Contents

0-8493-0052-5/00/$0.00+$.50

List of Figures

0-8493-0052-5/00/$0.00+$.50

Part III

Technologies

Chapter 2

Enabling Context-Aware Web Services: A Middleware Approach for Ubiquitous Environments

Abstract In ubiquitous environments, mobile applications should sense and react to environmental changes to provide a better user experience. In order to deal with these concerns, *Service-Oriented Architectures* (SOA) provide a solution allowing applications to interact with the services available in their surroundings. In particular, context-aware Web Services can adapt their behavior considering the user context. However, the limited resources of mobile devices restrict the adaptation degree. Furthermore, the diverse nature of context information makes difficult its retrieval, processing and distribution. To tackle these challenges, we present the CAPPUCINO platform for executing context-aware *Web Services* in ubiquitous environments. In particular, in this chapter we focus on the middleware part that is built as an autonomic control loop that deals with dynamic adaptation. In this autonomic loop we use FraSCAti, an implementation of the *Service Component Architecture* (SCA) specification, as the execution kernel for *Web Services*. The context distribution is achieved with SPACES, a flexible solution based on REST (*REpresentational State Transfer*) principles and benefiting from the COSMOS (*COntext entitieS coMpositiOn and Sharing*) context management framework. The application of our platform is illustrated with a mobile commerce application scenario that combines context-aware *Web Services* and social networks.

0-8493-0052-5/00/$0.00+$.50

2.1 Introduction

Context awareness enables nowadays systems to sense and react to changes observed in their physical environment (*e.g.*, location, connectivity, brightness, resource availability). This capability is particularly critical in ubiquitous environments, where context is the central element of mobile applications. In order to cope with these concerns, *Service-Oriented Architectures* (SOA) have provided an attractive solution to build distributed applications, which can communicate with services discovered in their surroundings. In particular, *Web Services* [2, 8] enable mobile devices to interact spontaneously with services available on the Internet (weather forecast, news, etc.) in order to improve the *Quality of Service* (QoS) offered to end-users. Context-aware Web Services can therefore tune dynamically their behavior in order to better integrate the user's context. Nevertheless, on a mobile device, the degree of adaptation of these applications is often restricted by the device capacities. In order to meet this challenge and provide an uniform approach to context-aware adaptation, we propose to exploit the *Service Component Architecture* (SCA) technology [5] for enabling end-to-end adaptive applications. We believe that SCA provides an elegant solution to build self-adaptive ubiquitous applications based on Web Services.

This chapter introduces CAPPUCINO[1], which is a SCA-based middleware platform for executing context-aware Web Services in ubiquitous environments. The CAPPUCINO platform is divided into two technical parts: context-aware software product line[2] and a middleware platform for context-aware Web Services. The architecture of our Web Services platform is built as an *autonomic control loop* [16]. The principle of the control loop from the autonomic computing domain brings several characteristics, such as self-configuration, self-optimization, self-healing, self-reparation [28, 17]. We rely on several of these characteristics to build our platform architecture, which is structured around the following threefold contribution:

The *autonomic control loop* is built of three parts: *i)* A "knowledge" part for abstracting modelling of autonomous systems; *ii)* an "autonomic manager" part for setting adaptation rules; and *iii)* a "managed resources" composed of context management, deployment, and reconfiguration services.

The *execution kernel* is based on FraSCAti, our implementation of the SCA specification and its FraSCAme version for Java ME based mobile devices.

[1]The development of this platform is supported by the project of the same name from the research agency PICOM, Nord-Pas-de-Calais, France.

[2]The context-aware software product line is published as a separate chapter to this book [22].

The *context-awareness* of Web Services is dealt by a *context mediation service*: SPACES. This service applies the principles of *REpresentational State Transfer* (REST) in order to distribute COSMOS context policies in ubiquitous environments.

The remainder of this chapter is organized as follows. Section 2.2 motivates the use of context-awareness in mobile commerce applications. Section 2.3 introduces the principles and the background of the contribution. Then, the threefold contribution, that is the realization of an autonomous SOA system with the control loop, the SCA runtime, and the distributed RESTful context manager, is described in Section 2.4 and then illustrated in a mobile commerce scenario in Section 2.5. Finally, Sections 2.6 and 2.7 discuss the contribution with regard to related work and conclude the chapter, respectively.

2.2 Motivating Scenario

In order to motivate the challenges of ubiquitous applications development, this section introduces a mobile commerce application scenario. This scenario takes advantage of some of the new opportunities brought by ubiquitous systems, such as the availability of context-aware Web Services to *On-The-Go* users and context-aware social networks.

> *Mary uses an application on her mobile phone to search and buy different items from an online catalog. This application exploits Mary's profile to offer her personalized services. For example, when subscribing to the reward point program, Mary automatically receives special offers and prices. She can also register important events in her calendar like her best friends birthdays. Using this information, customized notifications with a focused product offer are pushed to her mobile phone. Mary can also find gift ideas by using a special service that connects to a social network to retrieve a list of items selected by her friends as their favorite products. Additionally, according to her location, Mary is informed of Flash Sale offers currently running in the shops nearby. If she does not have an Internet access when she finally makes her choice, she can still complete her order off-line. The application registers the order in memory, as an SMS message, and sends it automatically once the network coverage becomes sufficient again.*

Challenges This scenario highlights some of the challenges of the development of context-aware web services systems concerning the capacity to detect

context changes in the user environment and to dynamically adapt the provided services. We now discuss how the CAPPUCINO platform can fulfill these challenges.

Dynamic adaptation: The scenario requires that applicative services are context-aware and can be dynamically reconfigured after a context change. The customized catalog depends on the user's context, including both profile and location information. When a special event registered in Mary's profile (a birthday approaching) occurs, the CAPPUCINO platform prepares a new customized catalog. In order to do that, the platform considers the account preferences and product suggestions coming from several sources (*e.g.*, a social network). A more challenging situation results from combining Mary's location and the current date and time. When Mary comes close to a shop, where there is a Flash Sale offer running, her mobile device is detected and a new version of the customized catalog is proposed to her. The Flash Sale offer is only available at a given place and during a precise time period. This situation demonstrates the capacity of the CAPPUCINO platform to deal with time and space constraints. Furthermore, the purchase is usually performed online. This means that the client remains connected during the whole purchase process. However, the network connection is not necessarily guaranteed all the time. For this reason, the CAPPUCINO platform provides service continuity in case of a network disconnection. To do that, the order can be continued off line and registered as an SMS message. This message will be sent automatically when the network coverage becomes sufficient again.

Device limitations: As mobile devices vary in terms of hardware and software capabilities, tailoring software components to run on multiple devices is a difficult and time-consuming task that must be dealt with carefully. Therefore, the deployment of software components on mobile devices and the dynamic reconfiguration of the application depending on context variation represents a key challenge of the CAPPUCINO approach.

Context mediation: The context information manipulated in this scenario is associated to the user profile, location and network connectivity. Mary's profile aggregates a very rich set of information including her favorite products (published in a social network), the characteristics of her mobile device, and her current subscriptions to the reward point program and to the gift idea service. These pieces of information are not all registered at a single place but are rather distributed on different devices. Location information can be determined using a GPS-like system in outdoor conditions. However, other geo-location mechanisms are necessary for indoor environments involving sensors dispersed in the monitored area. Network connectivity information is critical for *On-The-Go* users. Therefore, it is required to monitor different network interfaces, such as WiFi, Bluetooth, or long-distance access and to anticipate potential disconnections.

2.3 Principles and Background

This section presents the foundations of our contributions, namely the MAPE-K control loop (cf. Section 2.3.1), the SCA component model (cf. Section 2.3.2) and the COSMOS context framework (cf. Section 2.3.3). The combination of these three building blocks represents the foundations of the CAPPUCINO platform.

2.3.1 MAPE-K: Control Loop for Autonomic Computing

The CAPPUCINO runtime follows the principle of autonomic computing, which is a computing environment able to manage itself and dynamically adapt to change according to business policies and objectives. Autonomic systems are characterized by their ability to devise and apply counter-measures when necessary, including the ability to detect these adaptation situations. Self-managing environments can perform such activities based on situations they observe or sense in the IT environment rather than requiring IT professionals to initiate the task. The following properties are the ones that we elect in the design of the CAPPUCINO runtime (taken from [13, 16, 17]). *Self-configuration* stipulates that the system shall be capable of adapting its behavior to the execution context. For instance, it shall be possible to add or remove some functionalities (*e.g.*, a business component providing a service) without a complete interruption of all the services. Moreover, system parts not directly impacted by the changes shall be able to progressively adapt themselves to these changes. *Self-optimization* states that the system shall control and monitor the system resources it consumes. It shall then detect a degradation of service and cater with the necessary reconfigurations to improve their own performance and efficiency. *Self-healing* establishes that the detection, analysis, prevention, and resolution of damages shall be managed by the system itself. The system shall be capable to survive hardware and software failures. *Self-protecting* aims at anticipating, detecting, identifying and protecting against threats. System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent system-wide failures.

These properties are generally reached in autonomic computing by applying the principles of the MAPE-K control loop (upper side of Figure 2.1). The Knowledge is the standard data shared among the Monitor, Analyze, Plan and Execute functions of an *Autonomic Manager*, such as symptoms and policies. This run-time knowledge must be complete—*i.e.*, including the whole aspects influencing adaptation decisions—, modifiable—*i.e*, following the application changes—, and at a high-level of abstraction—*i.e*, comprising only relevant information. The Monitor part provides the mechanisms that collect, aggregate, filter and report details (such as metrics and topologies) collected from a man-

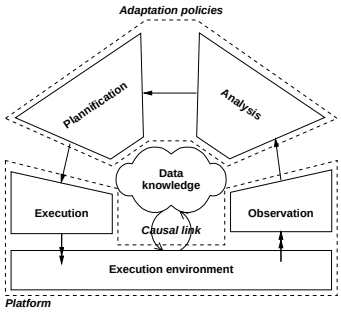

Figure 2.1: Overview of the MAPE-K autonomic control loop.

aged resource. The Analyze part contains the mechanisms that correlate and model complex *adaptation situations*. The Plan function encloses the mechanisms that construct the actions needed to achieve goals and objectives. The planning mechanism uses *adaptation policies* information to guide its work. The Execute function groups the mechanisms that control the execution of an *adaptation plan* with considerations for dynamic updates. The *Managed Resources* are entities that exist in the run-time environment of an IT system and that can be managed. This encloses the Execution Environment, Supporting Platform, and Application entities. All these managed resources provide Sensors to introspect the entity states, and Effectors to reconfigure them dynamically.

2.3.2 SCA: Developing Web Service Components

Initiated in 2005, *Service Component Architecture* (SCA) [21] is a set of specifications for building distributed applications using the principles of SOA and CBSE. The model is promoted by a group of companies, including BEA, IBM, IONA, Oracle, SAP, Sun, and TIBCO. The specifications are now defined and hosted by the *Open Service Oriented Architecture* (OSOA) collaboration[3] and promoted in the Open CSA section of the OASIS consortium[4].

SOA, *e.g.*, when based on Web Services, provides a way for exposing coarse-grained and loosely-coupled services, which can be remotely accessed. Yet, the SOA approach seldom addresses the issue of the way these services should be implemented. SCA fills this gap by defining a distributed component model. SCA entities are software *components*, which may provide interfaces (called *services*), require interfaces (called *references*), and expose *properties*. References and services are connected through *wires*. The model is hier-

[3]OSOA collaboration: `http://www.osoa.org`
[4]OASIS consortium: `http://www.oasis-opencsa.org`

archical with components being implemented either by primitive language entities or by subcomponents (the component is then said to be a *composite*). Figure 2.2, which is taken from the SCA specifications [21], provides the graphical notation for these concepts. An XML-based assembly language is available to configure and assemble components.

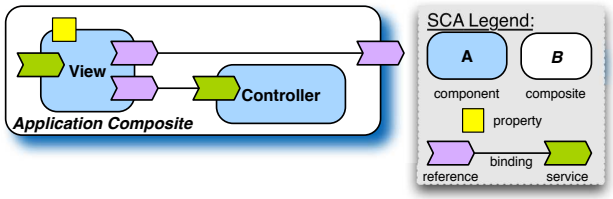

Figure 2.2: SCA component architecture [21].

In order to cover a wide range of SOA applications, the SCA specifications are not bound to a particular middleware technology, nor a programming language. They have been kept as open as possible thanks to four main principles: Independence from *programming languages*, from *interface definition languages*, from *communication protocols*, and from *non-functional properties*. Firstly, SCA does not assume that components will be implemented with a unique programming language. Rather, several language mappings are supported and allow programming SCA components in Java, C++, PHP, BPEL, or COBOL.

Secondly, SCA components provide (*resp.* require) functionalities through precisely defined interfaces. SCA does not assume that a single *Interface Definition Language* (IDL) will fit all needs. Rather, several IDL are supported, such as WSDL and Java interfaces.

Thirdly, although Web Services are the preferred communication mode for SCA components, this solution may not fit all needs. In some cases, protocols with different semantics and properties may be needed. For that, SCA provides the notion of a *binding*: a service or a reference will be bound to a particular communication protocol, such as SOAP [3, 8] for Web Services, Java RMI, Sun JMS, or REST [10].

Lastly, concerning the non-functional properties, they may be associated to an SCA component with the notion of a *policy set* (this notion is also referred to with the term *intent*). The idea is to let a component declare the set of policies (non-functional services) that it depends upon. The platform is then in charge of guaranteeing that these policies are enforced. So far, security and transactions have been included in the SCA specifications. Yet, developers may need other types of non-functional services. For that, the body of supported policy sets may be extended with user-specified values.

Therefore, these principles offer a broad scope of solutions for implementing SCA-based applications. Developers may think of incorporating new forms of language mappings (*e.g.*, SCA components programmed with EJB or OSGi, which are two solutions currently investigated by the community), IDLs (*e.g.*, CORBA IDL), communication bindings (*e.g.*, *Java Business Integration* [JBI] bindings) and non-functional properties (*e.g.*, persistence, logging).

2.3.3 COSMOS: Collecting and Composing Context Data

COntext entitieS coMpositiOn and Sharing (COSMOS) is a component-based framework for managing context information in ubiquitous environments for context-aware applications [6, 26]. In particular, COSMOS identifies the contextual situations to which a context-aware application is expected to react. These situations are modeled as *context policies* that are hierarchically decomposed into fine-grained units called *context nodes* (cf. Figure 2.3).

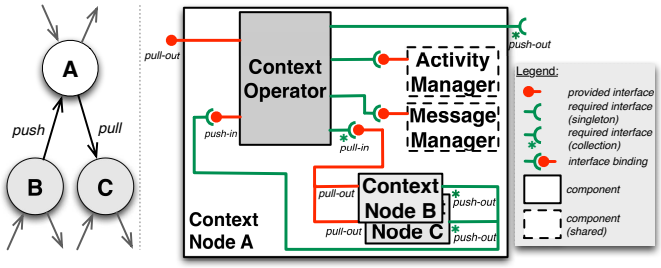

Figure 2.3: COSMOS context node and context policy.

A Context Node represents context information modeled by a software component. The relationships between context nodes are *sharing* and *encapsulation*. The sharing of a context node—and, by implication, of a partial or complete hierarchy—corresponds to the sharing of a part of or a whole context policy. Context nodes leaves (the bottom-most elements, with no descendants) encapsulate raw context data obtained from collectors, such as operating system probes, sensors in the vicinity of the user terminal, user preferences in profiles, and remote devices. Context nodes should provide all the inputs necessary for reasoning about the execution context. For this reason COSMOS considers user preferences as context information. Thus, the role of a context node is to isolate the inference of high-level context information from lower architectural layers responsible for collecting context information.

Context nodes also possess *properties* that define their behavior with respect to the context policy: *(i)* **active/passive** An active node is associated with a thread of control (attached to the Activity Manager). Typical examples of

active nodes include a node in charge of the centralization of several types of context information, a node responsible for the periodic computation of higher-level context information, and a node to provide the latter information to upper nodes. A passive node gets information upon demand; *(ii)* **observation/notification** Communication into a context node's hierarchy can be top-down or bottom-up. The former—implemented by the interface `Pull`—corresponds to observations that a parent node triggers, while the latter—realized by the interface `Push`—corresponds to notifications that context nodes send to their parents; *(iii)* **pass-through/blocking** Pass-through nodes propagate observations and notifications while blocking nodes stop the traversal. In other words, for observations, a blocking node transmits the most up-to-date context information without polling child nodes, and for notifications, a blocking node uses context data to update the node's state, but it does not notify parent nodes.

COSMOS provides the developer with predefined generic context operators, organized following a typology: elementary operators for collecting raw data; memory operators, such as averaging and translation operators; data mergers; and abstract or inference operators, such as adders or thresholds operators. The only programming is in the Context Operators. If a sufficiently large library of context operators is available and well targeted to a developer's business, no programming should be necessary—only declarative composition of context nodes.

The context nodes exchange *context reports* (messages) which are created and manipulated by the Message Manager. The context nodes send and receive these context reports through the `Pull` and `Push` interfaces. Each context report is composed of a set of `chunks` and encloses a set of sub-messages, while each chunk reflects a context information as a 3-tuple `(name, value, content type)`.

2.4 CAPPUCINO: Enabling Context-aware Adaptive Services

The CAPPUCINO runtime combines the strengths of both the SCA and COSMOS models to provide a versatile infrastructure for supporting context-aware adaptations in SOA environments. After an introduction of the overall infrastructure (cf. Section 2.4.1), this section focuses on the reconfiguration capabilities offered by the FraSCAti platform (cf. Section 2.4.2) as well as on the distribution of context policies using SPACES (cf. Section 2.4.3).

2.4.1 Overview of the CAPPUCINO Runtime

The novelty of CAPPUCINO lies in the exploitation of SCA as an unified model to build both applications and their associated control loops. Each control loop is in charge of monitoring the execution of an SCA application and of adapting it with regards to the evolutions of the ambient context. Figure 2.4 illustrates our contributions and introduces the CAPPUCINO distributed runtime as well as its core constituting components. This illustration considers a deployment on three distinct nodes: the *adaptation server*, a *mobile client*, and an *application server*. The adaptation server hosts the control loop dedicated to an application. This control loop is deployed as a SCA system using the FraSCAti platform. The control loop interacts with the mobile client(s)—*i.e.*, hand-held device(s) hosting a lightweight version of FraSCAti (so called FraSCAme)—to deploy and reconfigure dynamically the client-side application. The server-side application can also be deployed and reconfigured dynamically using the FraSCAti platform available on the application servers.

Specifically, these control loops are implemented as distributed COSMOS context policies using SPACES in order to describe end-to-end adaptations of the system. In particular, such a distributed context policy correlates the ambient context information to infer the reconfiguration actions to execute for adapting the SCA application. The reconfiguration scripts are automatically produced depending on changes observed in the surrounding environment of a mobile device. As the adaptation decision is taken remotely, the context information sensed by the mobile device requires therefore to be propagated to the remote server. Thus, the combination of the SCA and COSMOS concepts enables CAPPUCINO to *i)* build context-aware SCA applications and *ii)* distribute the adaptation policies across a set of distributed nodes.

Inspired by the MUSIC definition [25], an *adaptation domain* is a partition of CAPPUCINO runtime instances executing a distributed application under the control of one adaptation server. This node acts as the nucleus around which the adaptation domain organizes itself as satellite nodes, such as mobile devices and application servers, enter and leave the domain. The movement of satellite nodes or changes in the network topology also cause the dynamic configuration of an adaptation domain. Adaptation domains may overlap in the sense that satellite nodes may be members of multiple adaptation domains. This adds to the dynamics and increases the complexity since the amount of resources the satellite nodes are willing to provide to a given adaptation domain may vary depending on the needs of other served domains. The user of a mobile device may start and stop applications or shared services. The set of running SCA components is therefore safely adapted by the Reconfiguration Engine according to the end-user actions and context changes, while taking into account the resource constraints.

This Reconfiguration Engine pilots the SCA Platform to deploy the SCA application components, as further described in Section 2.4.2. Once deployed,

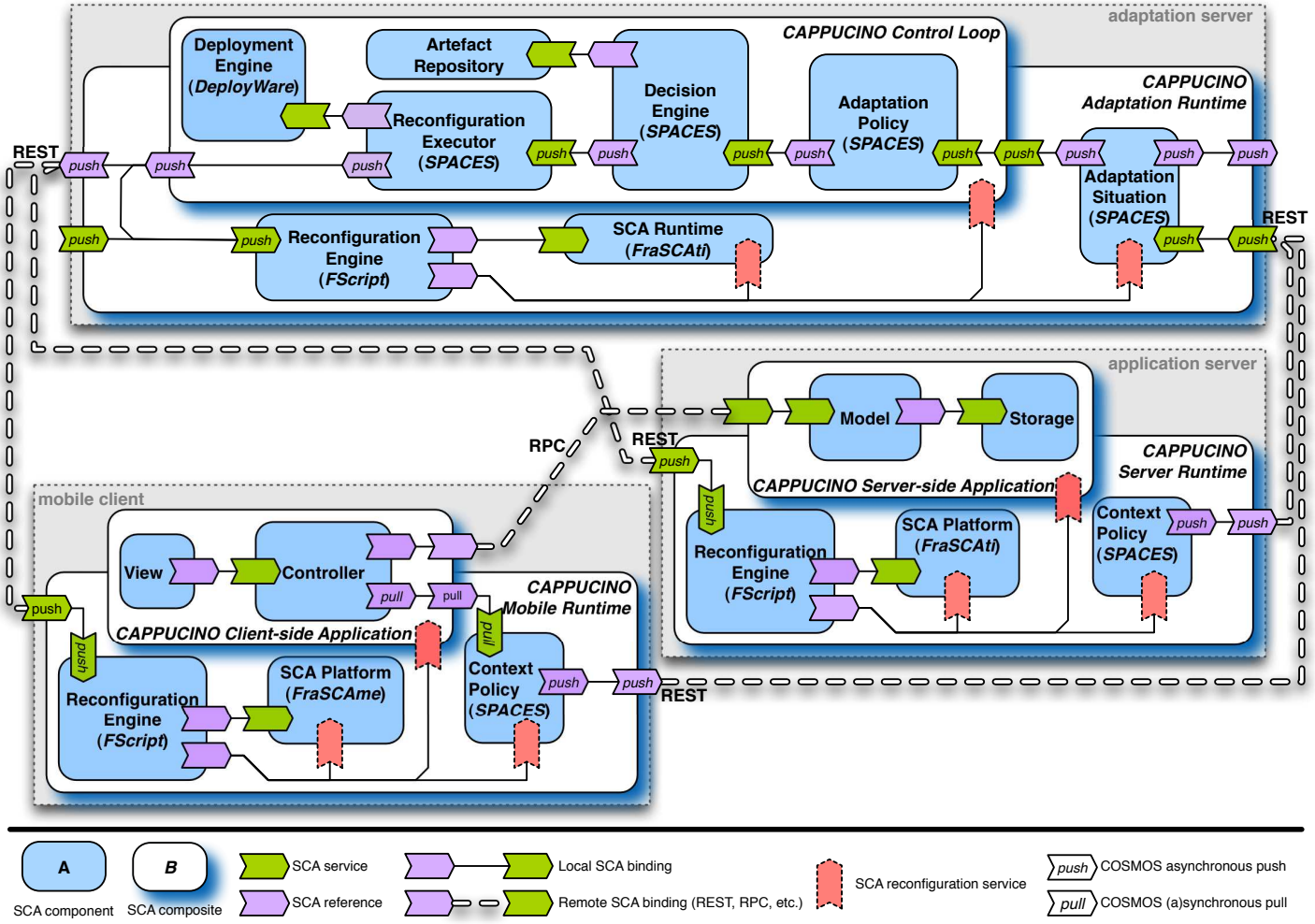

Figure 2.4: Overview of the CAPPUCINO distributed runtime.

these components can be reconfigured dynamically via a SCA reconfiguration service, provided by the SCA component, in order to reflect changes in the computing environment. Reconfigurations in this application layer include the deployment of new SCA components as well as their bindings to available services. This capability is an original SCA feature enabled by the integration of the FSCRIPT reconfiguration language [9] within the CAPPUCINO runtime. Besides, thanks to the adoption of a uniform component model for all the parts of the control loop (including the execution kernel), the Reconfiguration Engine can also exploit the SCA reconfiguration service provided by the SCA Platform and the Context Policy to dynamically reconfigure the CAPPUCINO runtime. Possible reconfigurations in this infrastructure layer include the deployment of new communication protocols (*e.g.*, SOAP) and new context collectors (*e.g.*, GPS sensor).

The Context Policy is responsible for *i)* providing contextual information and *ii)* detecting changes in the device's context. The former supports the realization of context-aware applications, while the latter initiates the adaptation process by sending the reified context information to the context policy hosted on the adaptation server. The distribution of a context policy among hosts is based on the REST architecture style as described in Section 2.4.3. In particular, SPACES enriches COSMOS with a flexible RESTful architecture for supporting lightweight communication protocols and alternative resource representations. Once received by the Adaptation Situation, the context information—in addition to application variation points and device characteristics—is processed by the Adaptation Policy to filter out the input data relevant for the Decision Engine. Note that for the sake of clarity, Figure 2.4 does not depict observation interactions between context policies performed using the `pull` calls of COSMOS; the notification mode—*i.e.*, COSMOS `push` calls—suffices to demonstrate the principles of the control loop.

The Decision Engine applies an application-specific reasoning algorithm (*e.g.*, situation-action rules or constraint satisfaction problems) to produce the *adaptation plans* (*e.g.*, the reconfiguration script) to execute. The application artefacts as well as the adaptation knowledge are provided by the Artefact Repository. If a reconfiguration is requested, the Reconfiguration Executor can deploy the adaptation artefacts using the Deployment Engine, which is based on the DEPLOYWARE framework [11], and then trigger a reconfiguration by sending the generated reconfiguration script(s) to the Reconfiguration Engine(s) available in the adaptation domain. This typically means that the result of an adaptation may impact not only the mobile device(s), but also the application server(s) as well as the adaptation server. For instance, adaptations performed in the application server can balance the load and handle potential failures.

2.4.2 FraSCAti: Deploying Adaptive Distributed Systems

The FraSCAti platform [27] enables the execution of SOA applications with advanced properties, such as the ability to reconfigure at runtime existing assemblies of SCA components (cf. Section 2.3.2). Other available features include a binding factory supporting various communication protocols, a transaction service, a semantic trader service, a service for deploying autonomic SCA-based applications, and a graphical administration and management console. As illustrated in Figure 2.5, the FraSCAti platform is composed of the following elements. The SCA Meta Model defines all the concepts available in the SCA specifications. The meta-model is used by the FraSCAti Assembly Factory to load component assembly descriptors. The Tinfi Runtime Kernel is in charge of hosting the SCA components. This kernel implements the execution semantics defined by the SCA specifications and provides some additional properties, such as dynamic reconfiguration. This kernel is based on generative programming techniques where most of the code for managing components is generated on-demand at runtime. The Binding Factory is in charge of establishing communication channels between SCA components. This factory supports various protocols, such as SOAP [3, 8], REST [10], and RMI. The Trading Service provides functionalities to perform semantic trading on components published in a registry. This service associates to each external element of a component (service, reference, property, implementation) a concept defined in a semantic model. This service allows, for example, the replacement of components which would have become inaccessible, by some semantically equivalent ones. The Assembly Factory is in charge of analyzing, validating, and instantiating assembly descriptors. The analysis step uses the SCA metamodel to load and validate descriptors. The instantiation step uses the runtime kernel and the assembly factory for installing and connecting components.

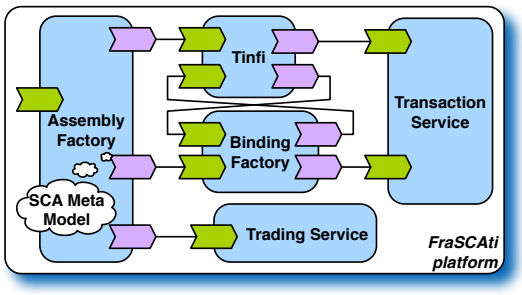

Figure 2.5: Architecture of the FraSCAti execution kernel.

Besides deploying and instantiating an SCA application, the FraSCAti

platform provides some runtime services for dynamic reconfiguration. These features are of particular importance for adapting applications to new and unforeseen requirements. The runtime adaptation actions, which are enabled by FraSCAti, are described below. They concern the architecture of an SCA application. They can take place at any time during the execution of the application.

Firstly, FraSCAti enables instantiating new components. For example, these new components can provide additional functionalities or provide alternate implementations for existing ones. Secondly, new wires can be created and existing wires can be removed. Thirdly, component hierarchies can be queried and modified. The query functionality enables traversing an existing application much like what is available with component browsers. The modification of the hierarchy, in conjunction with the creation of new wires, enables creating different component compositions. Finally, components can be selectively started and stopped. This enables performing maintenance operations in a safe manner.

2.4.3 SPACES: Deploying Distributed Context Policies

This section introduces SPACES: a *distributed processing architecture integrating context as RESTful services*. SPACES is a lightweight middleware solution enabling the versatile and efficient mediation of context information. Context policies have been introduced by the COSMOS context framework as a scalable model for processing context information (cf. Section 2.3.3). Therefore, we propose to extend COSMOS with the principles of *REpresentational State Transfer* (REST) [10] in order to distribute context policies in ubiquitous environments. These RESTful context policies leverage the provision of *Context as a Service* and enable the efficient mediation of context information among heterogeneous devices.

Figure 2.6 illustrates the integration of SPACES into a COSMOS context policy. Specifically, SPACES acts as a mediation middleware, which is responsible for disseminating the context information between two physical entities. This distribution is implemented as a connector based on the *HyperText Transfer Protocol* (HTTP). This connector is composed of *i)* an HTTP server publishing part of the context nodes as REST resources and *ii)* an HTTP client sending context requests as REST requests. By considering the context mediation as a software connector, the realization of the context policy is not impacted by the distribution concerns. In addition to that, the use of a software connector opens up for multiple implementations of the dissemination mechanism, including *message-oriented* [19] and *peer-to-peer* [14] middleware approaches.

In this section, we report the design of SPACES using HTTP to facilitate the integration of adaptation policies with legacy systems. In particular, we can monitor the context surrounding a mobile device from a traditional Internet browser and access this information from any systems supporting HTTP. This

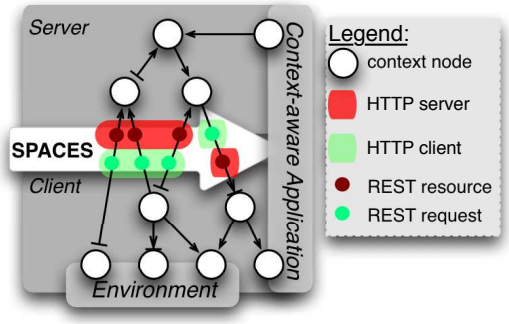

Figure 2.6: Principles of the SPACES approach.

integration is facilitated by the support for multiple representation of a given resource. This means that depending on the client requirements, a REST resource can be retrieved as an HTML document (when accessed from an Internet browser) or as an XML document (when accessed from a legacy system). Therefore, the integration of the REST principles within the context policies requires to map the REST triangle of nouns, verbs, and content types to COSMOS context nodes, which are the first class entities of CAPPUCINO adaptation policies.

SPACES Nouns. The advertisement of context nodes as REST resources requires the definition of *context identifiers*. These context identifiers are unique nouns described using the *Uniform Resource Identifier* (URI) format [1]. Therefore, context identifiers include a communication scheme, a server address, a context path, and a sequence of request parameters: `scheme://context-server/context-path?request-parameters`. The *communication scheme*, such as HTTP, FTP (*File Transfer Protocol*), URN (*Uniform Resource Name*), RTSP (*Real Time Streaming Protocol*), or file, describes the communication protocol used to transfer the resource representation between the hosting server and the requesting client. Then, the *server address* description is specific to a given scheme. For example, an HTTP server address can be specified using the syntax `user:password@host:port` to describe the web server host and port as well as the credentials for accessing the context information. The *context path* points to context node name under which the context information is published. This context path can be hierarchically organized in context domains using the syntax `parent-domain/child-domain/context`. Finally, the *request parameters* are used to specify the representation of the requested (or submitted) context information using the syntax `format=application/xml`.

SPACES Verbs. The dissemination of context information is based on *observation* and *notification primitives*. COSMOS implements these primitives

as `Pull` and `Push` interfaces (cf. Section 2.3.3). In SPACES, resource observations are implemented as side-effect free `GET` requests, while notifications are based on `POST`. For example, the context information *geolocation-dps* can be retrieved from a mobile device using the HTTP request `GET http://device.inria.fr:8080/geolocation-dps?format=application/xml`, which returns the requested context information as an XML document. Context information can also be pushed into a remote entity by using an HTTP `POST` request. In this case, the submitted request points to the server-side resource associated to the context policy. For example, the HTTP request `POST http://server.inria.fr/adjusted-bit-rate?format=application/json` notifies the server-side context policy that the uploaded JSON document refers to a change of the context information *GPS geolocation*.

SPACES Content Types. Finally, SPACES supports various types of content for representing the context information. This representation multiplicity benefits from the structure of COSMOS messages used to encode the context information (cf. Section 2.3.3). Therefore, the content types supported by SPACES are based on the MIME media types classification [15]. In particular, SPACES promotes the *Java object serialization* as the default resource representation (`application/octet-stream`) for performance concerns. Nevertheless, SPACES provides also representations of resources as XML (`application/xml`) and JSON [7] (`application/json`) documents.

2.5 Illustrating Dynamic Context-aware Web Services with a Mobile Commerce Scenario

We present in this section the way the CAPPUCINO platform is able to realize the mobile commerce scenario presented in Section 2.2. Figure 2.7 shows the corresponding component architecture obtained using CAPucine (*Context-Aware Service-Oriented Product Line*), a Context-aware *Dynamic Service-Oriented Product Line* (DSOPL) approach described in [22]. We describe in Section 2.5.1 how deployment takes place on a mobile device. We then detail in Section 2.5.2 a reconfiguration scenario taking place on the mobile device after the detection of an adaptation situation. We focus on the deployment and reconfiguration on the mobile device side for more concision. Nonetheless, the application server side follows the same principles.

2.5.1 On-demand Deployment on a Mobile Device

This section presents the dynamic aspects of the CAPPUCINO architecture as UML activity diagrams. We consider that the deployment of the

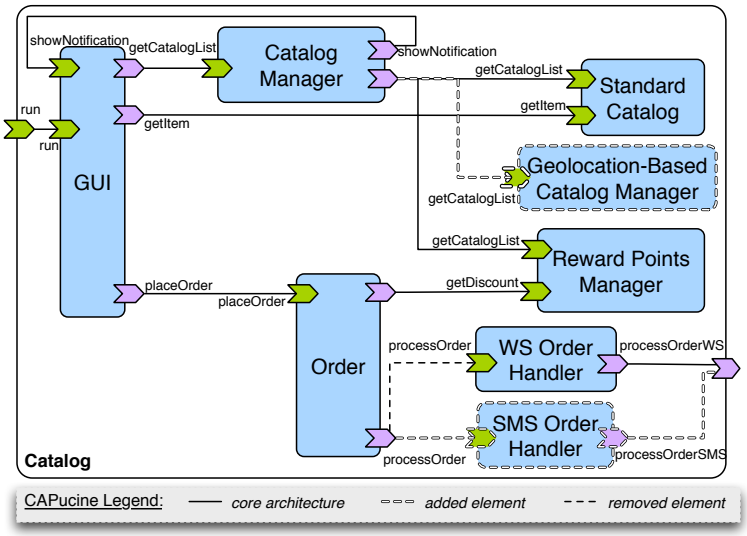

Figure 2.7: Mobile commerce application architecture using CAPucine.

CAPPUCINO adaptation server followed by the deployment and the start of the application server have already taken place. The deployment on a mobile device can then be made on-demand. For legal issues, the user intervention is required to request and then validate the deployment of software applications on their device. The artefacts deployed on a mobile device are lightweight versions of the ones deployed on an application server, *e.g.*, the lightweight version of FraSCAti called FraSCAme.

The deployment of the CAPPUCINO runtime on a mobile device is detailed in Figure 2.8. *(1)* We suppose that the service provider has preliminarily installed access points able to detect mobile devices in their vicinity. We do not detail here the functioning of these access points. We only assume that the user either declares their presence by passing their mobile device in front of a bluetooth antenna or has agreed beforehand to get automatically detected by providing the physical characteristics of their mobile device. Consequently, the interactive point notifies the adaptation server of the arrival of a mobile device. *(2)* The adaptation server sends to this mobile device an electronic message inviting the detected user to use available services. This message may be a SMS or a MMS message proposing to download the CAPPUCINO runtime to enable the access to personalized services. *(3)* The user gives an explicit authorization to download and deploy the CAPPUCINO runtime. *(4)* The adaptation server analyzes the type of the mobile device, prepares the adequate deployment plan and uploads the required software artefacts. *(5)* The adaptation server then transmits the CAPPUCINO runtime artefacts to the mobile device. *(6)* A volunteer action is required here from the end user:

The end user must explicitly accept the effective installation of the artefacts on their mobile device. *(7)* Following the installation of the artefacts, the CAPPUCINO runtime is automatically started. This step concludes the first phase of the deployment on a mobile device.

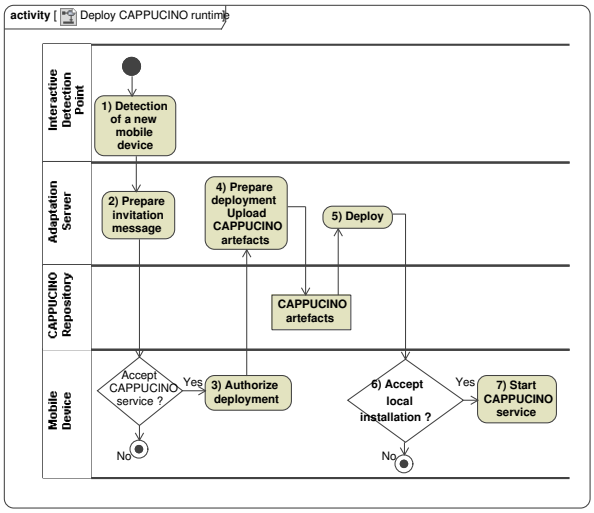

Figure 2.8: Deploying the CAPPUCINO runtime on a mobile device.

The second phase for deploying on a mobile device corresponds to the deployment of an application component. We show as an example the deployment of the `Order` component on Mary's mobile device (cf. Figure 2.7). The `Order` component will allow Mary to buy an item chosen in a catalog. The different deployment steps are depicted in Figure 2.9. *(1)* At the end of the first phase of the deployment of the CAPPUCINO runtime, this runtime informs the adaptation server that the deployment is completed. *(2)* The adaptation server then builds the list of the relevant Web Services to be proposed to Mary according to the type of her mobile device (*e.g.*, video feature available or not), her user profile (*e.g.*, recently accessed services, preferred services) and her user network connectivity (*e.g.*, Internet access and high rate network connection). *(3)* The adaptation server transmits the list of the available services to Mary. *(4)* Mary chooses the `Order` service in the list. *(5)* The adaptation server prepares the `Order` service deployment plan. It includes to upload the `WS Order Handler` and the `SMS Order Handler` components on the mobile device. The former allows an order to be handled directly by the remote web server when the network coverage is sufficient. The latter provides service continuity in case of network disconnection registering the order as a low-

memory consuming SMS message and placing it as soon as the network is available again. Both handlers are deployed but only `WS Order Handler` is initially bound to the `Order` service. *(6)* The deployment on the client side may require to preliminarily deploy a server part on the application server. *(7)* Moreover, the deployment on the server side may require to upload non CAPPUCINO artefacts from the repository of the business information system. For instance, Mary is registered as a new potential mobile clients and the payment service is activated. *(8)* Following the deployment of the required applicative services on the server side, the adaptation server sends the `Order` artefacts together with the `WS Order Handler` and the `SMS Order Handler` artefacts and drives their deployment on Mary's mobile device. This includes initially binding the `Order` service with the `WS Order Handler`. *(9)* Finally, Mary is notified of the availability of the `Order` service and can use it.

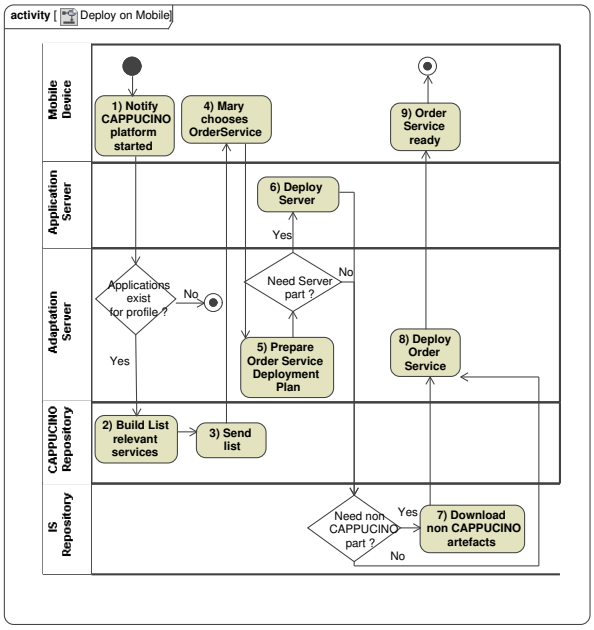

Figure 2.9: Deployment of an applicative service on a mobile device.

2.5.2 Dynamic Reconfiguration on a Mobile Device

We present in this section how reconfiguration is enabled at runtime illustrating the reaction path from the detection of an adaptation situation to the

execution of the reconfiguration actions on the mobile device. We describe two reconfiguration cases corresponding to different runtime reconfiguration services of the FraSCAti platform (see Section 2.4.2) namely the possibility to modify the wires between existing components and the possibility to instantiate new components.

Case 1: Change connections between existing components. As shown in Section 2.5.1, the `Order` service has been deployed when network connectivity was high, implying that the `Order` component has been bound to the `WS Order Handler`. However, the `SMS Order Handler` has also been deployed in order to tolerate any network disconnection and to provide service continuity transparently to Mary. When the network connectivity becomes lower than a threshold indicating there is a risk of disconnection, the adaptation server detects an adaptation situation and establishes the reconfiguration plan. The reconfiguration actions are performed on Mary's device corresponding to unbinding the `WS Order Handler` and binding instead to the `SMS Order Handler`.

Case 2: Deploy a new component and create connections. Consider Mary arriving nearby a shop where a Flash Sale offer is currently running. Mary has accepted to periodically notify the adaptation server with her geographical position. When the adaptation server detects that Mary can benefit from the nearby Flash Sale, it pushes the special offer by presenting a customized catalog. This new customized catalog is specifically prepared according to Mary's preferences as mentioned in her profile and requires the reconfiguration of the application components present on her mobile device. The `Geolocation-Based Catalog` component shown on Figure 2.7 is instantiated. It is then bound to the `Catalog Manager` component. These reconfiguration steps taking place on the mobile device are shown on Figure 2.10.

2.6 Related Works

This section compares our contribution with regards to the current state-of-the-art related to the definition of platform supporting the self-adaptation of service-oriented architectures.

MUSIC [25] defines a component-based platform supporting the seamless adaptation of mobile application according to *Quality of Service* variations. MUSIC integrates a versatile support for discovering, negotiating, and binding to remote services available in the surroundings of a mobile device. Unless CAPPUCINO, MUSIC adopts a device-centric approach where the device decides the adaptation to perform by applying a utility-based decision making

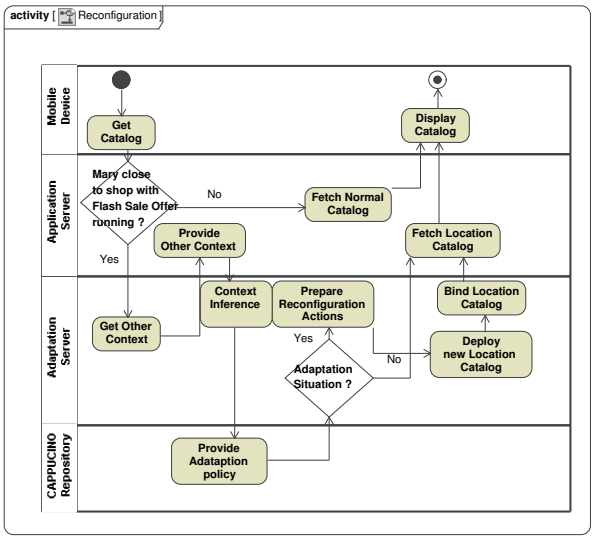

Figure 2.10: Reconfiguration of a mobile device.

mechanism [17]. CAPPUCINO offers a scalable adaptation model where the decision-making is under the control of an adaptation domain, which can decide to reason on the server-side or to deploy local and efficient control loops within the mobile devices.

The QuA project [12] investigates middleware-managed adaptation, which means that services are specified by their behavior, and then planned, instantiated and maintained automatically by middleware services in such a way that the behavioural requirements are satisfied throughout the service lifetime. Central in this approach, is a planning framework that searches through the space of possible configurations, and selects one that fits the current context of the application. The planning framework depends on the availability of meta-information about services that has not yet been instantiated, such as meta-information provided by techniques like MDA and ADL. While QuA integrates multiple implementation platforms at the adaptation middleware level, CAPPUCINO offers a versatile programming model based on SCA to leverage the adaptation middleware and thus offer a uniform model for the description and the adaptation of ubiquitous service-oriented architectures.

Adaptive Service Grids (ASG) [18] and VieDAME [20] are initiatives enabling dynamic compositions and bindings of services for provisioning adaptive services. In particular, ASG proposes a sophisticated and adaptive delivery life-cycle composed of three sub-cycles: *planning*, *binding*, and *enactment*. The entry point of this delivery life-cycle is a semantic service request, which consists of a description of what will be achieved and not which concrete ser-

vice has to be executed. VieDAME proposes a monitoring system that observes the efficiency of BPEL processes and performs service replacement automatically upon performance degradation. Compared to our CAPPUCINO platform, ASG and VieDAME focus only on the planning per request of service compositions with regards to the properties defined in the semantic service request. Thus, both approaches do not support a uniform session-based adaptation of both client-side and server-side services as our solution for ubiquitous applications does. Nevertheless, CAPPUCINO provides an extensible infrastructure to integrate ASG and VieDAME adaptive services and thus support the dynamic enactment of service workflows.

CARISMA is a mobile peer-to-peer middleware exploiting the principle of reflection to support the construction of context-aware adaptive applications [4]. In particular, services and adaptation policies are installed and uninstalled on the fly. CARISMA can automatically trigger the adaptation of the deployed applications whenever detecting context changes. CARISMA uses utility functions to select application profiles, which are used to select the appropriate action for a particular context event. If there are conflicting application profiles, then CARISMA proceeds to an auction-like procedure to resolve (both local and distributed) conflicts. However, CARISMA fails to integrate the diversity of service providers and legacy services. Nonetheless, the auction-like procedure used by CARISMA could be integrated in the CAPPUCINO control loop as a particular decision-making service.

Finally, R-OSGi extends OSGi with a transparent distribution support [23] and uses jSLP to publish and discover services [24]. The communication between a local service proxy and the associated service skeleton is message-based, while different communication protocols (*e.g.*, TCP or HTTP) can be dynamically plugged in. In contrast to R-OSGi, the discovery and binding frameworks of CAPPUCINO are open to support a larger range of discovery and communication protocols.

2.7 Conclusion

Ubiquitous environments strongly constrain the design and development of Context-aware Web Services. An adaptive runtime in such environments should be based on a control loop enclosing *i)* an adaptive execution kernel, *ii)* distributed context management facilities, *iii)* a context and situation adaptation analysis, *iv)* an adaptation policies and reconfiguration planning, and *v)* a reconfiguration engine orchestrating deployment and reconfiguration actions. The first contribution reported in this chapter is the CAPPUCINO runtime that provides the building blocks of a distributed control loop and relies on CBSE principles to ensure adaptation a service-oriented ubiquitous

systems. Moreover, the use of SCA as a common component model enables CAPPUCINO to perform adaptations of its own structure in order to better satisfy the requirements of the supported Web Service and therefore the end-user.

The second contribution of the chapter is FraSCAti, an execution kernel conforming to the SCA distributed component model for Web Services. SCA promotes independence from programming languages, interface definition languages, communication protocols, and non-functional properties. The FraSCAti execution kernel additionally enables the execution of SOA applications with advanced properties, such as the ability to reconfigure assemblies of SCA components at runtime.

The third contribution of this chapter builds upon the context manager COSMOS to build a lightweight middleware solution enabling the scalable and efficient dissemination of context information. SPACES acts as a communication middleware, which is responsible for disseminating the context information between distributed physical entities. This distribution is implemented in CAPPUCINO as an HTTP connector. By considering the context dissemination as a software connector, the realization of the composition of context information into software components is not impacted by the distribution concerns.

As a matter of future work, we plan to improve the scalability of the CAPPUCINO runtime by enabling the deployment of hierarchical control loops. In particular, the adaptation server should deploy a lightweight full-fledged local control loop within the mobile device in order to tolerate local autonomic adaptation like facing disconnection situations. The mobile devices could then react and reconfigure the application autonomously when leaving a given adaptation domain—*i.e.*, when loosing the connectivity with the adaptation server. In addition, adaptation domains could also be grouped as clusters to control a set of applications. Thus, a cluster of adaptation servers will belong to a common adaptation domain, which is dynamically reconfigured by the enclosing adaptation server. Another perspective consists in integrating the concerns of *Quality of Service* (QoS) in the development of context-aware Web Services. We foresee that this integration will be achieved at the application level and at the middleware level by reflecting the *Quality of Context* (QoC). Therefore, we plan to integrate the dynamic negotiation of QoS and QoC agreements during the adaptation process.

Bibliography

[1] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. `http://www.ietf.org/rfc/rfc3986.txt`, January 2005.

[2] David Booth, Hugo Haas, Francis Mccabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture. Working Group Note, February 2004.

[3] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP), May 2007. `http://www.w3.org/TR/SOAP`.

[4] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. CARISMA: Contex-Aware Reflective mIddleware System for Mobile Applications. 29(10):929–945, October 2003.

[5] David Chappell. Introducing SCA. white paper, Chappell & Associates, July 2007.

[6] Denis Conan, Romain Rouvoy, and Lionel Seinturier. Scalable Processing of Context Information with COSMOS. In *Proceedings of the 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, volume 4531 of *Lecture Notes in Computer Science*, pages 210–224, Paphos, Cyprus, June 2007. Springer-Verlag.

[7] Douglas Crockford. RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON). IETF RFC, 2006.

[8] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *Internet Computing, IEEE*, 6(2):86–93, 2002.

[9] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Springer Annals of Telecommunications, Special Issue on Software Components – The* Fractal *Initiative*, 64(1–2):45–63, January/February 2009.

0-8493-0052-5/00/$0.00+$.50

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

[11] Areski Flissi, Jérémy Dubus, Nicolas Dolet, and Philippe Merle. Deploying on the Grid with DeployWare. In *Proceedings of the 8th International Symposium on Cluster Computing and the Grid (CCGRID'08)*, pages 177–184, Lyon, France, May 2008. IEEE.

[12] Eli Gjørven, Romain Rouvoy, and Frank Eliassen. Cross-Layer Self-Adaptation of Service-Oriented Architectures. In *Proceedings of the 3rd International Middleware Workshop on Middleware for Service Oriented Computing (MW4SOC)*, pages 37–42, Leuven, Belgium, December 2008. ACM Press.

[13] Salim Hariri, Bithika Khargharia, Houping Chen, Jingmei Yang, Yeliang Zhang, Manish Parashar, and Hua Liu. The autonomic computing paradigm. *Cluster Computing*, 9(1):5–17, 2006.

[14] Xiaoming Hu, Yun Ding, Nearchos Paspallis, Pyrros Bratskas, George A. Papadopoulos, Paolo Barone, and Alessandro Mamelli. A Peer-to-Peer based infrastructure for Context Distribution in Mobile and Ubiquitous Environments. In *Proceedings of 3rd International Workshop on Context-Aware Mobile Systems (CAMS'07)*, Vilamoura, Algarve, Portugal, November 2007.

[15] IANA. MIME Media Types. `http://www.iana.org/assignments/media-types`, March 2007.

[16] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1), January 2003.

[17] Jeffrey O. Kephart and Rajarshi Das. Achieving Self-Management via Utility Functions. *IEEE Internet Computing*, 11(1):40–48, 2007.

[18] Dominik Kuropka and Mathias Weske. Implementing a Semantic Service Provision Platform — Concepts and Experiences. *Wirtschaftsinformatik Journal*, 1:16–24, 2008.

[19] Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. DREAM: a component framework for the construction of resource-aware, reconfigurable MOMs. In *Proceedings of the 3rd Workshop on Adaptive and reflective middleware (ARM'04)*, pages 250–255, Toronto, Ontario, Canada, 2004. ACM.

[20] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-intrusive monitoring and service adaptation for WS-BPEL. In *Proceedings of the 17th International Conference on World Wide Web (WWW)*. ACM, 2008.

[21] Open SOA. Service Component Architecture, November 2007. `http://www.osoa.org/display/Main/Service+Component+Architecture+Home`.

[22] Carlos Parra, Xavier Blanc, Laurence Duchien, Nicolas Pessemier, Rafael Leano, Chantal Taconet, and Zakia Kazi-Aoul. *Dynamic Software Product Lines for Context-Aware Web Services*, chapter 1. Chapman and Hall/CRC, 2009.

[23] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: Distributed Applications through Software Modularization. In *Proceedings of the*, volume 4834, pages 1–20, Newport Beach, CA (USA), November 2007.

[24] Jan S. Rellermeyer and Markus Alexander Kuppe. jSLP, 2009. `http://jslp.sourceforge.net`.

[25] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. *MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments*, volume 5525 of *Lecture Notes on Computer Science Hot Topics*, chapter 2, pages 164–182. Springer-Verlag, 2009.

[26] Romain Rouvoy, Denis Conan, and Lionel Seinturier. Software Architecture Patterns for a Context Processing Middleware Framework. *IEEE Distributed Systems Online (DSO)*, 9(6):12, June 2008.

[27] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *Proceedings of the 6th IEEE International Conference on Service Computing (SCC'09)*, Bangalore Inde, 2009. IEEE.

[28] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. A Concise Introduction to Autonomic Computing. *Advanced Engineering Informatics*, 19(3):181–187, July 2005.