Components in an Adaptive and QoS-based Architecture

Claudia Raibulet, Francesca Arcelli, Stefano Mussino, Mario Riva, Francesco Tisato, Luigi Ubezio
Università degli Studi di Milano-Bicocca, DISCo – Dipartimento di Informatica Sistemistica e Comunicazione
Via Bicocca degli Arcimboldi, 8, I-20126, Milan, Italy
Tel: +39-02-6448-7803
{raibulet, arcelli, stefano.mussino, mario.riva, tisato, ubezio}@disco.unimib.it

ABSTRACT

This paper aims to present the main software components we have developed in the context of the ARM (Adaptive Resource Management) project at University of Milano-Bicocca for an adaptive, distributed, service-oriented architecture. The goal of ARM is to manage the resources of a system in a way that enables it to dynamically identify and execute services on the available resources. Our approach chooses the most appropriate resource that is able to execute a service with the requested qualities of service (QoSs). To achieve adaptivity, ARM uses reflection at the architectural level. Exploiting the reflective representation of the system’s resources and their related QoSs, ARM may organize them accordingly to various criteria and evaluate them based on their QoSs features and their potentiality in executing a requested service with the requested QoSs. To validate the ARM concepts, a prototype based on the peer-to-peer paradigm is currently under development. It aims to provide an adaptive support when using the resources available in our department.

Categories and Subject Descriptors
Primary Classification: D. Software; D2: Software Engineering; D2.11: Software Architectures; Subject: Domain-specific architectures

Additional Classification: D. Software; D2: Software Engineering; D2.11: Software Architectures; Subject: Data Abstraction

General Terms: Design.

Keywords: Adaptivity, reflection, quality of service, view, strategy.

1. INTRODUCTION

One of the most challenging issues raised by nowadays information systems is to adapt themselves dynamically and automatically in the attempt to accomplish the anytime, anywhere paradigm in a constantly changing reality. In this context, through our approach we provide support to identify, choose, and exploit the appropriate available resources able to satisfy users’ requests with different levels of qualities of services (QoSs) [1, 9, 20] in a dynamic mobile-enabled environment.

The idea behind the ARM project has emerged from common scenarios, which are met within companies or universities such as: print on the most nearest A3-format printer, display an image on a monitor supporting a resolution greater than a specified value, provide the list of the available scanners in this building at the first floor, send this message using the most appropriate available device (e.g., mobile-phone, PDA) and network connection (e.g., wireless, LAN), and so on. These requirements are common when resources are shared among users who, typically, do not have a fixed location and, thus, they do not know which are the suitable available resources they can exploit to accomplish their needs.

Our approach to address runtime adaptivity and its related issues exploits reflection at the architectural level. Reflection [15] is defined as the activity performed by a system when doing computation about itself. Essentially, reflection enables a system to observe and control itself through appropriate metadata. This is particularly useful in auto-adaptive systems [6, 16], which should modify their own features and behavior at runtime as a result of the changes occurred internally (i.e., modifications of their own structure) or externally (i.e., modifications in their running environment).

Generally, architectural reflection [8, 23] may introduce additional layers [5, 7] or software components [14, 17]. These additional elements play an intermediary role between the representation of a system and applications. They enable applications to adapt to the systems’ features, and vice versa, systems to adapt (whenever possible) to the applications’ requirements.

In our opinion, the main issue raised by the reflective architectures is to identify, design, and manage the reflective knowledge (the set of meta-representations of the system’s entities) in such a way to overcome the possible disadvantages of such systems (i.e., a significant increase of the number of software components which reduces the overall efficiency, or modifications within the reflection components which may cause overall damages).

The aim of this paper is to present the most important components of our reflective architecture for adaptive, service-oriented [12] systems. Our approach defines reflective layers which represent and manage the QoSs of the components of a system. In addition to the QoSs, our reflective layers capture further properties of the system’s components. The role of these properties is twofold: (1) they model important aspects of the system (i.e., structural, topological, location) which are exploited in a similar way to the QoSs and which are necessary to achieve adaptivity, and (2) they allow an efficient organization and management of the reflective knowledge associating it various views with various semantics.
The rest of the paper is organized as following. Section 2 provides an overview of the ARM architecture describing its main components. Section 3 presents the views and their roles. Section 4 describes how ARM achieves adaptivity. A prototype implementing the ARM concepts and exploiting the JXTA technology is presented in Section 5. Conclusions and further works are dealt within Section 6.

2. ARM OVERVIEW

ARM defines various architectural layers. Each layer interacts with its neighbours, being independent of the upper one and representing an abstraction of the underlying one. As shown in Figure 1, there is an application component that presents a list of services to the user. The available services depend on the resources of the underlying system. Users’ requests arrive to the service manager, which may interrogate both the local ARM node and/or the remote ARM-enabled nodes by forwarding the request to the network manager. Once the service manager identifies the type of the request (which may be of one of the following three types: service execution, inspection of available services/resources, and change of the state of one or more resources), the last arrives to the reflective view manager. At this level, complex services are decomposed into their sub-services and the related QoSs into sub-QoSs (for example, a send e-mail service may be decomposed into type message and send message). Based on the service required and its QoSs, the reflective view manager determines how to catalog better the reflective knowledge to search for the most appropriate resource that provides the service.

Figure 1. The Architecture of ARM

The mechanism that allows the management of the reflective knowledge is represented by views. Views play a fundamental role in our approach because they organize the reflective knowledge based on various semantics, and improve the search of the best solution for the current request. The reflective view manager may create dynamically views on the reflective entities to address efficiently a request. Views identify the most appropriate resources to execute sub-services. By composing these partial results we obtain the resource or resources which satisfy properly a request on the local ARM-enabled node. This local result is compared with the remote results provided by the network manager in order to obtain the best overall solution. Eventual modifications performed on the identified reflective entities for the execution of the requested services are propagated to the functional objects through a causal connection [15] mechanism. Finally, the service manager requires the executor manager to execute the service exploiting the resources identified by the reflective view manager.

2.1 Modularity of the Architecture

Our architecture may be exploited in a wide range of application domains, which range from multimedia to telemedicine, from telecommunications to disaster recovery. Thus, we have considered necessary to introduce a communication interface between the entire platform and the possible applications that may exploit it. This interface is represented by the service manager (see Figure 2), which offers services not only to the local applications, but also to the other ARM-enabled nodes. In this way, local and remote request are addressed identically by the service manager.

Figure 2. The Communication Interface of ARM

2.2 Representation of the Reflective Knowledge

Reflective knowledge is modeled through reflective objects and their related QoSs (see Figure 3). In addition, we have introduced the concept of property of a resource. Properties express the characteristics of the resources that are not directly measurable at runtime (the measurable information represent the state of a resource), but which are exploited together with the QoSs to achieve adaptivity.

Figure 3. Reflective Objects, QoS, and Properties

ARM defines three main properties:

- the structural property, which represents the physical structure of a resource;
- the topological property, which represents the connections a resource has with other network resources;
- the location property, which represents the current location of a resource.

Additional properties can be easily added. For example, the cost or the provider of a service may be modeled as properties (see Figure 4). When asking for a service, a user may indicate a maximum cost s/he agrees to pay for receiving the service. Or, sub-services of a composed service, if supplied by the same provider, may have a lower cost than those supplied by different providers.
To summarize, a reflective object has associated its QoSs and properties. A reflective object is aware only of its QoSs. Even if properties are directly linked to the reflective object, the last is not aware of them. From the implementation point of view, this means that a reflective object has a reference to its QoSs, but it has no references to its properties. In this way, reflective objects and QoS model that information which is causally connected to the base objects (we called them Functional Objects – see Figure 1), while properties provide that information which cannot be obtained through a causal connection, but which is exploited at the reflective layer to achieve dynamic adaptivity: how resources are composed, where resource are physically, and which their structure is.

Figure 5. ARM Views

3. MANAGEMENT OF THE REFLECTIVE KNOWLEDGE
The ability of ARM to identify the most appropriate available resources to execute a service is based on the mechanisms introduced by the views on the reflective knowledge. A view is defined as an organizational structure on the reflective objects which has its own semantics and its own computational strategies to evaluate the elements under its control [19].

ARM defines four views: service, structural, location, and topological (see Figure 5). Further views can be easily added into the architecture because the modularity of ARM does not limit our solution to only four views, but enables its extension by implementing the abstract interface of the reflective views.

As all the reflective entities, also views have associated QoSs, their values representing an abstraction and a summary of the QoSs of the reflective objects (see Figure 5). Furthermore, each view has a set of strategies to manipulate the reflective objects it manages.

A reflective object may be used in various views, but it has only one representation in the system. Each view contains references to its reflective objects. A view can be seen as a reflective entity at a higher level of abstraction, which is characterized by the absence of the causal connection mechanism and by the types of its QoSs. At this level, QoSs are not measured, but calculated based on the QoSs of the underlying reflective objects.

3.1 The Services View
The main view of ARM is the services view (see Figure 6). It catalogs the resources of a system based on the services they provide. For each type of service, a view that represents its features is created. For instance, we may have a view for the print service, one for the transmit service, one for the display service, and so on. Each view manages a collection of heterogeneous reflective objects, which are able to provide the same type of service.

Figure 6. The Services View

The QoSs of a service view represent the high level QoSs associated to a service at the application level. The mapping between the high-level QoSs associated to services and the low-level QoSs associated to the reflective objects is performed by the strategies of each view (ServiceStrategy – see Figure 6). A view strategy may inspect the reflective objects to obtain information about their abilities to perform a service, or may identify the most appropriate resource(s) that can fulfil a requested service (BestEffortStrategy – see Figure 6).

An example of the print view is described in Figure 7. The QoSs we have considered for a print service are: resolution, printing color, and format. A possible request may claim for a print service with a high resolution, coloured, and on an A3-format. The high
resolution QoS of the service is translated by strategies into a resolution greater than 1200x1200 dpi of the reflective objects. The strategy chooses from the available printers the one that meets better the user request.

Figure 7. The Print Service View

3.2 The Structural View
The structural view enables the evaluation of the resources of a system based on their physical structure. This view is used to extract information about the physical composition that influences the values of the QoSs. For example, it is possible to verify whether a printer is connected directly to a PC being part of its structure, or whether it is accessible through a network interface.

Each view represents a reflective composite component, which manipulates a collection of sub-components, which may be either simple components or composite views (see Figure 8). This structure creates reflective trees able to represent any hardware configuration.

Figure 8. An Example of a Composite Structural View

The QoSs related to the structural view represent abstractions of the QoSs of the resources it manages. The most appropriate resource is identified based on the structure and its structural features, and not on the functionalities it provides.

3.3 The Location View
The view of locations is exploited to evaluate the location of the resources based on the current location of the user and the location where the service will be executed. One of the main issues of ARM is to address mobility, hence to be able to manage mobile resources as well as resources available at various locations.

The view of locations catalogs the reflective objects based on their locations. In the context of ARM, locations may be expressed both in geographical coordinates and in building coordinates, which consist of the name of the street, civic number, flour number, and room number [2]. This view groups together the resources available at the same location. It enables the extraction of information regarding the possible changes of the location of the resources and the distance between various resources.

For example, a user may want to print an image on the most nearest color-enabled printer. The location of the printer plays an important role in solving this request. In this case, location has priority in front of other QoSs of a printer.

3.4 The Topological View
The topological view manages the reflective objects based on the connections between the various resources (see Figure 9). It provides information about which and how resources are connected to each other, through which communication channels, and which are the QoSs of these channels. This view plays an important role in the communication between the ARM-enabled nodes in the network, particularly in a peer-to-peer solution.

Figure 9. An Example of a Topological View

For example, printing a large file on a high-speed printer connected to the user’s computer through a reduced bandwidth connection may be slower than printing a large file on a medium-speed printer connected directly to the user’s computer.

3.5 Modularity of Views
All views have been designed to be easily changeable and extensible. They are very important components of our architecture, but they should not limit the functionalities and the extensibility of ARM. To achieve these goals, views implement a common abstract interface, which defines their base behaviour and have associated a set of strategies to compute the best results. The strategies associated to a view exploit the Strategy [13] design pattern, which defines not only their interface, but allows also a dynamic choice of the strategy to apply to fulfill a request.

4. ACHIEVING ADAPTIVITY IN ARM
In ARM, adaptivity is achieved through three main elements: (1) the reflective objects, which capture the non-functional features of the system’s resources and which are causally connected to the functional objects, (2) the views, which organize efficiently the reflective objects based on various semantics, and (3) the strategies, which provide the decision support for various tasks at various layers. All these elements of our approach are defined in such a way to be easily reusable and extensible. Their implementation exploits four design patterns [13] (chain of
responsibility, strategy, composite, and observer), which have further improved their reusability and extensibility.

4.1 Reflective Objects
Reflective objects represent the basic elements of our reflective infrastructure. They are modelled independently of the other components of the architecture. The causal connection between them and the functional objects (see Figure 3) allows reflective objects to automatically adapt themselves to the modifications occurred at the functional objects, and vice versa, allows functional objects to adapt themselves (whenever possible) to the reflective objects (which mirror the application’s need for adaptivity). Note that resources are observable or controllable, or both, hence adaptivity takes into consideration also these aspects.

The causal connection is modelled through the Observer design pattern as shown in Figure 10. This pattern provides an efficient mechanism for the synchronization between objects, or more precisely, for the synchronization of the state of objects. The update() method allows reflective objects to adapt themselves according to their corresponding functional objects, while force() tries to constrain functional objects to adapt (whenever possible) themselves to the reflective objects.

![Figure 10. The Causal Connection Mechanism in ARM](image)

4.2 Views
Views represent our fundamental mechanism to exploit reflective objects. The main view is the services one. As previously mentioned, it organizes reflective objects based on the services they related functional objects may execute. In an architecture without views, reflective objects should have associated a description of the services they provide. Through views we obtain at least three advantages: a uniform and common mechanism to organize reflective objects based on various criteria, the possibility to extend the number of views on the reflective objects, and to not include external information in the representation of the reflective objects (i.e., location, structure, topology, etc.). Note that the services view is used in every request that claims for adaptivity, while other views may be not always exploited.

The implementation of the services view is based on the chain of responsibility design pattern (see Figure 11), which addresses two main problems: the dynamic control of a collection of service views, and the execution of composed services.

The first problem occurs during the initialization phase of the architecture when the services view is created based on the functionalities offered by the resources in the system. The collection of views has significant dimensions, and using additional structures to store each type of view may become very expensive. The chain of responsibility pattern suggests using a chain of views connected between them by references (see Figure 11). The reference of the first view in the chain is hold by the reflective view manager.

![Figure 11. The Chain of Service Views](image)

4.3 Strategies
In the context of ARM, decisions are modelled through strategies. Each view has associated one or more strategies that implement the logic necessary to take decisions (see Figure 12). For example, each reflective view has associated best-effort strategies through which the most appropriate resources to execute an actual service are chosen. Or, the service manager has strategies to decide, based on the service type and its related QoSs, whether a request involves local and/or remote resources.

![Figure 12. Strategies](image)
classes representing strategies have to provide an actual implementation of this interface (see Figure 12).

4.3.1 Composed Strategies

Strategies may have complex structures. Furthermore, they cannot be defined exhaustively until the implementation phase. To prevent and address their complexity we have chosen the Composite design pattern to define their structure. Figure 13 shows an example of a service strategy that exploits this pattern. In this case, a service strategy may be a simple one or a composition of strategies, which further may be simple or composite. This mechanism is particularly useful when complex services are requested. The strategy that analysis the complex service is actually a composition of strategies related to the sub-services of the required one. This improves significantly the implementation of strategies, by requiring the definition of the simple strategies and the definition of the complex one as composition of the already defined simple strategies. Furthermore, modifications of the simple strategies are automatically propagated to the complex once that exploits them.

5. VALIDATION OF ARM

A prototype implementing the ARM concepts is currently under development. We have chosen to implement a totally distributed solution based on the peer-to-peer (P2P) approach provided by JXTA [22]. Our choice has been determined by the following considerations. JXTA defines the P2P basic concepts and a common infrastructure for P2P applications. It defines also a set of core protocols to publish and discover services and resources, and to communicate between peers. Communication between peers exploits the XML standard format, which may be used also to describe services, resources, and QoSs.

In the context of our prototype a peer is defined as a component that has computational abilities and that implements the ARM concepts [22]. For example, a laptop or a PC is considered a peer, while a printer cannot be considered a standalone peer (it is part of a peer, which contains also the server/PC to which it is connected).

We have defined an ARM group that contains all the ARM-enabled peers. Currently, the ARM group is composed of two types of peers: simple and rendezvous. The last provides support to discover other available peers in the network. Services provided by peers are published through advertisements. Requests of services are modeled as described in Section 5.1.

The current version of our prototype considers two services: printing and displaying. Printers are connected both to servers and to various PCs. As printers, also monitors and projectors cannot be considered standalone peers. A display service may request the visualization of an image with various possible QoSs.

5.1 Requests of Services

During the development of the prototype the necessity of a proper specification of the users’ requests has raised. The definition of requests is described by the class diagram shown in Figure 14. The user may specify only the service and eventually the input data. For example, print this document. Or, s/he may choose also the resource that will execute the service. For example, print this document on the hp347bn1 printer. These are two cases in which no adaptivity is required. Hence, the service arrives to the service manager, which forwards the request to the execution manager (see Figure 1).

Requests that claim for adaptivity contain QoSs associated to the required service. If the user specifies on which resources to execute the service, adaptivity consists in setting the QoSs of the indicated resources on values equals or as close as possible to the desired ones. For example, print this image on the hp347bn1 printer with a high resolution and coloured.
to a type of service or to a type of view [19]. This makes them highly reusable. Moreover, the organization of the reflective objects in MAIS is based on their physical structure and not on the services they provide. ARM introduces properties and views, and dynamic mechanisms to manage the reflective knowledge which leads to the improvement of the runtime adaptivity. Moreover, ARM makes use of a set of design patterns, which enhance its flexibility and enable its extensibility.

The idea of considering various aspects of the reflective knowledge occurs also in [11], which describes the features of the next generation reflective middleware. As required by the authors of this paper, our views capture independent and orthogonal aspects of the reflective knowledge organizing the reflective objects based on their QoSs and properties.

The aim of our current prototype is to validate the ARM concepts. Performance is not our main issue at this stage. Further work will focus on extending the range of services and to integrate ARM in various application domains such as multimedia, telemedicine, and disaster recovery. Moreover, we will address the resource allocation and negotiation aspects. Future work includes also an implementation of ARM using the Jini technology and a comparison between these two different solutions in order to evaluate their advantages and disadvantages.

7. REFERENCES