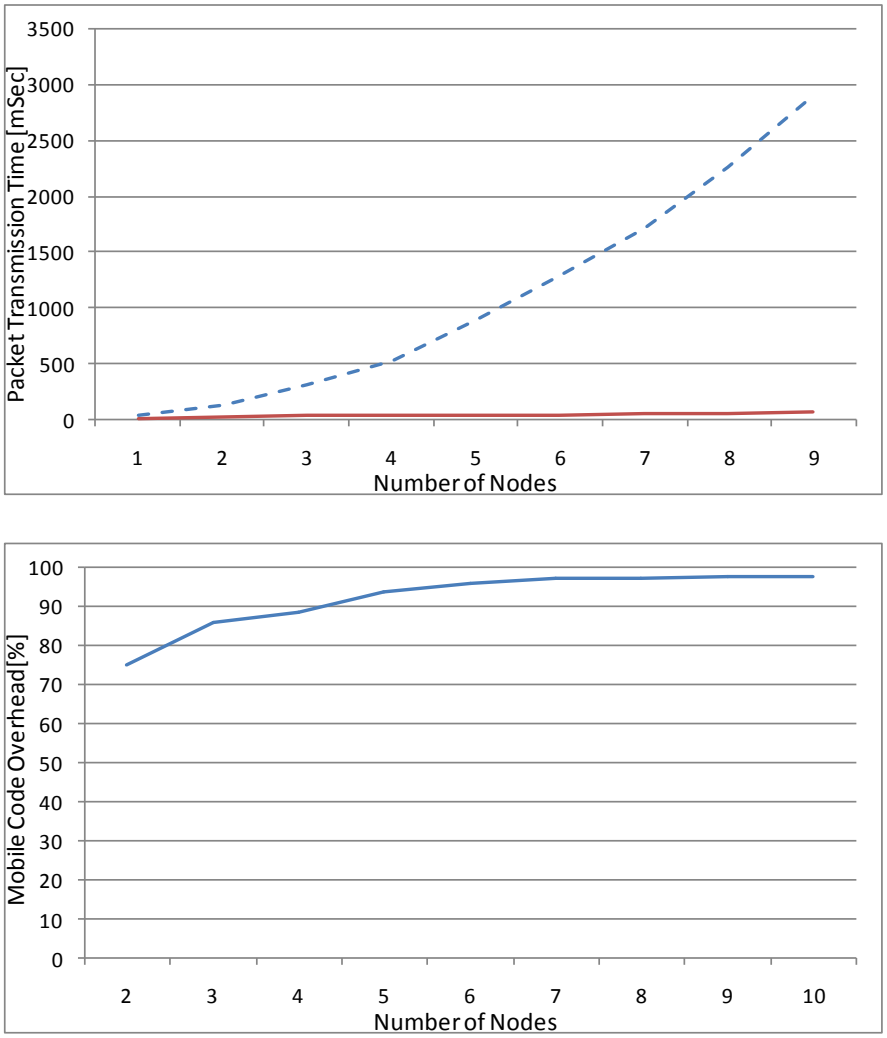


Figure 6.6.: Experiments with linear topology

transfer time with and without bytecode transfer. Both with and without bytecode moving the transfer time grows exponentially with respect to the number of nodes. Though, it is very interesting the results rendered in the overhead diagram. Indeed, when the number of nodes grows the percentage of time dedicated to bytecode transfer becomes more and more negligible with respect to the total transfer time. Indeed, in this case, we passed from values over 90% for 15 nodes to values under 20% for 31 nodes.

The last experiment has been conducted with a complete ternary tree

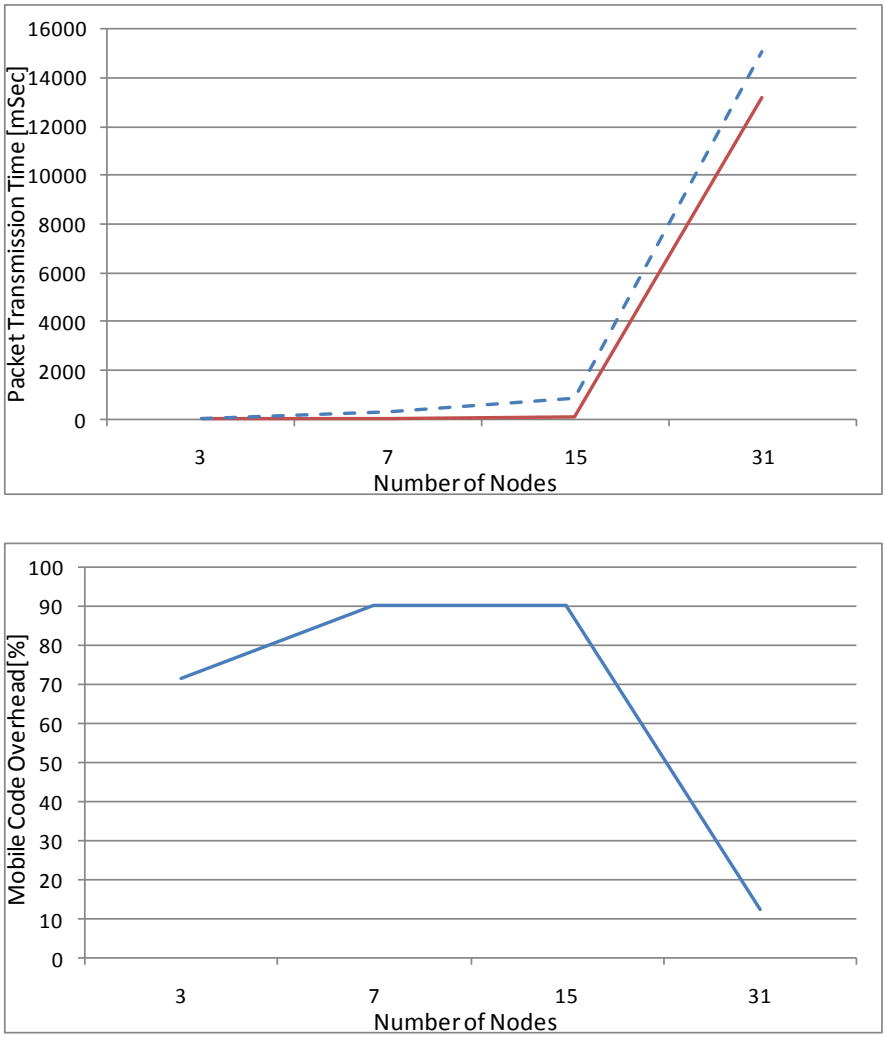


Figure 6.7.: Experiments with a binary complete tree topology

topology with a depth ranging from 2 to 4. The max number of nodes involved in this case is 40 and the sequence is 4,13,40.

Also in this case the highest arrival time grows sensibly with the number of nodes (see Figure 6.8). Still, it is interesting to notice that in the experiments with 40 nodes, the time taken to deliver the packet for the first time is slightly lower than the time needed for the second time. It just means that the first set of runs was "luckier" then the second one and the overhead for bytecode mobility is smaller than the normal fluctuation of REDS performance. The overhead diagram confirms the trend

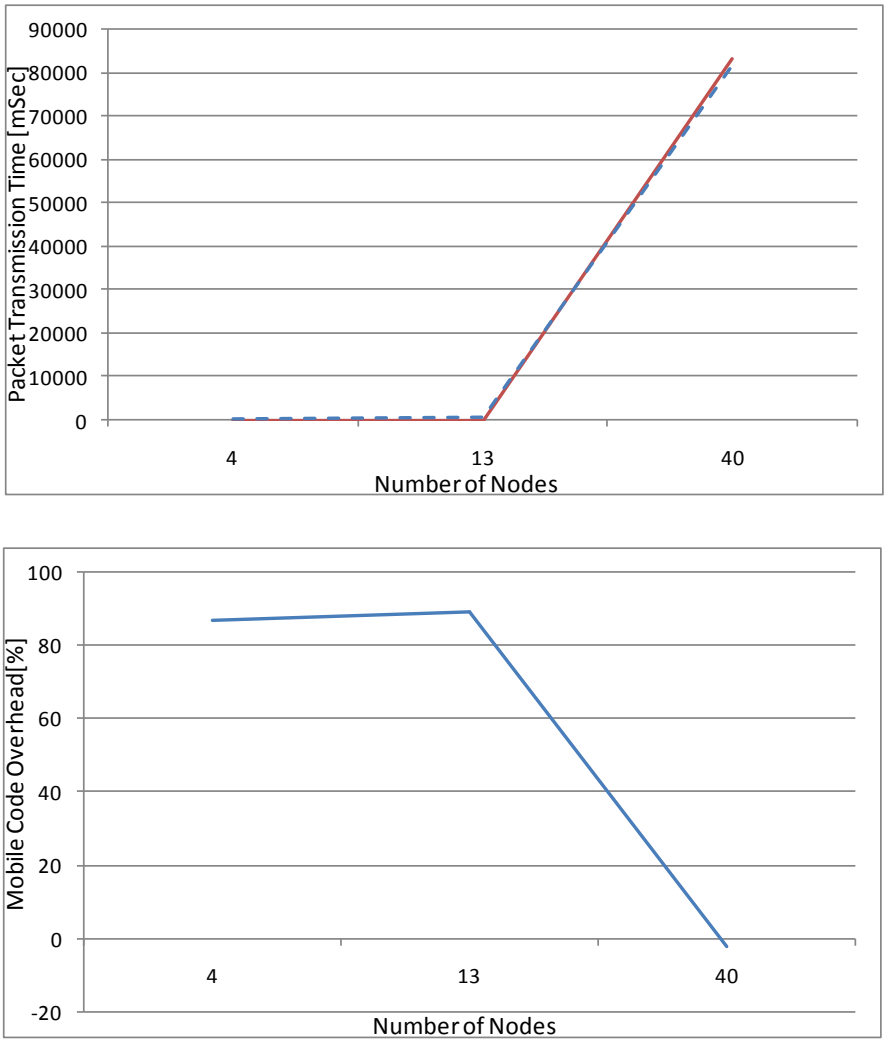


Figure 6.8.: Experiments with a ternary complete tree topology

already noticed in Figure 6.7, that is, the modified version of REDS (the one with the bytecode mobility facilities) scales as well as the standard REDS when the number of nodes grows enough.

# Part III.

# 7. Conclusion

In this thesis we have addressed the problem of designing, developing and orchestrating applications operating in pervasive environments. Such applications are required to support adaptive and evolutionary behaviors to deal with changes occurring in the environment. Changes are mainly the result of both the dynamic appearance/disappearance of functionalities and the interaction with an uncertain physical context.

In this context, we presented in Chapter 2 our model-centric conceptual model, which identifies the building blocks of self-adaptive pervasive systems dealing with both adaptation and evolution. In Chapter 3, 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 design model for pervasive applications derived from REST. We also provided a meta-model for P-REST meant to be instantiated to design applications that follow the P-REST principles. In this context, a middleware should provide proper abstractions and mechanisms satisfying the *flexibility*, *genericity* and *dynamicity* requirements. To this extent, in Chapter 4 we presented the PRIME middleware that provides the programming abstractions for the development and the execution of pervasive applications adhering to the P-REST principles. Further, in Chapter 5 a coordination language called P*a*CE has been introduced to ease the task of orchestrating resources deployed within the PRIME middleware. P*a*CE interpreter also provides primitives to modify the script at run time to implement the canonical adaptation actions. Finally, to prove the effectiveness of the whole framework, in Chapter 6, we presented a case study in which we went through the whole development process from the design phase to the run-time adaptation phase.

This work opens several research lines that can be worth following in future. The first and more pressing need for the our framework is a standard way to describe resources. Currently, the implementation of such feature is left to the developers that can define their own description systems and the appropriate filter functions to be used by the lookup service. A standard description framework would enable the creation of default and configurable lookup functions and, as a consequence, would further relieve the development effort.

Moving on to P*a*CE, right now every instruction is executed as soon

as possible according to the data-flow execution model. This mechanism favors the parallelism but, in some cases, this is not the most suitable behavior. For instance, if a script must retrieve the state of a remote resource and the developer wants an updated result, PaCE cannot provide any guarantee in this sense. Thus, we want to investigate the insertion in PaCE of a call-by-name semantics for remote operations. In this way the execution of an instruction becomes lazy and does not take place as soon as possible anymore but it is delayed as much as possible. Clearly this mechanism does not substitute the standard invocation semantics but is meant to work side by side with it. Further work can be carried out to achieve a higher degree of parallelism by improving the parallel execution of the iterations of a cycle and of the external functions. As for the cycles, if iterations are completely independent they can be executed concurrently according to the patterns identified by Pautasso and Alonso in [69]. Cases in which the iterations are possibly not independent are when a `read` or an external function are involved.

Last but not least, it would be very useful providing a user-friendly interface to handle a PRIME instance. Currently, indeed, every interaction is handled through either the command line or the bootstrap parameters.

# Bibliography


[1] Dimitri Papadimitriou. Future internet - the cross-etp vision document. `http://www.future-internet.eu`, Jan 2009. version 1.0.

[2] Debashis Saha and Amitava Mukherjee. Pervasive computing: A paradigm for the 21st century. *Computer*, 36(3):25–31, 2003.

[3] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, 2006.

[4] B.H.C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee. Software engineering for self-adaptive systems: A research road map. In *Dagstuhl Seminar Proceedings*, volume 8031. Springer, 2008.

[5] Danilo Ardagna, Carlo Ghezzi, and Raffaela Mirandola. Rethinking the Use of Models in Software Architecture Quality of Software Architectures. Models and Architectures. In Steffen Becker, Frantisek Plasil, and Ralf Reussner, editors, *Quality of Software Architectures. Models and Architectures*, volume 5281 of *Lecture Notes in Computer Science*, chapter 1, pages 1–27. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[6] Mary Shaw. Beyond objects: a software design paradigm based on process control. *SIGSOFT Softw. Eng. Notes*, 20:27–38, January 1995.

[7] Hausi Müller, Mauro Pezzè, and Mary Shaw. Visibility of control in adaptive systems. In *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, ULSSIS '08, pages 23–26, New York, NY, USA, 2008. ACM.

[8] S.W. Cheng, D. Garlan, and B. Schmerl. Making self-adaptation an engineering reality. *Self-star Properties in Complex Information Systems*, pages 349–349, 2005.

[9] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1:223–259, December 2006.

[10] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, pages 41–50, 2003.

[11] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing — degrees, models, and applications. *ACM Comput. Surv.*, 40:7:1–7:28, August 2008.

[12] Antonio Bucchiarone, Cinzia Cappiello, Elisabetta Di Nitto, Raman Kazhamiakin, Valentina Mazza, and Marco Pistore. Design for adaptation of service-based applications: Main issues and requirements. In *ICSOC/ServiceWave Workshops*, pages 467–476, 2009.

[13] Mauro Caporuscio, Marco Funaro, and Carlo Ghezzi. Architectural issues of adaptive pervasive systems. In Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schïurr, and Bernhard Westfechtel, editors, *Graph Transformations and Model Driven Enginering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, volume 5765 of *Lecture Notes in Computer Science*, pages 500–520. Springer, 2010.

[14] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, April 1996.

[15] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Tran. Soft. Eng.*, 16(11):1293–1306, 1990.

[16] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng.*, 33(12):856–868, 2007.

[17] R. N. Taylor, N. Medvidovic, and P. Oreizy. Architectural styles for runtime software adaptation. In *WICSA/ECSA 09*, 2009.

[18] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., and Jason E. Robbins. A component- and message-based architectural style for gui software. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 295–304, New York, NY, USA, 1995. ACM.

[19] R.N. Taylor, N. Medvidovic, K.M. Anderson, Jr. Whitehead, E.J., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for gui software. *Software Engineering, IEEE Transactions on*, 22(6):390 –406, jun 1996.

[20] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[21] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[22] R. T. Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[23] Justin R. Erenkrantz, Michael Gorlick, Girish Suryanarayana, and Richard N. Taylor. From representations to computations: the evolution of web architectures. In *European Software Engineering Conference -Symposium on the Foundations of Software Engineering*, pages 255–264, Dubrovnik, Croatia, 2007.

[24] Object Management Group (OMG). The common object object request broker architecture, 3.1.1, September 2011.

[25] Jim Waldo. *The Jini Specifications.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[26] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 21–26, New York, NY, USA, 2002. ACM.

[27] Paul Grace, Gordon S. Blair, and Sam Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9:2–14, January 2005.

[28] Mauro Caporuscio, Pierre-Guillaume Raverdy, and Valerie Issarny. ubiSOAP: A Service Oriented Middleware for Ubiquitous Networking. *IEEE Transactions on Services Computing*, 99(PrePrints), 2011.

[29] Fabio Mancinelli. Leveraging the web platform for ambient computing: An experience. *IJACI*, 2(4):33–43, 2010.

[30] Tim Kindberg and John Barton. A web-based nomadic computing system. *Computer Networks*, 35(4):443 – 456, 2001.

[31] Dan R. Olsen, Jr., Sean Jefferies, Travis Nielsen, William Moyes, and Paul Fredrickson. Cross-modal interaction using xweb. In *13th annual ACM symposium on User interface software and technology*, UIST '00, pages 191–200, 2000.

[32] Daniele Bonetta and Cesare Pautasso. An architectural style for liquid web services. *Software Architecture, Working IEEE/IFIP Conference on*, 0:232–241, 2011.

[33] Mauro Caporuscio, Marco Funaro, and Carlo Ghezzi. Restful service architectures for pervasive networking environments. In Erik Wilde and Cesare Pautasso, editors, *REST: From Research to Practice*. Springer, 2011.

[34] Daniel Romero, Romain Rouvoy, Lionel Seinturier, and Pierre Carton. Service discovery in ubiquitous feedback control loops. In *DAIS*, pages 112–125, 2010.

[35] D. Guinard and V. Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), Madrid, Spain*, 2009.

[36] D. Guinard, V. Trifa, and E. Wilde. A resource oriented architecture for the web of things. In *Internet of Things (IOT), 2010*, pages 1 –8, 29 2010-dec. 1 2010.

[37] Leonard Richardson and Sam Ruby. *Restful web services*. O'Reilly, 2007.

[38] W. Keith Edwards, Mark W. Newman, Jana Z. Sedivy, and Trevor F. Smith. Experiences with recombinant computing: Exploring ad hoc interoperability in evolving digital networks. *ACM Trans. Comput.-Hum. Interact.*, 16(1):1–44, 2009.

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

[40] Gruia-Catalin Roman, Gian Pietro Picco, and Amy L. Murphy. Software engineering for mobility: a roadmap. In *FOSE '00*, pages 241–258, New York, NY, USA, 2000. ACM.

[41] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile enviroment. In *2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 27–34, 2001.

[42] S. Ben Mokhtar, A. Kaul, N. Georgantas, and V. Issarny. Efficient semantic service discovery in pervasive computing environments. *Middleware 2006*, pages 240–259, 2006.

[43] Network Working Group. Role of the Domain Name System (DNS). RFC3467.

[44] Rohit Khare and Richard N. Taylor. Extending the representational state transfer (rest) architectural style for decentralized systems. In *ICSE '04*, pages 428–437, Washington, DC, USA, 2004. IEEE Computer Society.

[45] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 2001.

[46] Cesare Pautasso. Composing restful services with jopera. In *International Conference on Software Composition 2009*, volume 5634, pages 142–159, Zurich, Switzerland, July 2009. Springer.

[47] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98*.

[48] The Scala programming language. `http://www.scala-lang.org/`.

[49] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[50] Diego Doval and Donald O'Mahony. Overlay networks: A scalable alternative for p2p. *IEEE Internet Computing*, 7(4):79–82, July-Aug. 2003.

[51] Gianpaolo Cugola and Gian Pietro Picco. Reds: a reconfigurable dispatching system. In *6th international workshop on software engineering and middleware*, 2006.

[52] Paolo Santi. Topology control in wireless ad hoc and sensor networks. *ACM Comput. Surv.*, 37(2):164–194, 2005.

[53] Luca Mottola, Gianpaolo Cugola, and Gian Pietro Picco. A self-reparing tree topology enabling content-based routing in mobile ad hoc networks. *Transaction on Mobile Computing*, 7(8):946–960, 2008.

[54] Mauro Caporuscio and Alfredo Navarra. CoP3D: Context-aware overlay tree for content-based control systems. In *International Conference on Informatics in Control, Automation and Robotics, Intelligent Control Systems and Optimization*, 2009.

[55] UPnP Forum. UPnP Device Architecture, 2008. Version 1.1.

[56] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35:97–107, February 1992.

[57] John Backus. Acm turing award lectures. chapter Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs, pages 1977–. ACM, New York, NY, USA, 2007.

[58] Arvind and D E Culler. Dataflow architectures. *Annual Review of Computer Science*, 1(1):225–253, 1986.

[59] A.L. Davis and R.M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, 1982.

[60] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[61] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36:1–34, March 2004.

[62] William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[63] Cesare Pautasso and Gustavo Alonso. The jopera visual composition language. *Journal of Visual Languages & Computing*, 16(1-2):119 – 152, 2005. 2003 IEEE Symposium on Human Centric Computing Languages and Environments.

[64] Cesare Pautasso and Gustavo Alonso. Jopera: A toolkit for efficient visual composition of web services. *Int. J. Electron. Commerce*, 9:107–141, January 2005.

[65] M. Bernini and M. Mosconi. Vipers: a data flow visual programming environment based on the tcl language. In *AVI: Proceedings of the workshop on Advanced visual interfaces*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 1994.

[66] R. Jagannathan. Coarse-grain dataflow programming of conventional parallel computers. In *in Advanced Topics in Dataflow Computing and Multithreading*, pages 113–129. IEEE Computer Society Press, 1995.

[67] W.B. Ackerman. Data flow languages. *Computer*, 15(2):15 – 25, feb 1982.

[68] A. Moors, F. Piessens, and M. Odersky. Parser combinators in scala. *CW Reports, volume CW491, Department of Computer Science, KU Leuven*, 2008.

[69] C. Pautasso and G. Alonso. Parallel computing patterns for grid workflows. In *Workflows in Support of Large-Scale Science, 2006. WORKS'06. Workshop on*, pages 1–10. IEEE, 2006.