Department of Computer Science
University College London
University of London

Reflective Mobile Middleware for Context-Aware Applications

Licia Capra

Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
at the University of London

October 2003

Abstract

The increasing popularity of mobile devices, such as mobile phones and personal digital assistants, and advances in wireless networking technologies, are enabling new classes of applications that present challenging problems to application designers. Applications have to be *aware* of, and *adapt* to, variations in the execution context, such as fluctuating network bandwidth and decreasing battery power, in order to deliver a good quality of service to their users.

We argue that building applications directly on top of the network operating system would be extremely tedious and error-prone, as application developers would have to deal with these issues explicitly, and would consequently be distracted from the actual requirements of the application they are building. Rather, a middleware layered between the network operating system and the application should provide application developers with abstractions and mechanisms to deal with them.

We investigate the principle of *reflection* and demonstrate how it can be used to support context-awareness and dynamic adaptation to context changes. We offer application engineers an abstraction of middleware as a dynamically customisable service provider, where each service can be delivered using different policies when requested in different contexts. Based on this abstraction, current middleware behaviour, with respect to a particular application, is reified in an *application profile*, and made accessible to the application for run-time inspection and adaptation. Applications can use the *meta-interface* that the middleware provides to change the information encoded in their profile, thus tailoring middleware behaviour to the user's needs. However, while doing so, *conflicts* may arise; different users may have different quality-of-service needs, and applications, in an attempt to fulfil these needs, may customise middleware behaviour in conflicting ways. These conflicts have to be resolved in order to allow applications to come to an agreement, and thus be able to engage successful collaborations.

We demonstrate how *microeconomic* techniques can be used to treat these kinds of conflicts. We offer an abstraction of the mobile setting as an *economy*, where applications compete to have a service delivered according to their quality-of-service needs. We have designed a mechanism where middleware plays the role of the auctioneer, collecting bids from the applications and delivering the service using the policy that maximises social welfare; that is, the one that delivers, on average, the best quality-of-service.

We formalise the principles discussed above, namely reflection to support context-awareness and microeconomic techniques to support conflict resolution. To demonstrate their effectiveness in fostering the development of context-aware applications, we discuss a middleware architecture and implementation (CARISMA) that embed these principles, and report on performance and usability results obtained during a thorough evaluation stage.

Acknowledgements

First and foremost, I would like to thank my supervisors, Wolfgang Emmerich and Cecilia Mascolo. When I first contacted them about three and a half years ago, I was rather confused about what to do in my life; I wanted to come to London, but I was not really planning to pursue a PhD. Their enthusiasm in doing research convinced me to join the Software Systems Engineering group, first as a Research Assistant, and then as a PhD student. Their guidance, experience and encouragement have been invaluable for my research. I would like to thank Cecilia and Wolfgang even more for their friendship, for having been so supportive, for the long talks we had when I was in a crisis, and for having given me a house to live in when I was left homeless (I did not forget it . . . thanks again :)

Special thanks to Anthony Finkelstein for his very creative spirit that made our discussions so stimulating and profitable. I would like to thank Steve Hailes for the precious feedback he gave me in my first and second year viva; Ken Binmore and Pedro Rey-Biel for disclosing me some of the mysteries behind microeconomic theory, and Clare Gryce for the immense work she did in proof reading my thesis. I would also like to thank the Department of Computer Science of University College London, for providing me with the resources to develop this thesis.

I have been very fortunate to be in the company of a wonderful group of people here at UCL that have made this period of my life so enjoyable. In particular, I would like to thank Stefanos for always fixing my computer with his technical skills (what linux kernel version am I running?); Rami for the year planners and the baklava (I still owe you a couple of kilos); Judy for the crazy shopping we did in Orlando, and Nicola for being such a great gossip-mate :). Thanks to Christian, Martin, Nima, Danila, Carina, Gena, Torsten, Daniel, Mirco, Andy D., Andy H., and the whole 104 and 203.

I am indebted to my family, Loredana, Manillo and Marco, for their love and unconditioned support, and for more than I will ever be able to express.

Last but not least, a very special thank to Luca, the best anti-stress I ever came across with :)

Financial support for this work has been provided by Zuhlke Engineering Ltd.

A Loredana e Manillo
per avermi messo sulla bicicletta
e insegnato a pedalare

A Luca
per essere il mio vento a favore

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Portable devices, such as palmtop computers, mobile phones, personal digital assistants, digital cameras and the like, are gaining wide popularity. Their computing capabilities are growing quickly, while their size is shrinking, allowing their pervasive use in our everyday life. Wireless networks of increasing bandwidth allow these mobile units to aggregate and form complex distributed system structures, as well as to seamlessly connect to fixed networks while they change location. The combined use of these technologies enables people to access their personal information as well as public resources *anytime* and *anywhere*.

The image that we used to associate to distributed systems, that is, of a relatively static network structure made of fixed and powerful hosts, permanently connected to the network and executing in a stable environment, cannot be applied to the mobile scenario. Although the computing capabilities of portable devices are growing quickly, their CPUs, memory size, as well as network bandwidth and latency, will continue to lag one or two orders of magnitude behind their fixed counterpart. The mobile network topology is now very dynamic, as nodes may come and leave freely. Hosts can be added, deleted or moved in a distributed system too, but the frequency at which this happens is orders of magnitude lower than in mobile settings; the size of wireless devices, in fact, has shrunk to the point that most of them can be carried in a pocket and moved around easily. In a fixed distributed environment, context is more or less static: bandwidth is high and stable, location almost never changes, services may vary, but the discovery of available services is easily performed by forcing service providers to register with well-known location services, such as LDAP or DNS. The execution environment of a portable device is extremely dynamic and subject to rapid, unpredictable and drastic changes, instead: location is no longer fixed and, depending on location, the services and hosts in reach vary, as well as the quality of the network connection. The performance of wireless networks (i.e., GSM, GPRS networks, satellite links, WaveLAN [Held, 2000], HiperLAN [Networks, 2000], Bluetooth [Bray and Sturman, 2000]) vary greatly depending on the protocols and technologies being used; for example, if a PDA is equipped with both a WaveLan network card and a GSM interface, connection may drop from 11Mbs bandwidth when close to a base station (e.g., in a meeting room), to less than 9.6 Kpbs when outdoor in a GSM cell (e.g., in a car on our way home). Reasonable bandwidth may be achieved if, for instance, the hosts are within a few (hundred) meters of their base station, and if they are few in number in the same base station cell (some technologies provide a shared bandwidth to the hosts in the same cell, so that, if they grow in number, the quality-of-service rapidly drops). However, if a device moves to an area with no coverage or with high interference, bandwidth may suddenly drop to zero and the connection may be lost. Unpredictable disconnections can no longer be considered an exception, but rather they become part of normal wireless communication.

Mobility breaks down the concept of *stability* that was prevalent in traditional distributed systems, and a new concept of *dynamicity* arises. This loss of stability in the physical infrastructure introduces challenging problems to application designers. Applications are required to react to frequent changes in the environment, such as change of location or of resource availability, variability of network bandwidth (that will remain by orders of magnitude lower than in fixed networks), and so on. They need to face temporary and unannounced loss of network connectivity, when their host is on the move. They are usually engaged in rather short connection sessions; they need to discover other hosts and services in an ad-hoc manner. They have to support different communication protocols, according to the wireless links they are exploiting.

Building mobile applications directly on top of the network operating system available on portable devices, would be extremely tedious and error-prone. Application developers would have to deal explicitly with all the requirements introduced by mobility (partly listed above). As a result, constructing and maintaining mobile applications would become less efficient and cost-effective, as developers, overwhelmed by the new complexities, would be distracted from the actual requirements of the application they are building.

When developing applications for traditional distributed systems, designers do not have to explicitly deal with the problems related to distribution, such as heterogeneity, scalability, resource sharing and fault tolerance. *Middleware* layered between the network operating system and the application, provide application designers with abstractions, such as remote evaluation, message passing, transactions, exception handling, and so on, that hide away the complexity of distributed system construction from application developers as much as possible. For example, developers do not need to know the location of a distributed component to request a service from it, nor do they need to care of potential network failures: the communication primitives that the middleware provides already take care of these issues, so that developers are offered an image of the distributed system as a single integrated computing facility [Emmerich, 2000]. In other words, distribution becomes *transparent* [ANSA, 1989].

Different types of middleware technologies exist on the market that can be roughly classified based on the abstractions they implement. For example, the remote evaluation abstraction is offered by current object-oriented technologies, such as implementations of OMG CORBA [Pope, 1998] (e.g., IONA'S Orbix [Baker, 1997] and Borland's VisiBroker [Natarajan et al., 2000]), the CORBA Component Model (CCM) [OMG, 1997], Microsoft COM [Rogerson, 1997], Java/RMI [Pitt and McNiff, 2001] and Enterprise JavaBeans [Monson-Haefel, 2000]). The message passing abstraction has been implemented by message-oriented technologies, like IBM MQSeries [Redbooks, 1999] and Sun's Java Message Queue [Monson-Haefel et al., 2000]. Transactions are offered by transaction-oriented middleware, such as IBM CICS [Hudders, 1994] and BEA's Tuxedo [Hall, 1996]. These technologies are widely adopted in industry, thus proving the effectiveness of the abstractions they implement in enhancing the development of distributed applications.

Although successfully used in stationary distributed systems, we argue that these abstractions and mechanisms have only limited applicability in the mobile setting. Traditional distributed systems are characterised by a stable network infrastructure, where hosts are permanently connected to the network through high-bandwidth links. This infrastructure naturally promotes synchronous communication styles, such as distributed transactions, object requests or remote procedure calls. For example, object-oriented middleware systems, such as CORBA [Pope, 1998], support synchronous point-to-point communication abstractions that require a rendez-vous between the client asking for a service, and the server delivering that service. Mobile devices, instead, often connect to the network opportunistically for short periods of time, mainly to access some data or to request a service. Even during these periods, the available bandwidth is by orders of magnitude lower than in fixed distributed systems, and it may suddenly drop to zero if an area with no network coverage is entered. It is often the case that the client asking for a service, and the server delivering that service, are not connected at the same time, because of voluntary disconnections (e.g., to save battery power), or forced ones (e.g., loss of network coverage). Mobile computing middleware should take the dynamicity of the network infrastructure into account and promote asynchronous communication styles instead.

Moreover, the principle of transparency that has driven the design of traditional middleware can no longer be applied. The execution context of a fixed distributed system is rather static: the location of a device seldom changes, the topology of the system is preserved over time, bandwidth tends to be stable, and so on. Because of the static nature of their context, traditional middleware systems have taken the approach of hiding environmental information from application developers, and of managing internally the rare changes that happen, so that developers do not have to deal with them explicitly. For example, one way to achieve scalability in distributed systems is through replication: a component is copied onto different hosts, and these copies are kept synchronised with the master. If the copy that an application is currently accessing is no longer available, because, for example, of a sudden network failure, middleware exploits its (mainly static) knowledge about network topology to transparently redirect application's requests to another accessible copy. While this approach is plausible in a stable environment, it cannot be adopted in a mobile setting. By providing transparency, in fact, middleware must take decisions on behalf of the application; this is inevitably done using built-in mechanisms and policies that cater for the common case rather than the high levels of heterogeneity and dynamicity intrinsic in mobile environments. In addition, applications may have valuable information that could enable the middleware to execute more efficiently. For example, because of frequent disconnections, replica cannot always be kept synchronised in a mobile setting; if access to a copy is suddenly lost, application knowledge should be exploited to decide which of the available copies to contact (e.g., a synchronised copy may be less preferred to an out-of-date one, if accessible only through a poor quality network link).

Finally, the interaction primitives provided by traditional middleware systems are usually too heavy-weight to be used on portable devices. In order to deliver high quality-of-service to applications, traditional middleware systems offer powerful abstractions (e.g., transactions) that often prescribe heavy-weight primitive implementations. While this burden can be borne by the powerful machines that constitute the fixed scenario, resource limitations on portable devices demand abstractions that can be provided through light-weight primitive implementations instead.

In order to enhance the development of mobile computing applications, different principles (other than transparency) have to be investigated, and new abstractions and mechanisms have to be developed that can be used by middleware practitioners to build a new class of middleware tailored to the mobile setting.

1.2 Towards a Mobile Computing Middleware

In order to understand what abstractions and mechanisms are needed to enhance the development of mobile computing applications, we take an application developer perspective. In the following, we isolate some of the new complexities that application developers have to face in building and maintaining this class of applications, and that we aim to tackle in this thesis.

Context-awareness and adaptation to context changes

Unlike fixed distributed systems, mobile systems execute in an extremely dynamic context. By context, we mean everything that can influence the behaviour of an application, from resources within the physical device, such as memory, battery power, screen size, and processing power, to resources outside the physical device, such as location, remote service available, network bandwidth and latency. Mobile applications need to be *aware* of changes occurring in their execution context, and *adapt* to these changes, to continue to deliver a high quality-of-service to their users.

In order to achieve context-awareness, and enable adaptation to context changes, application developers would have to face the following issues: first, they would be required to deal with heterogeneous physical sensors to gather context information. This information can vary considerably from sensor to sensor (for example, location information can be gathered using the Global Positioning System outdoors, or infrared and radio frequency indoors), and can require various processing in order to be interpreted correctly by different applications (e.g., absolute location, relative location). Once this information has been gathered and processed, the behaviour of the application has to be adapted to the newly available context configuration; a mechanism would therefore be needed to detect context changes of interest to the application, and to carry out the adaptation strategy required. However, this mechanism cannot be simply based on a static mapping between possible context configurations and corresponding adaptation strategies. First, we cannot expect application developers to foresee all possible execution contexts, second, user's needs may vary over time, and thus require different adaptation strategies at different moments.

The construction and maintenance of mobile applications would be hampered and considerably slowed down if application developers had to deal with these issues. Rather, a middleware software layer developed between the network operating system and the application should provide application developers with an abstraction of context that hides the heterogeneity of physical sensors from applications, and enables them to easily specify which portion of context they are interested in. Also, it should provide a mechanism to dynamically specify which adaptation strategies have to be undertaken in answer to context changes, based on varying user's needs. Middleware, on behalf of the applications, should then interact with the physical sensors to gather context information, process this information in an application-specific manner, check whether a configuration of interest to the application has been entered, and if so, execute the adaptation strategy required.

Quality-of-Service (QoS) conflict resolution

Applications running on mobile devices often have limited amounts of resources at their disposal, so that they need to carefully exploit these resources, in order to deliver to their users a good quality-of-service. When context changes, and adaptation is required, it is likely that different users will have different quality-of-service needs, and this will cause applications to compete in order to use the available resources and adapt as their user demands.

When these conflicts arise, a conflict resolution mechanism must be put in place, so that applications will agree on which adaptation strategy to apply (i.e., which quality-of-service to deliver), and successful cooperations can be established.

When building mobile applications, we do not expect application engineers to design a conflict resolution mechanism, as this would overburden them with additional complexities. These conflicts, in fact, cannot be easily resolved at design time by assigning, for example, priorities to various adaptation strategies, as this would not take into consideration the current user's needs, and thus may fail to deliver actual benefit to the parties involved. Rather, a dynamic conflict resolution mechanism must be put in place that takes into consideration both the current availability of resources, and the importance that applications (and their users) associate to the quality-of-service levels involved.

Middleware, having direct knowledge of both the above running applications and of the available resources, is in the best position to function as an arbiter in these disputes. Suitable abstractions and mechanisms have therefore to be provided at the middleware layer to enable dynamic conflict resolution based on user's needs.

1.3 Thesis Contributions

The goal of this thesis is to investigate new principles, and design new abstractions and mechanisms that, embedded in a mobile computing middleware software layer, facilitate the development of context-aware applications. The following is an overview of the main contributions of this thesis.

1.3.1 Reflection for Context-Awareness and Dynamic Adaptation

We have investigated the principle of *reflection* to offer application developers abstractions and mechanisms that allow them to achieve context-awareness and adaptation to context changes [Capra et al., 2001a, Capra et al., 2001b, Capra et al., 2002a].

Reflection is a technique that first emerged in the programming language community to support the design of more open and extensible languages (e.g., see [Kiczales et al., 1991]). By definition, reflection allows a program to access, reason about and alter its own interpretation. The key to the approach is to offer a meta-interface supporting the *inspection* and *adaptation* of the underlying virtual machine (the meta-level). We have borrowed the idea of using reflection from the programming language community, and adapted it to middleware in the following way.

Reification of middleware behaviour: we provide application developers with an abstraction of middleware as a dynamically customisable service provider, where each *service* that the application is willing to customise can be delivered using different *policies* when requested in different *contexts*. For example, an instant messaging application may wish to exchange messages in plain text when bandwidth is high, while using compressed messages when bandwidth is low.

Middleware behaviour, with regard to a particular application it serves, is encoded in an *application profile*, which contains associations between the services that middleware delivers to the application, the policies that can be used to deliver the services, and the context conditions that must hold in order for a policy to be applied. This meta-encoding of middleware behaviour is then made available to applications (the base-level) for inspection, so that they can dynamically interrogate the middleware to know its current configuration, that is, its current behaviour. The process of making some aspects of the internal representation of the middleware explicit, and, hence, accessible to the application, is called *reification*.

Absorption of middleware behaviour: as user's needs may vary over time, applications must be allowed to customise middleware behaviour (i.e., to change the way services are delivered) while executing. Middleware provides applications with a meta-interface that enables run-time inspection and modification of the associations previously made explicit. The process where some aspects of the system are altered or overridden is called *absorption*.

The variety and heterogeneity of physical sensors is transparent to application engineers, as they are offered an image of context as a uniform set of resources. Once they have defined, in an application profile, which resources they are interested in, and how services should be delivered in different contexts, middleware takes charge of periodically querying the physical sensors and of determining the applicability of policies to deliver services. It is however outside the scope of this research to investigate what happens underneath the middleware, at the network operating system layer. We therefore do not research on how contextual information is actually retrieved; instead, we assume that the network OS provides an interface middleware designers can exploit to capture this information.

Context-aware computing is not a new computing paradigm; however, many of the solutions developed to date are of limited applicability, either because of the narrow view of context that they imply, or because of the lack of a systematic and principled approach, so that mainly ad-hoc, rather than reusable solutions, have evolved. There is also a growing community that is investigating the use of reflection in middleware; we will discuss our position compared to related work in these areas in Chapter 3, after having presented our approach.

1.3.2 Auctions for Dynamic QoS Conflict Resolution

Applications participating in the delivery of a service (e.g., instant messengers exchanging lines of text) may come to disagreements with regards to the policy that has to be used, that is, the QoS level they are willing to achieve in their current context. These conflicts cannot be statically resolved when application profiles are written, as they manifest themselves only in relation to the particular context configuration available at the time the service is requested, and to the profiles of participating applications. Therefore, a dynamic solution is needed.

We have designed a conflict resolution mechanism that exploits micro-economic techniques to dynamically solve these conflicts. We propose an image of the mobile distributed system as an economy, where a set of consumers must make a collective choice over a set of alternative goods. Goods represent the various policies that can be used to deliver a service, and the different QoS levels associated with them. Consumers are applications seeking to achieve their own goals, that is, to have the middleware deliver a service using the policy that achieves the best quality-of-service, according to user's preferences.

At the centre of the conflict resolution mechanism we have designed is an *auction mechanism* that allows applications to express their own preferences (i.e., how much they value the use of each conflicting policy), and therefore influence the way conflicts are resolved [Capra et al., 2002b]. Whenever a service that incorporates a conflict is requested, middleware plays the role of the auctioneer, collecting bids from the applications and delivering the service using the policy that maximises social welfare, that is, the policy that delivers, on average, the best quality-of-service.

The mechanism we have designed is simple, as it requires only a low computational overhead; it is dynamic, as it solves conflicts at run-time, when a service that incorporates a conflict is invoked, and it is customisable, as it takes application preferences into account.

The problem of resolving conflicts is a general one and different communities have investigated it over the years. We will discuss our position compared to related work in Chapter 4, after having presented our approach.

1.3.3 Formalisation

In order to provide a better understanding of the abstractions and mechanisms we have designed, we have provided a mathematical model that formalises their behaviour, using denotational semantics [Capra et al., 2002b].

Dynamic adaptation. Dynamic adaptation to context changes is achieved by means of reflection and metadata (i.e., the associations between services, policies and context encoded at the meta-level). These associations define which policies can be used to deliver a service and when. In order to understand exactly what happens when a service is invoked, we have provided the abstract syntax used to encode profiles, and the denotational semantics of service invocation. In particular, we have defined a semantic function that, given the name of a service, the associations attached to it in the meta-encoding, and current context, determines the way in which the service is delivered, that is, what policy is applied. Moreover, we have defined the set of operations that the meta-interface provides to applications to access the meta-encoding, together with their semantics.

Dynamic QoS agreement. Whenever a service is requested, the policy that must be used to deliver that service can be determined using the previously defined semantic function. However, a conflict may arise when different policies are enabled at the same time (i.e., in the same context). We have provided a formalisation of our auctioning mechanism, in order to understand the way conflicts are detected and solved. Conflict detection is based on the idea of sets. Each time a service is requested, the set of enabled policies is determined. If the cardinality of this set is exactly one, there is an agreement on the policy that must be applied, and no conflict resolution mechanism is therefore necessary. If the cardinality is zero, a conflict exists that cannot be automatically solved, as applications do not agree on a common policy to be applied. Finally, if the cardinality is greater than one, there is a conflict that can be resolved using one of the policies in the previously computed set. In order to understand which of these policies is selected, we have formalised the auctioning process. In particular, we have provided the semantic functions that, given the set of enabled (i.e., conflicting) policies, determine which policy is finally applied.

1.3.4 System Design and Implementation

To prove the effectiveness of the principles we have investigated in developing mobile context-aware applications, we have designed and implemented a reflective middleware architecture, CARISMA, that realises our middleware model. Only a minimal set of components have to be installed on a portable device, together with a meta-level description of the system. This core has a very small footprint, and therefore it is particularly well-suited for use on portable devices. According to the functionalities that the application needs, this set of components can be easily extended (e.g., new policies can be delivered, and new sensors can be monitored), and the middleware behaviour dynamically re-configured (by changing, through a reflective API, the meta-level description of the system).

We have designed two distributed algorithms to implement the conflict resolution mechanism, one that optimises time performance, the other that minimises communication costs. Both algorithms either complete successfully, or fail gracefully as a result of loss of connection.

1.3.5 Evaluation of Results

CARISMA implementation has been evaluated with a number of case studies, so to gauge the suitability of our reflective approach in achieving the main goals of this thesis [Capra et al., 2003]. In particular, we prove that the overheads introduced by reflection and by the conflict resolution mechanism are moderate, so that, as far as performance is concerned, CARISMA is well suited to run on portable devices. From the performance analysis we have run, we have also derived heuristics on how to encode associations (i.e., to what level of detail) in order to achieve the best performance results.

To estimate the flexibility and usability of our approach, we have asked students to implement context-aware applications on top of CARISMA and to report on their experience. The results have shown that the application development time is rather short, as the complexities of dealing with changes happening in the environment have been confined inside the middleware. Application engineers are only exposed to a small and easy-to-use interface through which they can access and modify a high-level meta-encoding of middleware behaviour.

1.4 Thesis Outline

Chapter 2 defines the motivation and the scope of our work. The characteristics of our target applications are elicited, and our assumptions declared, together with the aims and objectives of our research.

Chapter 3 introduces the concepts of reflection and metadata, and illustrates how they have been exploited to achieve dynamic adaptation to context. The conceptual model at the basis of our approach is first illustrated, and then formalised. Our position to related work is discussed.

Chapter 4 introduces the concept of conflicts. A classification of the conflicts we are interested in is first presented, followed by the requirements that our conflict resolution mechanism aims to achieve. Both a high-level description and a formalisation of the mechanism are presented, followed by a set of clarifying examples. Our position to related work is also discussed.

Chapter 5 discusses the construction of CARISMA, focusing on two major aspects: the reflective architecture we have designed to achieve dynamic adaptability, and the distributed algorithms that implement the conflict resolution process.

Chapter 6 offers a thorough evaluation of CARISMA, both quantitative, in terms of performance (i.e., time requested to answer a service request), and qualitative, in terms of usability. A critical evaluation of the results achieved concludes the chapter.

Chapter 7 summarises and evaluates the contributions of CARISMA to mobile computing middleware and explores directions for future work.

Appendix A contains the semantics of the reflective meta-interface.

Appendix B contains the grammar (XML Schema) used to encode meta-information.

Chapter 2

Motivation

In order to enhance the development of mobile computing applications, application engineers should be provided with abstractions and mechanisms to deal with context changes. This thesis investigates the principles needed to support the construction of context-aware applications, and how they can be best offered to application engineers through a middleware software layer.

Building a context-aware middleware requires the investigation of a broad set of aspects: from techniques to sense context changes, to formalisms to encode context information, from mechanisms to adapt application execution, to mechanisms to dynamically set the quality-of-service level to be delivered in current context. In approaching the issue of context-awareness, the following questions have to be answered in order to outline our research perspective: what context is considered? Who adapts to context changes? Who drives this adaptation?

In this chapter, we first introduce an example that illustrates the class of applications our middleware model aims at supporting; we define what context is relevant to these applications, and what kind of adaptation to context changes they need. Once the setting of our research has been established, we define the assumptions our work is based upon, and declare our research aims and objectives.

2.1 Introducing a Running Example

In this section, we sketch a conference application that is representative of the class of context-aware mobile applications we target. In particular, we focus on the many levels of complexity raised by mobility, to understand the requirements that context-aware middleware for mobile computing would need to fulfil.

2.1.1 Conference Application

Let us imagine a researcher Alice travelling to a conference with her own PDA. When arriving at the conference location, no paper information (e.g., conference proceedings, technical programme, local information, etc.) is provided; rather, a wireless network has been put in place that allows attendees to access this information dynamically from their portable devices.

Among the functionalities provided, Alice may:

1. Access the electronic proceedings to read papers of interest, browse through the technical programme to find out which talks to attend, search for local restaurants and places of interest.
2. Select the talks she wishes to attend and be alerted of those selected 10 minutes before they start, to have time to reach the conference room where they are being held.
3. Exchange messages with other attendees, for example, to share opinions, to arrange meetings, to organise dinners, and so on.

1. Access to the electronic proceedings
Once she arrives at the conference location, Alice starts browsing through the technical programme to select the talks she wants to attend the following day. It is difficult to decide, as there are many good speakers and interesting topics. To make up her mind, she accesses the on-line proceedings and has a look at the abstracts of the talks. While doing so, she meets her old friend Bob, also attending the conference. The two decide to have a look at the programme together, but as the weather is so nice, they want to do that by the swimming pool. When they arrive there, however, the quality of the network connection has worsened considerably and it takes them a significant amount of time to access the content of a paper. They decide to move to find a place where they have better connectivity, but suddenly they loose connection altogether, and they cannot access the conference programme anymore.

Problem statement: the system was designed considering that portable devices do not have much memory but they are connected with reasonable bandwidth (e.g., using an indoor WaveLAN connection). Therefore, the proceedings were not replicated on Alice's device, but always accessed on demand. Although saving memory, the system failed to provide Alice with the information she wanted at the time she needed it. What was more important *here*? Saving memory or granting data availability?

The system should be designed in such a way that the conference proceedings, as well as any other piece of information, can be accessed in different ways when requested in different contexts. For example, access to the proceedings may be obtained through network reference (e.g., no data is cached on the PDA), when executing indoors (i.e., good network connection) and battery availability is high (i.e., the user can stay connected for a long period of time). Title and abstract of the talks may be cached locally instead, to grant Alice data availability upon disconnections, either triggered by the user (e.g., to save battery power) or by the environment (e.g., degradation of network connectivity). However, this is not enough. Only Alice knows what her needs are, and therefore what behaviour is desired of the system. Moreover, her needs may change over time; availability may become more important than saving resources, and, as a result, network link techniques should be replaced by caching ones, possibly improved with compression mechanisms. It must therefore be possible for Alice to express her needs (maybe depending on context), and for the application to adapt according to these needs.

2. Reminder of the next talk

Alice and Bob go back inside the conference hotel, access the proceedings again, and select the talks they wish to attend the following day. At 8.50am the next day, Alice's PDA starts beeping: the first talk to attend starts in 10 minutes, and a reminder appears on the screen with information about the speaker name, the title of the talk, and the location where it is going to be held. At 9.20am, the speaker is about to finish when Alice's PDA starts beeping again to remind her to change room to attend her next talk.

> Problem statement: the reminder functionality of the system was designed to capture user's attention in noisy places as effectively as possible. Therefore, a sound alert was activated each time a reminder had to be issued. It was successful in capturing Alice's attention, but it also captured the attention of everybody else in her auditorium. What was more important *here*? User's responsiveness or discretion?

The reminder for the next talk must be delivered in different ways in different situations. For example, Alice may wish to use a sound alert in open air environments (e.g. by the swimming pool), a vibrating alert when attending a talk, and a silent alert when actively using the PDA (i.e., a blinking message is enough to capture user's attention). This is just one possible solution. Bob, for example, or Alice herself sometime later, may wish to use a sound alert also when attending a talk, so to be sure not to miss the alert, provided that a head-set is being used. Once again, dynamic and user-driven customisation of the system behaviour must be provided.

3. Exchange of messages

Alice leaves the room and tries to talk privately to her friend Bob. She turns on the messaging functionality of the conference application to see whether Bob is online. He is, so they start chatting. However, together with Bob's replies, Alice receives messages from other attendees and friends who want to say hello.

Problem statement: the messaging functionality of the system was designed to let people interact with each other; for example, to find each other, to exchange opinions, arrange meetings, etc. Therefore, when Alice was connected, all users of the system could see her online. This was successful for finding out where Bob was; however, it also made Alice visible to everybody else. What was more important *here*? User reachability or privacy?

Alice's visibility to other users of the messaging system should change according (for example) to her status (e.g., 'on-line' means that everyone can see her, 'busy' means that only her closest friends can see her, 'invisible' allows Alice to hide from everyone, for example, while giving a talk herself). Moreover, the way messages are exchanged may vary: messages may be sent in plain text, or encrypted, depending on the importance Alice confers to issues such as privacy and resource consumption. What represents a major concern to Alice, however, may not be a major concern to Bob, or to Alice herself at a different time. Users must be allowed to specify what is of concern to them, and to change this information at any time.

2.1.2 Research Perspective

The previous examples help us illustrate the perspective we take when designing mobile computing middleware for context-aware applications. This perspective can be defined as *application-centric*, that is, we look at the issue of context-awareness putting the application (and its user) at the centre of our interest, as explained below.

What context do we consider?

Context has been defined in a variety of ways in the mobile computing arena. [Schilit et al., 1994] emphasise three aspects of context: 'where you are, who you are with, and what resources are nearby'. [Dey et al., 1999] define context as 'any information that can be used to characterise the situation of an entity, where an entity can be a person, place, or physical or computational object'. According to the entity we consider, some aspects of context are therefore more important than others. [Chalmers and Sloman, 1999b] take the user's perspective and give a definition of context as comprising 'location, relative location, device characteristics, environment, and user's activity'.

In the conference application example, aspects such as location, device characteristics, network characteristics, and user's activity are discussed more frequently than others. However, we prefer to give a general definition of context, that is not restricted by the particular examples we use:

> *Context is any information that is of interest to the execution of an application.*

This definition includes information that is local to the device the application is executing on, such as available memory, available battery, screen size, and CPU speed; information that is external to the device, such as location, available bandwidth, hosts, services and resources in reach; and information that is application-specific, such as user's mood, user's activity, and similar.

Note that the notion of context changes radically if we take a middleware (rather than an application) perspective. From a middleware point of view, context comprises, for example, the communication protocol used to interact with other devices, the service discovery protocol put in place to dynamically discover resources and services, etc.

Who adapts to context changes?

In our application-centric perspective, we aim to support *application adaptation* to context changes. In order to provide application engineers with abstractions and mechanisms that facilitate the development of context-aware applications, we first need to understand more precisely what 'application adaptation' means and what it demands. In particular, we distinguish between *reactive adaptation* and *proactive adaptation*.

With *reactive adaptation*, we refer to the ability of the application to alter and re-configure itself as a result of (i.e., in reaction to) context changes; the application can then present itself to the users in different ways when executing in different context. For example, the 'print' option of an application main menu can be automatically enabled and disabled based on the proximity to a printer; users of a chat application may be automatically highlighted or overshadowed depending on their connectivity status, and so on.

With *proactive adaptation*, we refer to the ability of the application to deliver the same service in different ways when requested in different contexts and at different points in time. For example, in our conference application, access to the electronic proceedings may be granted using caching techniques when executing in an unstable environment and disconnections are likely to happen, while a network reference is preferred when executing in more stable and richer environments.

There have been approaches in the literature (e.g., OpenORB [Blair et al., 2001], ReMMoC [Capra et al., 2002a], UIC [Román et al., 2001]) that enable middleware adaptation to context changes, where context is defined according to a middleware point of view. For example, ReMMoC provides primitives that support adaptation of the service discovery protocol used by middleware, based on the service discovery infrastructure available in its current context. In this thesis, we take an application perspective instead, and aim at providing application engineers with abstractions and mechanisms that facilitate the achievement of both reactive and proactive application adaptation to context changes.

In this thesis, we do not target multimedia applications. For multimedia, adaptation has to be performed in a continuous way, while information (e.g., voice and video) flows over time [Coulson et al., 1992]. Although it would be possible to model continuous media flows as repeated service invocations, there could be no concept of a long term quality-of-service level to be achieved, as each invocation would be a separate, isolated event; moreover, it would not be possible to specify synchronisation constraints that apply to these sequences of requests. On the contrary, the class of applications we target requires a 'per-request' adaptation. We believe this represents an interesting class to study, as there exists a large number of applications that fall into this category, and that would therefore benefit from primitives that support this type of adaptation to context: from data-sharing applications, where adaptation is performed each time some (piece of) data is accessed, to web browsers, that adapt each time a new page is loaded, to e-mail clients, that adapt the download and display of e-mail contents, and so on.

Who drives adaptation?

In our application-centric perspective, context-changes trigger adaptation, and the way adaptation (both reactive and proactive) is carried out is driven by applications.

When installing an application on a mobile device, there is no static configuration that can be used by the application itself to learn how to execute in different contexts. One reason for this is that the variety of context configurations is so wide that one cannot possibly cover all of them; another reason is that applications, or rather the users of the applications, may change their minds, and therefore they may wish to adapt to context changes in different ways, not only in different contexts but also at different moments. Referring once again to the conference application example, at one moment Alice may wish to exchange messages with peers in plain text if her user's mood is set to 'on-line', while using encryption when 'busy' (i.e., when willing to communicate privately with her closest friends). Sometimes later, Alice may wish to give up encryption, because it is too resource demanding, and want to exchange messages in plain text only.

As applications alone know what is important to their users and when, applications must be put in a position that allow them to drive the adaptation process. We therefore need to provide application engineers with primitives to dynamically state how context changes should be handled.

2.2 Assumptions

As we have pointed out in the previous section, our research aims to investigate principles and techniques that enable the development of context-aware mobile computing middleware. In doing so, we take an application-centric perspective, therefore focusing on context information that is relevant to the application.

One can argue that a software layer that provides support for context-awareness only is not a middleware, as it does not tackle fundamental issues, such as enabling communication of distributed components. However, as we argued in Chapter 1, providing application engineers with primitives that facilitate the communication of distributed components is not enough: mobility introduces new complexities (e.g., context-awareness), that should not be tackled by application engineers directly, as this would slow down productivity and would compromise product quality. We believe middleware should be structured into different layers, each focusing on a specific issue (e.g., communication, context-awareness, etc.). In this thesis, we investigate one of these layers, while making the following assumptions as far as other layers are concerned (see Figure 2.1).

1. First, we assume the existence of a communication (sub)layer that enables components to coordinate via some wireless data link. We are not interested in what particular communication paradigm is actually provided; we only require that it fits mobile settings, and therefore that it tackles issues such as frequent discon-

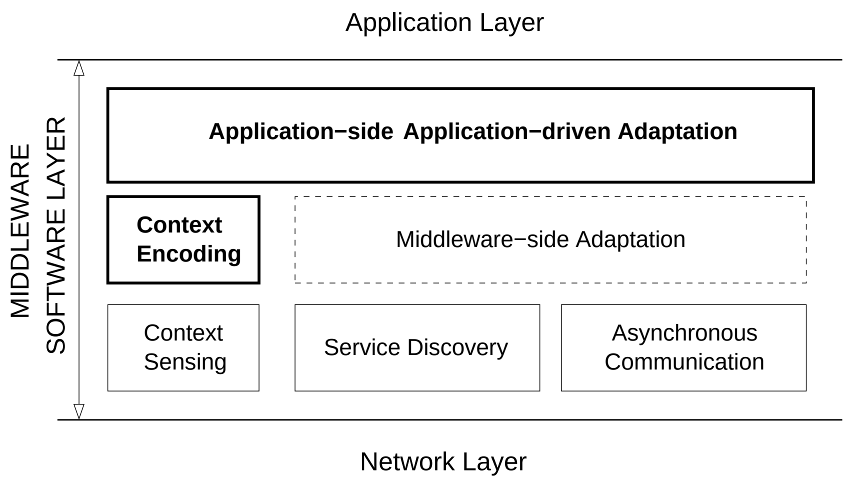

Application Layer
MIDDLEWARE SOFTWARE LAYER
Application−side Application−driven Adaptation
Context Encoding
Middleware−side Adaptation
Context Sensing
Service Discovery
Asynchronous Communication
Network Layer

Figure 2.1: Middleware (Sub)Layers.

nections. Asynchronous communication paradigms, such as message-based (e.g., iBus [Softwired, 2002]), event-based (e.g., Elvin [Segall and Arnold, 1997], iBus [Softwired, 2002], JEDI [Cugola and Nitto, 2001]), and tuple space-based (e.g., Lime [Murphy et al., 2001], TSpaces [Wyckoff et al., 1998], JavaSpaces [Waldo, 1998], L2imbo [Davies et al., 1998]), would therefore fit.

2. Second, we assume the existence of an infrastructure to advertise and discover hosts, services and resources in an ad-hoc network; this may be a simple broadcasting mechanism implemented on each mobile device, or a more sophisticated approach (e.g., Jini [Arnold et al., 1999], JMatos [Psinaptic, 2001] SLP [Guttman et al., 1999], UPnP [UPnP Forum, 1998], Salutation [Salutation Consortium, 1999]). When discussing the implementation of our middleware model (see Chapter 5), we will provide details about the communication and service discovery (sub)layers we build upon.

3. Although not necessary to our work, on top of these two layers there may be an infrastructure that support middleware-side adaptation; assuming a middleware-centric perspective, this infrastructure may enable, for example, customisation of the communication and service discovery protocols, based on context (e.g., ReMMoC [Capra et al., 2002a]).

4. In the application-centric perspective we take, context information we are interested in includes, for example, local resource availability (e.g., memory, battery, etc.), location, bandwidth, hosts within reach, application-specific information (e.g., user's activity and mood), and so on. Middleware gathers this information by interacting with a variety of heterogeneous sensors; for example, available battery can be obtained invoking a primitive of the underlying operating system, location can be computed interacting with various sensor technologies (e.g., GPS, infrared and radio frequency), user's mood can be discovered invoking a method that the above application provides. In this thesis, we are not interested in how (accurate) context sensing is actually performed; we assume that each sensor, be it a location sensor, a memory sensor, or a user's mood sensor, provides an interface that middleware can use to get the value of the associated resource (as accurately as possible). As long as such an interface exists, the resource can be part of the context our middleware monitors.

2.3 Research Aims and Objectives

As we have defined in Section 1.3, the goal of this thesis is to investigate new principles, and define new abstractions and mechanisms that, embedded in a mobile computing middleware software layer, facilitate the development of context-aware applications. Given the assumptions enumerated above, and the application-centric perspective we adopt, our research aims and objectives can be summarised as follows.

Adaptation to Context Changes. We aim to support both reactive and proactive adaptation to context changes. Although the way this adaptation takes place is application-dependent, we aim to simplify this task, by providing application engineers with abstractions and mechanisms to deal with both types of adaptation. In particular, for reactive adaptation, applications should only define the context configurations they are interested in, and the adaptations that must be performed when these configurations are entered; similarly, for proactive adaptation, applications should define how services should be delivered when requested in different contexts. The tasks of monitoring the environment, detecting context configurations of interest to the application and triggering the associated adaptation, and finding out how to deliver services in the context in which they are requested, are delegated to a middleware software layer that embeds the mechanisms we aim to provide. These mechanisms should also enable applications to dynamically change the context configurations of interest, the adaptation strategies adopted, and the way services are delivered, as the application's needs may vary during its lifetime.

In investigating the principles, and designing the mechanisms, that support application adaptation to context changes, we are only marginally concerned with context sensing (see Figure 2.1). It is our goal, instead, to provide application engineers with a uniform representation of context, and with a well-defined and general interface that they can use to inspect its status. Applications should therefore not be concerned with how each sensor encodes status information about the associated resource, and how this information can be retrieved: our middleware layer aims to provide them with a high-level encoding that abstracts from sensor-specific encodings, so that, be it a memory sensor, a location sensor, or any other sensor, the way context is queried and the way resource status is encoded, appears (to the application) to be the same.

QoS Conflict Resolution. In adapting the way services are delivered in various context configurations, applications may come to quality-of-service disagreements. For example, instant messaging application instances may disagree on the way messages are exchanged: while one peer may wish to send encrypted messages to protect the information moved around, another peer may prefer to send plain text messages only, to minimise resource consumption. As quality-of-service needs vary, applications are likely to use the adaptation mechanism provided by the middleware to express conflicting preferences. We aim to provide a mechanism that allows automatic conflict detection, and that enables applications to drive the conflict resolution process with as little effort as possible.

In order to be effective, mechanisms for dynamic adaptation to context and for QoS conflict resolution should not impose additional complexities on application engineers. We aim to provide powerful abstractions and mechanisms that minimise application engineer efforts in using them, while maximising automation (e.g., applications do not have to repeatedly check the status of their context to detect changes as middleware does it on their behalves).

Chapter 3

Reflection in Mobile Computing

As we have outlined in the previous chapter, our research in the area of mobile computing middleware tackles the issue of context-awareness from an application-centric perspective; applications determine our definition of context, as any piece of information that is of interest to their execution. Moreover, applications are responsible for driving the adaptation process (both reactive and proactive) that causes applications to dynamically tune their behaviour when context changes occur.

In this chapter, we discuss the principles we exploit to support application-driven adaptation to context changes. We illustrate the conceptual model we have developed based on these principles, and provide a formalisation of the model itself, to avoid ambiguities in its interpretation.

3.1 Principles

Application-driven adaptation refers to the ability of applications to *configure* and *re-configure* their behaviour, in order to adapt to different context conditions and needs.

With *application-driven configuration*, we refer to the ability of applications to control their own behaviour in a set of pre-defined context configurations. For example, referring to our conference application, an application-driven configuration may require access to the electronic proceedings using a network reference when bandwidth is stable and battery is high, while using caching techniques when bandwidth is variable and battery is low. Also, an application may want to be notified when some pre-defined context configurations are entered; for example, when running out of battery, or when new hosts come within reach. Applications determine both the set of behaviours they want to adhere to, and the context configurations that must hold in order for these behaviours to be applied.

With *application-driven re-configuration*, we refer to the ability of applications to dynamically change the set of possible behaviours, as well as the associations between these behaviours and their corresponding enabling contexts, in order to cope, for example, with varying application needs and unforeseen context conditions. Considering once again our conference application, the set of behaviours associated with the ‘access proceedings’ service may be altered from ‘copy’ (i.e., caching techniques) and ‘link’ (i.e., network reference), to ‘copy’ and ‘compress and copy’ (i.e., first compress the proceedings and then cache them locally), if availability of information when disconnected becomes a user’s primary need. Also, an application may alter the set of context configurations of interest, and may ask also to be alerted when running out of memory.

Note that, although strictly related, application-driven configuration and re-configuration are two distinct processes; in particular, one can have configuration without supporting re-configuration, but not viceversa.

We aim to support both application-driven configuration and re-configuration in a principled way, so that a reusable, rather than an ad-hoc solution, is promoted. On the one hand, this solution should minimise the application engineer’s efforts in developing context-aware applications, so that a high degree of automation should be provided; on the other, applications should be allowed to control the adaptation process, whenever and to the extent that they need to. We intend to reach this goal by means of a *middleware software layer* that provides automation, and by exploiting the principles of *metadata* and *reflection* to provide applications with the mechanisms and primitives to control the adaptation process. Before discussing how we exploit these principles in details, we provide the rationale behind this approach.

As we argued in the previous chapter, in supporting context-awareness we do not want application engineers to deal with low-level tasks, such as monitoring the environment, detecting context changes, and so on. These repetitive tasks can be easily performed by a middleware software layer that hides these complexities, thus providing the level of automation we call for. However, applications alone know how to properly adapt and when; middleware cannot have this knowledge a priori, because of the variety of applications, user’s needs, and context. Therefore, applications must be allowed, at any time, to dynamically specify what context information they are interested in, which context configurations they wish to react to, and so on. In other words, applications must be able to dynamically customise middleware behaviour, with respect to the resources it monitors, the alerts it issues, and so on. Metadata and reflection support this customisation in a very effective and elegant manner. By definition [Eliassen et al., 1999], reflection allows a program to access, reason about and alter its own interpretation. The key to the approach [Smith, 1982] is to make some aspects of the internal representation of the system, or meta-level (i.e., the middleware), explicit, and hence accessible from the base-level (i.e., the application), through a process called *reification*. Applications are then allowed to dynamically inspect middleware (introspection), and also to dynamically change it (adap-

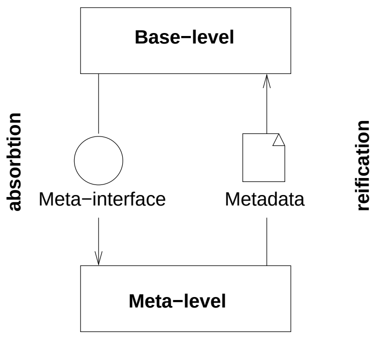


Figure 3.1: The Reflective Process.

tation), by means of a meta-interface that enables run-time modification of the internal representation previously made explicit. The process whereby some aspects of the system are altered or overridden is called *absorption*. In principle, a reflective system could be structured into a (potentially unbounded) number of logical meta/base levels, leading to what is known as the “reflective tower” [Smith, 1982]; in practice, there are seldom more than two of them. The whole process is depicted in Figure 3.1.

The principles we exploit to support application adaptation to context mirror the asymmetric dependability between the goals we aim to fulfil: we use *metadata* (i.e., reification of middleware behaviour) to achieve application-driven configuration, and *reflection* to add support for application-driven re-configuration.

3.1.1 Configuration through Metadata

Application-driven configuration concerns both reactive and proactive adaptation. As far as reactive adaptation is concerned, configuration refers to the ability of the application to define which aspects of context are of interest to the application itself (e.g., battery availability, location, etc.); to identify which context changes are relevant (e.g., available battery power falls below 10%, a printer becomes reachable, etc.), and to associate behaviours the application is willing to adhere to when such changes occur (e.g., switch off the back light of a PDA, fire an alert, etc.).

In the case of proactive adaptation, configuration refers to the ability of the application to identify the services that it wishes to adapt to context changes (e.g., access to the electronic proceedings); to decide on a set of possible behaviours that can be used to deliver these services (e.g., network reference, caching, etc.), and to specify the context configurations that must hold in order for a behaviour to be selected and applied when the service is actually requested (e.g., high battery power for a network reference, high memory availability for a caching technique).

To ease application development, middleware takes care of performing low-level and repetitive tasks, such as monitoring physical sensors, detecting context changes, selecting the behaviour enabled in the current context, etc. However, the set of resources that must be monitored, the changes that must be detected, the behaviours that must be enabled, and so on, are known to the application alone, and not to the middleware. Configuration information, specifying how the application wishes the middleware to behave (with regard to these issues) in its current context, should therefore be encoded in *middleware metadata* and used by the middleware to perform its tasks. In other words, middleware metadata reify middleware behaviour with respect to a particular application. This meta-information is divided into two parts:

Reactive metadata. Applications define associations between the context configurations of interest to them, and the behaviours that have to be triggered when such configurations are entered (e.g., switch off the back light of the PDA when battery falls below 20%). The task of interacting with the (heterogeneous) physical sensors, to periodically obtain updated information about context, of checking if one of the encoded configurations is entered, and then of executing the corresponding behaviour, is automatically performed by the middleware, without the application having to care about these low-level tasks. These behaviours can be simple notifications of a context configuration being entered (e.g., an alert that the device is running out of battery), or more complex behaviours that cause the application to execute differently (e.g., disable the 'print' functionality when there are no printers in reach). The extent to which applications adapt to context changes depend on the nature of the behaviours they associate in the metadata.

Note that the normal behaviour of the application is altered as a consequence of entering a particular context configuration, without the application performing any particular action for this to happen (apart, of course, from specifying an association between the new behaviour and that particular context configuration in the metadata).

Proactive metadata. Applications define associations between the services they wish to customise, the set of behaviours that can be used to deliver the services, and the context configurations that enable each of these behaviours. Each time one of these services is requested, the middleware is in charge of getting updated context information by interacting with the physical sensors, of checking which of the encoded configurations is currently valid, and then of delivering the service with the associated behaviour. In the conference application example, customisable services (i.e., services that should be delivered in different ways when requested in different context) include the 'access proceedings' service, the 'talk reminder' service, and the 'messaging' service. For the 'talk reminder' service, possible behaviours (i.e., possible ways of delivering the service) include a 'sound alert' and a 'vibrating alert'.

In this case, adaptation is performed only when a service is requested; as long as the application does not request a service, proactive metadata is not used, and, therefore, it has no effect on the way the application behaves.

Once configuration information has been encoded in metadata and made available to the middleware, the adaptation process is transparent to applications: middleware, in fact, implements mechanisms that use this information to perform both reactive and proactive adaptation of application behaviour. Unlike middleware systems that have been built adhering to the principle of the black-box, our middleware has no static behaviour it adheres to, either in response to context changes, or to answer a service request: in both cases, our middleware *learns* how to behave from the application, by looking at the configuration information that the application has encoded as middleware metadata. We discuss what metadata is encoding and how in Section 3.2.1 and 3.3.1.

3.1.2 Dynamic Re-configuration through Reflection

Configuration information is likely to need updating during the lifetime of an application; for example, an application may wish to be notified of new context changes, as a result of having entered unforeseen context configurations, or it may wish to adapt the delivery of a service in different ways, because of changes in application's user needs.

As far as reactive adaptation is concerned, re-configuration refers to the ability of applications to dynamically change the way the application reacts to context changes, that is, both to alter the set of resources that make up the context of interest to the application, and to re-define the associations between relevant context configurations and adaptation behaviours. Similarly, for proactive adaptation, re-configuration refers to the ability of applications to dynamically change the way services are delivered in different contexts. This refers to the ability to redefine both the set of services that require customisation, as well as the associations between customised services, possible behaviours to deliver these services, and context configurations that enable the various behaviours.

We argue that reflection is the best possible way to achieve dynamic re-configuration, as it makes the system it is applied to adaptable to its environment and better able to cope with changes. Brian Cantwell Smith, the originator of the early work on reflection [Smith, 1982], defines the reflective process as follows:

> *"In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures."*

The approach demands that an explicit representation of middleware behaviour, with respect to the above running applications, is maintained; this is what we call middleware metadata (i.e., reification of middleware behaviour). Reflection [Maes, 1987] then allows both dynamic *inspection* and *adaptation* of middleware behaviour, by means of a meta-interface that the middleware offers to applications to access this explicit representation. In particular, the meta-interface allows applications to read the currently active configuration (i.e., introspection), and to dynamically alter the information here encoded (i.e., adaptation). As middleware metadata reify middleware actual behaviour, changes in the behaviour are materialised in the meta-level description and, similarly, changes in the meta-level description immediately reflect back into the underlying middleware behaviour. This "closed loop" approach is called *causal connection*. Once we will have detailed what information we encode in middleware metadata and how (Section 3.2.1 and 3.3.1), we will discuss what part of metadata can be accessed and altered using the meta-interface, and with what effects (Section 3.2.2 and 3.3.2).

3.2 Conceptual Model

Having described the principles of our middleware model, we now introduce the primitives that support the application of these principles in practice. In particular, we define in this section what information we encode in middleware metadata (what we call *application profile*), and how this information can be accessed and altered using the meta-interface that the middleware provides (what we call the *reflective API*). Figure 3.2 illustrates how the reflective process has been adapted in our approach.

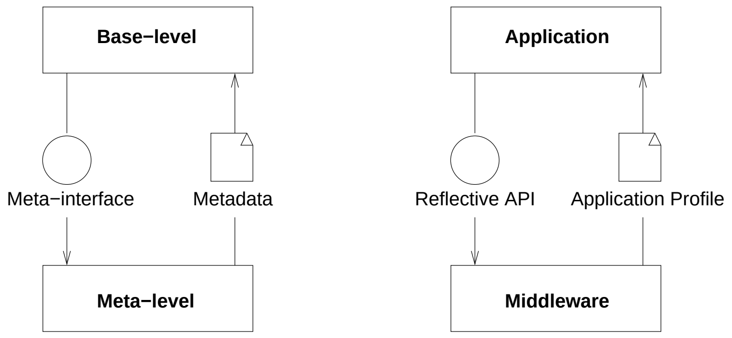


Figure 3.2: The Reflective Process (adapted).

3.2.1 Application Profile

We assume a single user for each mobile device, though there may be many applications running simultaneously on that device, hence, on the same middleware instance. This assumption is reasonable for many portable devices, such as palmtop computers, PDAs and mobile phones. In order to adapt to context changes, each application encodes *how* the middleware should behave *when* executing in particular execution contexts, in what we call an *application profile*, that is, middleware metadata.

Each application profile is divided into two parts, one containing *reactive metadata*, for reactive adaptation, and one containing *proactive metadata*, for proactive adaptation.

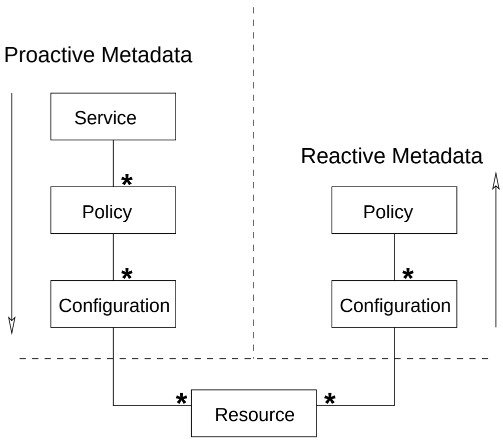


Figure 3.3: ER Diagram of Application Profile.

Reactive Metadata

Applications use reactive metadata to ask the middleware to listen for changes in the execution context and to react accordingly, independently of the task the application is performing at the moment. For example, the application may ask the middleware to switch off the back-light of a PDA when running out of battery, or to be alerted when the memory availability drops below a certain value. We therefore establish *associations* between particular *context configurations* that depend on the value of one or more resources that the middleware monitors, and *policies* (i.e., behaviours) that the middleware has to trigger when such configurations are entered, as depicted on the right-hand side of Figure 3.3. The arrow in the figure is pointing upward to represent an execution flow that goes *from* the middleware *to* the application: the middleware monitors context, and as a reaction to particular changes, it fires policies that reach the applications, for example, to simply notify that something has happened, or to modify the current behaviour of the application itself.

Proactive Metadata

Proactive metadata specify how the application wants to have a service delivered in different contexts. In particular, each application creates *associations* between the *services* the application is willing to customise, the *policies* that can be applied to deliver the services, and the *context configurations* that must hold in order for a policy to be applied, as shown on the left-hand side of Figure 3.3. For example, the ‘access proceedings’ service can be delivered either using a ‘network reference’ policy, when bandwidth is stable, or a ‘cache’ policy, when bandwidth is fluctuating. In this case, the arrow is pointing downward to indicate that control flows *from* the application *to* the middleware: each time an application requests a service, the middleware consults its application profile to determine which policy should be applied to deliver the requested service in the current context. We make here an assumption that must be remembered in the following chapter: each service can be delivered using exactly one policy at a time. Multiple policies can logically be combined, for example, to compress data first and then to cache it locally. However, we regard the combined ‘compress and cache’ policy as a new one, and in the profile we will refer to this new policy, not to the sequential execution of two distinct policies, ‘compress’ and ‘cache’.

Context Encoding

We represent context configurations, for both reactive and proactive metadata, in terms of the resources that the middleware monitors and that the application is interested in. These resources vary greatly, both in the sensing technologies the middleware has to interact with to obtain resource status information, as well as in the obtained information itself. For example, the resources we deal with can be local to the device, such as memory or battery availability, whose status can be obtained by invoking a primitive of the underlying network operating system; resources can be external to the device, such as location, whose status is usually obtained by interacting with a physical sensor (not necessarily available on the device); and, finally, resources can be application-specific, such as user’s mood, whose status can be obtained by interacting with an application-defined software agent. Not only does the way status information is obtained vary greatly, but also the nature of this information is extremely heterogeneous. For example, memory or battery availability can be easily expressed with a numeric value that is interpreted roughly in the same way by any application (e.g., either absolute value or percentage). Location information, instead, may be represented in different ways (e.g., coordinates in the space, proximity to physical objects, etc.) by different sensing technologies (e.g., GPS, infrared, radio frequency, etc.), and may require various processing to become meaningful to the applications (e.g., from a (x, y, z) point in the space to a room in the conference hotel). Context configurations encoded in a profile abstract away from the specific sensing technique used, and the raw and heterogeneous status information obtained; rather, these configurations assume that context information has already been gathered and processed, and that a uniform representation of resource status has being used. Details of this uniform encoding are given in Section 3.3.1.

3.2.2 Reflective Mechanism

Our middleware model provides a meta-interface to read and alter the information encoded in an application profile, thus allowing applications to dynamically control middleware behaviour. This meta-interface is divided into two parts: one that gives access to reactive metadata, and the other that gives access to proactive metadata.

Reactive Meta-Interface

The meta-interface to the reactive metadata encoded in a profile allows applications to read information about currently encoded associations, that is, which policies can be executed, and in which context configurations. Associations between policies and context configurations can also be removed, added, and updated. Updates may happen at any level of granularity: of the resources that make up a configuration, of the set of configurations associated with a policy, and of the policies that make up the reactive metadata. For example, an application may ask to be notified when the battery falls below 10%; some time later, it may use the reactive meta-interface to request the middleware to trigger a low battery alert when battery falls below 5% too, and to trigger a low memory alert when falling short of memory. These changes have immediate impact on middleware behaviour as, for example, a resource may not be of interest to the application anymore, and therefore middleware does not need to periodically interact with the corresponding physical sensor. Viceversa, when a new resource becomes of interest to the application, its physical sensor is activated and middleware starts to poll it periodically. Reactive metadata may also be empty, in case the application is not interested in adapting its behaviour to context changes.

Proactive Meta-Interface

The meta-interface to the proactive metadata encoded in a profile allows applications to know which services are currently customised, which policies are associated with such services, and which context configurations are associated with the policies. As before, associations can be added, removed and updated, and these updates may happen at any level of granularity: from the resources that make up a configuration, to the services currently customised, thus giving applications full control over middleware execution. For example, the association that requires the 'access proceedings' service to be delivered using a network reference policy may be removed, in order to maximise data availability when disconnected. The effects of these changes are visible only when a customised service is requested; as long as these services are not invoked, the behaviour of the system is not affected. If there is no proactive metadata, services are always delivered in the same way (i.e., using a default behaviour).

It should be clear by now that reflection does not enforce any particular behaviour; it is up to the applications to decide whether to use the adaptation mechanism the middleware provides; if context adaptation is not needed, applications may simply leave their profile blank.

3.3 Formal Model

In this section, we illustrate a formalisation of our conceptual model. In particular, we define the abstract syntax that we use to encode application profiles, and the semantics of its reactive and proactive metadata. A formalisation of the meta-interface that we provide to access the profiles is also illustrated; its semantics is reported for completeness in Appendix A.

3.3.1 Application Profile

Each application $profile$ is divided into two parts: a *reactive* part and a *proactive* part.

$$profile \; ::= \; \textit{reactive proactive}$$

Reactive Metadata

The reactive part of the profile (Figure 3.4) encodes associations (*policyList*) between policies (identified by name *pname*) and context configurations (*contextList*) that determine when each policy should be fired. Each context configuration (*context*) is uniquely identified inside the profile (*cid*), and describes the status of one or more resources (*resourceList*). Each resource (*resource*) is identified by a unique name (*rname*), and its status is described by the result of applying a resource-specific operator (*oname*) to a set of associated values (*valueList*). For example, (*memory*, *inBetween*, $\{20\%, 25\%\}$) describes resource *memory* having one of the possible values $\{20\%, 21\%, 22\%, 23\%, 24\%, 25\%\}$.

We use a model for context that is based on boolean algebra, which allows us to easily

$$\begin{array}{rcl}
\textit{reactive} & ::= & \textit{policyList} \mid \varepsilon \\
\textit{policyList} & ::= & \textit{policy policyList} \mid \textit{policy} \\
\textit{policy} & ::= & \textit{pname contextList} \\
\textit{contextList} & ::= & \textit{context contextList} \mid \textit{context} \\
\textit{context} & ::= & \textit{cid resourceList} \\
\textit{resourceList} & ::= & \textit{resource resourceList} \mid \textit{resource} \\
\textit{resource} & ::= & \textit{rname oname valueList} \\
\textit{valueList} & ::= & \textit{value valueList} \mid \varepsilon
\end{array}$$

Figure 3.4: Application Profile's Abstract Syntax - Reactive Part. $\textit{pname} \in \mathrm{P}$, $\textit{rname} \in \mathrm{R}$, $\textit{cid} \in \mathbb{N}$, $\textit{value} \in \mathrm{V}$, $\textit{oname} \in \mathrm{O}$, being P, R, $\mathbb{N}$, V, and O the domain sets illustrated in Figure 3.5.

Σ	:	alphabet
$S \subset \Sigma^*$	:	set of all service names
$P \subset \Sigma^*$	:	set of all policy names
$\mathbb{N}$	:	set of all natural numbers
$R \subset \Sigma^*$	:	set of all resource names
$O \subset \Sigma^*$	:	set of all operator names
V	:	set of all values of resources in R
$E \subset \wp(R \times V)$	:	set of all possible execution contexts

Figure 3.5: Application Profile - Domain Sets.

construct more complex context configurations starting from atomic formulae, using the $\wedge$ (logical *and*) and $\vee$ (logical *or*) operators. An atomic context is represented by the 3-ary predicate: $<$ *rname oname valueList* $>$. Atomic formulae can then be combined using (implicit) $\wedge$ operators, to form more complex context configurations to which a new *cid* is assigned; finally, various context configurations can be combined using (implicit) $\vee$ operators and are then associated with a policy. In other words, each context configuration expresses the set of resource conditions that must simultaneously hold ($\wedge$ operator) for a policy to be applied; these context configurations are then put in $\vee$ relation, as the same policy may be enabled in different contexts.

Figure 3.6 shows an example of reactive metadata; at this stage, we are not interested in implementation details (in particular, in the language used to encode profiles); we therefore use the abstract syntax illustrated above to discuss the following examples. The application with which this profile is associated requires notification when executing in a resource constrained environment (`constrainedContextAlert`); this is defined to mean when memory availability drops below 10% *and* battery power is between 5% and 10%. Also, it requires notification when running out of battery (`lowBatteryAlert`), i.e., battery power falls below 10%, *or* when below 5%.

```
lowBatteryAlert
    1
        battery lessThan 10%
    2
        battery lessThan 5%

constrainedContextAlert
    3
        memory lessThan 10%
        battery inBetween {5% 10%}
```

Figure 3.6: Reactive Metadata - Example.

$$\begin{array}{rcl}
\mathit{bool} & : & \{\top,\ \bot\} \\
\mathit{resourceStatus} & : & \mathrm{R} \times \mathrm{O} \times \wp(V) \times \mathit{bool} \\
\mathit{resourceStatusList} & : & \wp(\mathit{resourceStatus}) \\
\mathit{contextStatus} & : & \mathbb{N} \times \wp(\mathit{resourceStatusList}) \\
\mathit{contextStatusList} & : & \wp(\mathit{contextStatus}) \\
\mathit{policyStatus} & : & \mathrm{P} \times \wp(\mathit{contextStatusList}) \\
\mathit{policyStatusList} & : & \wp(\mathit{policyStatus})
\end{array}$$

Figure 3.7: Reactive Semantic Functions - Domain Sets.

Once battery power has dropped below 10% (but not yet below 5%), and this context change has been notified, the application should not keep receiving notifications of the fact that battery power is below 10% every t time units (i.e., at the frequency at which middleware checks the context). Only when available battery falls below 5% another alert should be fired, followed by no other. The semantics we attach to each reactive association is therefore the following. Context configurations are independent: if a policy has more than one configuration associated with it, any of them ($\vee$ semantics) can cause the policy to be fired. For example, `lowBatteryAlert` is fired both when battery drops below 10% (`cid = 1`), and when it falls below 5% (`cid = 2`). Moreover, each configuration determines a partition of the context space; for example, `cid=1` determines a partition between a context where battery is greater than 10%, and one where its value is less than 10%. When battery falls below 10% (i.e., when context enters into the first element of the partition), the policy is fired; before this policy is fired again, context changes must have happened that have brought context into a different element of the partition. For example, when battery power falls below 10% for the first time, `lowBatteryAlert` is fired. Before `cid=1` causes the policy to be fired again, the battery must have been recharged and its value brought above 10%; therefore, while battery power level keeps decreasing (i.e., 9%,8%,7%, ...), the policy is not fired repeatedly. When battery drops below 5%, the same policy is fired again, but this time `cid=2` enables it. We therefore prevent triggering undesired sequences of policies.

A formalisation of this behaviour is shown in Figures 3.8 to 3.10. The domain sets of the semantic functions presented here can be found in Figure 3.7. When an application is started, the middleware fetches its profile and processes it, in order to associate a boolean value equal to *true* to each *resource* of the context configurations encoded in the reactive part of the profile itself (function *init* in Figure 3.8). In order for a context to be enabled, and the associated policy to be fired, all resource conditions expressed in the configuration must hold, and the associated boolean values set to true. As we are going to show, we use these boolean values to prevent cascading policies, as they represent a sort of 'firability' pre-condition.

In order for a policy to be fired, two conditions must be met, as shown in Figure 3.9: at least

$$\begin{aligned}
init &: policyList \rightarrow policyStatusList \\
init_{cl} &: contextList \rightarrow contextStatusList \\
init_{rl} &: resourceList \rightarrow resourceStatusList
\end{aligned}$$

$$\begin{aligned}
init[\![policy\ policyList]\!] &= init[\![policy]\!] \cup init[\![policyList]\!] \\
init[\![policy]\!] &= init[\![pname\ contextList]\!] \\
init[\![pname\ contextList]\!] &= \{(pname,\ init_{cl}[\![contextList]\!])\} \\
init_{cl}[\![context\ contextList]\!] &= init_{cl}[\![context]\!] \cup init_{cl}[\![contextList]\!] \\
init_{cl}[\![context]\!] &= init_{cl}[\![cid\ resourceList]\!] \\
init_{cl}[\![cid\ resourceList]\!] &= \{(cid,\ init_{rl}[\![resourceList]\!])\} \\
init_{rl}[\![resource\ resourceList]\!] &= init_{rl}[\![resource]\!] \cup init_{rl}[\![resourceList]\!] \\
init_{rl}[\![resource]\!] &= init_{rl}[\![rname\ oname\ valueList]\!] \\
init_{rl}[\![rname\ oname\ valueList]\!] &= \{(rname,\ oname,\ valueList,\ \top)\}
\end{aligned}$$

Figure 3.8: Application Profile Semantics - Reactive (*init*).

one of its associated context configurations evaluates to *true* (i.e., it is enabled) in its current context (i.e., all resource conditions hold in the current context), *and* all the boolean values of resources that make up this configuration are *true*. We make use here of the auxiliary boolean function *eval*, such that $eval((rname, oname, vList), e)$ returns *true* if the value of resource *rname* in the execution context e is among the values obtained by applying the operator *oname* to *vList*. For example, $eval((memory, inBetween, \{20\%, 25\%\}), \{(memory, 22\%)\}) = \top$, while $eval((memory, lessThan, \{5\%\}), \{(memory, 22\%)\}) = \bot$.

Once a policy has been executed, we prevent it from being repeatedly fired by setting the boolean value of each resource that makes up the enabling context configuration to $false$ (Figure 3.10).

Boolean values representing firability pre-conditions are re-set to *true* by the *update* function. Every t time units, the middleware checks the status of context and updates the boolean value previously associated with each resource in the following way: if the boolean value is set to $false$, and the current resource value does not respect the condition expressed by that resource in the profile, the value is changed to *true*; otherwise, the boolean value is not altered (Figure 3.11).

Let us refer to the example shown in Figure 3.6, where we have encoded a condition $(battery, lessThan, 10\%)$ in context 1. When the application is started, a boolean value

$$
\begin{array}{rcl}
fire & : & policyStatusList \rightarrow \mathrm{E} \rightarrow \wp(\mathrm{P} \times \mathbb{N}) \\
fire[\![pStatus\ psList]\!]_e & = & fire[\![pStatus]\!]_e \cup fire[\![psList]\!]_e \\
fire[\![pStatus]\!]_e & = & fire[\![(pname,\ csList)]\!]_e \\
fire[\![(pname,\ csList)]\!]_e & = & \begin{cases} \{(pname, cid)\} \text{ if } \exists\ cStatus = (cid,\ rsList) \in csList \mid \\ \quad \forall\ rStatus = (rname, oname, vList, b) \in rsList, \\ \quad eval((rname, oname, vList), e) = \top\ \wedge\ b = \top \\ \\ \emptyset \text{ otherwise} \end{cases}
\end{array}
$$

Figure 3.9: Application Profile Semantics - Reactive ($fire$).

$$
\begin{array}{rcl}
reset & : & policyStatusList \rightarrow \wp(\mathrm{P} \times \mathbb{N}) \rightarrow policyStatusList \\
reset_{csl} & : & contextStatusList \rightarrow \wp(\mathrm{P} \times \mathbb{N}) \rightarrow contextStatusList \\
reset_{rsl} & : & resourceStatusList \rightarrow \wp(\mathrm{P} \times \mathbb{N}) \rightarrow resourceStatusList
\end{array}
$$

$$
\begin{array}{rcl}
reset[\![pStatus\ psList]\!]_{pcl} & = & reset[\![pStatus]\!]_{pcl} \cup reset[\![psList]\!]_{pcl} \\
reset[\![pStatus]\!]_{pcl} & = & reset[\![(pname, csList)]\!]_{pcl} \\
reset[\![(pname, csList)]\!]_{pcl} & = & \begin{cases} \cup\{(pname, reset_{csl}[\![csList]\!]_{\{(p_i,c_i)\}})\}\ \forall\ i \mid \\ \quad \exists\ (p_i, c_i) \in pcl,\ pname = p_i \\ \\ \{(pname, csList)\} \text{ otherwise} \end{cases} \\
reset_{csl}[\![cStatus\ csList]\!]_{\{(p,c)\}} & = & reset_{csl}[\![cStatus]\!]_{\{(p,c)\}} \cup reset_{csl}[\![csList]\!]_{\{(p,c)\}} \\
reset_{csl}[\![cStatus]\!]_{\{(p,c)\}} & = & reset_{csl}[\![(cid, rsList)]\!]_{\{(p,c)\}} \\
reset_{csl}[\![(cid, rsList)]\!]_{\{(p,c)\}} & = & \begin{cases} \{(cid, reset_{rsl}[\![rsList]\!]_{\{(p,c)\}})\} \text{ if } cid = c \\ \{(cid, rsList)\} \text{ otherwise} \end{cases} \\
reset_{rsl}[\![rStatus\ rsList]\!]_{\{(p,c)\}} & = & reset_{rsl}[\![rStatus]\!]_{\{(p,c)\}} \cup reset_{rsl}[\![rsList]\!]_{\{(p,c)\}} \\
reset_{rsl}[\![rStatus]\!]_{\{(p,c)\}} & = & reset_{rsl}[\![(rname, oname, vList, b)]\!]_{\{(p,c)\}} \\
reset_{rsl}[\![(rname, oname, vList, b)]\!]_{\{(p,c)\}} & = & \{(rname, oname, vList, \bot)\}
\end{array}
$$

Figure 3.10: Application Profile Semantics - Reactive ($reset$).

equal to *true* is associated with the battery status condition; as soon as a context change brings battery value below 10%, the `lowBatteryAlert` policy is fired, and the boolean value is changed to $false$; as long as battery values stay below 10%, context configuration 1 does not enable this policy, as the boolean value stays $false$. A change in the context that brings battery above 10%, will cause the boolan value to be set to *true* (*update*

$$\begin{aligned}
update &: policyStatusList \rightarrow \mathrm{E} \rightarrow policyStatusList \\
update_{csl} &: contextStatusList \rightarrow \mathrm{E} \rightarrow contextStatusList \\
update_{rsl} &: resourceStatusList \rightarrow \mathrm{E} \rightarrow resourceStatusList
\end{aligned}$$

$$\begin{aligned}
update[\![pStatus\ psList]\!]_e &= update[\![pStatus]\!]_e \cup update[\![psList]\!]_e \\
update[\![pStatus]\!]_e &= update[\![(pname, csList)]\!]_e \\
update[\![(pname, csList)]\!]_e &= \{(pname, update_{csl}[\![csList]\!]_e)\} \\
update_{csl}[\![cStatus\ csList]\!]_e &= update_{csl}[\![cStatus]\!]_e \cup update_{csl}[\![csList]\!]_e \\
update_{csl}[\![cStatus]\!]_e &= update_{csl}[\![(cid, rsList)]\!]_e \\
update_{csl}[\![(cid, rsList)]\!]_e &= \{(cid, update_{rsl}[\![rsList]\!]_e)\} \\
update_{rsl}[\![rStatus\ rsList]\!]_e &= update_{rsl}[\![rStatus]\!]_e \cup update_{rsl}[\![rsList]\!]_e \\
update_{rsl}[\![rStatus]\!]_e &= update_{rsl}[\![(rname, oname, vList, b)]\!]_e \\
update_{rsl}[\![(rname, oname, vList, b)]\!]_e &= \begin{cases} \{(rname, oname, vList, \top)\} \\ \quad \text{if } eval((rname, oname, vList), e) = \top \\ \quad \wedge\ b = \bot \\ \\ \{(rname, oname, vList, b)\} \\ \quad\quad \text{otherwise} \end{cases}
\end{aligned}$$

Figure 3.11: Application Profile Semantics - Reactive (*update*).

function), and the next time the battery value drops below 10% the `lowBatteryAlert` policy will be fired again.

After an initialisation process that takes place when an application is started (time t_0), the reactive encoding of the application profile determines the set of policies that are fired in the following way:

$$\begin{aligned}
t_0 &: PSL_0 = init[\![policyList]\!] \\
\\
t_1 &: P_1 = fire[\![update[\![PSL_0]\!]_{e_1}]\!]_{e_1} \\
&\quad\ PSL_1 = reset[\![PSL_0]\!]_{P_1} \\
&\ldots \\
t_i &: P_i = fire[\![update[\![PSL_{i-1}]\!]_{e_i}]\!]_{e_i} \\
&\quad\ PSL_i = reset[\![PSL_{i-1}]\!]_{P_i}
\end{aligned} \tag{3.1}$$

where $PSL_i \in policyStatusList$ associates boolean values with resource conditions, and

$P_i \in \wp(\mathrm{P} \times \mathbb{N})$ states which policies have been fired, and which context configurations have enabled them.

Proactive Metadata

The proactive part of the profile (Figure 3.12) encodes associations (*serviceList*) between the services the application wishes to customise (identified by name *sname*), the policies (identified by name *pname*) that can be applied to deliver each service, and context configurations (*contextList*) that determine when each policy can be applied. Similarly to what we discussed for the reactive encoding, each context configuration (*context*) is identified by a unique number inside the profile (*cid*), and describes the required status for one or more resources in order for the corresponding context configuration to be valid, and its associated policy to be applied.

Figure 3.13 shows various examples of proactive encodings for the conference application. Three services have been customised here: as shown, more than one resource can be associated with the same context (e.g., `cid = 4`), and more than one context configuration may be associated with the same policy (e.g., policy `plainMsg`). Note that `compressCache` is a combination of two other policies, `compress` and `cache`; however, in the profile, only the name of the combined policy appears, without reference to its constituents. The rationale for this is that, at the profile level, we are not interested in how various policies can be combined, and what the semantics of the combination is; policies can be combined at below the profile level, but this requires assigning a new name to the combined policy, and using this name in the profile.

The semantics we associate to proactive metadata is the following: whenever a customised

$$
\begin{array}{rcl}
\mathit{proactive} & ::= & \mathit{serviceList} \mid \varepsilon \\
\mathit{serviceList} & ::= & \mathit{service}\ \mathit{serviceList} \mid \mathit{service} \\
\mathit{service} & ::= & \mathit{sname}\ \mathit{policyList} \\
\mathit{policyList} & ::= & \mathit{policy}\ \mathit{policyList} \mid \mathit{policy} \\
\mathit{policy} & ::= & \mathit{pname}\ \mathit{contextList} \\
\mathit{contextList} & ::= & \mathit{context}\ \mathit{contextList} \mid \varepsilon \\
\mathit{context} & ::= & \mathit{cid}\ \mathit{resourceList} \\
\mathit{resourceList} & ::= & \mathit{resource}\ \mathit{resourceList} \mid \mathit{resource} \\
\mathit{resource} & ::= & \mathit{rname}\ \mathit{oname}\ \mathit{valueList} \\
\mathit{valueList} & ::= & \mathit{value}\ \mathit{valueList} \mid \varepsilon
\end{array}
$$

Figure 3.12: Application Profile's Abstract Syntax - Proactive Part. *sname* $\in$ S, *pname* $\in$ P, *rname* $\in$ R, *cid* $\in \mathbb{N}$, *value* $\in$ V, *oname* $\in$ O, being S, P, R, $\mathbb{N}$, V, and O the domain sets illustrated in Figure 3.5.

```
accessProceedings                      messagingService
     networkReference                      plainMsg
         4                                     7
             bandwidth greaterThan 1               userMood equals online
             battery greaterThan 10%           8
                                                   battery lessThan 30%
                                                   userMood notEquals online

     cacheAbstract                         encryptedMsg
         5                                     9
             bandwidth lessThan 1                  userMood equals busy
             battery lessThan 10%                  battery greaterThan 30%

     compressCache
         6
             memory lessThan 10\%


talkReminder
     vibraAlert

     soundAlert
         10
             location equals outdoor
```

Figure 3.13: Proactive Metadata - Examples.

service is invoked, the policy that will be used to deliver the service is the one that has got *at least one* context configuration enabled in the current context. A configuration is enabled *if and only if all* resource conditions that make up the configuration hold in the current context. For example, if the `accessProceedings` service is invoked in a context where bandwidth is less than 1 and battery less than 10%, than the policy `cacheAbstract` is applied. Note that if no context is associated with a policy (e.g., policy `vibraAlert` in Figure 3.13), then that policy is always enabled, regardless of the current execution context.

The semantics of proactive metadata is formally presented in Figure 3.14. The domain sets of these semantic functions can be found in Figures 3.5 and 3.7; as described before, *eval* is a boolean function such that $eval((rname, oname, vList), e)$ returns *true* if the value of resource *rname* in the execution context e is among the values obtained by applying the operator *oname* to *vList*. According to this semantics, whenever an application requests a customised service of the middleware, a *set* of policies is determined. So far, we have assumed that each service is delivered with exactly one policy at a time: we will discuss what happens if the cardinality of this set is not exactly one in the next chapter.

$$\begin{array}{rcl}
\mathcal{F} & : & \mathit{service} \rightarrow \mathrm{E} \rightarrow \wp(\mathrm{P}) \\
\mathcal{F}_{pl} & : & \mathit{policyList} \rightarrow \mathrm{E} \rightarrow \wp(\mathrm{P}) \\
\mathcal{F}[\![\mathit{sname\ policyList}]\!]_e & = & \mathcal{F}_{pl}[\![\mathit{policyList}]\!]_e \\
\mathcal{F}_{pl}[\![\mathit{policy\ policyList}]\!]_e & = & \mathcal{F}_{pl}[\![\mathit{policy}]\!]_e \ \cup \ \mathcal{F}_{pl}[\![\mathit{policyList}]\!]_e \\
\mathcal{F}_{pl}[\![\mathit{policy}]\!]_e & = & \mathcal{F}_{pl}[\![\mathit{pname\ contextList}]\!]_e \\
\mathcal{F}_{pl}[\![\mathit{pname\ contextList}]\!]_e & = & \{\mathit{pname}\} \ \texttt{if} \ \mathit{valid}[\![\mathit{contextList}]\!]_e = \top \\
 & & \quad \emptyset \qquad \texttt{if} \ \mathit{valid}[\![\mathit{contextList}]\!]_e = \bot
\end{array}$$

$$\begin{array}{rcl}
\mathit{valid} & : & \mathit{contextList} \rightarrow \mathrm{E} \rightarrow \mathit{bool} \\
\mathit{valid}_{rl} & : & \mathit{resourceList} \rightarrow \mathrm{E} \rightarrow \mathit{bool} \\
\mathit{valid}[\![\mathit{context\ contextList}]\!]_e & = & \mathit{valid}[\![\mathit{context}]\!]_e \ \vee \ \mathit{valid}[\![\mathit{contextList}]\!]_e \\
\mathit{valid}[\![\mathit{context}]\!]_e & = & \mathit{valid}[\![\mathit{cid\ resourceList}]\!]_e \\
\mathit{valid}[\![\mathit{cid\ resourceList}]\!]_e & = & \mathit{valid}_{rl}[\![\mathit{resourceList}]\!]_e \\
\mathit{valid}_{rl}[\![\mathit{resource\ resourceList}]\!]_e & = & \mathit{valid}_{rl}[\![\mathit{resource}]\!]_e \ \wedge \ \mathit{valid}_{rl}[\![\mathit{resourceList}]\!]_e \\
\mathit{valid}_{rl}[\![\mathit{resource}]\!]_e & = & \mathit{valid}_{rl}[\![\mathit{rname\ oname\ valueList}]\!]_e \\
\mathit{valid}_{rl}[\![\mathit{rname\ oname\ valueList}]\!]_e & = & \mathit{eval}((\mathit{rname}, \mathit{oname}, \mathit{valueList}), e) \\
\mathit{valid}[\![\varepsilon]\!]_e & = & \top
\end{array}$$

Figure 3.14: Application Profile Semantics - Proactive.

3.3.2 Reflective Mechanism

In this section, we define how the meta-interface we provide acts upon application profiles, either to inspect them (i.e., to inspect middleware behaviour) or to alter them (i.e., adaptation of middleware behaviour).

Reactive Meta-Interface

Inspection of reactive metadata may happen at different levels of granularity: from the resources associated with a context configuration, to the configurations associated with a policy, up to the policies themselves. Figure 3.15 lists the semantic functions available to inspect this information; inspection is based on unique names for policies and resources, and on ids for context configurations; we use *null* to indicate an empty search result. The semantics of these functions is trivial, and is reported in Appendix A for completeness.

Adaptation of proactive metadata takes place by adding, removing and updating the associations encoded in a profile. As for inspection, we allow these operations to work at different levels of granularity: the resources associated with a context, the configurations associated with a policy, and the policies themselves. The semantics of these functions,

$$\begin{aligned}
readRP &: profile \times \mathrm{P} \rightarrow policy \cup \{null\} \\
readRC &: profile \times \mathrm{P} \times \mathbb{N} \rightarrow context \cup \{null\} \\
readRR &: profile \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow resource \cup \{null\}
\end{aligned}$$

Figure 3.15: Reflective Mechanism - Inspection of Reactive Metadata.

with respect to the way they change the meta-encoding, is also rather simple, and is reported in Appendix A.

Informally speaking, remove operations are based on unique policy names, context ids, and resource names; if the names/ids are not found, the profile is not altered. Add operations require that the policy to be added does not already exist; that the context id of the new context does not already exist (while the policy to which it has to be added must already be listed in the profile), and that the resource name is not already listed in the particular context and policy to which it must be added. Finally, update operations require the existence of the policy/context/resource to be modified.

These operations alter the *policyStatusList* PSL_i (see equation 3.1 on page 49) that is used by the middleware to decide which policies to fire as a result of context changes. In order for a reactive policy to be fired, both the information encoded in a profile, and the 'firability' status of its associated resources are checked. Therefore, each time an operation that changes the reactive metadata is performed, the *policyStatusList* has to

$$\begin{aligned}
remRP &: profile \times \mathrm{P} \rightarrow profile \\
remRC &: profile \times \mathrm{P} \times \mathbb{N} \rightarrow profile \\
remRR &: profile \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow profile \\
\\
addRP &: profile \times policy \rightarrow profile \\
addRC &: profile \times \mathrm{P} \times context \rightarrow profile \\
addRR &: profile \times \mathrm{P} \times \mathbb{N} \times resource \rightarrow profile \\
\\
updRP &: profile \times \mathrm{P} \times policy \rightarrow profile \\
updRC &: profile \times \mathrm{P} \times \mathbb{N} \times context \rightarrow profile \\
updRR &: profile \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \times resource \rightarrow profile
\end{aligned}$$

Figure 3.16: Reflective Mechanism - Adaptation of Reactive Metadata.

$$\begin{aligned}
remPSP &: policyStatusList \times \mathrm{P} \rightarrow policyStatusList\\
remPSC &: policyStatusList \times \mathrm{P} \times \mathbb{N} \rightarrow policyStatusList\\
remPSR &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow policyStatusList\\
\\
addPSP &: policyStatusList \times policy \rightarrow policyStatusList\\
addPSC &: policyStatusList \times \mathrm{P} \times context \rightarrow policyStatusList\\
addPSR &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times resource \rightarrow policyStatusList\\
\\
updPSP &: policyStatusList \times \mathrm{P} \times policy \rightarrow policyStatusList\\
updPSC &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times context \rightarrow policyStatusList\\
updPSR &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \times resource \rightarrow policyStatusList
\end{aligned}$$

Figure 3.17: Reflective Mechanism - Adaptation of Reactive Metadata (*policyStatusList*).

be updated consequently, using the semantic functions listed in Figure 3.17; the new *policyStatusList* obtained is then used in equation 3.1 to decide which policies to fire. The semantics of these functions is trivial and is reported in Appendix A for completeness; simply, remove/add/update operations respectively remove/add/update the specified policy/context/resource status from the *policyStatusList*.

Proactive Meta-Interface

As before, the meta-interface to proactive metadata accesses associations at various levels of granularity, both during inspection and adaptation: from the resources associated with a context configuration, to the configurations associated with a policy, the policies associated with a service, up to the services themselves. Figures 3.18 and 3.19 list the semantic functions that make up this meta-interface; their full semantics is reported in Appendix A for completeness.

$$\begin{aligned}
readPS &: profile \times \mathrm{S} \rightarrow service \cup \{null\}\\
readPP &: profile \times \mathrm{S} \times \mathrm{P} \rightarrow policy \cup \{null\}\\
readPC &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \rightarrow context \cup \{null\}\\
readPR &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow resource \cup \{null\}
\end{aligned}$$

Figure 3.18: Reflective Mechanism - Inspection of Proactive Metadata.

Read operations are based on unique service, policy, resource names and context ids;

null is used for an empty search result. Remove, add and update operations exhibit a semantics very similar to the one discussed for reactive encoding: remove and update operations succeed if and only if the specified entity (be it a service, policy, context or resource) exists in the profile, while add operations require its non-existence.

$$\begin{aligned}
remPS &: profile \times \mathrm{S} \rightarrow profile \\
remPP &: profile \times \mathrm{S} \times \mathrm{P} \rightarrow profile \\
remPC &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \rightarrow profile \\
remPR &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow profile \\
\\
addPS &: profile \times service \rightarrow profile \\
addPP &: profile \times \mathrm{S} \times policy \rightarrow profile \\
addPC &: profile \times \mathrm{S} \times \mathrm{P} \times context \rightarrow profile \\
addPR &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \times resource \rightarrow profile \\
\\
updPS &: profile \times \mathrm{S} \times service \rightarrow profile \\
updPP &: profile \times \mathrm{S} \times \mathrm{P} \times policy \rightarrow profile \\
updPC &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \times context \rightarrow profile \\
updPR &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \times resource \rightarrow profile
\end{aligned}$$

Figure 3.19: Reflective Mechanism - Adaptation of Proactive Metadata.

While the set of reactively fired policies depends on both the information encoded in a profile, and the status of the associated resources (their 'firability' condition), the way a proactive service is delivered depends solely on its currently encoded associations. Therefore, the semantics of adaptation of proactive information is fully captured by these functions.

3.4 Related Work

It is not the aim of this thesis to provide a detailed literature review in the area of mobile computing middleware, as this would require the discussion of issues such as asynchronous communication, service discovery, and so on, that have not been investigated in this thesis. For a more exhaustive discussion on the state-of-the-art of mobile computing middleware, the interested reader may refer to [Mascolo et al., 2002a].

In this section, we discuss instead our position compared to the works done in the area of mobile computing, as far as context-awareness and dynamic adaptation to changes are concerned. We structure the discussion into three main parts, based on what currently available systems achieve: *context sensing*, *application adaptation*, and *middleware adaptation.*

3.4.1 Context Sensing

One of the first issues that context-aware systems have to tackle is gathering context information and processing it in a manner that is meaningful to the (context-aware) application. Researchers in context-aware computing [Schilit et al., 1994] have studied and developed systems that collect context information, such as location, with varying accuracy depending on the positioning system used; relative location, such as proximity to printers and databases; device characteristics, such as processing power and input devices; physical environment, such as noise level and bandwidth; and user's activity, such as driving a car or sitting in a lecture theatre.

In particular, location has attracted a lot of attention and many examples exist of applications that exploit location information in order to: offer travellers directional guidance, such as the Shopping Assistant [Asthana and Krzyzanowski, 1994], CyberGuide [Long et al., 1996] and GUIDE [Davies et al., 1999]; find out neighbouring devices and the services they provide, such as Teleporting [Bennett et al., 1994]; send advertisements depending on user's location, such as People and Object Pager [Brown, 1998]; send messages to anyone in a specific area, such as Conference Assistant [Dey et al., 1999]; and so on. Most of these systems interact directly with the underlying network operating system to extract location information, process it, and present it in a convenient format to the user. These approaches suffer from two major drawbacks: first, they are mainly ad-hoc solutions, that can be used for the development of one particular application, but cannot be reused, thus requiring application engineers to 're-invent the wheel' each time a new location-aware application has to be developed. Second, they do not cope with heterogeneity of coordinate information, and therefore different versions have to be released that are able to interact with specific sensor technologies, such as the Global Positioning System (GPS) outdoors, and infrared and radio frequency indoors.

To enhance the development of location-based services and applications, and reduce their development cycle, middleware systems have been built that integrate different positioning technologies by providing a common interface to the different positioning systems. Examples include Oracle iASWE [Oracle Technology Network, 2000], Nexus [Fritsch et al., 2000], Alternis [Alternis S.A., 2000], SignalSoft [SignalSoft, 2000], and CellPoint [CellPoint, Inc., 2000]. [Leonhardt and Magee, 1996], for example, describe a framework for a general location service where multiple sources of location information are integrated for location information acquisition, processing and presentation. First, raw location data is acquired from the physical sensors and represented as cells in a symbolic space. Then, the cell space is processed into a zone space, that is a set of disjoint cells used for object tracking and movement prediction; finally, zones are mapped into hierarchical domains, with associated access control policies to protect users' privacy. There exist also more general approaches, that do not only cater for location information: the Context Toolkit [Salber et al., 1999], for example, provides a set of context servers, each responsible for the sensing and processing of different context information.

We take a similar approach to context sensing, by investing the middleware with the task of gathering and maintaining context information. While these tasks are trivial when considering local resources, such as battery and memory, difficulties arise when context extends across multiple mobile hosts in an ad-hoc network, to include resources whose status cannot be measured directly by the host itself (because, for example, it does not possess the sensors necessary to gather such information). In presenting our approach, we have voluntarily left these issues under-specified: we assume (see Figure 2.1 in Chapter 2) that middleware is able to maintain context information, by interacting with an entity that can be a primitive of the operating system, a local physical sensor, or a more advanced software agent that is able to interact with other hosts and sensors in the network. [Roman et al., 2002, Julien and Roman, 2002] provide a formal definition of context, together with algorithms for computing and maintaining a specified context. The solution maps nodes of an ad-hoc network to points in a multi-dimensional space, and considers context as the set of all nodes whose distance from the host where the application of interest is running, is below an application-defined bound. This solution could be used by our middleware model too, to broaden the context of interest to an application to an entire ad-hoc network; however, the problem of binding and re-binding of external sensors while on the move has not yet been completely solved and remains therefore an open issue.

3.4.2 Application Adaptation

Being able to gather and process context information is only the first step towards the development of context-aware applications. Applications, in fact, must be provided with mechanisms and primitives to adapt to context changes. Various systems exist that achieve this goal to different extents and in different ways. In particular, we distinguish between systems that support: *reactive adaptation*, *multimedia adaptation* and *application interface* adaptation.

Reactive Adaptation

There are systems that provide application adaptation to context changes only as far as what we called reactive adaptation is concerned. They provide applications with primitives to specify what portion of context they are interested into, and with mechanisms to be alerted when context configurations of interest to the application are entered.

[Welling and Badrinath, 1998], for example, discuss a publish-subscribe architecture, where applications register their interest in particular context changes, and then an event delivery mechanism notifies registered applications of relevant changes happening in the environment. The set of events that can be detected and delivered is extensible and can be dynamically modified by the application. It is then entirely up to the application to decide what to do (i.e., how to adapt) once these changes have been notified.

Odyssey [Satyanarayanan, 1996] goes one step further: not only does it notify applications of context changes, but it also provides applications with primitives to register the behaviours that the system should automatically invoke when specific context configurations are entered. Odyssey is a platform for mobile data access: first, applications register an interest in particular resources, by defining the acceptable upper and lower bounds on the availability of that resource, and by registering an 'up-call procedure' that must be invoked whenever the availability of the resource falls outside the window of acceptance. A 'Viceroy' component is then responsible for monitoring resource usage and notifying applications of significant changes, using the registered up-calls. When an application is notified of a change in resource availability, it must adapt its access. 'Warden' components are responsible for implementing the access methods on objects of their type: they provide customised data access behaviour (e.g., different replication policies) according to type-specific knowledge.

Gaia [Román et al., 2002] offers a more general approach to reactive adaptation to context changes, as it does not focus on one particular service (data access). It is built on top of the 2k [Kon et al., 2000a] reflective, component-based meta-operating system. On top of this framework, Gaia converts physical spaces and the ubiquitous computing devices they contain into active spaces. An active space hides the complexities of dealing with heterogeneous devices and sensors, and provides application engineers with a generic interface that allows them to interact with any physical space in a uniform way. Gaia then adapts application requirements to the properties of its associated active space, without the application having to explicitly deal with the particular characteristics of every possible physical space where they can be executed.

A common limitation of these approaches is the lack of support for what we have defined as proactive adaptation to context changes. That is, they do not provide the application with mechanisms to facilitate the customisation of the services the application delivers to its user, based on context. In the case of services that require adaptation to context (e.g., the instant messaging functionality of the conference application, the talk reminder service, etc.), the application has to perform the tedious and repetitive tasks of querying its context and finding out which policy suits the current context. We go a step further, by providing applications with primitives to transfer this knowledge to the middleware (i.e., proactive metadata), thus automating these tasks.

Multimedia Adaptation

Although we are not concerned with multimedia data and applications, we briefly discuss a couple of examples from this area, as this is from where most of the early work on dynamic adaptation comes from. In the area of multimedia, dynamic adaptation to context changes becomes vital in order to achieve reasonable quality-of-service. Researchers have devised a number of interesting approaches to quality-of-service provision to mobile hosts [Chalmers and Sloman, 1999a]. Most of the time, the hosts are considered terminal nodes and the clients of the service provision, and the network connectivity is assumed fluctuating but almost continuous (like in GSM settings).

Probably the most significant example of QoS-oriented middleware is Mobiware [Angin et al., 1998], which uses CORBA, IIOP and Java to allow service quality adaptation in the delivery of multimedia to hosts in mobile settings. In Mobiware, mobile hosts are seen as terminal nodes of the network, and the main operations and services are developed on a core programmable network of routers and switches. Mobile hosts are connected to access points and can roam from one access point to another. The main idea of Mobiware is that mobile hosts will have to probe and adapt to the constantly changing resources over the wireless link. Mobiware mostly assumes a service provision scenario where mobile hosts are roaming but permanently connected, with fluctuating bandwidth. Even in the case of the ad-hoc broadband link, the host is supposed to receive the service provision from the core network through the cellular links first, and then some ad-hoc hops. In more extreme scenarios, where links are all ad-hoc, these assumptions cannot be made and different middleware technologies need to be applied.

[Chalmers et al., 2001] present a model for multimedia data adaptation to context changes and user's needs. The approach assumes that a document can be seen as a collection of elements, each of which has a type and can be represented by multiple variants (e.g., different scale, resolution, etc.). Each variant is described by an encoding format and a set of parameters (e.g., size). Whenever a user requests access to a document, the elements to be displayed are first dynamically chosen, based on 'weights' that the user associates with the element types. Each of these elements is then displayed using the variant that maximises user's preferences. These preferences are represented as values computed using a 'utility function' over variant parameters. Both weights and utility functions may vary to take into account context information, such as user's activity, screen size, etc. Similar to this approach, we aim to build a middleware model that achieves adaptation both to context changes and varying user's needs. However, what we aim to adapt is not static data but application behaviours, for which different abstractions and mechanisms are needed.

Application Interface Adaptation

In the area of Human-Computer Interaction (HCI), researchers have investigated the issue of application interface adaptation to context. Although the subject of adaptation (i.e., application interfaces), and the perspective they take (i.e., user's perspective) are fundamentally different from ours (i.e., we take an application's perspective to adapt application functionalities), some of the issues encountered, and the solutions proposed, manifest similarities that are worth discussing.

Some researchers have focused on accurate sensing of context information, such as the user's task, its social environment, and so on. This information has then been used to adapt the user interface to a set of pre-defined situations; for example, [Schmidt et al., 1999] illustrate a User Interface (UI) rotation based on awareness of the device orientation and user's task, to improve human-computer interaction.

[Eisenstein et al., 2001] discuss a more general approach to help UI designers to develop mobile application interfaces that adapt to the characteristics of the devices they run on (e.g., what kind of graphical capabilities they have, what kind of interaction capabilities, etc.), and to the context of use (e.g., noise level, light, etc.). Rather than developing unique UIs for each platform and usage, they propose to start from an abstract, platform-neutral, formal description of the UI; this description should then be understood and analysed by a software system to automatically produce a usable UI matching the requirements and constraints of each context of use. This research is at an early stage: so far, they have defined the abstract model of the UI, but they still lack the definition of the requirements and constraints it has to satisfy, and therefore the automatic mapping from the model to an actual interface. The process they are following is very similar to the one we have developed to adapt application functionalities: first an abstract description of the behaviour of the application (i.e., the abstract model) in different contexts (i.e., the constraints) is provided through application profiles; then the middleware (i.e., the software system) automatically adapts application behaviour to current context, based on the abstract description.

3.4.3 Middleware Adaptation

Instead of adapting applications, many researchers have taken a middleware-centric perspective and have investigated principles, and designed mechanisms, to achieve middleware adaptation to context. The definition of context they have is completely different to the one we gave, as they look at context from a different point of view. For example, context includes the communication paradigms being used, the service discovery protocols available, and so on. As discussed in Chapter 2 (see Figure 2.1 on page 31), this type of adaptation is complementary to the one we provide, and middleware could be structured into layers, with mechanisms to provide middleware adaptation at the bottom, and mechanisms to provide application adaptation on top. We discuss some of the most relevant approaches to middleware adaptation in this section, as they exploit the same principles we investigated, that is, reflection.

The concept of reflection was first introduced by Smith in 1982 [Smith, 1982] as a principle that allows a program to access, reason about and alter its own interpretation. Initially, reflection emerged as a technique to support the design of more open and extensible languages (e.g., [Kiczales et al., 1991]). Reflection is also increasingly being applied to a variety of other areas including operating system design [Yokote, 1992], concurrent languages [Watanabe and Yonezawa, 1988], and distributed systems (e.g., [McAffer, 1996], [Okamura et al., 1992]). There is now a growing community working on the area of reflective middleware too, mainly to provide a principled (as opposed to ad-hoc) means of achieving openness of the underlying middleware platform. A reflective middleware may bring about modifications to itself by means of inspection and/or adaptation. Through inspection, the internal behaviour of the system is exposed, so that it becomes straightforward to insert additional behaviour to monitor the middleware implementation. Through adaptation, the internal behaviour of the system can be dynamically changed, by modification of existing features or by adding new ones. This is particularly useful when dealing with portable devices which require light-weight middleware implementation, due to resource limitations: a middleware core with only a minimal set of functionalities can be installed on a device, and then, through reflection, the system can be re-configured dynamically to adapt to context changes. Examples of middleware built around the principle of reflection include, but are not limited to, OpenORB [ExoLab, 2001], OpenCorba [Ledoux, 1999], dynamicTAO [Kon et al., 2000b], Blair et al. work [Blair et al., 1998], MULTE-ORB [Plagemann et al., 1999], Flexinet [Hanssen and Eliassen, 1999], Globe [van Steen et al., 1999, Bakker et al., 1999], UIC [Román et al., 2001]. Most of the platforms developed to experiment with reflection were based on standard middleware implementations (e.g., CORBA [Pope, 1998]), and therefore targeted to a wired distributed environment. Some noticeable exceptions include OpenORBv2 [Blair et al., 2001], ReMMoC [Capra et al., 2002a], UIC [Román et al., 2001] and LegORB [Román et al., 2000]. They all share the idea of exploiting reflection and components to achieve dynamic reconfigurability of middleware.

The ReMMoC project and the Universally Interoperable Core (UIC) aim at overcoming the problems of heterogeneous middleware technology in the mobile environment. They offer developers the ability to specialise the middleware to suit different devices and environments. ReMMoC, in particular, enables dynamic re-configuration of the middleware structure and behaviour so that it can interoperate with a range of middleware platforms (e.g., RPC, message-oriented and event-based paradigms) and can discover services advertised using different service discovery protocols (e.g., SLP and UPnP). UIC, instead, concentrates on synchronous communication paradigms, less suited to mobile settings, and does not directly address the key property of heterogeneous service discovery.

LegORB exploits reflection and component technology to provide a minimal CORBA implementation for portable devices; its core implements the low-level functionalities required to guarantee CORBA interoperability. On top of this core, it provides a 'Configurator Component' that allows on-the-fly, consistent instantiation of other components, for example, for marshaling and unmarshaling, in order to dynamically suit application needs.

OpenORBv2 offers a general approach that achieves both backward compatibility with middleware standards (in particular, Microsoft's COM component model and the OMG's CORBA distributed programming environment), and dynamic and efficient middleware re-configurability. At the core is a light-weight component model, called OpenCOM, built on top of a subset of Microsoft's COM where high-level features, such as distribution, persistence, security and transactions, are discarded in favour of efficiency. On top of this core, OpenORBv2 adds support for pre- and post- method call interception, that enables the injection of monitoring code (e.g., to drive re-configuration policies), and the addition of new behaviours. In order to constraint the scope and effect of re-configuration, OpenORBv2 is structured as a set of nested component frameworks; by changing the component frameworks contained in the various layers, new platform architectures can be defined and customised to support, for example, transactions, security, etc.

As we have shown in this chapter, reflection can be effectively used at a higher level of abstraction, to customise the way middleware delivers services to applications (i.e., proactive adaptation), and not only 'low-level' middleware services, such as communication and service discovery. In [Capra et al., 2002a], we discuss how CARISMA, a realisation of our reflective middleware model, could sit on top of ReMMoC: ReMMoC presents the ability to develop applications independently from specific middleware technologies that may be encountered over time, by allowing the service discovery and communication implementation to be adapted dynamically. On top of it, CARISMA controls application-driven adaptation, based on context information.

3.5 Summary

The development of context-aware mobile applications can be enhanced by a middleware software layer that provides application engineers with primitives to define which aspects of context are relevant to the execution of the application, and to describe how applications should adapt to relevant context changes. Middleware then implements mechanisms to detect relevant context changes, and to perform adaptation as required by the application.

This chapter has shown how the principles of reflection and metadata can be exploited to fulfil this goal. Applications encode in profiles (i.e., metadata) two types of information: reactive metadata and proactive metadata. Reactive metadata encode associations between context configurations that are of interest to the application itself, and behaviours that have to be triggered when such configurations are entered. Proactive metadata contain associations between the services the application is willing to customise, the policies used to deliver these services, and the context configurations that enable these policies. Reactive metadata is used to perform reactive adaptation, while proactive metadata is responsible for proactive adaptation. A reflective meta-interface has been defined that allows applications to dynamically inspect and alter the information encoded in their profile, thus enabling dynamic re-configuration. A formalisation of our middleware model has been presented, to unambiguously define the semantics we associate with application profiles and the meta-interface.

There are a number of known potential drawbacks of the reflective approach that need to be carefully addressed; in particular, *integrity* and *performance*. By dynamically changing the associations encoded in an application profile, the integrity of middleware and application behaviour could be compromised. The problem of maintaining integrity is minimised in our approach by highly structuring the information encoded in a profile, and providing a meta-interface that minimises the scope of changes. Moreover, different applications have got different profiles, so that changes to a profile cannot affect the behaviour of the middleware with respect to other applications. As for performance, we will demonstrate in Chapter 6 that the overhead imposed by our reflective middleware is rather limited, and can be accommodated by currently available mobile devices.

Chapter 4

QoS Conflict Resolution

In Chapter 3, we presented a mobile middleware model that exploits the principles of reflection and metadata to simplify the development of context-aware applications. Applications encode in application profiles (i.e., middleware metadata) information about how they wish context changes to be handled using policies; middleware, on behalf of the application, maintains updated context information and then adapts application behaviour dynamically, by selecting the context-suitable policy, as specified in the profiles. As a result of entering unforeseen context conditions, and/or of varying user needs, applications may wish to modify the information encoded in their profile, that is, the policies they want to be executed in different contexts; a reflective meta-interface allows applications to inspect and alter this information at run-time, thus achieving dynamic re-configuration. However, while doing so, applications may introduce ambiguities, contradictions, and other logical inconsistencies. For example, applications cooperating in the delivery of a service may not agree on a common behaviour (i.e., policy) to be applied to deliver that service, or they may request the execution of different and contradictory behaviours in the same context. We refer to these inconsistencies as *conflicts*.

In this chapter, we classify the types of conflicts that may arise in our mobile setting. We present and formalise a conflict resolution mechanism based on microeconomic techniques, and demonstrate its suitability to our problem.

4.1 Conflicts

In this section, we characterise the types of conflicts that may emerge in mobile computing, and provide examples taken from our conference application. Based on our application-centric perspective, middleware can be considered by applications as a dynamically customisable service provider; the customisation takes place by means of the associations that applications encode in their profile, and that they can dynamically alter through the reflective mechanism that our middleware provides. These associations state how applications want the middleware to deliver a service, that is, which policies should be used in different contexts. Through reflection, applications are allowed to alter the set of policies associated with a service, as well as the context conditions that lead one policy to be preferred over others; in other words, reflection allows applications to customise middleware behaviour (with respect to application service delivery) based on current user's needs. For example, in our conference application, access to the electronic proceedings may be granted using a 'cache' policy, if the user is interested in data availability when disconnected, while using a 'network reference' if the user cares more about memory consumption.

This model enhances the development of context-aware applications, by providing application designers with a mechanism to adapt to changes in context and user's needs; however, it also gives rise to conflicts.

> *In our model, a conflict exists when different policies can be used in the same context to deliver a service, and therefore the middleware does not know which one to apply.*

This definition of conflicts is based on our assumption that a service can be delivered using only one policy at a time. Note that, by removing this assumption, we do not avoid the issue of conflicts, we just need to formulate it under different terms. In particular, conflicts would appear as different *sets* of policies enabled at the same time; in this case, a definition of what *different* means should be given too (e.g., is the order in which policies appear relevant, or is the execution of policies commutative?).

Reflection gives applications the 'intelligence' that transparency takes away in traditional middleware systems. Applications, however, may not be smart enough to cope with the new power, and may encode associations that lead to conflicts. In particular, when setting up application profiles, two basic kinds of conflicts may be created: *intra-profile* conflicts and *inter-profile* conflicts.

4.1.1 Intra-profile

We call a conflict an *intra-profile conflict* if it occurs inside the profile of an application running on a particular device. This class identifies conflicts that are *local* to a host.

Let us consider, for example, the 'talkReminder' service of the conference application. As shown in Figure 4.1, Alice may instruct the middleware to use a silent alert (e.g., blinking message on the screen) when she is interacting with the system; a vibration alert when she is not actively using the system; and a sound alert when she is not in a conference room (i.e., she is not attending a talk). What happens when Alice is having a coffee during a conference break, and a reminder has to be triggered to remind her of the next talk she wishes to attend? The middleware checks which policy should be applied and determines that more than one policy suits the current context, that is, both the `vibraAlert` policy and the `soundAlert` one. As we made the assumption that each service is delivered using one and only one policy at a time, the middleware is unable to choose which of the context-suitable policies to apply. This is an example of intra-profile conflict.

```
talkReminder
    silentAlert
        1
            userFocus equals on
    vibraAlert
        2
            userFocus equals off
    soundAlert
        3
            location notEquals conferenceRoom
```

Figure 4.1: Example of Intra-profile Conflict.

4.1.2 Inter-profile

We call a conflict an *inter-profile conflict* if it occurs between the profiles of applications running on different devices and that wish to interact. This class identifies conflicts that are *distributed* among various hosts.

As a particular example of inter-profile conflict, we consider the case in which a conflict arises between applications running on *two* different devices. This scenario is typical in a mobile setting, where interactions take place between peers. Let us consider the messaging service of our conference application, and the case where Alice and Bob start exchanging messages. As shown in Figure 4.2, Alice wishes to send encrypted messages when busy (i.e., when willing to communicate privately with her closest friends), due to the confidentiality of the information exchanged, and plain text messages when on-line. Bob, on the other hand, is more concerned with resource consumption than with privacy of information, and therefore he prefers to exchange plain text messages, unless bandwidth is very low, in which case he instructs the middleware to compress messages first. What happens if Alice starts exchanging messages with Bob when busy? Or if the available bandwidth is low? As they do not agree on which policy to use to exchange messages, the communication fails. We call this situation an *inter-profile conflict*, as the conflict is not incorporated in one particular profile, but spans more than one (in this case, two). A particular case of inter-profile conflict happens when applications run on the same device; we refer to this situation as an *N-on-1* (i.e., *N* applications on 1 device) *conflict*.

```
/* Alice profile */                    /* Bob profile */

messagingService                       messagingService
    plainMsg                              plainMsg
        1                                     1
            userMood equals online                bandwidth greaterThan 0.56
    encryptedMsg                          compressedMsg
        2                                     2
            userMood equals busy                  bandwidth lessThan 0.56
```

Figure 4.2: Example of Inter-profile Conflict.

4.1.3 On the Nature of Conflicts

Before describing the requirements that a conflict resolution mechanism should meet, we discuss some important aspects of the conflicts we have exemplified.

Quality-of-Service Conflicts

Both intra- and inter- profile conflicts are Quality-of-Service (QoS) conflicts. Each of the conflicting policies associated with a service, in fact, requires different amounts of resources to be executed, and achieves different QoS levels: for example, a caching policy for the 'access proceedings' service consumes more memory than a network reference, but delivers a better quality-of-service in terms of data availability.

We choose not to define the set of quality-of-service parameters we consider further, as we do not want to constrain ourselves by any set in particular. It is likely that different applications will be concerned with different QoS parameters: availability of information may be a QoS parameter for a data-sharing application, privacy may be a QoS parameter for an e-shopping application, and so on. Also, different users will value these parameters in different ways, at different times; for example, depending on how much the user cares about data availability and memory consumption, caching policies may or may not be preferred to network reference ones.

The conflict resolution mechanism we aim to design should be able to find an optimal solution (i.e., one that delivers the best quality-of-service according to the current user's preferences) regardless of what the actual set of QoS parameters contains; in other words, these parameters should be treated as variables in our model. We cannot opt, for example, for a non-deterministic choice from the set of enabled policies, and leave these parameters behind the scene. A random choice, in fact, would not take user's preferences into account, and would be likely to fail to deliver to the user the QoS he/she expects from the system. For example, non-deterministically choosing a `soundAlert` policy instead of a `vibraAlert` may not please Alice, if she is talking to other conference attendees and discretion is a major concern to her.

Service Delivery (Proactive) Conflicts

As we have illustrated in the previous chapter, the information encoded in an application profile is divided into two parts: reactive metadata and proactive metadata. The QoS conflicts we are interested in only occur within the proactive encoding; whenever a service request is issued, different policies, achieving different QoS, may be enabled in the current context, thus requiring the middleware to choose which of them to apply to deliver the service. That is, the enabled policies are in conflict with respect to the service they have been associated with. In the reactive encoding, instead, policies are not related to each other, but only to the context conditions that enable them: if more than one policy is enabled in the current context, our middleware model expects that all of them are fired (e.g., both a 'lowMemory' alert and a 'lowBattery' alert can be triggered at the same time, as a result of drops in memory and battery availability). This chapter is therefore concerned with proactive metadata, as this is where quality-of-service conflicts occur.

4.2 Requirements

Whenever a service with conflicting policies, either intra- or inter- profile, is requested, a conflict resolution mechanism has to be run to solve the conflict and find out which policy to use to deliver the requested service, otherwise applications cannot execute. The following requirements can be associated with such a mechanism.

Dynamicity. Neither intra- nor inter- profile conflicts can be detected and resolved statically, that is, at the time the profile is written by the application. In case of intra-profile conflict, a possible static approach would require us to check whether there is any intersection between the different contexts of the policies associated with each service. Due to the complex nature of context (the number of monitored resources may be large), a static conflict analysis would produce an explosion in the context information that must be checked, and would require a consumption of resources (especially in terms of battery, memory and processing power) that portable devices cannot bear. Providing the conflict resolution as an external service on a powerful machine that is contacted on-demand is not feasible either, as this would require persistent connectivity that in mobile settings cannot be taken for granted. As for inter-profile conflicts, the situation is even worse; mobile devices connect opportunistically and sporadically. We cannot foresee which devices are going to be encountered and, even if we could, we would be unable to assume that all of them were connected and in reach at the time a profile is modified; this means that the middleware cannot statically check whether the new configuration is conflict-free. Even assuming that this distributed check could be statically performed, it would not be worth the effort, as we would find many more potential conflicts than we would actually need to be concerned with. We are only interested in conflicts that manifest themselves in the particular context in which the service is requested, and according to the current peer profiles. As a consequence, a dynamic solution is needed: conflicts may exist inside or among profiles, but both applications and middleware can live with these conflicts until a service which involves such a conflict is invoked.

Simplicity. The conflict resolution mechanism must be simple in the sense that it must not unduly consume resources that are already scarce on a mobile device. Only a low computation and communication overhead should be imposed, even if this may occasionally prevent an optimal solution to the conflict being found.

Customisation. Different applications may have different preferences, as may even the same application at different times. Asking the middleware to solve conflicts independently of the applications that requested the conflicting service would hardly do any better than a non-deterministic choice, as it would not take into consideration how much applications value the execution of the various policies. On the one hand, we do not want applications to be questioned each time a conflict is detected, that is, middleware should be in charge of carrying on the conflict resolution process and keeping it as transparent as possible to applications and users. On the other, it must be possible for the applications to customise the conflict resolution mechanism, thus influencing which policy is chosen and applied, and which others are discarded, based on how much they value the QoS parameters associated with the conflicting policies.

In the following section, we describe and formalise the conflict resolution mechanism we have designed to meet these requirements.

4.3 Microeconomic Mechanism

When applications participating in the delivery of a service cannot agree on which policy to apply, a conflict resolution scheme is necessary to resolve the dispute. We have explored microeconomic techniques [Binmore, 1992] and used them to design a conflict resolution mechanism. The motivating idea is that a mobile distributed system can be seen as an *economy*, where a set of *consumers* must make a collective choice over a set of alternative *goods*. Goods represent the various policies that can be used to deliver a service; for example, policies 'plainMsg', 'encryptedMsg' and 'compressedMsg' are the goods associated with service 'messagingService'. Consumers are applications seeking to achieve their own goals, that is, to have a service delivered using the policy that provides the best quality of service, according to application-specific preferences.

Simple schemes include, for example, priority assignment or per capita distribution. Priority assignment could be used to solve intra-profile conflicts: it would require, for example, to statically assign a priority to the various policies, so that a conflict is solved by selecting and applying the policy with the highest associated priority among the conflicting ones. Per capita distribution would solve inter-profile conflicts instead, where each application wins a conflict in turn and gets its preferred policy executed. However, these schemes do not suit situations where participation in exchange of goods is voluntary on the part of all parties (i.e., the applications), so that action requires a consensus and mutual perception of benefit. A better scheme would use an *auction protocol*. Auctions allow parties to make distributed decisions independently, on the basis of private state, revealing only offers and acceptance of the offers made by others. Applications may vary greatly in their preferences and decision processes. An auction permits greater degrees of heterogeneity than simpler schemes.

The question we have to answer next is which auction protocol to use. This is known in microeconomic theory as a mechanism design problem [Mas-Colell et al., 1995]. A *protocol*, or mechanism, consists of a set of rules that govern interactions, and by which agents (i.e., our applications) will come to an agreement. It constrains the deals that can be made, as well as the offers that are allowed. We argue that the auction protocol we have designed [Capra et al., 2002b] can be successfully applied in a mobile setting, where the requirements listed in Section 4.2 must be satisfied.

4.3.1 The Protocol: an Informal Description

In this section we provide a high-level description of the protocol we have designed to automatically resolve QoS conflicts. Using microeconomic terms, the rules of our auction can be described as follows.

> *Given a setting with N agents that must make a collective choice from a set of P possible alternatives, each agent submits a single sealed bid for each element in P. The auctioneer collects the bids and selects the alternative in P that maximises social welfare, that is, the alternative with the highest sum of bids received. Each agent then pays the auctioneer an amount of money that is proportional to the bid they placed on the winning alternative.*

The Agents

In our case, applications are the agents, and the goods they are competing for is the execution of the policy they value most, among a set of alternatives that correspond to the policies that can be applied in a particular context to deliver a service. In a human auction, the maximum amount of money an agent can bid to win an auction is limited by the maximum amount of money he/she has got. In our computer-based auction, we simulate this concept by assigning each application a certain *quota*, that is, a maximum amount of virtual money an application owns. Based on how much an agent values the goods put up for auction, different amounts of money can be offered; in our computer-based auction, we use *utility functions* to derive an offer, that is, to decide what portion of the application's quota to bid, based on current application's preferences. We will provide a detailed discussion of utility functions and quota management in Section 4.3.3 and 4.3.4 respectively.

The Auctioneer

In our case, the role of the auctioneer is played by the middleware, which we assume is a trusted entity whose code and behaviour cannot be interfered with. The aim of the middleware is not to select the policy that received the highest bid, that is, the one that maximises the selling price: virtual monies, in fact, are worthless; they are simply an abstraction we use to allow applications to express preferences in a computer-understandable way. Rather, the goal of the middleware is to satisfy the largest number of applications involved in the conflict. In our case, in fact, applications are participating in the delivery of the same service, rather than competing for it (i.e., the service will be delivered to all of them, not only to one or some of them). In these collaborative, or at least compromise scenarios, a solution that satisfies the total benefit of all the applications is preferred to one that maximises the benefit of a single one.

The auction

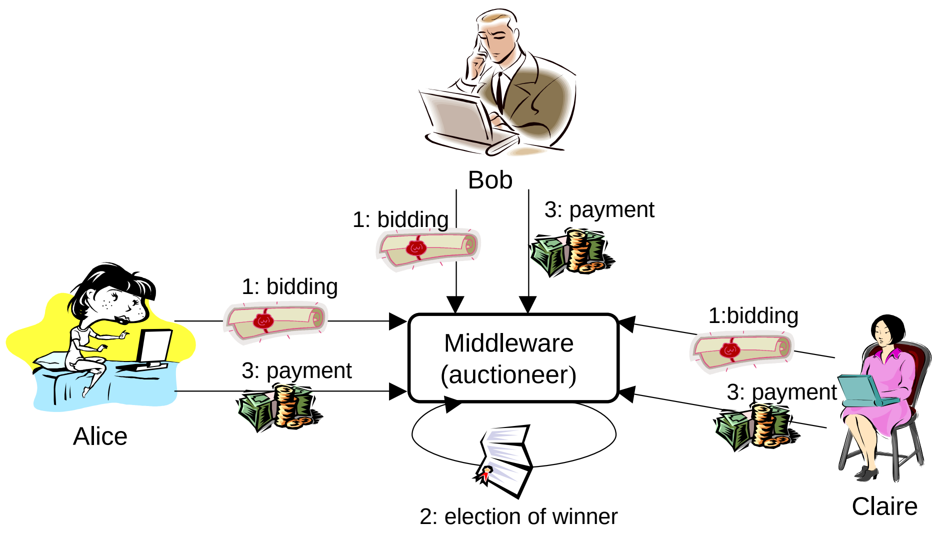


Figure 4.3: The Auction Protocol.

Let us consider the scenario depicted in Figure 4.3, where three peers (Alice, Bob and Claire) are participating in a service delivery, and they do not agree on which policy to use to deliver that service. In this case, our conflict resolution mechanism proceeds as described below.

1: Bidding: each peer uses its utility function to decide how much to bid on each conflicting policy. These values are secretly communicated to the auctioneer (that is, to the middleware) by all peers.

2: Election of the Winner: the auctioneer collects the offers, sums the bids received on each conflicting policy, determines which policy received the highest sum of bids, and selects this one as the winning policy.

3: Payment: in a human setting, the auction completes with the auctioneer collecting the money and assigning the goods to the winning agent. In our computer-based game, the goods are not uniquely assigned to one peer, as all applications will have the service delivered, but with different degrees of satisfaction. Each peer then pays the middleware an amount of money that is proportional to the added benefit obtained by applying the winning policy over the other peers. Middleware puts this money in what we call *middleware accounts*. Details about the payment scheme can be found in Section 4.3.2

Each time an application participates in an auction, money is taken from its current quota and used to pay the middleware. If we do not provide a way for money to flow back to the application, this process will leave the application without money, thus unable to participate in other auctions. To avoid this situation, middleware runs a quota redistribution service that, at regular time intervals, returns money from the middleware accounts to the applications in a way that rewards collaborative, rather than dictatorial, behaviours. We will give full details of how middleware manages this self-stabilising quota redistribution service in Section 4.3.4.

4.3.2 The Protocol: Formalisation

Having provided a high-level description of the microeconomic mechanism we have designed to resolve QoS conflicts, we can now formalise it and unambiguously define its rules. The following discussion applies to both intra- and inter- profile conflicts. To avoid confusion between an application, which may exist on different devices, and an application instance, which runs on a particular device, we will identify an application instance and the device it is executing on as a 'peer'. Peers are partners in the communication process. We call PEER the set of all possible peers; the other domain sets we refer to in the following discussion are the ones introduced in the previous chapter and reproduced in Figure 4.4. We do not describe here how coordination among different devices takes place; details of the algorithms that realise this coordination can be found in Chapter 5.

Step 1 - Initialisation. As part of an initialisation process, for every peer $peer_i$, $i \in [1, N]$, a utility function $u_i : \mathrm{P} \rightarrow \mathbb{R}^+$ that represents the user's goals (e.g., minimisation of

PEER	:	set of all peers
Σ	:	alphabet
$\mathrm{S} \subset \Sigma^*$	:	set of all service names
$\mathrm{P} \subset \Sigma^*$	:	set of all policy names
$\mathbb{N}$	:	set of all natural numbers
$\mathrm{R} \subset \Sigma^*$	:	set of all resource names
$\mathrm{O} \subset \Sigma^*$	:	set of all operator names
V	:	set of all values of resources in R
$\mathrm{E} \subset \wp(\mathrm{R} \times \mathrm{V})$	:	set of all possible execution contexts

Figure 4.4: Application Profile - Domain Sets.

battery consumption, maximisation of data availability, etc.) is determined. Peers use a utility function to specify how much they value the use of a policy $p_j \in \mathrm{P}$ during an auction, that is, $u_i(p_j) = u_{i,j}$. Each peer is also assigned a quota q_i by the middleware. The quota q_i represents the maximum amount of money that $peer_i$ can bid during a bidding process, that is, the bid placed by peer $peer_i$ on policy p_j is a number $b_{i,j} = \min\{u_{i,j},\ q_i\}$.

Step 2 - Service Request. Whenever an application requires the middleware to execute a service, a command like the one illustrated below is issued:

$$\begin{aligned} command &::= sname\ peerList \\ peerList &::= peer\ peerList \mid peer \end{aligned}$$

where *sname* $\in$ S is the name of the requested service, and *peerList* the set of peers involved in the service execution. Assuming that service *sname* requires the cooperation of $n \leq N$ peers, each peer (or, better, the middleware instance operating on the device of the peer) computes P_i as the set of policies that the above running application instance A_i has associated with service *sname* in its profile, and that can be applied in the current context (i.e., according to current resource availability). More formally, P_i can be defined as follow:

$$P_i = \mathcal{F}[\![serv(sname, peer_i)]\!]_{\mathcal{E}nv(peer_i)}$$

where $\mathcal{F}$ is the semantic function defined in Chapter 3 (see Figure 3.14 on page 52); $serv : \mathrm{S} \times \mathrm{PEER} \rightarrow service$ is a function that, given a service name and a peer, returns the corresponding service specification, and $\mathcal{E}nv : \mathrm{PEER} \rightarrow \mathrm{E}$ a function that computes the current execution environment of a peer.

Step 3 - Computation of the Solution Set. Middleware instances then cooperate to compute the *solution set* P^*, that is, the set of policies that all peers involved in the

$$\begin{array}{rcl} \mathcal{I} & : & S \rightarrow \wp(\text{PEER}) \rightarrow \wp(\text{P}) \\ \mathcal{I}[\![sname]\!]_{\{peer\ peerList\}} & = & \mathcal{I}[\![sname]\!]_{\{peer\}} \cap \mathcal{I}[\![sname]\!]_{\{peerList\}} \\ \mathcal{I}[\![sname]\!]_{\{peer\}} & = & \mathcal{F}[\![serv(sname, peer)]\!]_{\mathcal{E}nv(peer)} \end{array}$$

Figure 4.5: Semantics of the Computation of the Solution Set.

execution of the service have agreed upon:

$$P^* = \mathcal{I}[\![sname]\!]_{\{peer_1 \ldots peer_n\}}$$

$\mathcal{I}$ being the semantic function described in Figure 4.5.

If the cardinality of P^* is zero, that is, the solution set is empty, a conflict exists that cannot be automatically solved, as peers do not agree on a common policy to be applied; the conflict resolution process is terminated with a failure and peers are notified. If the cardinality is exactly 1, there is mutual agreement on the policy to apply (i.e., there is no conflict). Finally, if the cardinality is greater than 1, there is a conflict that can be resolved using one of the policies in P^*. In this case, the auctioning process proceeds as below, to decide which of these policies should be applied.

Step 4 - Computation of Bids. For every peer $peer_i$ participating in the communication process, and for every agreed policy $p_j \in P^*$, $j \in [1, m]$, a bid $b_{i,j}$ is computed, based on the peer utility function u_i and quota q_i. Unlike human auctions, we make the assumption that all peers participating in a bidding process bid a price, that is, they cannot refuse to bid. Middleware instances of bidding peers exchange the bids they have received, producing a merged set of tuples $\mathcal{B}^*$ specifying how much each peer valued the use of each agreed policy:

$$B^* = \mathcal{B}[\![\{p_1, \ldots, p_m\}]\!]_{\{peer_1, \ldots, peer_n\}}$$

$\mathcal{B}$ being the semantic function shown in Figure 4.6.

Step 5 - Election of the Winner. From the set B^*, middleware instances participating in the conflict resolution process select the winning policy $p_{\tilde{j}}$ as the one with the highest sum of the bids placed:

$$p_{\tilde{j}} = \mathcal{W}[\![B^*]\!]$$

where $\mathcal{W}$ is the semantic function defined in Figure 4.7. We make use here of the following auxiliary functions: π_i, to project a tuple onto the i^{th} value (i.e., $\pi_i(a_1, a_2, \ldots, a_n) = a_i$); $\#$ to compute the cardinality of a set (i.e., $\#\{a_1, a_2, \ldots, a_n\} = n$); $q_{mw}(i)$ to retrieve the middleware account on top of which peer $peer_i$ is executing; and, finally, pay, to move money from the application to the middleware

$$
\begin{array}{rcl}
\mathcal{B} & : & \wp(\mathrm{P}) \to \wp(\mathrm{PEER}) \to \wp(\mathrm{P} \times \mathrm{PEER} \times \mathbb{R}^+) \\
\mathcal{B}[\![\{p_1, \dots, p_m\}]\!]_{\{peer\ peerList\}} & = & \mathcal{B}[\![\{p_1, \dots, p_m\}]\!]_{\{peer\}} \cup \mathcal{B}[\![\{p_1, \dots, p_m\}]\!]_{\{peerList\}} \\
\mathcal{B}[\![\{p_1, \dots, p_m\}]\!]_{\{peer\}} & = & \bigcup_{j=1}^{m} \{(p_j,\ peer,\ \min\{q_{peer}, u_{peer,j}\})\} \\
\mathcal{B}[\![\{p\}]\!]_{\{peerList\}} & = & \{(p,\ \_\,, 0)\} \ \text{No conflict} \\
\mathcal{B}[\![\emptyset]\!]_{\{peerList\}} & = & \emptyset \qquad\qquad \text{No agreement}
\end{array}
$$

Figure 4.6: Semantics of the Computation of Bids.

$$
\begin{array}{rcl}
\mathcal{W} & : & \wp(\mathrm{P} \times \mathrm{PEER} \times \mathbb{R}^+) \ \to \mathrm{P} \\
\mathcal{W}[\![\{(p_j, peer_i, b_{i,j}), & & \\
\forall i \in [1,n], j \in [1,m]\}]\!] & = & p_{\tilde{\jmath}} \mid \\
 & & p_{\tilde{\jmath}} \in \{\pi_1(p_j, peer_i, b_{i,j}),\ \forall i \in [1,n], j \in [1,m]\} \\
 & & \wedge \sum_{i=1}^{n} \pi_3(p_{\tilde{\jmath}}, peer_i, b_{i,\tilde{\jmath}}) \ = \ \max_{j \in [1,m]} \sum_{i=1}^{n} \pi_3(p_j, peer_i, b_{i,j}) \\
 & & \wedge\ pay(q_{mw}(i), f_i, q_i),\ \forall i \in [1,n] \\
\mathcal{W}[\![\{(p,\ \_\,, 0)\}]\!] & = & p \ \text{ No conflict} \\
\mathcal{W}[\![\emptyset]\!] & = & \epsilon \ \text{ No agreement}
\end{array}
$$

$$
f_i \ = \ \begin{cases}
\text{a. } 0 \text{ if } \forall k \in [1,n]\ \pi_3(p_{\tilde{\jmath}}, peer_k, b_{k,\tilde{\jmath}}) = \max_{j \in [1,m]} \pi_3(p_j, peer_k, b_{k,j}) \\
\text{b. } \sum_{\substack{l \in \{s \mid s \in [1,n] \\ \wedge b_{s,\tilde{\jmath}} \leq b_{i,\tilde{\jmath}}\}}} \frac{b_{l,\tilde{\jmath}} - \max(\ \{b_{s,\tilde{\jmath}} \mid b_{s,\tilde{\jmath}} < b_{l,\tilde{\jmath}},\ s \in [1,n]\} \cup \{b_{min,\tilde{\jmath}}\}\ )}{\#\{b_{s,\tilde{\jmath}} \mid b_{s,\tilde{\jmath}} \geq b_{l,\tilde{\jmath}},\ s \in [1,n]\} * \#\{b_{s,\tilde{\jmath}} \mid b_{s,\tilde{\jmath}} = b_{l,\tilde{\jmath}},\ s \in [1,n]\}}, \\
b_{min,\tilde{\jmath}} = \min\{b_{i,\tilde{\jmath}}, i \in [1,n]\} \text{ otherwise}
\end{cases}
$$

Figure 4.7: Semantics of the Election of the Winning Policy.

$(pay(q_1, x, q_2) = (q_1 + x, q_2 - x))$.

As shown, each peer pays an amount of money that is proportional to the 'added' benefit obtained by applying the winning policy over the other peers. To understand how the payment is split, let us consider three peers x, y and z, who bid $b_x < b_y < b_z$ respectively on a winning policy p. Applying p gives an equal benefit of b_x to each peer; moreover, y and z share an added benefit of $b_y - b_x$ over x, and z enjoys an extra benefit equal to $b_z - b_y$ over both x and y. Our payment scheme demands that x, y and z pay 0, $(b_y - b_x)/2$, and $(b_y - b_x)/2 + (b_z - b_y)/1$ respectively. Note that, if the winning policy is the one that has been valued most by all peers (i.e., $b_x = \max_i b_{i,x}$, $b_y = \max_i b_{i,y}$, $b_z = \max_i b_{i,z}$), then no payment is demanded, as there was no real conflict to be solved. Note also that, in case of intra-profile conflicts, the payment is always zero, as the winning policy is never 'imposed' on anyone, that is, there is no added benefit over anyone. The rationale for this payment scheme is that applications are not paying for the resources they use when applying a policy, but, rather, for the (added) quality-of-service level the policy gives them. We assume that ties are broken by selecting a policy randomly (i.e., a k-way tie is decided by flipping a 'k-sided coin', where each policy is chosen with probability $1/k$).

If a service *sname* is requested that requires the cooperation of a set of peers *peerList*, then the whole conflict resolution mechanism can be summarised as follows:

$$\begin{aligned} \mathcal{G} &: \textit{command} \rightarrow \mathrm{P} \\ \mathcal{G}[\![\textit{sname peerList}]\!] &= \mathcal{W}\ [\![\ \mathcal{B}\ [\![\ \mathcal{I}[\![\textit{sname}]\!]_{\{peerList\}}\ ]\!]_{\{peerList\}}\ ]\!] \end{aligned}$$

A service request may then produce one of the following two results:

$\mathcal{G}[\![\textit{sname peerList}]\!] = \textit{pname}$**:** service *sname* is delivered using policy *pname* (either because all peers agreed on the policy, or because *pname* was the policy selected during a conflict resolution process);

$\mathcal{G}[\![\textit{sname peerList}]\!] = \epsilon$**:** the service request fails as no policy can be found that is both agreed on by all peers and valid in the current context.

Of this five-step procedure, step 1 is performed only at application startup; step 2 and 3 are part of every service request; only in the event of a solution set P^* of cardinality greater than one, the conflict resolution process (step 4 and 5) is actually run.

The auctioning mechanism has been described in the general situation where there are different applications running on different hosts (inter-profile conflict). N-on-1 conflicts are detected and solved in the same way as inter-profile conflicts. However, as the application instances involved are running on the same host, no communication overhead is required, and both the solution set P^*, the bids B^* and the winning policy $p_{\bar{\jmath}}$ can be computed locally. Intra-profile conflicts can be considered a degeneration of inter-profile conflicts, where the number n of bidders is 1, and the solution set coincides with P_1 (i.e., the set of policies that can be applied in the current execution context, according to $peer_1$ application profile). The auction proceeds as described above, selecting the policy that maximises this peer utility, without communication costs.

The microeconomic mechanism we have designed can resolve conflicts, but it cannot remove them. Conflicts, in fact, are usually not local to a profile but distributed among the profiles of different peers. If the peers involved change, or if the context changes, there may be no conflict at all. Also, the result of the auction is not stored as each auction is carried out in isolation and cannot be repeated: we cannot assume that next time the same conflict arises, the winning policy will be the same one, as the result depends on current peer quotas, utility functions and application profiles. Therefore, each conflict resolution process stands alone.

There are a few questions that need to be answered about the process described above; in particular, how is a utility function defined, and how is the quota managed? We answer these questions in the following sections.

4.3.3 Utility Function

Whenever an intra- or inter- profile conflict is detected, user goals, such as availability of information for the 'access proceedings' service of the conference application, or privacy for the instant messaging functionality, must be taken into account. In other words, users should be allowed to influence the conflict resolution process operated by the middleware as they are the only ones who know what their goals are at that moment, and how different outcomes are valued.

Utility functions serve this purpose. A utility function u_i translates peer i's goals into a value $u_{i,j}$, that represents the price the peer is currently willing to pay to have policy p_j applied, that is, to see its goals fulfilled. The following holds:

$$u_{i,j} \geq 0,\ \forall i \in [1, n],\ j \in [1, m].$$

As in human auctions, values cannot be negative; a value $u_{i,j} = 0$ means that policy p_j is not relevant to peer i, that is, the peer does not receive any benefit from applying p_j (this is a plausible 'machine' representation of a human who refrains from bidding).

Utility functions vary *dynamically* to reflect changes in the user goals; however, the value they return is computed over *static* policy specifications which estimate the *consumption* of resources that applying the policy entails, and the *benefits* it gives in terms of quality-of-service. If $\mathrm{R} \subset \Sigma^*$ defines the set of resource names that the middleware monitors, and $\mathrm{Q} \subset \Sigma^*$ the set of benefits achieved by applying policies in P, then a policy specification can be described as a domain set:

$$\mathrm{PSPEC} = \wp(\{\mathrm{R} \cup \mathrm{Q}\} \ \times \ level)$$

where $level ::= \ '1' | \ldots | 'LMAX'$ is an estimate of resource consumption/benefit achieved that the policy developers compute before delivering the policy. Policy specifications are therefore divided into two parts: one that provides an estimate of the consumption of resources that applying the policy would cause, and one that estimates the level of quality-of-service that is expected to result from the application of this policy. We assume these ratings are computed by policy developers and provided together with the policy code.

The more accurate these ratings are, the more faithfully user goals will be translated into bids; however, we do not require these numbers to be 'absolute', so that the specifications of all possible policies can be compared. The accuracy of estimates is local to the policies associated with a service. For example, the estimated consumption of battery to send encrypted messages must be higher than the one estimated to send plain messages; however, we do not care how battery consumption for the 'encryptedMsg' policy of the 'messaging' service compares to the 'cacheAbstract' policy of the 'access proceedings' service.

For each application, a utility function is then defined that makes explicit the resources the user is interested in preserving, and the quality-of-service parameters he/she wishes to achieve. Users associate weights with each of these resources/parameters that represent the importance they attribute to them (the higher the weight, the more important the resource/QoS parameter). The abstract syntax of a utility function is given in Figure 4.8, where *cb_name* $\in (\mathrm{R} \cup \mathrm{Q})$ is a name that uniquely identifies a resource or benefit inside a policy specification, and *weight* $::= \ '1'|\ldots|'\mathrm{WMAX}'$ represents the importance the user associates with a particular resource/benefit.

$$\begin{aligned} ufunction &::= addendList \mid \varepsilon \\ addendList &::= addend\ addendList \mid addend \\ addend &::= cb\_name\ weight \end{aligned}$$

Figure 4.8: Utility Function Abstract Syntax.

Both the resources/benefits listed in an utility function, and the weights associated with them, may vary over time, in order to represent current user needs, with regard to a particular application, as faithfully as possible. Although we consider the issue of generating weights that represent user needs as faithfully as possible a matter of future research, we will give a flavour of how these numbers can be obtained from users and be directly used in Chapter 6.

Whenever a peer $peer_i$ is involved in a bidding process, its utility function is retrieved and used to find the peer utility value $u_{i,j}$ for each conflicting policy p_j. The semantics of a utility function is presented in Figure 4.9. The following auxiliary functions have been used: $\mathcal{S} : (\mathrm{R} \cup \mathrm{Q}) \rightarrow \mathrm{PSPEC} \rightarrow level$, that, given a resource/benefit name *cb_name*, and a policy specification *ps*, fetches the *level* associated with *cb_name* in *ps* (if the utility function tries to retrieve a value for a resource/benefit that does not appear in the policy specification, the returned value is 0). *intval* is a function that, given a literal in $\{'1', \ldots, '\mathrm{MAX}'\}$, returns the corresponding integer value in $[1, \mathrm{MAX}]$. $LMAX * WMAX * RQMAX$ represents the maximum bid an application can place, where $RQMAX$ is the maximum number of resources/benefits of interest to an application.

Informally, for each resource/benefit that appears in the utility function, the corresponding

$$\begin{array}{rcl}
\mathcal{U} & : & \mathit{ufunction} \rightarrow \text{PSPEC} \rightarrow \mathbb{R}^+ \\
\mathcal{U}[\![\mathit{addend}\ \mathit{addendList}]\!]_{ps} & = & \mathcal{U}[\![\mathit{addend}]\!]_{ps} + \mathcal{U}[\![\mathit{addendList}]\!]_{ps} \\
\mathcal{U}[\![\mathit{addend}]\!]_{ps} & = & \mathcal{U}[\![\mathit{cb\_name}\ \mathit{weight}]\!]_{ps} \\
\mathcal{U}[\![\mathit{cb\_name}\ \mathit{weight}]\!]_{ps} & = & \dfrac{\mathit{intval}(\mathcal{S}[\![\mathit{cb\_name}]\!]_{ps}) * \mathit{intval}(\mathit{weight})}{\mathit{LMAX} * \mathit{WMAX} * \mathit{RQMAX}}, \\
\mathcal{U}[\![\varepsilon]\!]_{ps} & = & 0
\end{array}$$

Figure 4.9: Semantics of Utility Functions.

level (i.e., policy developer's estimate) is fetched from a policy specification. This value is then multiplied by the *weight* the user has associated with this resource/benefit in the utility function (the higher the multiplying factor, the more important the resource/benefit). These values are then added, and the sum is returned and interpreted as the price the application is willing to pay to have that policy applied. As shown, each value is normalised to vary in a range $[0, 1]$, so that different bids can be compared effectively, and money fairly redistributed (see Section 4.3.4).

Note that, to avoid incompatibility among the prices bid during a conflict resolution process, utility functions are locked at the beginning of an auction, and cannot be modified until the auction finishes. Thus, applications cannot 'cheat' and associate high bids with the policies they value most, while bidding zero for the others, to increase the chances of having the policy they value most finally applied, as this would require applications to change the weights of their utility functions during the auction.

4.3.4 Quota Allocation

When describing the rules of our mechanism (Section 4.3.2), we specified that each peer $peer_i$ is allowed to bid a value $b_{i,j}$ for policy p_j, given that this value is lower than its current quota q_i. We now explain how this quota is managed.

Whenever an application instance A_i is started, an initial quota $q_i = q_{init}$ is granted. Each time A_i participates in a bidding process, its current quota is decreased by an amount f_i that is proportional to the added benefit A_i got from applying the winning policy over other applications participating in the service delivery. A_i's underlying middleware collects A_i payments and stores them in an account $\overline{q}(i)$. We assume that there is no flow of money from one peer to another (i.e., each application instance pays its underlying middleware instance). Moreover, we assume that there is no explicit utility transfer among applications (e.g., no money can be transfered to a peer to compensate for a disadvantageous agreement).

Every t time units, each middleware instance redistributes the money it has collected in the accounts it manages $\overline{q}(i)$, $i \in [1, n]$, to the various application instances A_i, $i \in [1, n]$. The amount of money each application instance gets back is in direct relation to the number of interactions it has been involved in during the last t time units, and in inverse relation to the amount of money it has bid. We define an interaction as a service request which involves an inter-profile conflict (intra-profile conflicts are excluded from the quota recharging as no flow of money occurs).

In particular, if we indicate with $N_t(i)$ the number of interactions in which application instance A_i was involved in the last t time units, then the recharging process is carried out as described below:

$$\begin{aligned} q_i &= q_i + \left(\overline{q}(i) - \frac{\overline{q}(i)}{N_t(i)}\right) \\ \overline{q}(i) &= \frac{\overline{q}(i)}{N_t(i)} \end{aligned}$$

$\overline{q}(i)$ being the money currently stored by the middleware in the account associated with A_i, and q_i being A_i's current quota.

This quota redistribution scheme discourages dictatorial interactions: if an application instance bids very highly in a few interactions, 'imposing' its preferred policy over the others, then only a very low amount of money is returned during a recharging process. The only way to get money back from the middleware is to participate in other interactions in a more cooperative fashion (i.e., by bidding lower and interacting more). For example, let us assume that at time t_0, two applications instances A_1 and A_2 are started and are awarded the same quota $q_i = 3$, $i \in \{1, 2\}$. During the following t time units, they are involved in a number of interactions that cost them altogether the same amount of money; however, while A_1 bid aggressively, paying a lot of money in few interactions, A_2 was more cooperative, paying low amounts in many interactions. As a result, our quota redistribution scheme returns money to A_2 faster than to A_1 (see Figure 4.10).

The approach to quota redistribution that we have described could be defined as 'conservative': at any time, an application instance A_i has got the same amount of money, although split differently between its current quota q_i and the corresponding middleware account $\overline{q}(i)$. In other words:

$$\overline{q}(i) + q_i = q_{max},$$

q_{max} being a fixed amount that is the same for any application. At time t_0 when an application instance A_i is started, different choices of q_{init} and $\overline{q}(i)$ are possible. In particular,

Time / Action	$\mathbf{q_1}$	$\overline{\mathbf{q}}(\mathbf{1})$	$\mathbf{q_2}$	$\overline{\mathbf{q}}(\mathbf{2})$
t_0 / Start	3	0	3	0
t_1 / Bid	2.1	0.9	2.7	0.3
t_2 / Bid	1.2	1.8	2.4	0.6
t_3 / Bid			2.1	0.9
t_4 / Bid			1.8	1.2
t_5 / Bid			1.5	1.5
t_6 / Bid			1.2	1.8
t_7 / Redistribution	**2.1**	0.9	**2.7**	0.3

Figure 4.10: Example of Quota Redistribution (with $t_7 - t_0 = t$).

any assignment that complies with the following equations is acceptable:

$$\forall \alpha \in [0,1] \begin{cases} q_{init} = \alpha \cdot q_{max} \\ \overline{q}(i) = (1-\alpha) \cdot q_{max} \end{cases}$$

Setting $\alpha = 1$ favours newly started application instances, while setting $\alpha = 0$ favours applications that have been executing for a long while. The differences between these possibilities disappear over time.

4.4 Examples

In the previous section we have formally discussed our auctioning approach to the conflict resolution problem. We now illustrate how this mechanism can be instantiated and used to solve conflicts. In particular, we refer to our conference application and present both an example of intra-profile conflict, and two of inter-profile conflicts, and show how our auctioning mechanism can be successfully applied to resolve them.

4.4.1 Intra-profile Conflict: Talk Reminder

One of the services that our conference application can customise is the way talk reminders are issued. In particular, the application can choose one of three different policies: a `soundAlert` policy, a `vibraAlert` policy, and a `silentAlert` policy. Each of these policies requires different amounts of resources to be used (in particular, battery), and achieves a different quality of service (in terms of focusing and privacy). The corresponding policy specifications are shown in Figure 4.11; we are still not interested in implementation details (in particular, in the way policy specifications and utility functions are actually encoded); we therefore use the abstract syntax illustrated in the previous section to discuss the

```
soundAlert: {(battery,6),(privacy,1),(focusing,8)}
vibraAlert: {(battery,10),(privacy,7),(focusing,8)}
silentAlert: {(battery,1),(privacy,10),(focusing,2)}
```

Figure 4.11: Example of Policy Specifications.

```
talkReminder
    soundAlert
        1
            location equals outdoor
    vibraAlert
        2
            location equals indoor
    silentAlert
        3
            location equals indoor
            battery lessThan 15%
```

Figure 4.12: Example of Application Profile.

following examples.

Whenever a talk reminder is due, the application profile is consulted to find out which policy to apply. Let us assume that Alice's ($peer_1$) application profile is the one illustrated in Figure 4.12, and that the talk reminder service is invoked when the user is attending a talk (i.e., `location = indoor`), and battery is lower than 15%, so that both `vibraAlert` and `silentAlert` are enabled. This is an example of intra-profile conflict.

Note that, although it could be argued that such a conflict would not exist if the profile were properly written (i.e., if a line containing `battery > 15%` were added to the context of the `vibraAlert` policy), avoiding context overlaps is not so easy. When the number of resources associated with a context increases, the chances of making mistakes and of writing profiles with context overlaps increase quickly. As already argued, a static conflict analysis would be unmanageable on portable devices, and therefore a dynamic solution is needed. We now illustrate how our dynamic conflict resolution mechanism works effectively to solve this conflict. We assume that the initialisation process (step 1) has already been performed, and therefore a quota has been assigned, and the utility function illustrated in Figure 4.13 is in use. Also, we assume that a talk reminder service request (step 2) has

```
battery       2
privacy      10
focusing     10
```

Figure 4.13: Example of Utility Function Specification. $peer_1$ aims at maximising privacy and focusing, without paying too much attention to battery consumption.

been issued, and the set of locally enabled policies has been computed.

Step 3 - Computation of the Solution Set.

First, the solution set P^* is computed; as only one peer is involved, P^* coincides with P_1, that is, with the set of policies enabled in the current context by $peer_1$'s application profile:

$$\mathcal{I}[\![\texttt{talkReminder}]\!]_{\{peer_1\}} = \quad P_1 \quad = \{\texttt{vibraAlert}, \texttt{silentAlert}\}$$

As the cardinality of the solution set is greater than 1, a conflict exists that middleware can automatically solve using our conflict resolution mechanism.

Step 4 - Computation of Bids

For each policy in the solution set, a bid indicating how much $peer_1$ values the execution of that policy has to be computed. High weights associated with resources in utility function specifications mean that the user aims to preserve resources; however, policy specifications estimate the amount of resources consumed, not preserved. In order to give higher scores (i.e., higher bids) to the policies that reduce resource consumption, we therefore need to compute the value: LMAX$-$ *expected consumption.* For example, if we assume LMAX $= 10$, WMAX $= 10$, and RQMAX $= 6$ (i.e., memory, battery, bandwidth, focusing, availability and privacy), then:

$$\begin{aligned} u_{peer_1}(\texttt{vibraAlert}) &= \frac{(10-10)*2+7*10+8*10}{10*10*6} \\ &= 150/600 = 0.25 \end{aligned}$$

Assuming that the peer quota $q_{peer_1} > 1$ (i.e., the bid is not constrained by current quota, as each bid $b_{1,j} \in [0,1]$), we obtain:

$$\begin{aligned} \mathcal{B}[\![\{\texttt{vibraAlert}, \texttt{silentAlert}\}]\!]_{\{peer_1\}} &= \{(\texttt{vibraAlert}, peer_1, 0.25), \\ &\quad (\texttt{silentAlert}, peer_1, 0.23)\} \end{aligned}$$

Step 5 - Election of the winner

As only one peer is involved in an intra-profile conflict, maximising social welfare coincides with maximising individual utility. The winning policy is therefore the one that received the highest bid:

$$\mathcal{W}[\![\ \mathcal{B}[\![\{\texttt{vibraAlert}, \texttt{silentAlert}\}]\!]_{\{peer_1\}}]\!] = \texttt{vibraAlert}$$

No payment is needed for intra-profile conflicts, as there is no peer over which $peer_1$ had an extra benefit; the conflict resolution process simply ends with the execution of the `vibraAlert` policy.

4.4.2 Inter-profile Conflict: Messaging Service

Another service that our conference application can customise is the way messages are delivered among peers that have started a chat. In particular, the application designer provides four different policies among which the application can choose: a `charMsg` policy, which is used to exchange a character at a time; a `plainMsg` policy, which is used to deliver a line of characters at a time; a `compressedMsg` policy, which is used to exchange compressed messages, and finally an `encryptedMsg` policy used to exchange encrypted lines of characters. Once again, each of these policies requires a different amount of resources (in particular, battery and bandwidth), and achieves a different quality of service (in terms of availability and privacy of the message). The corresponding policy specifications are shown in Figure 4.14.

```
charMsg: {(battery,4),(bandwidth,10),(availability,10)}
plainMsg: {(battery,3),(bandwidth,6),(availability,7)}
compressedMsg: {(battery,5),(bandwidth,4),(availability,5)}
encryptedMsg: {(battery,6),(bandwidth,7),(availability,4),(privacy,10)}
```

Figure 4.14: Example of Policy Specifications.

Let us suppose that Alice ($peer_1$), Bob ($peer_2$), and Claire ($peer_3$) are now in reach of each other and want to start a chat. In order to do so, they have to agree on a common policy to be applied to exchange messages. During the lifetime of the chat, the policy used may change to adapt to new context configurations where the currently used policy is no longer suitable. However, when this happens, all the chatting peers must agree on the new policy to use.

The peers' application profiles are represented in Figure 4.15. The first peer enables each of the four policies in different contexts; the second peer prevents the use of the two heaviest policies, `charMsg` and `encryptedMsg`; finally, the third one prevents the use of `charMsg`, while leaving `plainMsg` always enabled (there is in fact no context associated with it). Leaving one or more policies always enabled is a good way to reduce the risk of ending a conflict resolution process with a failure because no agreed policy could be found. However, this increases the risk of conflicts and, consequently, the time used to resolve them. It is up to the application to decide which strategy suits it best (we will discuss in Chapter 6 some heuristics to write profiles, based on an experimental evaluation of our middleware model).

```
% peer 1                            % peer 3
messagingService                    messagingService
    charMsg                             plainMsg
        1
            bandwidth > 70%
    plainMsg                            compressedMsg
        2                                   1
            bandwidth < 70%                     bandwidth < 40%
    compressedMsg                       encryptedMsg
        3                                   2
            bandwidth < 35%                     battery > 60%
    encryptedMsg
        4
            battery > 50%

% peer 2
messagingService
    plainMsg
        1
            battery < 50%
    compressedMsg
        2
            bandwidth < 40%
```

Figure 4.15: Example of Application Profiles.

```
% peer 1             % peer 2           % peer 3
battery        4     battery    7       privacy  10
bandwidth      3     bandwidth  9
availability  10
```

Figure 4.16: Example of Utility Function Specifications. $peer_1$ aims at maximising availability without wasting resources; $peer_2$ aims at minimising resource consumption, and $peer_3$ aims at maximising privacy.

Assuming that the utility functions are the ones shown in Figure 4.16, and that the current execution context enables the following sets of policies:

$$\begin{aligned} P_1 &= \{\texttt{plainMsg}, \texttt{compressedMsg}, \texttt{encryptedMsg}\} \\ P_2 &= \{\texttt{plainMsg}, \texttt{compressedMsg}\} \\ P_3 &= \{\texttt{plainMsg}, \texttt{compressedMsg}, \texttt{encryptedMsg}\} \end{aligned}$$

for peers $peer_1$, $peer_2$ and $peer_3$ respectively, then the conflict resolution process proceeds as described below.

Step 3 - Computation of the Solution Set

First, the solution set P^*, that is, the set of commonly agreed policies is computed:

$$\begin{aligned}\mathcal{I}[\![\texttt{messagingService}]\!]_{\{peer_1,peer_2,peer_3\}} &= P_1 \cap P_2 \cap P_3 \\ &= \{\texttt{plainMsg}, \texttt{compressedMsg}\}\end{aligned}$$

Once again, the cardinality of the solution set is greater than 1, that is, our conflict resolution mechanism is needed to transparently solve the conflict.

Step 4 - Computation of Bids

Each peer is now required to compute a bid on both enabled policies. We assume, as before, that LMAX = 10, WMAX = 10, and RQMAX = 6, and that each peer has a quota $q_{peer_i} = 1$ (i.e., the bid is not constrained by the current quota, as each bid $b_{i,j} \in [0,1]$). For example:

$$\begin{aligned}u_{peer_1}(\texttt{plainMsg}) &= \frac{(10-3)*4+(10-6)*3+7*10}{10*10*6} \\ &= 110/600 = 0.183\end{aligned}$$

The whole set of bids is then the following:

$$\begin{aligned}&\mathcal{B}[\![\{\texttt{plainMsg}, \texttt{compressedMsg}\}]\!]_{\{peer_1,peer_2,peer_3\}} = \\ &\qquad \{(\texttt{plainMsg}, peer_1, 0.183), (\texttt{compressedMsg}, peer_1, 0.146), \\ &\qquad (\texttt{plainMsg}, peer_2, 0.142), (\texttt{compressedMsg}, peer_2, 0.15), \\ &\qquad (\texttt{plainMsg}, peer_3, 0), (\texttt{compressedMsg}, peer_3, 0)\ \}\end{aligned}$$

Step 5 - Election of the Winner

Bids received for each policy in the solution set are added, and the policy that maximises the sum (i.e., social welfare) is selected:

Policy	$peer_1$		$peer_2$		$peer_3$		Sum
plainMsg	0.183	+	0.142	+	0	=	0.325
compressedMsg	0.146	+	0.15	+	0	=	0.296

Therefore:

$$\mathcal{W}[\![\ \mathcal{B}[\![\{\texttt{plainMsg}, \texttt{compressedMsg}\}]\!]_{\{peer_1, peer_2, peer_3\}}]\!] = \texttt{plainMsg}$$

Note that, while `plainMsg` was the preferred policy by $peer_1$, $peer_2$ placed a higher bid on `compressedMsg` instead ($peer_3$ placed an equal bid on both). Our auctioning mechanism requires then that a payment that is proportional to the added benefit gained by each peer in applying `plainMsg` is made. Each peer quota is therefore adjusted in the following way:

$$\begin{aligned}
q_1 &= q_1 - \frac{0.183 - 0.142}{1} - \frac{0.142 - 0}{2} - \frac{0}{3} \\
q_2 &= q_2 - \frac{0.142 - 0}{2} - \frac{0}{3} \\
q_3 &= q_3 - \frac{0}{3}
\end{aligned}$$

At the same time, the middleware accounts associated with these peers are increased by the same amounts:

$$\begin{aligned}
\overline{q}_1 &= \overline{q}_1 + \frac{0.183 - 0.142}{1} + \frac{0.142 - 0}{2} + \frac{0}{3} \\
\overline{q}_2 &= \overline{q}_2 + \frac{0.142 - 0}{2} + \frac{0}{3} \\
\overline{q}_3 &= \overline{q}_3 + \frac{0}{3}
\end{aligned}$$

4.4.3 Inter-profile Conflict: Access Proceedings

As a last example, we consider the 'access proceedings' service of our conference application. Let us assume that the application developer provides three different policies among which the application can choose: a `networkReference` policy, which is used to access the proceedings without caching information locally, a `cacheAbstract` policy, which is used to replicate the abstracts of the papers, so that they can be accessed when the host is off-line, and a `compressCache` policy used to compress abstracts before caching them on the mobile device. Each of these policies requires a different amount of resources (in particular, memory, battery and bandwidth), and achieves a different quality of service (in terms of availability of data). The corresponding policy specifications are shown in Figure 4.17.

When Alice ($peer_1$) arrives at the conference site, the available wireless infrastructure

```
networkReference: {(battery,3),(bandwidth,8),(availability,2)}
cacheAbstract: {(memory,8),(battery,4),(bandwidth,8),(availability,10)}
compressCache: {(memory,5),(battery,7),(bandwidth,5),(availability,10)}
```

Figure 4.17: Example of Policy Specifications.

enables her to access the electronic proceedings, by requesting that service from a well-known, permanently connected server ($peer_2$). This is a typical client-server interaction; while $peer_1$ is willing to customise the way the `accessProceedings` service is delivered, we may assume that $peer_2$ is always willing to meet its clients' requests. Therefore, the client peer enables the three policies in different contexts, while the server peer enables all of them at all times; the corresponding application profiles are represented in Figure 4.18.

```
% peer 1 (client)                  % peer 2 (server)
accessProceedings                  accessProceedings
    networkReference                   networkReference
        1
            battery > 20%
    cacheAbstract                      cacheAbstract
        2
            battery < 20%
    compressCache                      compressCache
        3
            memory < 25%
```

Figure 4.18: Example of Application Profiles.

```
% peer 1 (client)            % peer 2 (server)
battery        5
availability  10
```

Figure 4.19: Example of Utility Function Specifications. $peer_1$ aims at maximising availability without wasting too much battery; $peer_2$ expresses no preferences.

Also, we may assume that, in the case that a conflict is detected, the server peer expresses no preferences as to which policy is actually chosen and applied; its utility function specification is therefore empty (Figure 4.19). Note that it entirely depends on the client whether a conflict is detected, and, if so, how it is solved.

Assuming that the current execution context enables the following sets of policies:

$$\begin{aligned} P_1 &= \{\texttt{cacheAbstract}, \texttt{compressCache}\} \\ P_2 &= \{\texttt{networkReference}, \texttt{cacheAbstract}, \texttt{compressCache}\} \end{aligned}$$

for peers $peer_1$ and $peer_2$ respectively, then the conflict resolution process as described below.

Step 3 - Computation of the Solution Set

First, the solution set P^*, that is, the set of commonly agreed policies is computed:

$$\begin{aligned}\mathcal{I}[\![\texttt{accessProceedings}]\!]_{\{peer_1,peer_2,\}} &= P_1 \cap P_2 \\ &= \{\texttt{cacheAbstract}, \texttt{compressCache}\}\end{aligned}$$

As expected, it coincides with P_1 (i.e., the client set of enabled policies), as the server gives carte blanche on how the service is delivered.

Step 4 - Computation of Bids

Both the client and the server peer are now required to compute a bid on both enabled policies. We assume, as before, that LMAX = 10, WMAX = 10, and RQMAX = 6, and that each peer has a quota $q_{peer_i} = 1$ (i.e., the bid is not constrained by the current quota, as each bid $b_{i,j} \in [0, 1]$). The whole set of bids is then the following:

$$\begin{aligned}&\mathcal{B}[\![\{\texttt{cacheAbstract}, \texttt{compressCache}\}]\!]_{\{peer_1,peer_2\}} = \\ &\qquad \{(\texttt{cacheAbstract}, peer_1, 0.216), (\texttt{compressCache}, peer_1, 0.196), \\ &\qquad (\texttt{cacheAbstract}, peer_2, 0), (\texttt{compressCache}, peer_2, 0),\end{aligned}$$

As discussed before, the server expresses no preferences and bids 0 on both policies.

Step 5 - Election of the Winner

Bids received for each policy in the solution set are added, and the policy that maximises the sum (i.e., social welfare) is selected:

Policy	$peer_1$		$peer_2$		Sum
`cacheAbstract`	0.216	+	0	=	0.216
`compressCache`	0.196	+	0	=	0.196

Social welfare coincides in this case with $peer_1$ preferences, therefore:

$$\mathcal{W}[\![\ \mathcal{B}[\![\{\texttt{cacheAbstract, compressCache}\}]\!]_{\{peer_1,peer_2\}}]\!] = \texttt{cacheAbstract}$$

Note that, in this case, `cacheAbstract` is the ‘preferred’ policy both by $peer_1$ (who placed its highest bid on it), and by $peer_2$ (who did not bid higher on any other policy). Our auctioning mechanism does not require any payment, as $peer_1$ did not 'impose' any policy on $peer_2$. The conflict resolution process simply ends with the `cacheAbstract` policy being executed.

4.5 Related Work

The problem of resolving conflicts is a general one and different research communities have investigated it over the years.

4.5.1 Resource Allocation

The operating systems community has studied the issue of conflicts in distributed environments, where conflicts manifest themselves as processes competing for shared resources. Microeconomic techniques, and auctions in particular, have been explored. [Malone et al., 1988] describe a market-like bidding mechanism which assigns tasks to processors that have given the lowest estimated completion time; similar techniques have been used to manage network traffic by [Sairamesh et al., 1995], and allocation of storage space by [Ferguson et al., 1993].

We have demonstrated that microeconomic techniques can also be successfully used to resolve QoS conflicts that arise in the mobile setting; however, the nature of conflicts is fundamentally different, thus requiring different conflict resolution algorithms. In particular, resource conflicts happening at the operating system level represent competitive situations where only one competitor obtains the resources, leaving all the others without them. In our case, collaboration characterises the nature of the auction better: peers participating in the delivery of a service will *all* get the goods (the delivery of the service), but with varying degrees of satisfaction. Traditional auctions cannot be applied in this setting, and a novel mechanism was required to deal with these conflicts.

4.5.2 Requirements Monitoring

The software engineering community has investigated the issue of conflicts too. Software development environments [Engels et al., 1992, Emmerich, 1996] have devised mechanisms for specifying consistency constraints between artifacts. They are able to detect static violations of these constraints and resolve them automatically (e.g., by propagating changes to dependent documents). Inconsistencies are often found in requirements documents, indicating conflicts between the different stakeholders involved. Requirements management methods and tools therefore include inconsistency detection and resolution mechanisms. The KAOS method [Dardenne et al., 1993] uses a goal-oriented approach to decompose requirements and formalises them using a temporal logic. Conflicts are detected by reasoning about the temporal logic formulae and conflict resolution strategies [van Lamsweerde et al., 1998] can be applied so that requirement conflicts are not propagated to system design. Other requirements engineering approaches [Hunter and Nuseibeh, 1998] leave inconsistencies in specifications and use an appropriate logic to continue reasoning, even in the presence of an inconsistency.

These approaches, however, are of limited use in a mobile setting where the nature of conflicts is such that they cannot be detected statically at the time an application is designed but, instead, they can only be detected and resolved at run-time. Also, they must be resolved, otherwise applications cannot execute.

Our work is more closely related to approaches that monitor requirements and assumptions during the execution of systems. Fickas and Feather's approach towards requirements monitoring [Fickas and Feather, 1995] uses a Formal Language for Expressing Assumptions (FLEA). FLEA is supported by a CLISP-based run-time environment, which can alert the user of requirement violations. For mobile systems, however, this is insufficient and a more proactive approach to resolving conflicts is required. [Robinson and Pawlowski, 1999] have developed a so-called "requirements dialog meta-model", which supports not only the definition and monitoring of goals, but also the re-establishment of a dialog goal in case of a goal failure. Goal monitoring is performed actively, so that violations are detected immediately.

Monitoring application profiles actively, as done for goals by the previous approaches, is however not feasible; this would require checking that the profile is conflict-free each time it is modified. The more complex the profile (in terms of the number of policies associated with a service, of contexts associated with policies, and of resources associated with contexts), the heavier the check, with a consumption of resources (especially memory and battery) that hand-held devices cannot bear. A more 'passive' solution is preferred in our case; conflicts can exist inside a profile and they are treated only when a service that incorporates a conflict is invoked.

4.5.3 Negotiation Mechanisms

In the Distributed Artificial Intelligence (DAI) community, game theory has been extensively applied to treat negotiation issues. Negotiation mechanisms have been used both to assign tasks to agents, to allocate resources, and to decide which problem solving tasks to undertake (e.g., [Zlotkin and Rosenschein, 1996], [Zlotkin and Rosenschein, 1993]). These scenarios typically involve a group of agents operating in a shared environment. Each agent has its own private goal; a negotiation process is put in place that, through a sequence of offers and counter-offers, explores the chances of agents achieving their (possibly conflicting) goals, at the lowest cost. Despite similarities with our scenario, there are a number of assumptions that differentiate our work from previous results obtained in the DAI community. In particular, in DAI the quality of the result is valued much more than the cost of achieving it; as a consequence, negotiation mechanisms are usually iterative processes which carry on until an (optimal) agreement is reached.

In a mobile setting, instead, resource constraints call for simple conflict resolution mechanisms that do not waste (scarce) resources. Moreover, the nature of goals is fundamentally different. In DAI, a goal can be seen as a task composed of atomic operations that the negotiation mechanism is able to assign to different agents; in our setting, goals are rather indivisible units that suggest the quality-of-service levels that applications are wishing to achieve.

4.5.4 QoS Provision

Despite the extensive research that has been carried out within the mobile middleware community, the issue of QoS conflicts has attracted little attention. On the one hand, many systems do not support dynamic adaptation, and thus they avoid the problem of conflicts a priori. On the other, systems which exploit reflection to improve flexibility and allow dynamic reconfigurability [Ledoux, 1999, Blair et al., 1998] generally target a stationary distributed environment, where context changes (and, consequently, adaptation) are much less frequent than in a mobile setting, so that the problem of conflicts is less pressing.

A survey on quality-of-service provision in a mobile computing environment is provided by [Chalmers and Sloman, 1999a]. QoS requirements are defined by all applications, and a negotiation mechanism is put in place to reach an agreement between all parties; as a result of context changes, a dynamic renegotiation of the contract may be necessary.

The approaches we have analysed usually target a specific domain (e.g., multimedia applications over broadband cellular networks), mainly focusing on bandwidth allocation [Campbell, 1997]. Moreover, applications have a rather limited way of influencing the policies that are chosen to meet QoS requirements. Our middleware aims to be general and gives applications the power to influence the way adaptation is achieved. This may lead to disagreements among applications to reach the quality-of-service level they wish.

4.5.5 Data Conflicts

In order to maximise data availability in mobile settings, where sudden disconnections may happen frequently, even for long periods of time, systems such as Coda [Satyanarayanan et al., 1990], its successor Odyssey [Satyanarayanan, 1996], Bayou [Terry et al., 1995] and Xmiddle [Mascolo et al., 2002b] give users access to replicas. They differ in the way they ensure that replicas move towards eventual consistency, that is, in the mechanisms they provide to detect and remove conflicts that naturally arise in mobile systems. Bayou, for example, reconciles application-specific information in an application-independent way, preventing the application from influencing the outcome of the reconciliation process. Xmiddle, on the contrary, exploits semantic knowledge about the information (the elements) encoded in a document, and allows the mobile application engineer to associate reconciliation policies to these elements, so that an application-specific reconciliation strategy is pursued.

Data conflicts are fundamentally different from the QoS conflicts we treat, and therefore these solutions can hardly be applied. In particular, inter-profile conflicts are not intrinsic in any profile, but manifest themselves only in relation to (some) other profiles *and* in particular contexts. Therefore, the aim of our conflict resolution mechanism is to dynamically solve them, not to remove them.

4.5.6 Policy Conflicts

To accommodate the dynamic change of behaviour of large-scale distributed systems, policy-based approaches have been investigated, that separate the management policies from the automated managers, so that the behaviour of the system can be dynamically changed without recoding the managers. While performing these changes, policy conflicts may arise. [Lupu and Sloman, 1999] describe an approach to policy-based management and conflict resolution that has some similarities with our model. There is a parallel between our (unchangeable) services, the (varying) policies with which they are delivered, and the context configurations that enable the policies, and their (fixed) managers, the (varying) policies, and the constraints that limit their applicability.

However, the approach to conflict resolution they undertake can hardly be applied in our setting for the following reasons: first, they perform an off-line conflict detection and resolution, based on static analysis techniques, while we argued in this chapter the necessity of a dynamic solution. Moreover, their conflict resolution mechanism is based on a static precedence relationship established between the policies, so that conflicts are always resolved in the same manner, while customisation, and therefore the need to take user's preferences into account, is a fundamental requirement of our solution.

4.6 Summary

The reflective middleware model we have described in the previous chapter enhances the development of context-aware mobile applications; however, it also opens the door to conflicts. In this chapter we have presented a conflict resolution mechanism that enables the automatic detection and resolution of QoS conflicts in our mobile setting. We assess that this mechanism is particularly well-suited for mobile computing scenarios as it meets our requirements of dynamicity, simplicity and customisation.

Our conflict resolution mechanism is dynamic as it is used ‘on-demand’, whenever an application invokes a service that involves a conflict. When an application profile is modified using the reflective meta-interface our middleware provides, no static analysis is performed to check that the new configuration is conflict-free (for example, using the tool that [Nentwich et al., 2002] describe), as this is not always possible (e.g., inter-profile conflicts), nor desirable, as mobile devices usually lack the resources necessary to run these checks.

The microeconomic technique that forms the basis of our conflict resolution mechanism has been inspired by traditional sealed bid auctions (e.g., first-price and second-price sealed bid auction [Vickrey, 1961]). Unlike ascending bid auctions, such as the standard English auction [Milgrom, 1989], where the auctioneer continuously raises the price of the good until only one bidder is willing to meet the price called, sealed bid auctions consist of a one-step bid that cuts down the computation and communication costs of iterative procedures, when the auction is distributed over space and time, as in our mobile setting. We can analytically assess that this meets also our requirement of simplicity, and we will demonstrate it experimentally in Chapter 6.

In case applications do not agree on a common set of enabled policies (i.e., the solution set is empty), our conflict resolution process fails and user intervention is required. Our auctioning mechanism could be extended so that, upon failures, applications may weaken their requirements, and enable a larger set of policies; a reward, in terms of quota recharging, could be given. However, this (iterative) approach would seriously compromise efficiency, and go against the simplicity requirement we advocated. We will discuss in Chapter 6 heuristics that considerably lower the risks of conflict resolution failures, without compromising performance; in particular, having a ‘fall-back policy’ that is always enabled by all peers, and that receives only very low bids, represents a good strategy.

Finally, our auctioning scheme meets the customisation requirement too, by means of utility functions. As we discussed in Section 4.3.3, while policy specifications are fixed, utility function specifications change over time. More precisely, applications can dynamically alter the resources and QoS parameters encoded in a profile, together with the weights they are associated with, in order to reflect current user needs. Application users therefore have the power to influence the way conflicts are solved, by customising the information encoded in utility functions. We will show in Chapter 6 how user’s preferences and needs can be translated into utility function specifications.

Chapter 5

CARISMA Architecture

In the previous chapters, we have presented a middleware model that, based on the principles of reflection and metadata, facilitates the development of context-aware applications. Applications encode in application profiles, that is, in middleware metadata, how they wish context changes to be handled, and use the reflective mechanism provided to dynamically change these preferences. In doing so, conflicts may arise. Whenever a service that incorporates a conflict (i.e., that may be delivered using different policies in the current context) is requested, a conflict resolution mechanism is executed, that selects and applies the policy that delivers, on average, the best quality-of-service to the applications involved.

In this chapter, we discuss some relevant issues we tackled during the construction of CARISMA, a mobile computing middleware architecture that realises our middleware model. In particular, we describe how the conceptual model presented in Section 3.2 has been mapped into a reflective architecture, and we analyse two different algorithms that implement the distributed auction protocol illustrated in Section 4.3.2.

5.1 Reflective Architecture

The CARISMA architecture is made up of four main components, as shown in Figure 5.1: the *Core* component provides basic functionalities, such as support for asynchronous communication, service discovery, etc.; the *Context Management* component is responsible for interacting with physical sensors and monitoring context changes; *Core Services* take care of answering service requests with application-defined QoS levels; and the *Application Model* defines a standard framework to create and run context-aware applications on top of our middleware model. We now describe each of these components in more detail. In the following figures, we use UML Component Diagrams [Rumbaugh et al., 1998] to illustrate how each of these components has been realised.

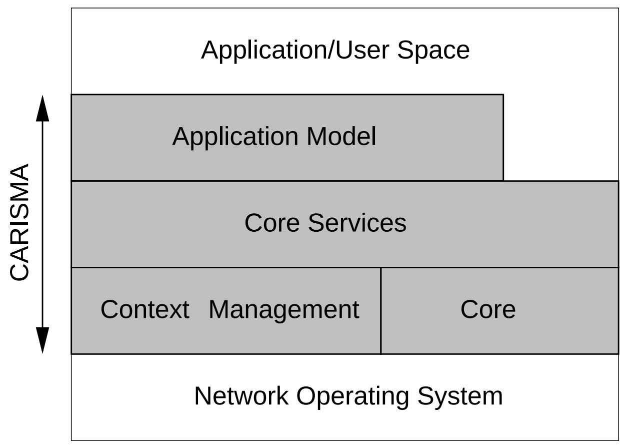


Figure 5.1: Middleware Architecture.

5.1.1 Core

As we pointed out in Chapter 2, middleware can be structured into different layers. While our research interests focus on high-level mechanisms to support application-driven adaptation of application behaviour, our architecture must include a lower-level core component to provide basic functionalities, such as communication, service discovery, etc. As Figure 5.2 illustrates, CARISMA Core comprises three main components: the *Connector* component, the *Publisher* component, and the *Register* component.

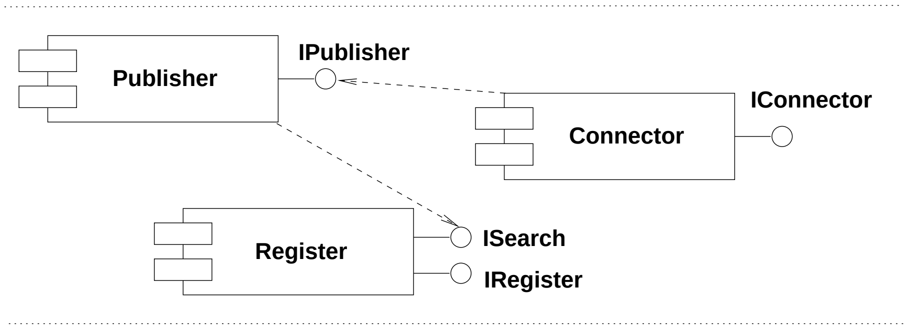


Figure 5.2: Core Architecture.

Connector. The Connector component supports communication among distributed applications. It exports a well-defined interface so that changes at this component level do not have repercussions on other high-level components. The current implementation of CARISMA provides a basic message-passing asynchronous communication model.

Publisher. The Publisher component is responsible for advertising the host presence to other peers, as well as the software components (e.g., applications, services, etc.) available on the device. Also, it keeps track of the presence of other entities and software components within reach.

Register. The Register component provides support for registering new entities on top of CARISMA, where an entity may be a new application, new available policies to deliver a service, new physical sensors CARISMA is able to deal with, and so on. Also, it maintains information about currently registered entities, and provides an interface to support searches for specific entity information.

5.1.2 Context Management

As we have discussed in Chapter 3, in order to build context-aware applications we need to break the principle of transparency, and provide applications with information about their execution context. However, we do not want applications to deal with heterogeneous sensors to gather such information. In our architecture, the Context Management component is responsible for doing so.

For each resource that the middleware monitors, a *wrapper* exists that is able to interact with the physical *sensor*, and to process the information thus obtaining a *value* that the application understands. Different wrappers may exist that interact with the same sensor, but that process information in a different manner; or, vice versa, the same wrapper may interact with various sensors, to synthesise context information in an application-specific manner. Only wrappers of resources that the running applications are interested in are loaded, to avoid wasting computational resources, already scarce on a portable device. Figure 5.3 illustrates a Memory Wrapper, interacting with a Memory Sensor (in this case, a primitive of the network operating system).

Context wrappers export an interface that is used for two main reasons: to register the interest of an application in specific context conditions, and to find out what the current context configuration is.

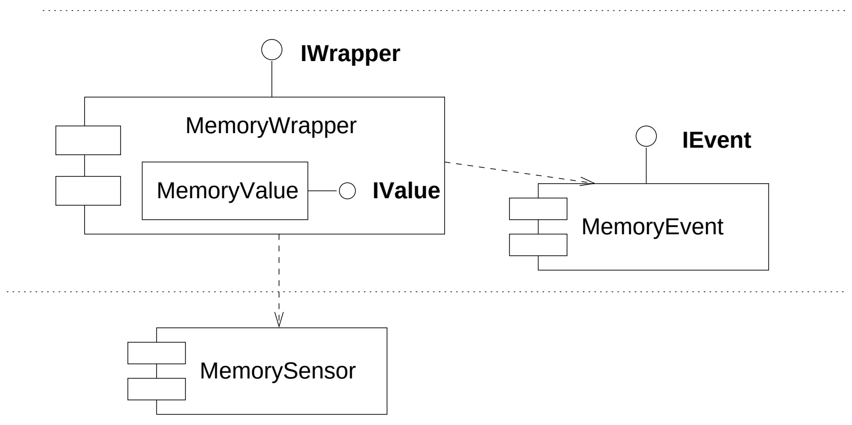


Figure 5.3: Context Management Architecture.

Registering application's interests. As we have seen, application profiles contain two kinds of information: reactive metadata and proactive metadata. Whenever an application is started, or whenever the profile of a running application is changed, the reactive metadata it encodes is processed (we will see in Section 5.1.3 who is in charge of doing this processing) and passed to the various wrappers; each wrapper periodically monitors the status of its sensor(s), and, whenever a context change occurs that is relevant to the application, a corresponding *event* is produced. As shown, wrappers, events and values export a set of well-defined interfaces, so that new resources can be defined and monitored by CARISMA, as long as suitable wrapper/event/value implementations are provided.

Querying current context. Whenever an application service request is issued, the set of policies enabled in the current context have to be determined. In order to do so, wrappers of relevant resources are queried, to obtain updated resource values. The decision of which policies are then enabled is the responsibility of a different component (see Section 5.1.3).

5.1.3 Core Services

On top of the Core and the Context Management components, there are three Core Services: the *ContextController* core service, the *Executor* core service and the *Shell* core service (Figure 5.4).

ContextController. The ContextController core service manages context information. It is responsible for activating/deactivating wrappers; it fetches the status of the current context, by interacting with the various wrappers through their well-defined interfaces; and it fires application-defined policies when particular context configurations are entered, implementing the semantics for reactive metadata we defined in Chapter 3.

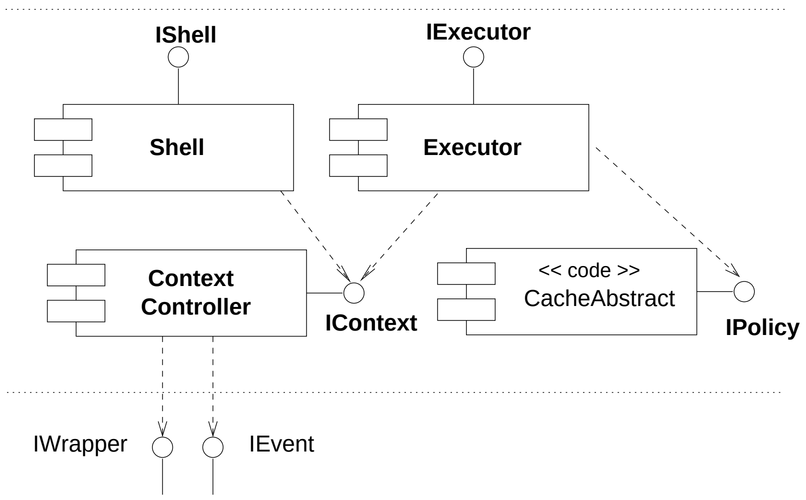


Figure 5.4: Core Services Architecture.

Executor. CARISMA executes application service requests by means of the Executor core service. Each time a service is invoked, the Executor fetches the profile of the requesting application, interacts with the ContextController to get updated information about the current context configuration, finds out which policy is suited in the current context and finally delivers the service with the selected policy, possibly after resolving conflicts.

Shell. The Shell core service provides users with functionalities to start applications on top of CARISMA, and to open/close network connections. Although the current implementation of CARISMA does not support it, the Shell core service may also be used to allow users to install/un-install applications, policies, and so on.

5.1.4 Application Model

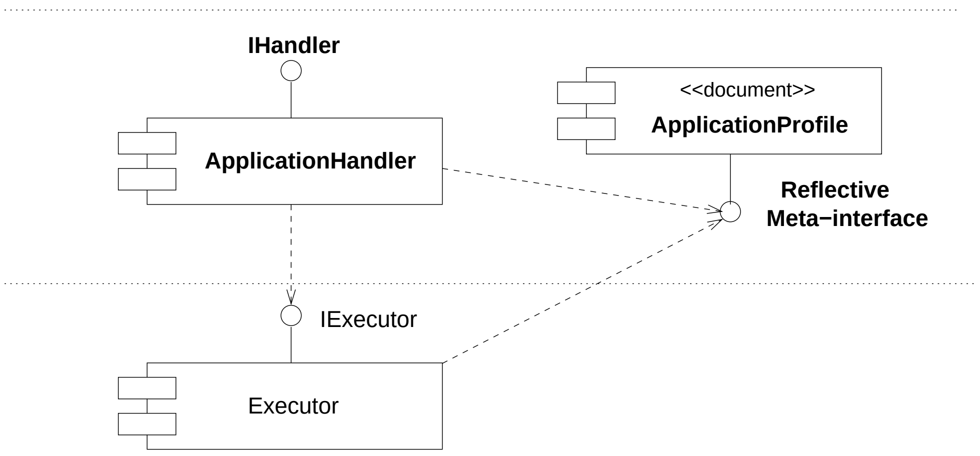


Figure 5.5: Application Model Architecture.

The CARISMA Application Model (Figure 5.5) provides support for building and running context-aware applications on top of our middleware model. Applications access middleware functionality through an *ApplicationHandler*: each time an application is started, an ApplicationHandler is created to allow interactions between the middleware and the application itself. In particular, application handlers allow applications to request services from the underlying middleware, and to access their own application profile through a well-defined *reflective meta-interface.*

5.1.5 Example

Figure 5.6 shows the components that may be active at a particular moment of the execution of the conference application. Though most of the active components are the same for any application, different configurations may occur at the context management level. In particular, the figure shows three wrappers running at the moment and probing their

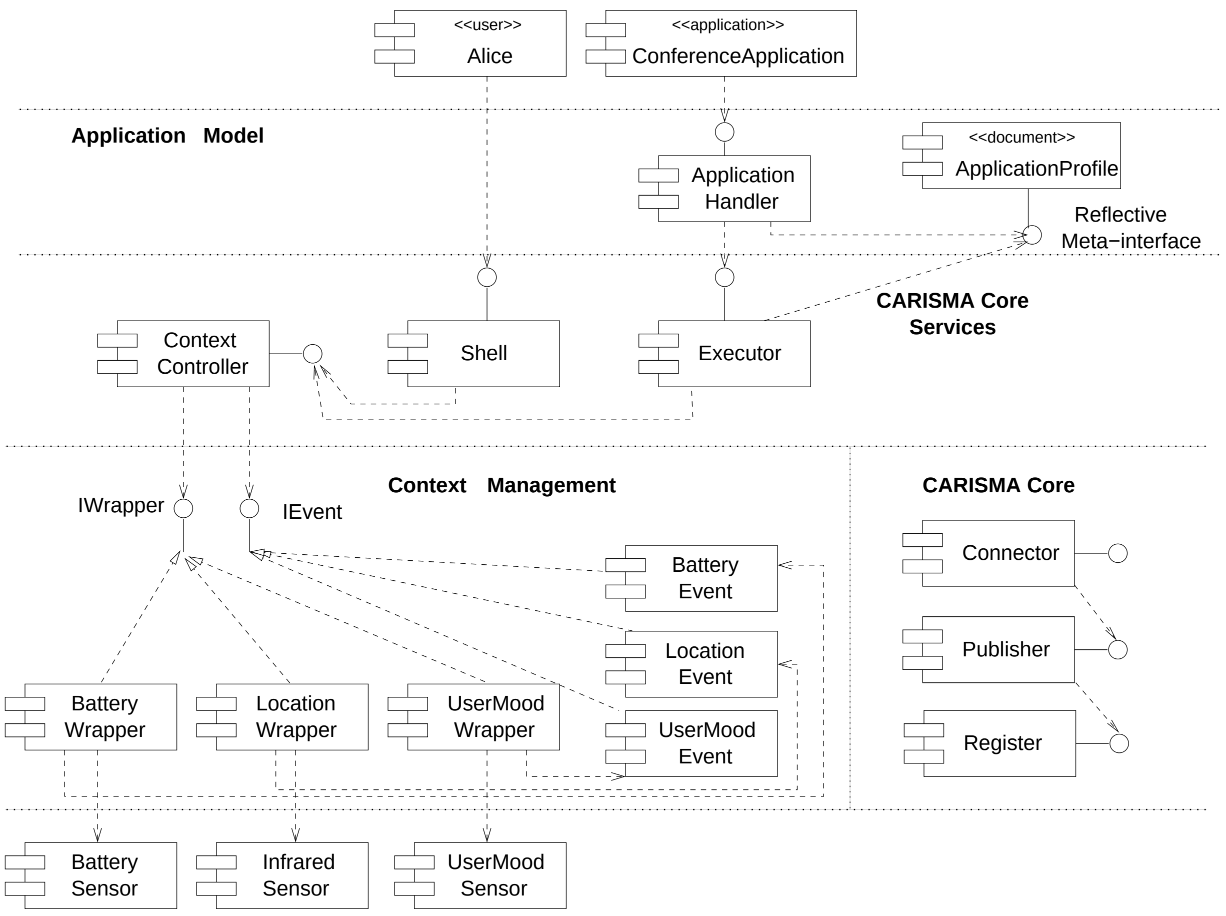


Figure 5.6: Middleware Architecture Example.

associated sensors (i.e., battery, location and user mood); this means that the application has encoded in its profile an interest in the status of these resources. The active wrappers and sensors vary dynamically, as a result of changes occurring in the profile, as well as activation and de-activation of applications.

5.2 Distributed Auction Protocol

In Chapter 4, we argued that the conflict resolution mechanism we have designed is 'simple', in the sense that the computation and communication overhead it imposes on mobile devices is low. Providing a light-weight implementation of this mechanism on a single machine, to solve intra-profile conflicts, is rather straightforward, as no communication is required. However, designing a light-weight distributed algorithm to solve inter-profile conflicts is not trivial, as minimising communication costs and minimising computation costs turn out to be conflicting requirements. In the following sections, we first provide a brief presentation of two different algorithms, the first one that attempts to minimise communication costs, and the second one that aims to minimise computation costs, and then we discuss their similarities and, mainly, their differences. We use UML Sequence Diagrams [Rumbaugh et al., 1998] to illustrate these algorithms.

5.2.1 Minimisation of Communication Costs

Our first attempt to design a distributed algorithm that implements our conflict resolution mechanism aims to minimise communication costs. This means both minimising the amount of data that is sent around, as well as reducing the number of messages exchanged among the involved hosts. Figure 5.7 illustrates the algorithm we designed. We assume here that $peer_1$ requests a service that involves the cooperation of n peers.

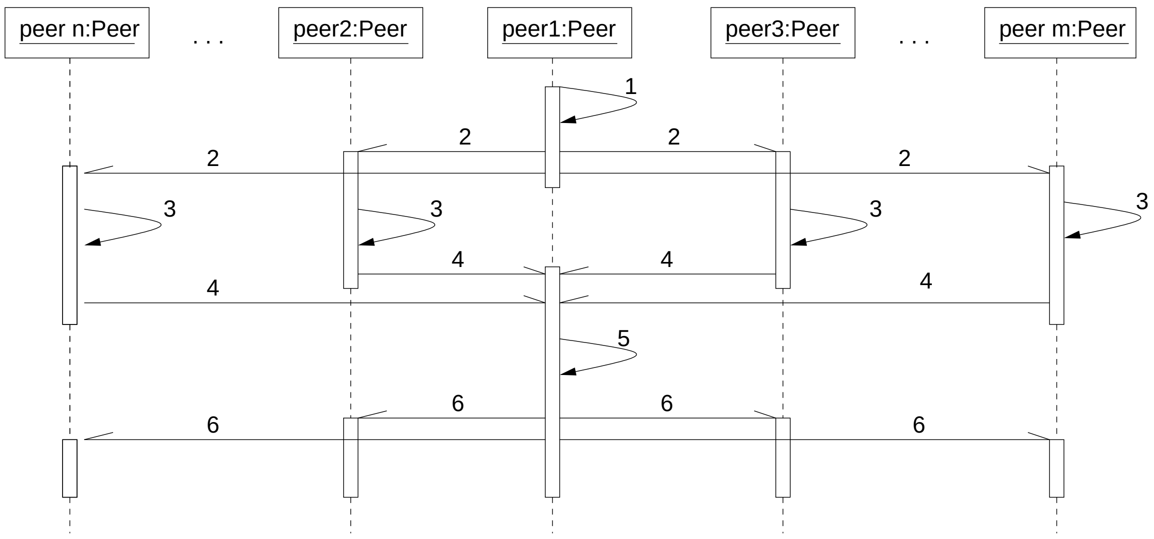


Figure 5.7: Algorithm for Minimisation of Communication Costs.

Step 1. When $peer_1$ requests a service from the underlying middleware, the first step is to compute the set of policies that are currently enabled to deliver the service, that is, P_1. Although no conflict has been detected yet, $peer_1$ pre-computes a bid for each of these policies, that is, it computes B_1.

Step 2. A service request message is sent to the $n-1$ participating peers, together with P_1 and B_1.

Step 3. Each of the $n-1$ peers involved in the service delivery computes the set of locally enabled policies P_i; also, for each policy in $P_i \cap P_1$, that is, for every policy agreed with the requesting peer, a bid is computed. We refer to the set of bids placed by $peer_i$ on the policies agreed by $peer_i$ and $peer_1$ as $B_{i \cap 1}$.

Step 4. Each of the $n - 1$ peers communicates $P_i \cap P_1$ and $B_{i \cap 1}$ back to $peer_1$.

Step 5. $Peer_1$ determines P^* and, in the case that a conflict is detected that can be solved using our auctioning mechanism, it uses the pre-computed bids to determine the winning policy $\tilde{p}$, as well as the sums $f_i,\ i \in [1, n]$ that each peer has to pay.

Step 6. Finally, $peer_1$ sends back to each peer information about what policy has been agreed to deliver the requested service, and how much money has to be paid.

We make the assumption that concurrent requests of the same service, originated by different peers, are identified and resolved prior to the execution of the protocol; in other words, leader election (i.e., $peer_1$) is carried out before this protocol is started. Also, we assume $peer_1$ to be available throughout the protocol; if it fails, the entire service request is aborted. We believe this assumption is reasonable, as $peer_1$ is the only peer to be in reach of all the others, and therefore in the position of being able to conduct the conflict resolution process; because in an ad-hoc network connection is not transitive, we cannot assume that the remaining $n - 1$ peers are connected among themselves, and therefore able to carry out the process without $peer_1$. On the contrary, individual failures of other participating peers taking place during the protocol execution do not compromise its success, as long as there are at least $0 < m \leq n$ peers connected until the end of the process (the minimum number of connected peers m is application dependent).

5.2.2 Minimisation of Computation Costs

The second algorithm that implements our conflict resolution scheme aims at minimising the overall computation costs, that is, minimising the total time that elapses between the service request and the final execution of the service itself. Figure 5.8 illustrates the algorithm; once again, we assume here that $peer_1$ requests a service that involves the cooperation of n peers.

Step 1. First, a service request is sent by $peer_1$ to all the peers participating in the service delivery.

Step 2. All n peers, including the one that started the service request, evaluate the set of policies that are locally enabled, that is, $P_i,\ i \in [1, n]$. As before, for each policy in P_i they pre-compute a bid, thus determining B_i.

Step 3. Each of the $n - 1$ participating peers replies to $peer_1$, communicating both P_i and B_i.

Step 4. The requesting peer uses the information gathered to calculate the solution set P^*, and, in case a conflict is detected, it uses the pre-computed bids to determine the winning policy $\tilde{p}$, as well as the sums $f_i,\ i \in [1, n]$ that each peer has to pay.

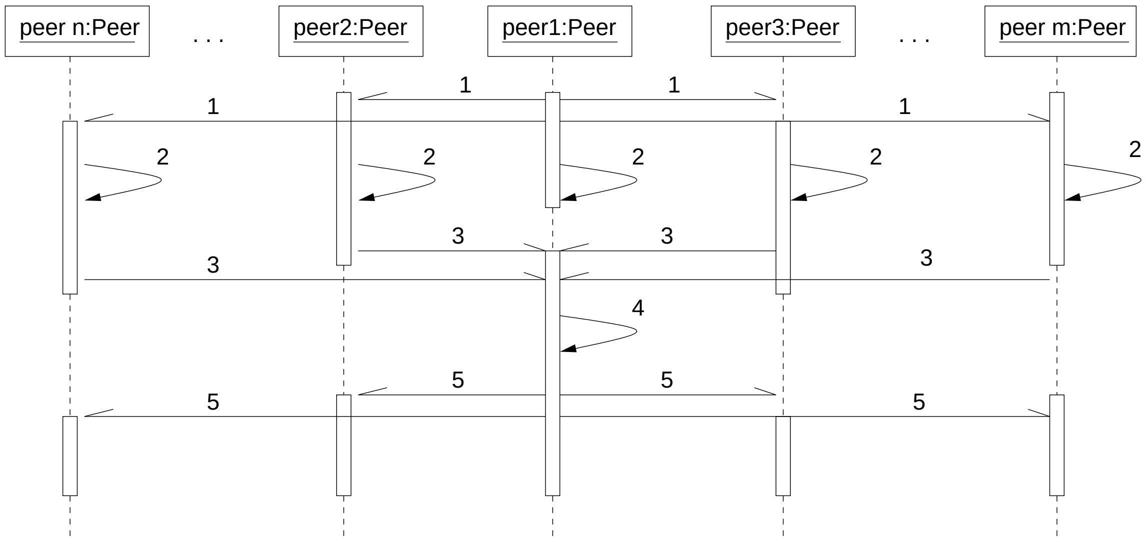


Figure 5.8: Algorithm for Minimisation of Computation Costs.

Step 5. Finally, $peer_1$ sends back to each peer information about what policy has been agreed to deliver the requested service, and how much money has to be paid.

As before, individual failures of participating peers taking place during the protocol execution do not compromise its success, as long as there are at least $0 < m \leq n$ peers connected until the end of the process. Again, m is an application-dependent parameter. If the requesting peer fails, the entire service request needs to be aborted.

5.2.3 Comparison

As shown, both algorithms perform a pre-computation of bids (Step 1 and 3 for the first algorithm, Step 3 for the second one) in order to avoid an extra round of messages once the solution set P^* has been computed (Step 5 in the first algorithm, Step 4 in the second), in case a conflict is detected. In this case, in fact, the requesting peer should communicate to the other participating peers the solution set P^*, and these should reply with their bids, thus requiring $2 * (n - 1)$ more messages to be sent around. As a drawback, bids are computed even if they are not necessary (i.e., when no conflict is detected). However, we consider this overhead negligible compared to the time taken by two additional message rounds, so that, as the evaluation chapter will prove (Chapter 6), the pre-computation is actually worthwhile.

The main difference between the two algorithms can be noticed in the early steps: while the first algorithm computes the locally enabled policies on $peer_1$ first, and then issues a service request passing these values to the other peers, the second algorithm issues a service request first, and then all the peers, including the requesting one, evaluate their locally enabled policies in parallel. By deferring the evaluation of P_1, the second algorithm maximises the parallelisation of computation of the various P_i, which we assume to be the heaviest task of the whole service request process. Computing P_i requires, in fact, the comparison of the current context configuration against all the ones encoded in the profile and associated with the requested service. In the case of complex context configurations and very detailed profiles (i.e., many contexts associated with a policy, and many policies associated with a service), this task may become much more time-consuming (and resource demanding) than an additional exchange of messages.

As far as communication is concerned, in case of interactions between $n > 2$ peers, both

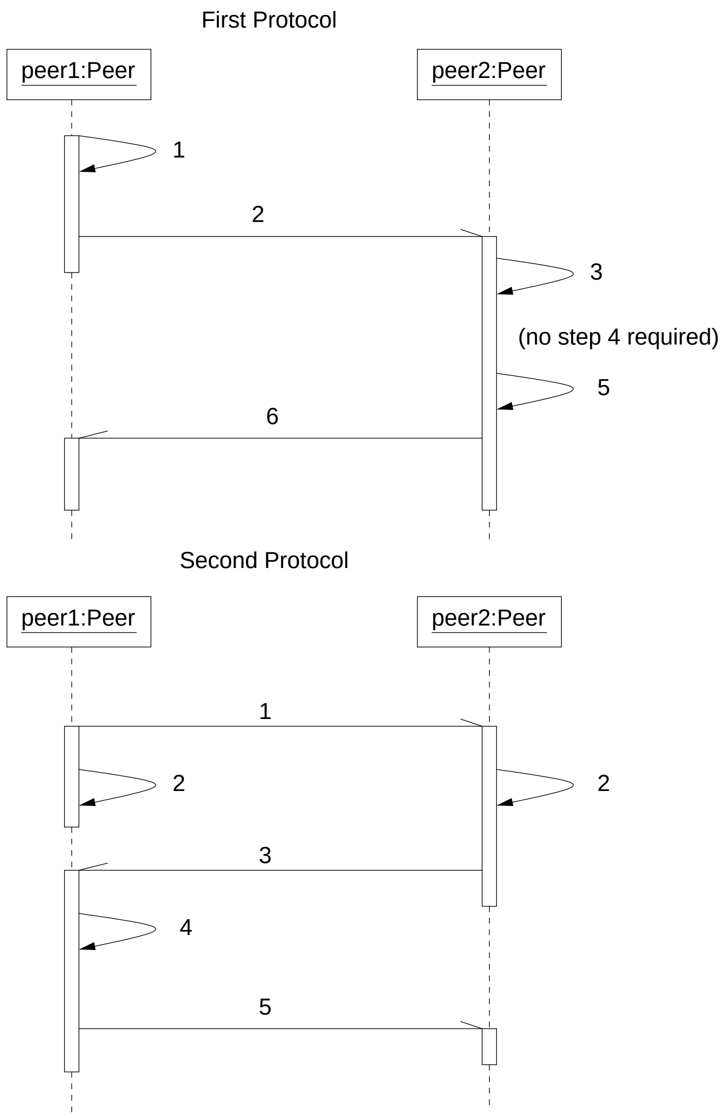


Figure 5.9: Peer-to-peer Interactions

protocols require three rounds of messages to be sent around; the advantage of the first protocol over the second one is that, by pre-computing P_1 in the first step, and by communicating this information to the $n-1$ peers, computation of the solution set is distributed among the various peers, so that the information each peer sends back to $peer_1$ is smaller (i.e., fewer enabled policies and associated bids), and therefore the time required by $peer_1$ to compute the solution set and the winning policy should be smaller.

The difference in the communication costs of the two protocols is however more striking when we consider peer-to-peer interactions, where only $n = 2$ peers are involved. In this (very common) case, the whole service request process is resolved with just two messages sent around if we use the first algorithm, while three messages are needed for the second protocol (see Figure 5.9). In particular, at Step 2 of the first protocol, both P_1 and B_1 are sent to $peer_2$, so that $peer_2$ has all the information needed to compute both the winning policy and the payments, and instead of replying with $P_1 \cap P_2$ and $B_{1\cap 2}$, it replies with $\tilde{p}$ and f_1, thus saving a third message round (step 4 is not required). On the contrary, three messages are needed by the second protocol, as $peer_2$ never gets any information about $peer_1$ enabled policies and bids.

In the next chapter, we will discuss the results of a thorough evaluation that we have conducted, and analyse whether cutting down the number and size of messages sent is more (or less) beneficial than parallelising computation.

5.3 Summary

In this chapter, we have presented how the reflective conceptual model of Chapter 3 can be mapped onto the CARISMA architecture, and how the distributed conflict resolution mechanism illustrated in Chapter 4 can be performed using two different distributed algorithms.

As shown, CARISMA relies on a core to deliver basic functionalities, in particular, communication and service discovery. On top of this core, a small number of CARISMA components are provided, that implement the reflective model and the conflict resolution mechanism. This architecture can be tailored to different application needs by dynamically changing the set of running wrappers that monitor the execution context.

Different algorithms can be designed to implement the conflict resolution mechanism; we have described two alternatives, one that minimises communication costs, and another one that minimises the overall computation costs of a service request. In both cases, our algorithms complete successfully even in the presence of failures, as required in a mobile setting, where disconnections can be frequent.

In the next chapter, we briefly discuss our implementation of CARISMA. Based on this implementation, we illustrate the results of a systematic evaluation of our approach, as far as usability of the system and performance are concerned.

Chapter 6

Implementation and Evaluation

In the previous chapter we have described CARISMA, a mobile middleware architecture that realises both the reflective model discussed in Chapter 3, and the conflict resolution mechanism illustrated in Chapter 4.

In this chapter, we provide insights into the implementation of CARISMA, and illustrate the results of its thorough evaluation. The evaluation results are divided into *qualitative* results and *performance* results. Our aim with regard to qualitative evaluation is to support the thesis that our model provides application engineers with powerful abstractions that are easy to use in the development of context-aware applications. As for performance, we aim to validate the thesis that our reflective middleware model can be realised through a light-weight implementation whose overhead is small enough for mobile devices to bear.

6.1 Implementation

We have implemented CARISMA in Java using jdk 1.4.1, and have encoded application profiles, utility functions and context information using the eXtensible Markup Language (XML [Bray et al., 1998]). The reasons that motivated these choices are manifold: Java is a portable language, it has embedded support for logical mobility and reflection, and more and more mobile devices are being released with J2ME [Sun Microsystem, 2000] technology enabled. Also, we have implemented CARISMA mainly as a proof of concept, to demonstrate the applicability of our middleware model and its suitability to the mobile setting; the many libraries available, as well as run-time support for Java have thus further motivated our choice.

As for the use of XML, we believe this meta-language may enhance context-aware and user-driven interactions between middleware and applications, supporting a representation of information that can be both easily manipulated by machines, and readily understood by humans. Also, XML related technologies have considerably reduced the development time. In particular, we have based our implementation of the reflective meta-interface on DOM [Apparao et al., 1998] and XPath [Clark and DeRose, 1999]; the parsing and retrieval of information encoded in XML is based on Apache Xerces 1.4.3, and Apache Xalan 1.2.2. Recently, new XML parsers (e.g., kXML2 [kObjects, 2002]) have been delivered that specifically target the mobile environment, as they are lighter and faster than traditional parsers; the performance results we discuss in this chapter could therefore be further improved by adopting these new technologies.

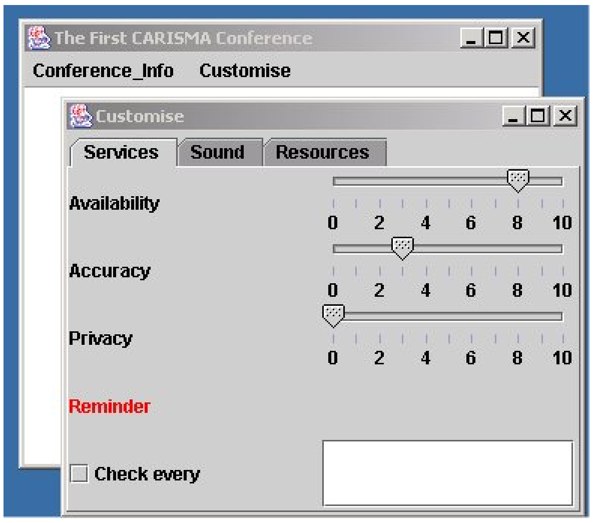
The middleware platform, including the Core, the Context Management, Core Services and the Application Model, currently requires 110Kb of persistent storage, and less than 800Kb of memory (without considering the memory required by the Java Virtual Machine and XML parser). The size of application profiles varies from a few hundreds of bytes, for simple configurations (e.g., 400 bytes for a service with three policies associated with one context, each with one resource), to tens of kilobytes, for very detailed configurations (e.g., 20Kb for a service with five policies associated with five contexts, each with five resources).

6.2 Qualitative Evaluation

One of the aim of the evaluation stage has been to support the thesis that our model provides application engineers with powerful abstractions that ease the development of context-aware applications. Validating this thesis is rather hard, as there are no quantitative parameters we can compute by running a set of experiments. Instead, we assigned an MSc student the task of implementing the Conference Application on top of CARISMA, to estimate the usability and effectiveness of our middleware model, in developing and running context-aware applications. In this section, we report on her experience.

One of the advantages of our middleware model is that application engineers do not have to deal explicitly with physical sensors in order to get context information; adaptation to context changes is carried out automatically by the middleware, using the information encoded in the application profile. As we have argued in Chapter 3, these profiles are dynamic, as they need to reflect user's needs as faithfully as possible. The first challenge the student (and application developers in general) has been exposed to, was therefore to provide a mechanism to gather this information dynamically from the user, and to translate it into XML meta-encoding, using the reflective meta-interface that our middleware provides. The student's experience in developing the conference application was that this task was rather straightforward, as most of the information she needed had already been obtained during the requirements elicitation process.

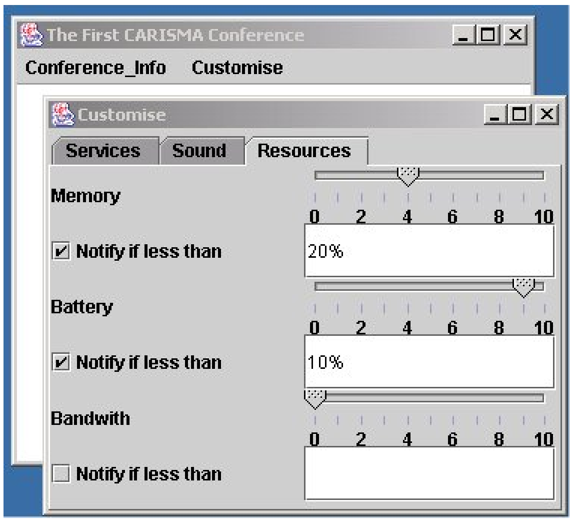
For the Conference Application, the student decided, during the requirements elicitation stage, that the non-functional requirements of interest to the user were availability of information, accuracy of information and privacy. In order to allow users to specify, at any point in time, how much they care about these parameters, she developed the "customisation window" shown in Figure 6.1.



Availability: 8 (0–10)
Accuracy: 3 (0–10)
Privacy: 0 (0–10)
☐ Check every

Figure 6.1: Conference Application Customisation - Non-functional Requirements.

Similarly, through the customisation window, the user may assign different levels of importance to available resources, such as memory, battery, and bandwidth. (Figure 6.2). Also, they may require notification when the availability of local resources fall below specific values.



Memory: 4 (0–10)
☑ Notify if less than 20%
Battery: 9 (0–10)
☑ Notify if less than 10%
Bandwith: 0 (0–10)
☐ Notify if less than

Figure 6.2: Conference Application Customisation - Resources.

Based on these preferences, the student implemented a synthesising algorithm to write application profiles. For example, with reference to the preferences expressed by the user as shown in Figure 6.2, the algorithm developed by the student exploited the meta-interface that CARISMA provides to derive the reactive metadata encoded in Figure 6.3

```
<REACTIVE frequency="5000">
      <POLICY name="lowMemoryAlert">
            <CONTEXT id="1">
                  <RESOURCE name="memory">
                        <OPERATOR name="lessThan"/>
                        <STATUS value="20%"/>
                  </RESOURCE>
            </CONTEXT>
      </POLICY>

      <POLICY name="lowBatteryAlert">
            <CONTEXT id="2">
                  <RESOURCE name="battery">
                        <OPERATOR name="lessThan"/>
                        <STATUS value="10%"/>
                  </RESOURCE>
            </CONTEXT>
      </POLICY>
</REACTIVE>
```

Figure 6.3: XML Profile - Reactive Encoding.

As shown, the user requires notification when memory and battery availability fall below 20% and 10% respectively, while they are not interested in bandwidth values; `frequency` represents the time interval in milliseconds between two consecutive context queries made by the ContextController Core Service, in order to get updated context information. In other words, it represents the speed of adaptation to context changes.

Figure 6.4 represents one of the possible XML proactive encodings for the 'accessProceedings' service. Based on the information entered by the user through the customisation windows shown in Figure 6.1 and 6.2, the student has derived an encoding that enables the local replication of talk abstracts when memory availability is relatively high, replication of talk titles only when low on memory, and a network reference if running out of memory, as a result of a strong user interest in data availability and in preserving memory. Note that `frequency` is not part of the proactive encoding, as context is queried only 'on-demand', when the service is actually requested. Figure 6.5 shows the result of requesting the `accessProceedings` service when available memory is between 10% and 20% (left-hand side), and when it is above 20% (right-hand side).

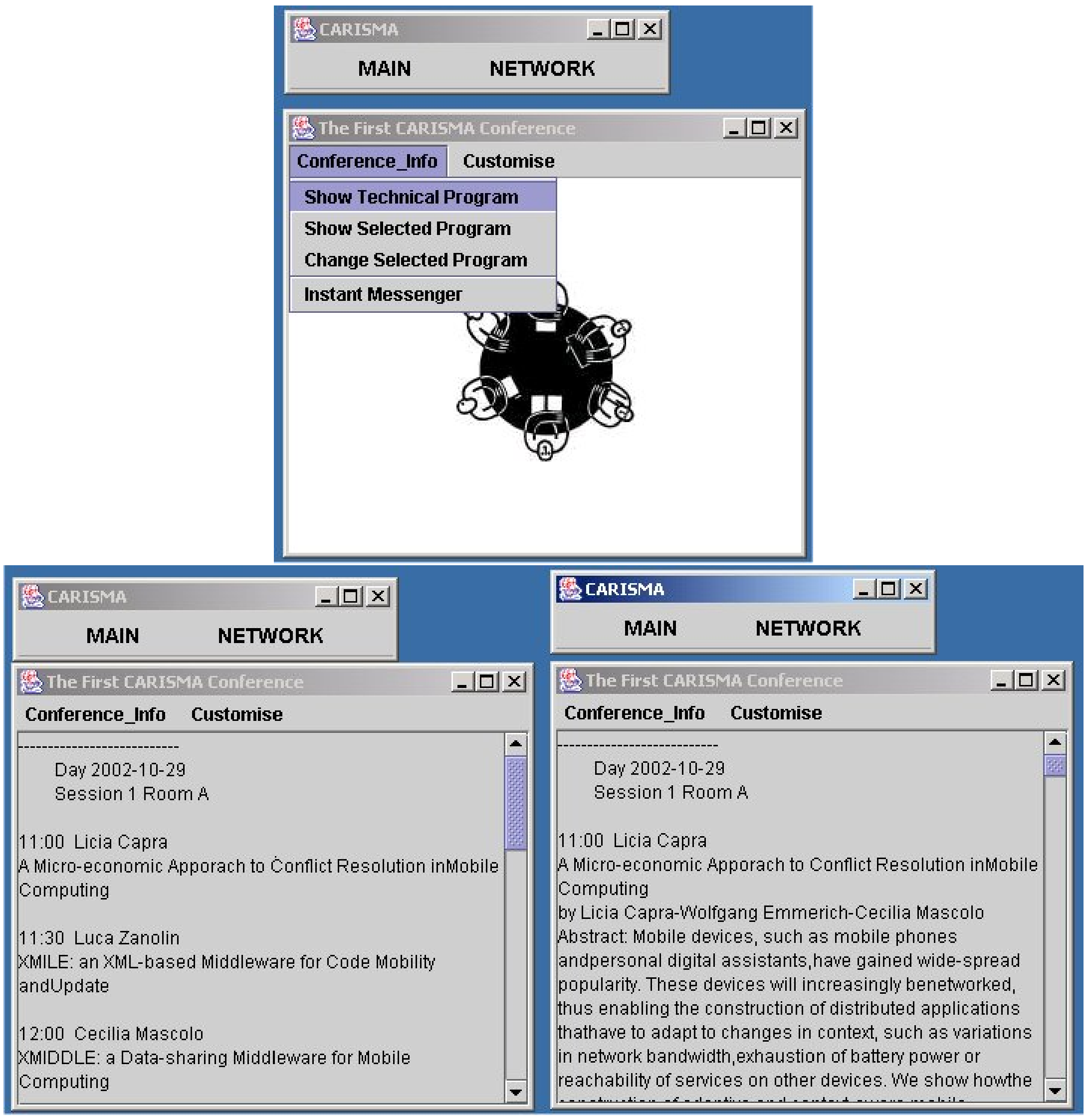
As another example, Figure 6.6 illustrates the XML encoding for the `messagingService` of the Conference Application. As the user is not interested in privacy issues, the `encryptedMsg` policy is never enabled; `charMsg` and `compressedMsg` are enabled in different contexts that depend on bandwidth and battery availability. `plainMsg` is always

```
<PROACTIVE>
      <SERVICE name="accessProceedings">
            <POLICY name="cacheAbstract">
                  <CONTEXT id="4">
                        <RESOURCE name="memory">
                              <OPERATOR name="greaterThan"/>
                              <STATUS value="20%"/>
                        </RESOURCE>
                  </CONTEXT>
            </POLICY>

            <POLICY name="cacheTitle">
                  <CONTEXT id="5">
                        <RESOURCE name="memory">
                              <OPERATOR name="inBetween"/>
                              <STATUS value="10%"/>
                              <STATUS value="20%"/>
                        </RESOURCE>
                  </CONTEXT>
            </POLICY>

            <POLICY name="networkReference">
                  <CONTEXT id="6">
                        <RESOURCE name="memory">
                              <OPERATOR name="lessThan"/>
                              <STATUS value="10%"/>
                        </RESOURCE>
                  </CONTEXT>
            </POLICY>
      </SERVICE>

      . . .

</PROACTIVE>
```

Figure 6.4: XML Profile - Proactive Encoding (Access Proceedings).

enabled; always enabling a policy is a way to reduce the risk of ending a conflict resolution process without an agreement. We will discuss in Section 6.3.6 heuristics on how to write 'good' profiles, that is, profiles with high chances of successfully completing a conflict resolution process, and that do not require heavy processing to determine which policy to apply in any current context.

Figure 6.7 shows the result of exchanging messages between peers; unlike the 'access proceedings' service, in this case it is transparent to the users which policy is actually applied (a delay may be noticed if messages are sent encrypted or compressed, due to the extra computation time required).



```
--------------------------
     Day 2002-10-29
     Session 1 Room A

11:00  Licia Capra
A Micro-economic Apporach to Conflict Resolution inMobile
Computing

11:30  Luca Zanolin
XMILE: an XML-based Middleware for Code Mobility
andUpdate

12:00  Cecilia Mascolo
XMIDDLE: a Data-sharing Middleware for Mobile
Computing
```

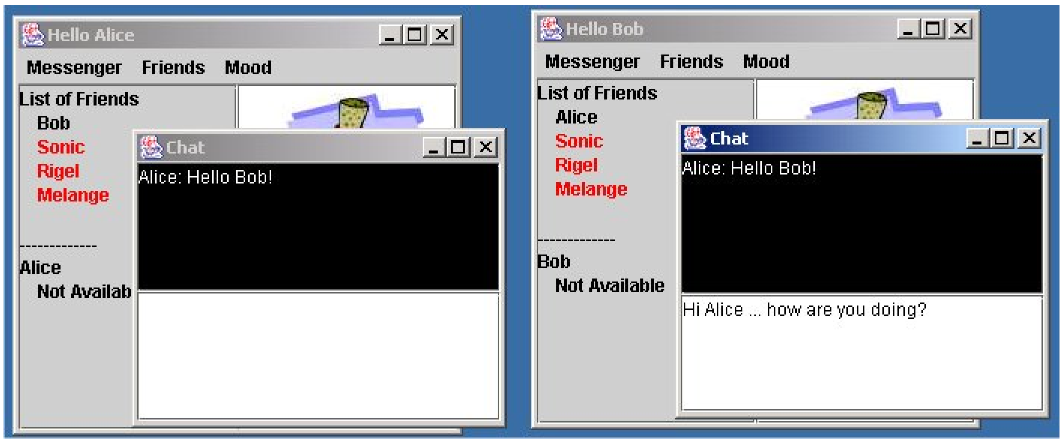
```
--------------------------
     Day 2002-10-29
     Session 1 Room A

11:00  Licia Capra
A Micro-economic Apporach to Conflict Resolution inMobile
Computing
by Licia Capra-Wolfgang Emmerich-Cecilia Mascolo
Abstract: Mobile devices, such as mobile phones
andpersonal digital assistants,have gained wide-spread
popularity. These devices will increasingly benetworked,
thus enabling the construction of distributed applications
thathave to adapt to changes in context, such as variations
in network bandwidth,exhaustion of battery power or
reachability of services on other devices. We show howthe
```

Figure 6.5: Conference Application Screenshot - Access Proceedings.

In the case that a service that involves a conflict is detected, the conflict resolution mechanism described in Chapter 4 is used to find out which policy to apply in the current context. This mechanism is based on utility functions that represent the user's needs as faithfully as possible. As for application profiles, these preferences may change over time; application developers must therefore provide a mechanism to gather this information dynamically from the user, and translate it into an XML meta-encoding. The student used the customisation window discussed before to achieve this task. Figure 6.8 illustrates an XML-encoded utility function that represents the information shown in Figures 6.1 and 6.2. As shown, `privacy` is not listed as it is of no importance to the user (i.e., it would have a weight equal to zero associated).

```
<PROACTIVE>
      <SERVICE name="messagingService">
            <POLICY name="charMsg">
                  <CONTEXT id="7">
                        <RESOURCE name="bandwidth">
                              <OPERATOR name="greaterThan"/>
                              <STATUS value="1"/>
                        </RESOURCE>
                  </CONTEXT>
            </POLICY>

            <POLICY name="plainMsg"/>

            <POLICY name="compressedMsg">
                  <CONTEXT id="10">
                        <RESOURCE name="battery">
                              <OPERATOR name="greaterThan"/>
                              <STATUS value="30%"/>
                        </RESOURCE>
                        <RESOURCE name="bandwidth">
                              <OPERATOR name="lessThan"/>
                              <STATUS value="1"/>
                        </RESOURCE>
                  </CONTEXT>
            </POLICY>
      </SERVICE>

      . . .

</PROACTIVE>
```

Figure 6.6: XML Profile - Proactive Encoding (Exchange of Messages).



Alice: Hello Bob!

Alice: Hello Bob!

Hi Alice ... how are you doing?

Figure 6.7: Conference Application Screenshot - Exchange of Messages.

```
<UTILITY_FUNCTION name="ConferenceApplication">
      <ADD>
            <VALUE_OF name="memory"/>
            <MULTIPLY_BY weight="4"/>
      </ADD>
      <ADD>
            <VALUE_OF name="battery"/>
            <MULTIPLY_BY weight="9"/>
      </ADD>
      <ADD>
            <VALUE_OF name="bandwidth"/>
            <MULTIPLY_BY weight="0"/>
      </ADD>
      <ADD>
            <VALUE_OF name="availability"/>
            <MULTIPLY_BY weight="8"/>
      </ADD>
      <ADD>
            <VALUE_OF name="accuracy"/>
            <MULTIPLY_BY weight="3"/>
      </ADD>
</UTILITY_FUNCTION>
```

Figure 6.8: XML Utility Function.

The XML Schema that define the grammars of the languages used to encode application profiles and utility functions are provided in Appendix B. In our implementation, however, we do not make use of validation, as this is a rather resource-demanding task. The integrity of profiles and utility functions is guaranteed by the interface our middleware provides to application engineers to change this information (i.e., applications do not have direct access to the XML encoding).

We can summarise the lessons we learnt from the student's report as follows. Using the abstractions and mechanisms that CARISMA provides is rather straightforward; the most difficult task that application developers are faced with is to decide which non-functional parameters the end-user should be allowed to tune, and to design a synthesising algorithm that maps user's preferences into application profiles. These tasks can be simplified with the help of application-domain experts. The tedious task of querying heterogeneous sensors to gather and maintain context information was completely transparent to application developers instead.

So far we have mainly focused our discussion on the interaction between middleware and applications, leaving the end-users of the system behind the scene. As Figure 6.9 illustrates, the middleware provides applications with a reflective API (i.e., meta-interface) that they can exploit to inspect and alter application profiles. The target users of our middleware model are therefore application developers. In customising an application's

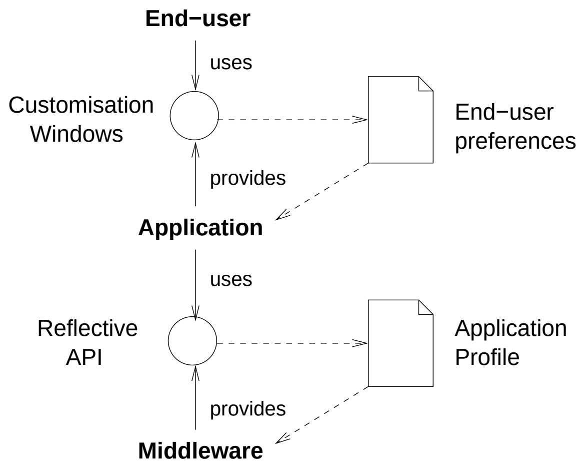


Figure 6.9: Roles and Responsibilities in the Reflective Process.

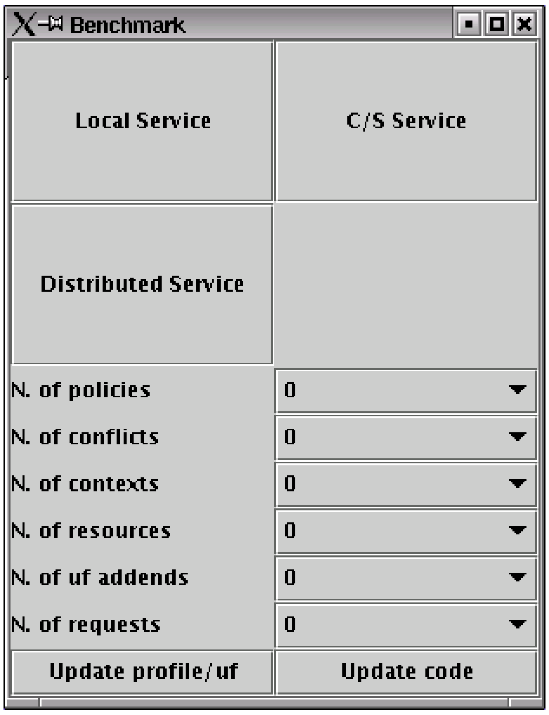
behaviour, however, end-user preferences must be taken into consideration. We have shown how applications built on top of CARISMA can capture end-users preferences by means of customisation windows. An important question is how much effort is required by the end-user to teach the system to behave according to his/her own expectations.

The answer to this question does not strictly depend on the middleware model we have developed, but on the user interface provided by application developers to gather end-user preferences, and on the synthesising algorithm used to translate these preferences into application profiles (i.e., into application behaviours). Also, we believe this issue is not intrinsic to our model, but applies, in general, to scenarios where adaptation to changing contexts and user requirements is needed. We do not have a definite and general answer, as we believe further research, that involves middleware, as well as HCI and requirement elicitation experts, is needed. In developing and using the Conference Application, we have learnt that the amount of human effort is in direct relation with the level of adaptation to context required. If many resources are being considered, and many non-functional requirements are taken into account, then it becomes difficult for the application developer to provide a mapping from user requirements to system behaviours (i.e., application profiles) that fulfils user expectations, and therefore the right balance has to be found.

6.3 Performance Evaluation

The aim of the performance evaluation of CARISMA is to validate the thesis that our reflective middleware requires only a very small overhead in terms of the elapsed time to respond to a service request, compared to a non-reflective approach where service delivery is not tailored to user's preferences and context conditions. In order to validate this hypothesis, we have implemented a synthetic benchmark (as defined by [Weicker, 1984]), and used it to run a large number of experiments. We report the main results in this section.

6.3.1 Experiment Design



N. of policies	0
N. of conflicts	0
N. of contexts	0
N. of resources	0
N. of uf addends	0
N. of requests	0

Figure 6.10: Benchmark User Interface.

In order to evaluate the performance of CARISMA, we have implemented a benchmark; the user interface we have developed to control benchmark executions is displayed in Figure 6.10. As the picture shows, the benchmark user interface allows us to customise both application profiles and utility functions; in particular, we can decide how many policies to associate with each service, how many contexts to associate with each policy, and how many resources with each context (each in a range $[0, 20]$), thus varying the extent to which reflection and context-awareness are used in the profiles. Also, the benchmark user interface allows us to tune the number of conflicting policies for each service, and the number of utility function parameters (again, in a range $[0, 20]$), thus controlling parameters that have an impact on the conflict resolution process. Once the application profile and utility function have been set up, we can select the number of consecutive service requests we are going to perform, and finally, by clicking on the buttons at the top, we may request either a local service (only one peer involved), a client/server service (two peers involved), or a distributed service ($n > 2$ peers involved).

Using this benchmark, we have run a large number of experiments and, in the following sections, we will provide a full report of the impact of reflection, context-awareness, conflict resolution, and distribution on CARISMA. The charts we are going to show represent the average elapsed time of 20 consecutive service requests, between the instant when a service request button is pressed, and the instant when a policy is selected and initialised to deliver the service. These performance results do not consider the time necessary to initialise a sensor and to process the information gathered through it. We assume, in fact, that wrappers are already running and monitoring their associated resources, when the Context Controller component queries them about current context, so that a resource value is immediately returned (i.e., no extra overhead is required by the wrapper to gather and process information from its sensors). We believe this approach is plausible, as sensors may greatly vary in nature, and therefore may introduce overheads of different orders of magnitude (e.g., knowing the amount of battery left requires much less time than gathering and processing location information) that we cannot compare. For each experiment, we have observed a standard deviation of approximately 20%.

All tests were performed on Dell Latitude laptops equipped with 128MB RAM, Intel Pentium II processors rated at 300MHz, and connected in an ad-hoc network using Cisco Aironet 340 10Mbps wireless cards. The operating system used was Microsoft's Windows2000 and the Java Virtual Machine version was 1.4.1. We believe these machines do not outperform currently available portable devices. For example, the Sony Ericsson P800 mobile phone is equipped with 12Mb internal storage, plus external memory stick, and ARM9 200MHz processor; the HP iPAQ Pocket PC h5450 is equipped with 64Mb RAM and Intel 400MHz processor; COMPAQ Tablet PC TC1000 is already extremely powerful, with 256MB RAM minimum and 1GHz processor. Therefore, the machines we used are well-suited to estimate the performance of our middleware in (what we believe) the mobile setting of the next 5-10 years.

In the following sections, we first consider a simple local service, i.e., a service that involves a single device. In this simple scenario, we illustrate the basic run-time performance overhead of *reflection*, by varying the number of policies associated with a service, of *context-awareness*, by varying the number of contexts and resources associated with each policy, and of *the conflict resolution mechanism*, by varying the number of conflicting policies and utility function parameters. We then move to a distributed setting and we analyse the run-time performance overhead introduced *by distribution*, while varying the number of devices involved in a service request, and using the two different algorithms to solve conflicts introduced in Chapter 5.

6.3.2 Impact of Reflection

One of the criticisms of reflection is it adds a level of indirection, and therefore reduces performance. In this section, we explore the run-time performance implications of reflection.

Figure 6.11 illustrates the effect that reflection has on the elapsed time of a local service request (i.e., a service that involves a single device), over a basic mechanism where a service is statically associated with exactly one policy. This lower bound is represented in the picture by the intersection of the curve with the Y axis. As the picture shows, the elapsed time (in milliseconds) between the instant when a service request is issued, and the instant when a policy is selected and initialised to answer the service request, is more or less linear in the number of policies associated with the service. This overhead includes also the evaluation of a simple context configuration made of one context with one resource associated with each policy (these associations are necessary to avoid conflicts).

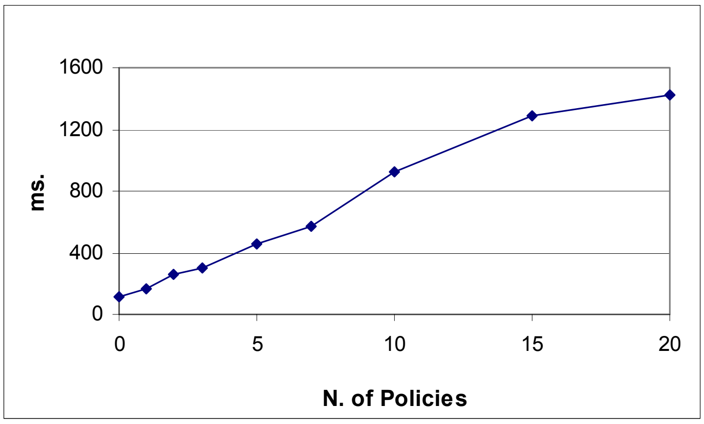


Figure 6.11: Impact of Reflection.

Note that, in realistic situations, it is unlikely that more than ten policies would be associated with the same service. In this case, the performance of CARISMA is rather good, as the elapsed time is below 1 second.

6.3.3 Impact of Context-Awareness

By increasing the number of resources associated with a context, and the number of contexts associated with a policy, the information encoded in an application profile gets very accurate, and adaptation more precisely tailored to application's and user's needs. However, there is a trade-off between accuracy of context information encoded in a profile, and performance. Increasing the number of contexts/resources implies increasing the number of comparisons that the middleware has to perform against the current context,

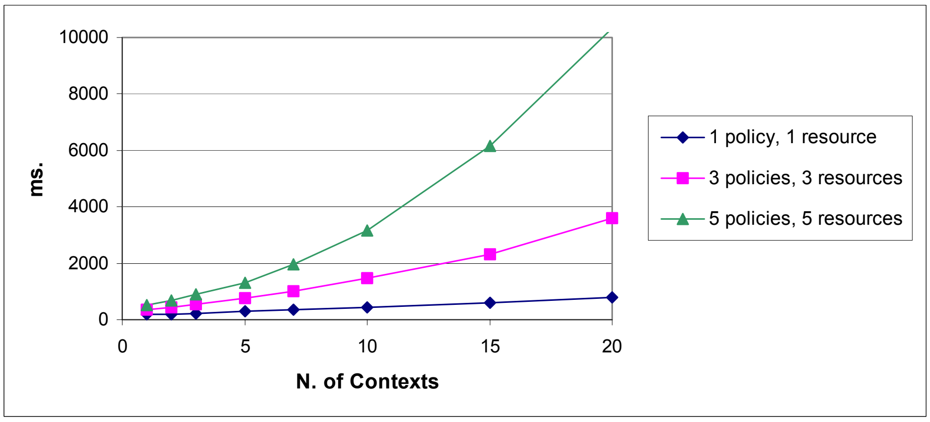


Figure 6.12: Impact of Context-Awareness - Contexts.

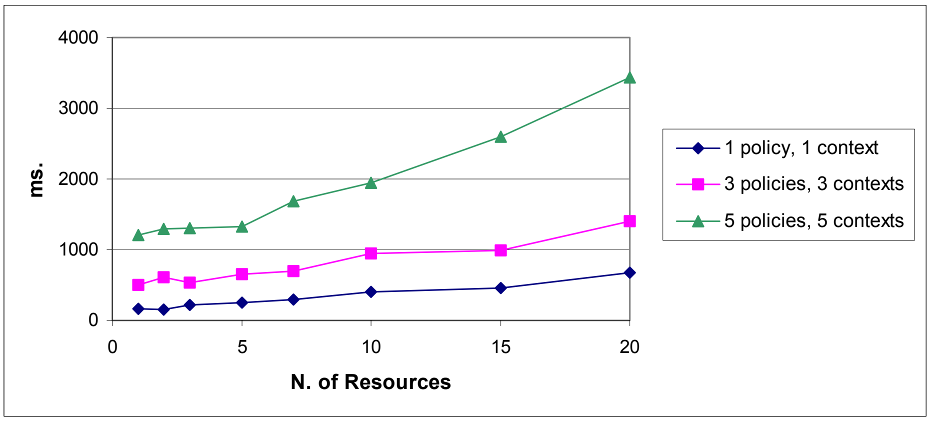


Figure 6.13: Impact of Context-Awareness - Resources.

to find out the set of enabled policies. Figures 6.12 and 6.13 illustrate the performance trend, first when we increase the number of contexts associated with a policy, and then when we increase the number of resources associated with a context.

As shown, increasing the number of contexts (while keeping the number of resources associated with each of them constant) causes a significant drop in performance; the reason is that *all* contexts have to be compared against the current one, as any of them could enable the associated policy ($\vee$ semantics). Increasing the number of resources associated with a context has a lower impact, as the resources associated with a context follow the $\wedge$ semantics, that is, as soon as one resource condition fails to be true, all remaining resource conditions associated with the same context need not be evaluated.

Note that, the elapsed times only seriously degrade for profiles that contain more than five to ten contexts associated with each policy. This means that CARISMA delivers good performance even when rather detailed profiles are used (with up to five policies, each with five to ten contexts, comprising up to five resources).

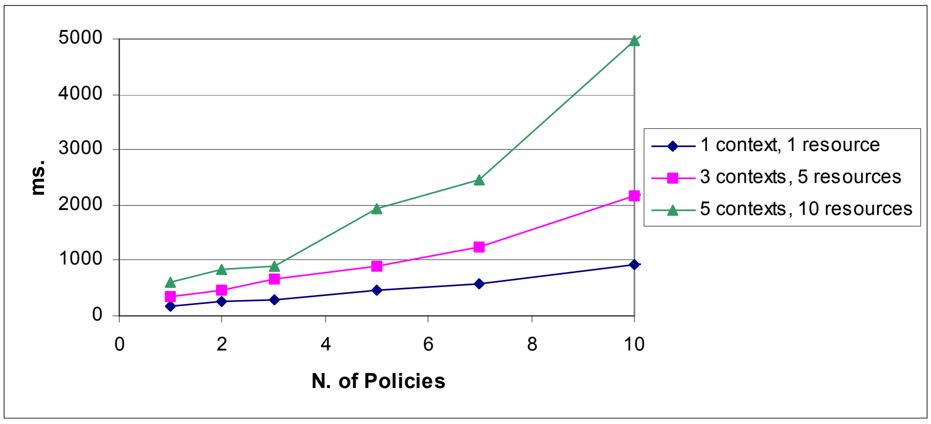


Figure 6.14: Impact of Context-Awareness.

Figure 6.14 combines Figure 6.12 and 6.13 in a number of meaningful combinations of pair values for the number of contexts and resources; we vary the number of policies associated with a service in a range $[1, 10]$, as we consider this value an upper bound on any plausible profile. As shown, having five or more contexts for each policy, and ten or more resources for each context, represents a boundary in the performance of CARISMA, in the sense that the elapsed time to answer a service request becomes unacceptably high if more complex profiles are used.

In our experience with the Conference Application, however, having five policies associated with three contexts with five resources each, already represented the maximum level of adaptation we needed (i.e., the worst-case scenario); in this case, the average amount of time to request a local service is still below one second.

6.3.4 Impact of Conflict Resolution

In measuring the run-time performance overhead introduced by the conflict resolution mechanism when executing a service locally, we found some interesting and encouraging results.

First, as Figure 6.15 shows, the number of utility function parameters does not influence the performance of a service request at all. Also, this chart depicts the performance of a local service request when no context is associated with the policies (i.e., they are always enabled), thus no context evaluation is performed. Let us compare this chart with the one shown in Figure 6.11, where a simple context, made of just one resource condition, was associated with each policy, so that only one policy was enabled at any time. The

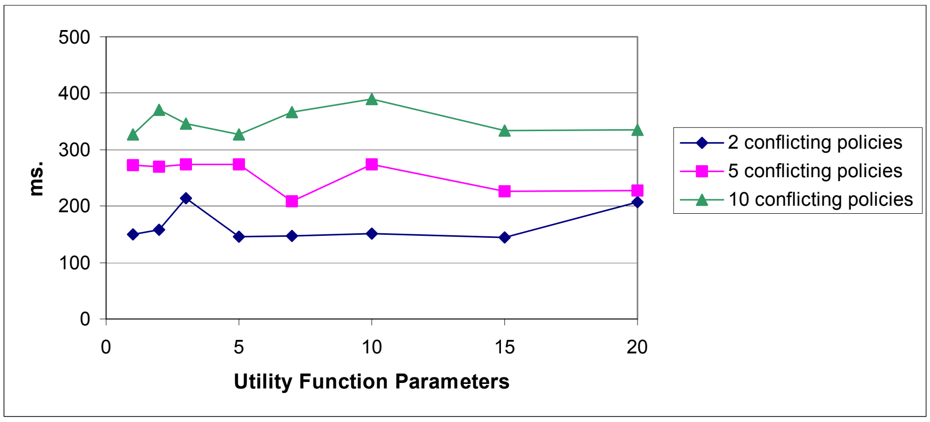


Figure 6.15: Impact of Utility Function Parameters.

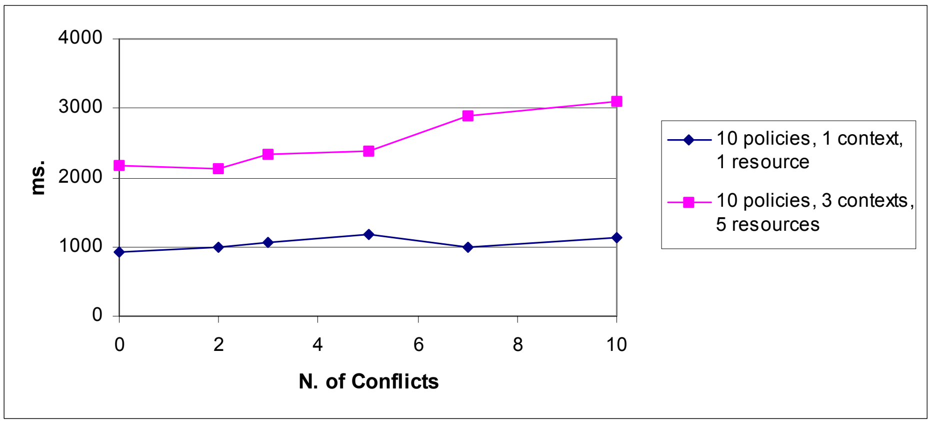


Figure 6.16: Impact of Conflict Resolution Mechanism - Number of Conflicting Policies.

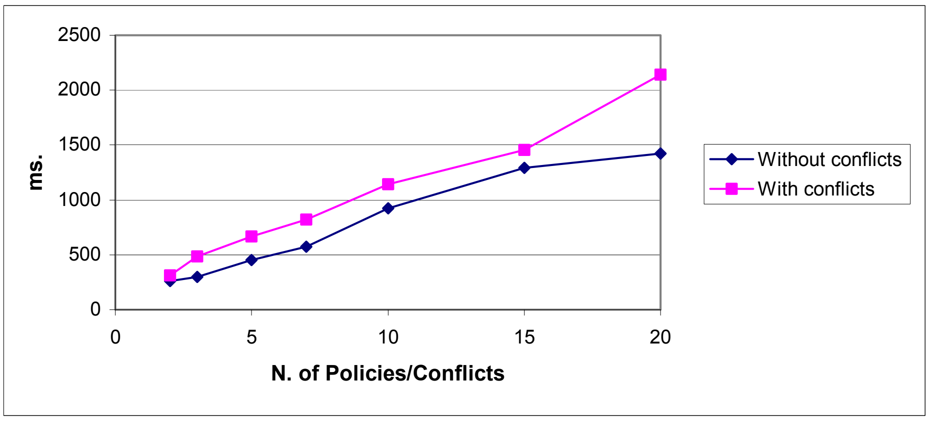


Figure 6.17: Impact of Conflict Resolution Mechanism.

numbers show that the conflict resolution mechanism introduces a much lower overhead than the simplest case of context-awareness. In fact, it takes about 900ms. to determine which policy to apply out of ten, if a very simple, mutually exclusive (i.e., no conflict) context is provided (one context with one resource), while it takes less than 400ms. if no context is provided and the conflict resolution procedure has to be executed to select which of the ten enabled policies to apply.

Second, the number of conflicting policies has almost no impact on the performance of the conflict resolution mechanism. In fact, as Figure 6.16 illustrates, the amount of time required to perform a service request is almost independent of the number of conflicting policies. In other words, the time taken to compute the bids, sum them up, and select the winning policy is little influenced by the cardinality of the solution set. Also, the conflict resolution mechanism adds only a negligible overhead over the standard mechanism, represented by the intersection of the curves with the Y axes.

As Figure 6.17 shows more clearly, in the case that each policy is associated with the same number of contexts and resources (in the picture, they are both set to 1), and all the policies are conflicting, the overhead introduced by the conflicts resolution mechanism is almost constant and in the order of 200ms. We will use these results to derive useful guidelines for writing 'good' application profiles in Section 6.3.6.

6.3.5 Impact of Distribution

The last set of charts shows the performance of CARISMA in answering a service request for some plausible profile configurations, while varying the number of devices involved in the delivery of the service. As shown, both in the presence of conflicts (Figure 6.20 and 6.21) and in their absence (Figure 6.18 and 6.19), the run-time performance overhead

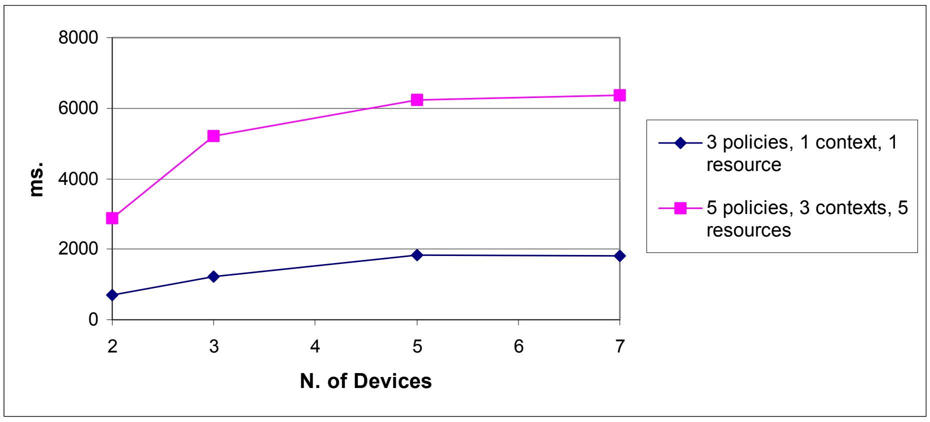


Figure 6.18: Impact of Reflection in a Distributed Setting - First Algorithm.

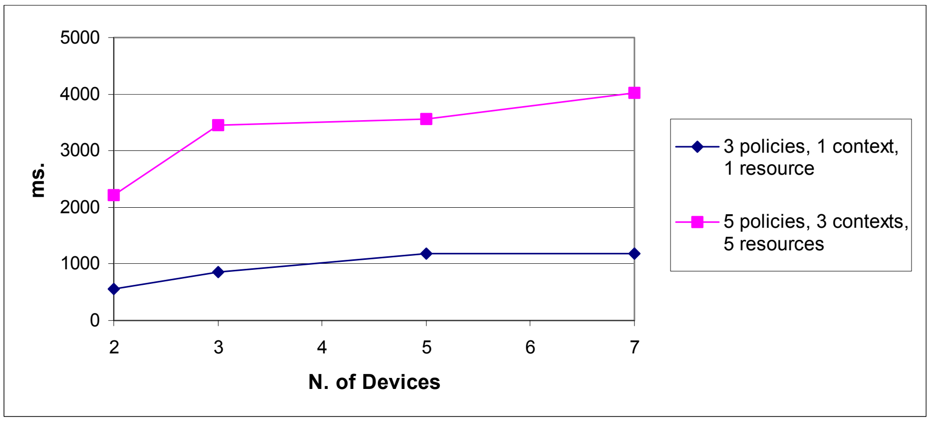


Figure 6.19: Impact of Reflection in a Distributed Setting - Second Algorithm.

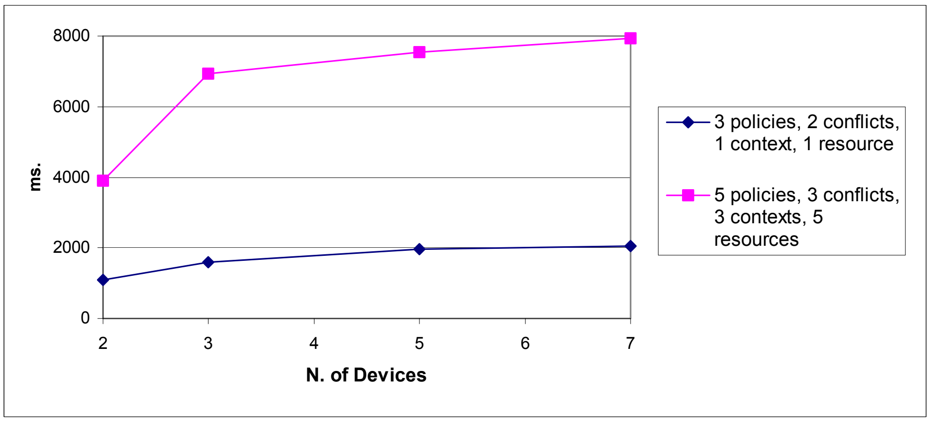


Figure 6.20: Impact of Conflicts in a Distributed Setting - First Algorithm.

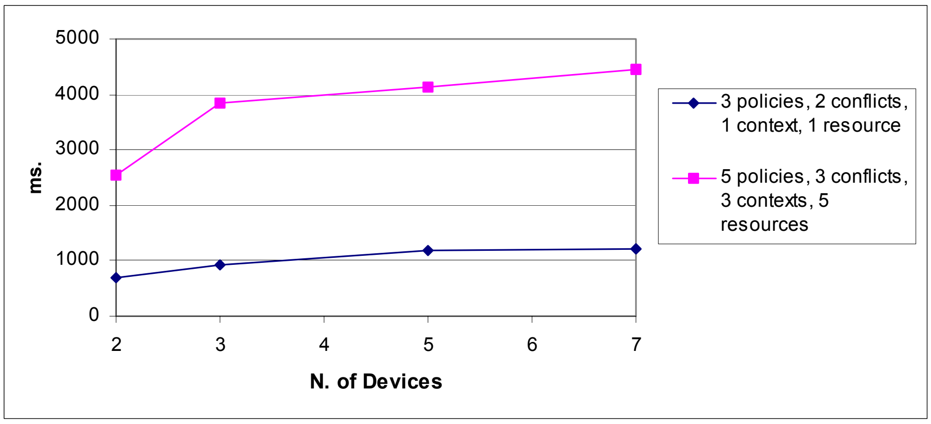


Figure 6.21: Impact of Conflicts in a Distributed Setting - Second Algorithm.

tends to be constant, and does not increase considerably when increasing the number of devices involved. The results shown here do not consider peer failures though; if peer failures are taken into account, the elapsed time depends on the timeout values used before acknowledging a peer is no longer within reach. Also, these charts show us that the evaluation of context and determination of the locally enabled policies is by far the most time-consuming task of the whole service request and conflict resolution process, as shown by the large gap between the two curves in each chart.

Figure 6.22 compares the performance of the two algorithms. The results strengthen our thesis that context evaluation is by far the heaviest task performed by our middleware model when responding to a service request. In fact, the second algorithm that maximises parallelisation of computation of P_i outperforms the first one, even in the case of peer-to-peer interactions, where the second algorithm requires one more step of communication. In particular, using the second algorithm, where all $P_i,\ i \in [1, n]$ are evaluated in parallel, it takes almost half of the time to answer a service request, compared to the first algorithm, where only $n - 1$ evaluations are executed in parallel.

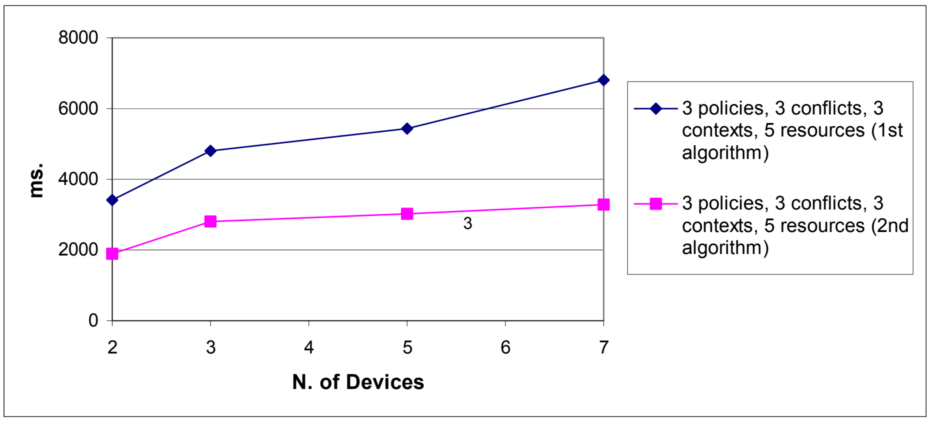


Figure 6.22: Algorithm Comparison.

We have also measured the size of the messages sent around during the conflict resolution process: it varies between 150 bytes when only two policies are conflicting, and 635 bytes when ten policies are enabled. The size of the information exchanged is therefore minimal.

6.3.6 Design Guidelines

From the performance evaluation results that we have presented in the previous sections, we have derived some important guidelines that application engineers should adhere to when developing applications on top of CARISMA.

First, a detailed and precise adaptation to context changes has a high impact on performance. In fact, as Figures 6.12, 6.13 and 6.14 illustrate, the higher the number of resources associated with a context, and of contexts associated with a policy, the heavier the performance overhead, and therefore the worse the performance. Although we have only given a flavour of how applications can derive application profiles from user preferences, leaving this issue to future research, our experience in developing the Conference Application has shown that synthesising detailed and precise context descriptions from user preferences may become difficult when the number of resources and contexts increases. However, we have found that meaningful context descriptions do not need to contain many details, and having three to five contexts, with more or less five resources associated with each of them, already enables a very high level of customisation.

Guideline 1 *In developing mobile context-aware applications, context descriptions should be as simple as possible, containing only the information that is needed to discriminate between different policies.*

Second, the conflict resolution mechanism we have designed and implemented adds a very low performance overhead over a standard service request. In fact, as Figures 6.15 and 6.16 have illustrated, both the number of utility function parameters, and the number of conflicting policies, only marginally influence the performance of CARISMA. Also, gathering information about user preferences (e.g., the importance the user gives to non-functional requirements and resources) is a rather straightforward task, compared to deriving precise context descriptions. This observation, combined with the previous one, has led us to the following guideline.

Guideline 2 *In developing mobile context-aware applications, only minimal context configurations should be associated with the policies, letting the conflict resolution mechanism efficiently solve potential conflicts.*

In addition, to reduce the risk of ending a conflict resolution process with a failure, because no agreed policy could be found, a good strategy is to leave one or more policies always enabled, if possible. These should be 'harmless' policies that, if applied, provide reasonable benefit in any context, although different policies, in different contexts, would have achieved a higher quality-of-service. These 'better' policies should be favoured during the conflict resolution process by higher bids.

Guideline 3 *In developing mobile context-aware applications, 'harmless' policies should always be enabled, thus reducing the risk of conflict resolution failures. These policies will however be less preferred than more context-specific ones.*

Following these guidelines, the difficulties that application engineers are exposed to in developing applications on top of CARISMA are rather limited; mainly, they must be able to gather numbers that represent the importance that users associate with non-functional requirements and resources at run-time. Once non-functional requirements and relevant resources have been isolated, this task can be easily accomplished using, for example, the customisation windows shown in Figure 6.1 and 6.2. Deriving context configurations that must be associated with different policies in different circumstances is more difficult, as different preferences must be combined, taking also policy specifications into account. However, adhering to the lessons we have elicited, only simple configurations are needed, and therefore application engineers do not need to design complex synthesising (i.e., from user preferences and policy specifications to application profiles) algorithms.

6.4 Summary

This chapter has presented the implementation of CARISMA, and discussed the results of its evaluation. The Conference Application has been used to discuss qualitative results, both in terms of usability and effectiveness, while a benchmark has been introduced and used to discuss performance results in non-trivial situations.

In terms of qualitative results, we have shown that CARISMA offers powerful abstractions, in particular, application profiles, that ease the development of context-aware mobile applications. Application engineers do not have to deal with heterogeneity of sensors, nor do they have to directly query their status; on the other hand, the task they are required to undertake is to gather information from the users about their preferences, and to process this information to synthesise application profiles and utility functions. With the help of application domain experts, who elicit non-functional application requirements, and with the use of the design guidelines discussed in Section 6.3.6, this task does not represent a burden.

As for performance results, we have demonstrated that the overhead imposed by CARISMA is not too heavy, and can be accommodated by currently available mobile devices. Both the reflective and conflict resolution mechanisms have an almost negligible impact on a basic service request where a service is statically associated with a policy; also, distribution has an impact on performance that tends to be constant with the number of devices involved in a service request. Supporting context-awareness is the most time-consuming and resource-demanding task; however, the overhead imposed is acceptable in plausible configurations.

We can conclude that CARISMA facilitates the development of context-aware mobile applications, exploiting powerful abstractions and efficient mechanisms that both ease the task of application engineers, and impose very little overhead. As far as performance is concerned, we are aware that this can be improved further, by adopting some new technologies that have become available; for example, the kXML parser [kObjects, 2002], a lighter and faster parser than the Xerces parser we have used, that specifically targets the mobile environment, and J2ME [Sun Microsystem, 2000] or PersonalJava [Sun Microsystem, 1997], instead of the heavier Java Standard Edition we have used. However, our goal in implementing CARISMA was to prove the validity and applicability of the principles we have discussed in Chapter 3 and 4 in a mobile setting, and we have achieved this, without needing to update our code with these latest technologies.

Chapter 7

Conclusions and Future Work

The main goal of the work presented in this thesis has been the development of abstractions and mechanisms that, embedded in a mobile computing middleware software layer, effectively enhance the construction and execution of context-aware mobile applications. More specifically, we have modelled application adaptation to context changes by means of a set of associations between the services that the application wishes to customise, the policies used to deliver these services, and the context configurations that enable these policies. We have then investigated the principle of reflection to allow applications to dynamically alter these associations, thus adapting their behaviour at run-time, in order to deliver the best quality-of-service in varying contexts and according to different user needs. In order to solve potential conflicts among the QoS needs of interacting applications, we have designed a dynamic, simple and customisable conflict resolution mechanism based on microeconomic techniques. Our work has resulted in CARISMA, a mobile distributed architecture that realises both the reflective model and the conflict resolution mechanism. We have provided an implementation of CARISMA and built applications on top of it, to demonstrate its effectiveness in developing context-aware mobile applications, as well as its suitability in resource-constrained settings. In this last chapter, we revise the main contributions of this thesis, provide a critical evaluation of the goals attained, and, finally, we discuss some open issues that we leave for future development.

7.1 Contributions

The contributions of this thesis to mobile computing middleware are summarised below.

Application Adaptation to Context Changes

Our work contributes the development of a model that, based on reflection and metadata, enables both reactive and proactive application adaptation to context changes. Adaptation takes place by means of metadata, or application profiles, that contain both reactive and proactive associations. Reactive associations relate context configurations to policies that have to be fired when such configurations occur; proactive associations relate the services that applications wish to customise, to the policies that must be used to deliver the services, and the context configurations that must hold in order for a policy to be enabled. Through reflection, applications can dynamically alter the meta-information, thus adapting their behaviour to varying context conditions and user needs.

We provide a general and flexible definition of context, that abstracts away from the heterogeneity of physical sensors and the information gathered through them. The middleware, and not the applications, takes charge of dealing with a diversity of sensors, and of encoding the information obtained in a uniform way. Applications then access this encoding through a well-defined interface; also, they exploit this flexible context abstraction to easily customise the set of resources that make up the context of interest at run-time, possibly adding application-specific resources too.

This model has been formalised and mapped onto the CARISMA reflective architecture. This architecture has then be implemented and thoroughly evaluated, to prove its suitability to the mobile setting.

Quality-of-Service Conflict Resolution

Our work contributes a mechanism for dynamically resolving QoS conflicts using microeconomic techniques. Applications participating in the delivery of the same service may disagree on the policy that has to be used to deliver the service, that is, the QoS level they are willing to achieve. In order to solve potential conflicts among (distributed) applications, we have designed a microeconomic mechanism; in this economy, applications are consumers seeking to achieve their own goals, that is, to have a service delivered using the policy that achieves the best quality-of-service, according to application-specific preferences. Our conflict resolution mechanism enables applications to express their own preferences (i.e., to associate values with the use of each conflicting policy), and therefore influence the way conflicts are resolved.

We have formalised the mechanism and designed distributed algorithms that realise it. We have provided an implementation of these algorithms and finally evaluated them, to prove the simplicity and effectiveness of our conflict resolution mechanism.

7.2 Critical Evaluation

As we have pointed out in Chapter 1, two fundamental requirements of mobile computing middleware are to be light-weight and to support context-awareness. In this section we evaluate CARISMA with respect to these criteria.

Light-Weight Middleware

The middleware model we have developed specifically targets mobile devices, that is, devices with resource limitations, in terms, for example, of memory and processor speed. Mobile computing middleware must therefore be light-weight, and impose the minimum overhead possible. In Chapter 6, we have demonstrated that our middleware model implementation meets this requirement.

Both reflection and the conflict resolution mechanism impose a very small overhead, both in terms of elapsed time to answer a service request, and memory requirements. The more resource-consuming task is the evaluation of context: the richer and more detailed the context description, the heavier the evaluation. However, the complexity of context configurations is limited in realistic situations, thus imposing an overhead that current mobile devices can bear. Moreover, our empirical evaluation has suggested guidelines (Section 6.3.6) that help in cutting down the overhead related to context-awareness further.

Context-Awareness

Our reflective middleware model supports context-awareness by means of powerful abstractions that model application behaviour in different contexts in a flexible and easy-to-manipulate manner. Proving the extent to which our middleware model fosters the development and deployment of context-aware applications, that is, measuring its effectiveness and usability, is rather hard, as there are no quantitative parameters we can compute by running a set of experiments. However, as we have discussed in Section 6.2, our experience in developing applications on top of CARISMA has revealed that both goals have been attained.

As far as usability is concerned, application engineers are not exposed to the difficulties of dealing with heterogeneous sensors, gathering and processing context information, etc. The only task they are required to undertake is to map user preferences into application profiles and utility functions: using the well-defined meta-interface our model provides, this task does not represent a problem.

As for effectiveness, application adaptation to context changes can be easily achieved by means of the abstractions provided. Context configurations of limited complexity (e.g., five resources associated with three contexts) were sufficient to capture the maximum level of adaptation needed by the Conference Application.

7.3 Future Work

The work presented in this thesis establishes a theoretical foundation of middleware primitives for enhancing the development and execution of context-aware mobile applications. Based on this model, future developments include the following.

Improvement of Current Model

Our current model can be improved in several respects:

- Although our definition of context is general enough to capture any resource available in the physical environment, we have mainly exploited a view of context that is local to the device, that is, it includes the information gathered from locally available sensors only. When broadening this view outside the physical device, to include context information gathered from any peer directly (or indirectly) connected, other issues arise; in particular, it becomes necessary to deal with the binding and re-binding of external sensors while on the move. In [Roman et al., 2002, Julien and Roman, 2002], a middleware that tackles the issues of a broad definition and maintenance of context has been presented; however, a final solution to the binding problem has not yet been found.
- To accommodate dynamicity requirements, services and policies may be installed and uninstalled on the fly; moreover, different application needs may result in different system configurations that vary over time. The changing interactions among distributed services and policies may alter the semantics of the applications built on top of our reflective middleware. The development of safe customisable middleware therefore becomes an issue. A first step towards the definition of a formal semantics for specifying and reasoning about the properties of, and interactions among, middleware components can be found in [Venkatasubramanian and Talcott, 1995]. These principles have been used, for example, in [Venkatasubramanian et al., 2001] to manage changes in large-scale distributed systems while ensuring application QoS requirements. The principles they use are based on a two-level architecture where the application, at the base level, interacts with the middleware, at the meta-level, via middleware-defined core services that are then used to initiate other activities. The similarity of this approach with our architecture makes us think that similar principles could be investigated to develop a formal semantics of composition within our reflective middleware framework.

An alternative may be to use an architectural approach: an architecture specification, containing constraints about how components are composed, is provided. During the system lifetime, the run-time architecture enforces that insertion and removal of components do not violate these constraints. Early work in this area can be found in [Georgiadis et al., 2002]. They propose to associate configuration managers with each component; these managers are in charge of verifying architectural constraints each time the configuration changes (because of, for example, component join/leave events, or because of binding/unbinding actions taken by component managers). Many research issues remain open in this direction; for example, the dynamic nature of the architectural constraints must be taken into account.

- Although we have paid little attention to the communication issue, it is important to state that the message-passing communication paradigm we provide needs to be improved, as it does not support persistence of message queues (i.e., it does not support disconnections). An important engineering exercise would be, therefore, to integrate available implementations of the Java Message Service for mobile settings, for example [Softwired, 2002], within our middleware. Currently available implementations, however, only target nomadic networks; an important research issue would be to port them to ad-hoc settings too.

Increased Flexibility through Logical Mobility

Despite being a very powerful concept, reflection enables adaptability and flexibility only in those contexts that middleware designers have considered likely to be unstable at design time. However, in a mobile ad-hoc setting, mobile hosts cannot forecast all the possible contexts they are going to encounter, and therefore which behaviours (i.e., policies) they are going to need. New behaviours may be delivered from time to time to cope with unforeseen context configurations and new application needs.

A major future direction of research is to exploit mobile code techniques to overcome this limitation, for example, by downloading new protocols either from a service provider or from other peers within reach that use the same behaviour [Capra et al., 2001c] [Zachariadis et al., 2002]. Moreover, only a minimum set of behaviours can be stored on a device so as to avoid consuming memory; by exchanging information about what services, code and resources are available with other peers, different behaviours can be downloaded only when needed (*if* needed). Reflection can be combined with mobile code techniques to allow applications to select where to download protocols from, based on application-specific information (e.g., trusted hosts, quality-of-service parameters, etc.).

QoS-aware Service Discovery and Delivery

Another major direction of research is service discovery and delivery. Traditional naming and trading service discovery techniques developed for fixed distributed systems cannot be successfully applied in mobile settings, where intermittent rather than continuous network connection is the norm. However, service discovery for mobile settings has not yet gained significant attention. Two notable exceptions are the Jini specification [Arnold et al., 1999] and the work by Handorean and Roman [Handorean and Roman, 2002]. A disadvantage of both approaches is that they do not take quality-of-service requirements into account when deciding which service to use.

We believe that QoS-aware service discovery would fit naturally in our framework, where application needs are made explicit and used to decide how a service should be delivered in the current context. Currently, these needs are taken into account only locally; a future direction of research would be to make use of this information to discover services available in an entire ad-hoc network that would deliver to the user the best QoS, according to current user-specific requirements.

Appendix A

Reflective API Semantics

A.1 Introspection

Semantics of $readRP$ (Read Reactive Policy)

$$\begin{aligned}
readRP &: profile \times \mathrm{P} \rightarrow policy \cup \{null\} \\
readRP[\![(react\ proact,\ pn)]\!] &= readRP[\![(react,\ pn)]\!] \\
readRP[\![((pn'\ cL)pL,\ pn)]\!] &= \begin{cases} (pn'\ cL) \text{ if } pn = pn' \\ readRP[\![(pL,\ pn)]\!] \text{ otherwise} \end{cases} \\
readRP[\![(pn'\ cL,\ pn)]\!] &= \begin{cases} (pn'\ cL) \text{ if } pn = pn' \\ null \text{ otherwise} \end{cases} \\
readRP[\![(\varepsilon, pn)]\!] &= null
\end{aligned}$$

Semantics of $readRC$ (Read Reactive Context)

$$\begin{aligned}
readRC &: profile \times \mathrm{P} \times \mathbb{N} \rightarrow context \cup \{null\} \\
readRC_{cl} &: contextList \times \mathbb{N} \rightarrow context \cup \{null\} \\
readRC[\![(react\ proact,\ pn,\ cid)]\!] &= readRC[\![(react,\ pn,\ cid)]\!] \\
readRC[\![((pn'\ cL)pL,\ pn,\ cid)]\!] &= \begin{cases} readRC_{cl}[\![(cL,\ cid)]\!] \text{ if } pn = pn' \\ readRC[\![(pL,\ pn,\ cid)]\!] \text{ otherwise} \end{cases} \\
readRC[\![(pn'\ cL,\ pn,\ cid)]\!] &= \begin{cases} readRC_{cl}[\![(cL,\ cid)]\!] \text{ if } pn = pn' \\ null \text{ otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
readRC_{cl}[\![((cid'\ rL)cL,\ cid)]\!] &= \begin{cases} (cid'\ rL) \text{ if } cid = cid' \\ readRC_{cl}[\![(cL,\ cid)]\!] \text{ otherwise} \end{cases} \\
readRC_{cl}[\![(cid'\ rL,\ cid)]\!] &= \begin{cases} (cid'\ rL) \text{ if } cid = cid' \\ null \text{ otherwise} \end{cases} \\
readRC[\![(\varepsilon, pn, cid)]\!] &= null
\end{aligned}$$

Semantics of $readRR$ (Read Reactive Resource)

$$\begin{aligned}
readRR &: profile \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow resource \cup \{null\} \\
readRR_{cl} &: contextList \times \mathbb{N} \times \mathrm{R} \rightarrow resource \cup \{null\} \\
readRR_{rl} &: resourceList \times \mathrm{R} \rightarrow resource \cup \{null\} \\
readRR[\![(react\ proact,\ pn,\ cid,\ rn)]\!] &= readRR[\![(react,\ pn,\ cid,\ rn)]\!] \\
readRR[\![((pn'\ cL)pL,\ pn,\ cid,\ rn)]\!] &= \begin{cases} readRR_{cl}[\![(cL,\ cid,\ rn)]\!] \text{ if } pn = pn' \\ readRR[\![(pL,\ pn,\ cid,\ rn)]\!] \text{ otherwise} \end{cases} \\
readRR[\![(pn'\ cL,\ pn,\ cid,\ rn)]\!] &= \begin{cases} readRR_{cl}[\![(cL,\ cid,\ rn)]\!] \text{ if } pn = pn' \\ null \text{ otherwise} \end{cases} \\
readRR_{cl}[\![((cid'\ rL)cL,\ cid,\ rn)]\!] &= \begin{cases} readRR_{rl}[\![(rL,\ rn)]\!] \text{ if } cid = cid' \\ readRR_{cl}[\![(cL,\ cid,\ rn)]\!] \text{ otherwise} \end{cases} \\
readRR_{cl}[\![(cid'\ rL,\ cid,\ rn)]\!] &= \begin{cases} readRR_{rl}[\![(rL,\ rn)]\!] \text{ if } cid = cid' \\ null \text{ otherwise} \end{cases} \\
readRR_{rl}[\![((rn\ on\ vL)rL,\ rn)]\!] &= \begin{cases} (rn\ on\ vL) \text{ if } rn = rn' \\ readRR_{rl}[\![(rL,\ rn)]\!] \text{ otherwise} \end{cases} \\
readRR_{rl}[\![((rn\ on\ vL),\ rn)]\!] &= \begin{cases} (rn\ on\ vL) \text{ if } rn = rn' \\ null \text{ otherwise} \end{cases} \\
readRR[\![(\varepsilon, pn, cid, rn)]\!] &= null
\end{aligned}$$

Semantics of $readPS$ (Read Proactive Service)

$$\begin{aligned}
readPS &: profile \times \mathrm{S} \rightarrow service \cup \{null\} \\
readPS[\![(react\ proact,\ sn)]\!] &= readPS[\![(proact,\ sn)]\!] \\
readPS[\![((sn'\ pL)sL,\ sn)]\!] &= \begin{cases} (sn'\ pL) \text{ if } sn = sn' \\ readPS[\![(sL,\ sn)]\!] \text{ otherwise} \end{cases}
\end{aligned}$$

$$
\begin{aligned}
readPS[\![(sn'\ pL,\ sn)]\!] &= \begin{cases} (sn'\ pL) \text{ if } sn = sn' \\ null \text{ otherwise} \end{cases} \\
readPS[\![(\varepsilon, sn)]\!] &= null
\end{aligned}
$$

Semantics of $readPP$ (Read Proactive Policy)

$$
\begin{aligned}
readPP &: profile \times \mathrm{S} \times \mathrm{P} \to policy \cup \{null\} \\
readPP_{pl} &: policyList \times \mathrm{P} \to policy \cup \{null\} \\
readPP[\![(react\ proact,\ sn,\ pn)]\!] &= readPP[\![(proact,\ sn,\ pn)]\!] \\
readPP[\![((sn'\ pL)sL,\ sn,\ pn)]\!] &= \begin{cases} readPP_{pl}[\![(pL,\ pn)]\!] \text{ if } sn = sn' \\ readPP[\![(sL,\ sn,\ pn)]\!] \text{ otherwise} \end{cases} \\
readPP[\![(sn'\ pL,\ sn,\ pn)]\!] &= \begin{cases} readPP_{pl}[\![(pL,\ pn)]\!] \text{ if } sn = sn' \\ null \text{ otherwise} \end{cases} \\
readPP_{pl}[\![((pn'\ cL)pL,\ pn)]\!] &= \begin{cases} (pn'\ cL) \text{ if } pn = pn' \\ readPP_{pl}[\![(pL,\ pn)]\!] \text{ otherwise} \end{cases} \\
readPP_{pl}[\![(pn'\ cL,\ pn)]\!] &= \begin{cases} (pn'\ cL) \text{ if } pn = pn' \\ null \text{ otherwise} \end{cases} \\
readPP[\![(\varepsilon, sn, pn)]\!] &= null
\end{aligned}
$$

Semantics of $readPC$ (Read Proactive Context)

$$
\begin{aligned}
readPC &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \to context \cup \{null\} \\
readPC_{pl} &: policyList \times \mathrm{P} \times \mathbb{N} \to context \cup \{null\} \\
readPC_{cl} &: contextList \times \mathbb{N} \to context \cup \{null\} \\
readPC[\![(react\ proact,\ sn,\ pn,\ cid)]\!] &= readPC[\![(proact,\ sn,\ pn,\ cid)]\!] \\
readPC[\![((sn'\ pL)sL,\ sn,\ pn,\ cid)]\!] &= \begin{cases} readPC_{pl}[\![(pL,\ pn,\ cid)]\!] \text{ if } sn = sn' \\ readPC[\![(sL,\ sn,\ pn,\ cid)]\!] \text{ otherwise} \end{cases} \\
readPC[\![(sn'\ pL,\ sn,\ pn,\ cid)]\!] &= \begin{cases} readPC_{pl}[\![(pL,\ pn,\ cid)]\!] \text{ if } sn = sn' \\ null \text{ otherwise} \end{cases} \\
readPC_{pl}[\![((pn'\ cL)pL,\ pn,\ cid)]\!] &= \begin{cases} readPC_{cl}[\![(cL,\ cid)]\!] \text{ if } pn = pn' \\ readPC_{pl}[\![(pL,\ pn,\ cid)]\!] \text{ otherwise} \end{cases}
\end{aligned}
$$

$$\begin{aligned}
readPC_{pl}[\![(pn'\ cL,\ pn,\ cid)]\!] &= \begin{cases} readPC_{cl}[\![(cL,\ cid)]\!] \text{ if } pn = pn' \\ null \text{ otherwise} \end{cases} \\
readPC_{cl}[\![((cid'\ rL)cL,\ cid)]\!] &= \begin{cases} (cid'\ rL) \text{ if } cid = cid' \\ readPC_{cl}[\![(cL,\ cid)]\!] \text{ otherwise} \end{cases} \\
readPC_{cl}[\![(cid'\ rL,\ cid)]\!] &= \begin{cases} (cid'\ rL) \text{ if } cid = cid' \\ null \text{ otherwise} \end{cases} \\
readPC[\![(\varepsilon, sn, pn, cid)]\!] &= null
\end{aligned}$$

Semantics of $readPR$ (Read Proactive Resource)

$$\begin{aligned}
readPR &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow resource \cup \{null\} \\
readPR_{pl} &: policyList \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow resource \cup \{null\} \\
readPR_{cl} &: contextList \times \mathbb{N} \times \mathrm{R} \rightarrow resource \cup \{null\} \\
readPR_{rl} &: resourceList \times \mathrm{R} \rightarrow resource \cup \{null\} \\
readPR[\![(react\ proact,\ sn,\ pn,\ cid,\ rn)]\!] &= readPR[\![(proact,\ sn,\ pn,\ cid,\ rn)]\!] \\
readPR[\![((sn'\ pL)sL,\ sn,\ pn,\ cid,\ rn)]\!] &= \begin{cases} readPR_{pl}[\![(pL,\ pn,\ cid,\ rn)]\!] \text{ if } sn = sn' \\ readPR[\![(sL,\ sn,\ pn,\ cid,\ rn)]\!] \text{ otherwise} \end{cases} \\
readPR[\![(sn'\ pL,\ sn,\ pn,\ cid,\ rn)]\!] &= \begin{cases} readPR_{pl}[\![(pL,\ pn,\ cid,\ rn)]\!] \text{ if } sn = sn' \\ null \text{ otherwise} \end{cases} \\
readPR_{pl}[\![((pn'\ cL)pL,\ pn,\ cid,\ rn)]\!] &= \begin{cases} readPR_{cl}[\![(cL,\ cid,\ rn)]\!] \text{ if } pn = pn' \\ readPR_{pl}[\![(pL,\ pn,\ cid,\ rn)]\!] \text{ otherwise} \end{cases} \\
readPR_{pl}[\![(pn'\ cL,\ pn,\ cid,\ rn)]\!] &= \begin{cases} readPR_{cl}[\![(cL,\ cid,\ rn)]\!] \text{ if } pn = pn' \\ null \text{ otherwise} \end{cases} \\
readPR_{cl}[\![((cid'\ rL)cL,\ cid,\ rn)]\!] &= \begin{cases} readPR_{rl}[\![(rL,\ rn)]\!] \text{ if } cid = cid' \\ readPR_{cl}[\![(cL,\ cid,\ rn)]\!] \text{ otherwise} \end{cases} \\
readPR_{cl}[\![(cid'\ rL,\ cid,\ rn)]\!] &= \begin{cases} readPR_{rl}[\![(rL,\ rn)]\!] \text{ if } cid = cid' \\ null \text{ otherwise} \end{cases} \\
readPR_{rl}[\![((rn\ on\ vL)rL,\ rn)]\!] &= \begin{cases} (rn\ on\ vL) \text{ if } rn = rn' \\ readPR_{rl}[\![(rL,\ rn)]\!] \text{ otherwise} \end{cases} \\
readPR_{rl}[\![((rn\ on\ vL),\ rn)]\!] &= \begin{cases} (rn\ on\ vL) \text{ if } rn = rn' \\ null \text{ otherwise} \end{cases} \\
readPR[\![(\varepsilon, sn, pn, cid, rn)]\!] &= null
\end{aligned}$$

A.2 Adaptation

A.2.1 Remove

Semantics of $remRP$ (Remove Reactive Policy)

$$\begin{aligned}
remRP &: profile \times \mathrm{P} \rightarrow profile \\
remRP[\![(react\ proact,\ pn)]\!] &= remRP[\![(react,\ pn)]\!] \cup proact \\
remRP[\![((pn'\ cL)pL,\ pn)]\!] &= \begin{cases} pL \text{ if } pn = pn' \\ \{(pn'\ cL)\} \cup remRP[\![(pL,\ pn)]\!] \text{ otherwise} \end{cases} \\
remRP[\![(pn'\ cL,\ pn)]\!] &= \begin{cases} \emptyset \text{ if } pn = pn' \\ \{(pn'\ cL)\} \text{ otherwise} \end{cases} \\
remRP[\![(\varepsilon, pn)]\!] &= \emptyset
\end{aligned}$$

Semantics of $remRC$ (Remove Reactive Context)

$$\begin{aligned}
remRC &: profile \times \mathrm{P} \times \mathbb{N} \rightarrow profile \\
remRC_{cl} &: contextList \times \mathbb{N} \rightarrow contextList \\
remRC[\![(react\ proact,\ pn,\ cid)]\!] &= remRC[\![(react,\ pn,\ cid)]\!] \cup proact \\
remRC[\![((pn'\ cL)pL,\ pn,\ cid)]\!] &= \begin{cases} \{(pn'\ remRC_{cl}[\![(cL,\ cid)]\!])\} \cup pL \text{ if } pn = pn' \\ \{(pn'\ cL)\} \cup remRC[\![(pL,\ pn,\ cid)]\!] \text{ otherwise} \end{cases} \\
remRC[\![(pn'\ cL,\ pn,\ cid)]\!] &= \begin{cases} \{(pn'\ remRC_{cl}[\![(cL,\ cid)]\!])\} \text{ if } pn = pn' \\ \{(pn'\ cL)\} \text{ otherwise} \end{cases} \\
remRC_{cl}[\![((cid'\ rL)cL,\ cid)]\!] &= \begin{cases} cL \text{ if } cid = cid' \\ \{(cid'\ rL)\} \cup remRC_{cl}[\![(cL,\ cid)]\!] \text{ otherwise} \end{cases} \\
remRC_{cl}[\![(cid'\ rL,\ cid)]\!] &= \begin{cases} \emptyset \text{ if } cid = cid' \\ \{(cid'\ rL)\} \text{ otherwise} \end{cases} \\
remRC[\![(\varepsilon, pn, cid)]\!] &= \emptyset
\end{aligned}$$

Semantics of $remRR$ (Remove Reactive Resource)

$$\begin{aligned}
remRR &: profile \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow profile \\
remRR_{cl} &: contextList \times \mathbb{N} \times \mathrm{R} \rightarrow contextList
\end{aligned}$$

$$
\begin{aligned}
remRR_{rl} &: resourceList \times \mathrm{R} \rightarrow resourceList \\
remRR[\![(react\ proact,\ pn,\ cid,\ rn)]\!] &= remRR[\![(react,\ pn,\ cid,\ rn)]\!] \cup proact \\
remRR[\![((pn'\ cL)pL,\ pn,\ cid,\ rn)]\!] &= \begin{cases} \{(pn'\ remRR_{cl}[\![(cL,\ cid,\ rn)]\!])\} \cup pL \\ \quad \text{if } pn = pn' \\ \{(pn'\ cL)\} \cup remRR[\![(pL,\ pn,\ cid,\ rn)]\!] \\ \quad \text{otherwise} \end{cases} \\
remRR[\![(pn'\ cL,\ pn,\ cid,\ rn)]\!] &= \begin{cases} \{(pn'\ remRR_{cl}[\![(cL,\ cid,\ rn)]\!])\} \text{ if } pn = pn' \\ \{(pn'\ cL)\} \text{ otherwise} \end{cases} \\
remRR_{cl}[\![((cid'\ rL)cL,\ cid,\ rn)]\!] &= \begin{cases} \{(cid'\ remRR_{rl}[\![(rL,\ rn)]\!])\} \cup cL \\ \quad \text{if } cid = cid' \\ remRR_{cl}[\![(cL,\ cid,\ rn)]\!] \text{ otherwise} \end{cases} \\
remRR_{cl}[\![(cid'\ rL,\ cid,\ rn)]\!] &= \begin{cases} \{(cid'\ remRR_{rl}[\![(rL,\ rn)]\!])\} \text{ if } cid = cid' \\ \{(cid'\ rL)\} \text{ otherwise} \end{cases} \\
remRR_{rl}[\![((rn'\ on\ vL)rL,\ rn)]\!] &= \begin{cases} rL \text{ if } rn = rn' \\ \{(rn'\ on\ vL)\} \cup remRR_{rl}[\![(rL,\ rn)]\!] \\ \quad \text{otherwise} \end{cases} \\
remRR_{rl}[\![((rn'\ on\ vL),\ rn)]\!] &= \begin{cases} \emptyset \text{ if } rn = rn' \\ \{(rn'\ on\ vL)\} \text{ otherwise} \end{cases} \\
remRR[\![(\varepsilon, pn, cid, rn)]\!] &= \emptyset
\end{aligned}
$$

Semantics of $remPS$ (Remove Proactive Service)

$$
\begin{aligned}
remPS &: profile \times \mathrm{S} \rightarrow profile \\
remPS[\![(react\ proact,\ sn)]\!] &= react \cup remPS[\![(proact,\ sn)]\!] \\
remPS[\![((sn'\ pL)sL,\ sn)]\!] &= \begin{cases} sL \text{ if } sn = sn' \\ \{(sn'\ pL)\} \cup remPS[\![(sL,\ sn)]\!] \text{ otherwise} \end{cases} \\
remPS[\![(sn'\ pL,\ sn)]\!] &= \begin{cases} \emptyset \text{ if } sn = sn' \\ \{(sn'\ pL)\} \text{ otherwise} \end{cases} \\
remPS[\![(\varepsilon, sn)]\!] &= \emptyset
\end{aligned}
$$

Semantics of $remPP$ (Remove Proactive Policy)

$$
remPP : profile \times \mathrm{S} \times \mathrm{P} \rightarrow profile
$$

$$
\begin{aligned}
remPP_{pl} &: policyList \times \mathrm{P} \rightarrow policyList \\
remPP[\![(react\ proact,\ sn,\ pn)]\!] &= react \cup remPP[\![(proact,\ sn,\ pn)]\!] \\
remPP[\![((sn'\ pL)sL,\ sn,\ pn)]\!] &= \begin{cases} \{(sn'\ remPP_{pl}[\![(pL,\ pn)]\!])\} \cup sL \text{ if } sn = sn' \\ \{(sn'\ pL)\} \cup remPP[\![(sL,\ sn,\ pn)]\!] \text{ otherwise} \end{cases} \\
remPP[\![(sn'\ pL,\ sn,\ pn)]\!] &= \begin{cases} \{(sn'\ remPP_{pl}[\![(pL,\ pn)]\!])\} \text{ if } sn = sn' \\ \{(sn'\ pL)\} \text{ otherwise} \end{cases} \\
remPP_{pl}[\![((pn'\ cL)pL,\ pn)]\!] &= \begin{cases} pL \text{ if } pn = pn' \\ \{(pn'\ cL)\} \cup remPP_{pl}[\![(pL,\ pn)]\!] \text{ otherwise} \end{cases} \\
remPP_{pl}[\![(pn'\ cL,\ pn)]\!] &= \begin{cases} \emptyset \text{ if } pn = pn' \\ \{(pn'\ cL)\} \text{ otherwise} \end{cases} \\
remPP[\![(\varepsilon, sn, pn)]\!] &= \emptyset
\end{aligned}
$$

Semantics of $remPC$ (Remove Proactive Context)

$$
\begin{aligned}
remPC &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \rightarrow profile \\
remPC_{pl} &: policyList \times \mathrm{P} \times \mathbb{N} \rightarrow policyList \\
remPC_{cl} &: contextList \times \mathbb{N} \rightarrow contextList \\
remPC[\![(react\ proact,\ sn,\ pn,\ cid)]\!] &= react \cup remPC[\![(proact,\ sn,\ pn,\ cid)]\!] \\
remPC[\![((sn'\ pL)sL,\ sn,\ pn,\ cid)]\!] &= \begin{cases} \{(sn'\ remPC_{pl}[\![(pL,\ pn,\ cid)]\!])\} \cup sL \text{ if } sn = sn' \\ \{(sn'\ pL)\} \cup remPC[\![(sL,\ sn,\ pn,\ cid)]\!] \text{ otherwise} \end{cases} \\
remPC[\![(sn'\ pL,\ sn,\ pn,\ cid)]\!] &= \begin{cases} \{(sn'\ remPC_{pl}[\![(pL,\ pn,\ cid)]\!])\} \text{ if } sn = sn' \\ \{(sn'\ pL)\} \text{ otherwise} \end{cases} \\
remPC_{pl}[\![((pn'\ cL)pL,\ pn,\ cid)]\!] &= \begin{cases} \{(pn'\ remPC_{cl}[\![(cL,\ cid)]\!])\} \cup pL \text{ if } pn = pn' \\ \{(pn'\ cL)\} \cup remPC_{pl}[\![(pL,\ pn,\ cid)]\!] \text{ otherwise} \end{cases} \\
remPC_{pl}[\![(pn'\ cL,\ pn,\ cid)]\!] &= \begin{cases} \{(pn'\ remPC_{cl}[\![(cL,\ cid)]\!])\} \text{ if } pn = pn' \\ \{(pn'\ cL)\} \text{ otherwise} \end{cases} \\
remPC_{cl}[\![((cid'\ rL)cL,\ cid)]\!] &= \begin{cases} cL \text{ if } cid = cid' \\ \{(cid'\ rL)\} \cup remPC_{cl}[\![(cL,\ cid)]\!] \text{ otherwise} \end{cases} \\
remPC_{cl}[\![(cid'\ rL,\ cid)]\!] &= \begin{cases} \emptyset \text{ if } cid = cid' \\ \{(cid'\ rL)\} \text{ otherwise} \end{cases} \\
remPC[\![(\varepsilon, sn, pn, cid)]\!] &= \emptyset
\end{aligned}
$$

Semantics of $remPR$ (Remove Proactive Resource)

$$
\begin{aligned}
remPR &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow profile \\
remPR_{pl} &: policyList \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow policyList \\
remPR_{cl} &: contextList \times \mathbb{N} \times \mathrm{R} \rightarrow contextList \\
remPR_{rl} &: resourceList \times \mathrm{R} \rightarrow resourceList \\
remPR[\![(react\ proact, sn, pn, cid, rn)]\!] &= react \cup remPR[\![(proact, sn, pn, cid, rn)]\!] \\
remPR[\![((sn'\ pL)sL, sn, pn, cid, rn)]\!] &= \begin{cases} \{(sn'\ remPR_{pl}[\![(pL, pn, cid, rn)]\!])\} \cup sL \\ \quad \text{if } sn = sn' \\ \{(sn'\ pL)\} \cup remPR[\![(sL, sn, pn, cid, rn)]\!] \\ \quad \text{otherwise} \end{cases} \\
remPR[\![(sn'\ pL, sn, pn, cid, rn)]\!] &= \begin{cases} \{(sn'\ remPR_{pl}[\![(pL, pn, cid, rn)]\!])\} \text{ if } sn = sn' \\ \{(sn'\ pL)\} \text{ otherwise} \end{cases} \\
remPR_{pl}[\![((pn'\ cL)pL, pn, cid, rn)]\!] &= \begin{cases} \{(pn'\ remPR_{cl}[\![(cL, cid, rn)]\!])\} \cup pL \\ \quad \text{if } pn = pn' \\ \{(pn'\ cL)\} \cup remPR_{pl}[\![(pL, pn, cid, rn)]\!] \\ \quad \text{otherwise} \end{cases} \\
remPR_{pl}[\![(pn'\ cL, pn, cid, rn)]\!] &= \begin{cases} \{(pn'\ remPR_{cl}[\![(cL, cid, rn)]\!])\} \text{ if } pn = pn' \\ \{(pn'\ cL)\} \text{ otherwise} \end{cases} \\
remPR_{cl}[\![((cid'\ rL)cL, cid, rn)]\!] &= \begin{cases} \{(cid'\ remPR_{rl}[\![(rL, rn)]\!])\} \cup cL \\ \quad \text{if } cid = cid' \\ remPR_{cl}[\![(cL, cid, rn)]\!] \text{ otherwise} \end{cases} \\
remPR_{cl}[\![(cid'\ rL, cid, rn)]\!] &= \begin{cases} \{(cid'\ remPR_{rl}[\![(rL, rn)]\!])\} \text{ if } cid = cid' \\ \{(cid'\ rL)\} \text{ otherwise} \end{cases} \\
remPR_{rl}[\![((rn'\ on\ vL)rL, rn)]\!] &= \begin{cases} rL \text{ if } rn = rn' \\ \{(rn'\ on\ vL)\} \cup remPR_{rl}[\![(rL, rn)]\!] \\ \quad \text{otherwise} \end{cases} \\
remPR_{rl}[\![((rn'\ on\ vL), rn)]\!] &= \begin{cases} \emptyset \text{ if } rn = rn' \\ \{(rn'\ on\ vL)\} \text{ otherwise} \end{cases} \\
remPR[\![(\varepsilon, sn, pn, cid, rn)]\!] &= \emptyset
\end{aligned}
$$

A.2.2 Add

Semantics of $addRP$ (Add Reactive Policy)

$$addRP \ : \ profile \times policy \rightarrow profile$$

$$\begin{aligned}
addRP[\![(react\ proact,\ pn\ cL)]\!] &= addRP[\![(react,\ pn\ cL)]\!] \cup proact \\
addRP[\![((pn'\ cL')pL,\ pn\ cL)]\!] &= \begin{cases} \{(pn'\ cL')\} \cup addRP[\![(pL,\ pn\ cL)]\!] \text{ if } pn \neq pn' \\ (pn'\ cL')pL \text{ otherwise} \end{cases} \\
addRP[\![(pn'\ cL',\ pn\ cL)]\!] &= \begin{cases} \{(pn'\ cL')\} \cup \{(pn\ cL)\} \text{ if } pn \neq pn' \\ \{(pn'\ cL')\} \text{ otherwise} \end{cases} \\
addRP[\![(\varepsilon, pn\ cL)]\!] &= \{(pn\ cL)\}
\end{aligned}$$

Semantics of $addRC$ (Add Reactive Context)

$$\begin{aligned}
addRC &: profile \times \mathrm{P} \times context \to profile \\
addRC_{cl} &: contextList \times context \to contextList \\
addRC[\![(react\ proact,\ pn,\ cid\ rL)]\!] &= addRC[\![(react,\ pn,\ cid\ rL)]\!] \cup proact \\
addRC[\![((pn'\ cL)pL,\ pn,\ cid\ rL)]\!] &= \begin{cases} \{(pn'\ cL)\} \cup addRC[\![(pL,\ pn,\ cid\ rL)]\!] \text{ if } pn \neq pn' \\ \{(pn'\ addRC_{cl}[\![(cL,\ cid\ rL)]\!])\} \cup pL \text{ otherwise} \end{cases} \\
addRC[\![(pn'\ cL,\ pn,\ cid\ rL)]\!] &= \begin{cases} \{(pn'\ cL)\} \text{ if } pn \neq pn' \\ \{(pn'\ addRC_{cl}[\![(cL,\ cid\ rL)]\!])\} \text{ otherwise} \end{cases} \\
addRC_{cl}[\![((cid'\ rL')cL,\ cid\ rL)]\!] &= \begin{cases} \{(cid'\ rL')\} \cup addRC_{cl}[\![(cL,\ cid\ rL)]\!] \text{ if } cid \neq cid' \\ (cid'\ rL')cL \text{ otherwise} \end{cases} \\
addRC_{cl}[\![(cid'\ rL',\ cid\ rL)]\!] &= \begin{cases} \{(cid'\ rL')\} \cup \{(cid\ rL)\} \text{ if } cid \neq cid' \\ \{(cid'\ rL')\} \text{ otherwise} \end{cases} \\
addRC[\![(\varepsilon, pn, cid\ rL)]\!] &= \emptyset
\end{aligned}$$

Semantics of $addRR$ (Add Reactive Resource)

$$\begin{aligned}
addRR &: profile \times \mathrm{P} \times \mathbb{N} \times resource \to profile \\
addRR_{cl} &: contextList \times \mathbb{N} \times resource \to contextList \\
addRR_{rl} &: resourceList \times resource \to resourceList \\
\begin{array}{r} addRR[\![(react\ proact, pn, cid, \\ (rn\ on\ vL))]\!] \end{array} &= addRR[\![(react, pn, cid, (rn\ on\ vL))]\!] \cup proact
\end{aligned}$$

$$
\begin{aligned}
\begin{array}{r} addRR[\![((pn'\ cL)pL, pn, cid,\\ (rn\ on\ vL))]\!]\end{array} &= \begin{cases} \{(pn'\ cL)\} \cup addRR[\![(pL, pn, cid, (rn\ on\ vL))]\!] \\ \quad \text{if } pn \neq pn' \\ \\ \{(pn'\ addRR_{cl}[\![(cL, cid, (rn\ on\ vL))]\!])\} \cup pL \\ \quad \text{otherwise} \end{cases} \\
addRR[\![(pn'\ cL, pn, cid, (rn\ on\ vL))]\!] &= \begin{cases} \{(pn'\ cL)\} \text{ if } pn \neq pn' \\ \\ \{(pn'\ addRR_{cl}[\![(cL, cid, (rn\ on\ vL))]\!])\} \\ \quad \text{otherwise} \end{cases} \\
addRR_{cl}[\![((cid'\ rL)cL, cid, (rn\ on\ vL))]\!] &= \begin{cases} \{(cid'\ rL)\} \cup addRR_{cl}[\![(cL, cid, (rn\ on\ vL))]\!] \\ \quad \text{if } cid \neq cid' \\ \\ \{(cid'\ addRR_{rl}[\![rL, (rn\ on\ vL)]\!])\} \cup cL \\ \quad \text{otherwise} \end{cases} \\
addRR_{cl}[\![(cid'\ rL, cid, (rn\ on\ vL))]\!] &= \begin{cases} \{(cid'\ rL)\} \text{ if } cid \neq cid' \\ \\ \{(cid'\ addRR_{rl}[\![(rL, (rn\ on\ vL))]\!])\} \\ \quad \text{otherwise} \end{cases} \\
addRR_{rl}[\![((rn'\ on'\ vL')rL, (rn\ on\ vL))]\!] &= \begin{cases} \{(rn'\ on'\ vL')\} \cup \\ \quad addRR_{rl}[\![(rL, (rn\ on\ vL))]\!] \\ \qquad \text{if } rn \neq rn' \\ \\ (rn'\ on'\ vL')rL \text{ otherwise} \end{cases} \\
addRR_{rl}[\![((rn'\ on'\ vL'), (rn\ on\ vL))]\!] &= \begin{cases} \{(rn'\ on'\ vL')\} \cup \{(rn\ on\ vL)\} \\ \quad \text{if } rn \neq rn' \\ \\ \{(rn'\ on'\ vL')\} \text{ otherwise} \end{cases} \\
addRR[\![(\varepsilon, pn, cid, (rn\ on\ vL))]\!] &= \emptyset
\end{aligned}
$$

Semantics of $addPS$ (Add Proactive Service)

$$
\begin{aligned}
addPS &: profile \times service \to profile \\
addPS[\![(react\ proact,\ sn\ pL)]\!] &= react \cup addPS[\![(proact,\ sn\ pL)]\!] \\
addPS[\![((sn'\ pL')sL,\ sn\ pL)]\!] &= \begin{cases} \{(sn'\ pL')\} \cup addPS[\![(sL,\ sn\ pL)]\!] \text{ if } sn \neq sn' \\ (sn'\ pL')sL \text{ otherwise} \end{cases} \\
addPS[\![(sn'\ pL',\ sn\ pL)]\!] &= \begin{cases} \{(sn'\ pL')\} \cup \{(sn\ pL)\} \text{ if } sn \neq sn' \\ \{(sn'\ pL')\} \text{ otherwise} \end{cases}
\end{aligned}
$$

$$addPS[\![(\varepsilon, sn\ pL)]\!] = \{(sn\ pL)\}$$

Semantics of $addPP$ (Add Proactive Policy)

$$\begin{aligned}
addPP &: profile \times \mathrm{S} \times policy \rightarrow profile \\
addPP_{pl} &: policyList \times policy \rightarrow policyList \\
addPP[\![(react\ proact,\ sn, pn\ cL)]\!] &= react \cup addPP[\![(proact,\ sn,\ pn\ cL)]\!] \\
addPP[\![((sn'\ pL')sL,\ sn,\ pn\ cL)]\!] &= \begin{cases} \{(sn'\ pL')\} \cup addPP[\![(sL,\ sn,\ pn\ cL)]\!] \text{ if } sn \neq sn' \\ \{(sn'\ addPP_{pl}[\![(pL',\ pn\ cL)]\!])\} \cup sL \text{ otherwise} \end{cases} \\
addPP[\![(sn'\ pL',\ sn,\ pn\ cL)]\!] &= \begin{cases} \{(sn'\ pL')\} \text{ if } sn \neq sn' \\ \{(sn'\ addPP_{pl}[\![(pL',\ pn\ cL)]\!])\} \text{ otherwise} \end{cases} \\
addPP_{pl}[\![((pn'\ cL')pL,\ pn\ cL)]\!] &= \begin{cases} \{(pn'\ cL')\} \cup addPP_{pl}[\![(pL,\ pn\ cL)]\!] \text{ if } pn \neq pn' \\ (pn'\ cL')pL \text{ otherwise} \end{cases} \\
addPP_{pl}[\![(pn'\ cL',\ pn\ cL)]\!] &= \begin{cases} \{(pn'\ cL')\} \cup \{(pn\ cL)\} \text{ if } pn \neq pn' \\ \{(pn'\ cL')\} \text{ otherwise} \end{cases} \\
addPP[\![(\varepsilon, sn, pn\ cL)]\!] &= \emptyset
\end{aligned}$$

Semantics of $addPC$ (Add Proactive Context)

$$\begin{aligned}
addPC &: profile \times \mathrm{S} \times \mathrm{P} \times context \rightarrow profile \\
addPC_{pl} &: policyList \times \mathrm{P} \times context \rightarrow policyList \\
addPC_{cl} &: contextList \times context \rightarrow contextList \\
addPC[\![(react\ proact, sn, pn, cid\ rL)]\!] &= react \cup addPC[\![(proact, sn, pn, cid\ rL)]\!] \\
addPC[\![((sn'\ pL')sL, sn, pn, cid\ rL)]\!] &= \begin{cases} \{(sn'\ pL')\} \cup addPC[\![(sL, sn, pn, cid\ rL)]\!] \\ \quad \text{if } sn \neq sn' \\ \{(sn'\ addPC_{pl}[\![(pL', pn, cid\ rL)]\!])\} \cup sL \\ \quad \text{otherwise} \end{cases} \\
addPC[\![(sn'\ pL', sn, pn, cid\ rL)]\!] &= \begin{cases} \{(sn'\ pL')\} \text{ if } sn \neq sn' \\ \{(sn'\ addPC_{pl}[\![(pL', pn, cid\ rL)]\!])\} \text{ otherwise} \end{cases} \\
addPC_{pl}[\![((pn'\ cL')pL, pn, cid\ rL)]\!] &= \begin{cases} \{(pn'\ cL')\} \cup addPC_{pl}[\![(pL, pn, cid\ rL)]\!] \\ \quad \text{if } pn \neq pn' \\ \{(pn'\ addPC_{cl}[\![(cL', cid\ rL)]\!])\} \cup pL \\ \quad \text{otherwise} \end{cases}
\end{aligned}$$

$$addPC_{pl}[\![(pn'\ cL', pn, cid\ rL)]\!] = \begin{cases} \{(pn'\ cL')\} \text{ if } pn \neq pn' \\ \{(pn'\ addPC_{cl}[\![(cL', cid\ rL)]\!])\} \text{ otherwise} \end{cases}$$

$$addPC_{cl}[\![((cid'\ rL')cL, cid\ rL)]\!] = \begin{cases} \{(cid'\ rL')\} \cup addPC_{cl}[\![(cL, cid\ rL)]\!] \\ \quad \text{if } cid \neq cid' \\ (cid'\ rL')cL \text{ otherwise} \end{cases}$$

$$addPC_{cl}[\![(cid'\ rL', cid\ rL)]\!] = \begin{cases} \{(cid'\ rL')\} \cup \{(cid\ rL)\} \text{ if } cid \neq cid' \\ \{(cid'\ rL')\} \text{ otherwise} \end{cases}$$

$$addPC[\![(\varepsilon, sn, pn, cid\ rL)]\!] = \emptyset$$

Semantics of $addPR$ (Add Proactive Resource)

$$\begin{aligned} addPR &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \times resource \rightarrow profile \\ addPR_{pl} &: policyList \times \mathrm{P} \times \mathbb{N} \times resource \rightarrow policyList \\ addPR_{cl} &: contextList \times \mathbb{N} \times resource \rightarrow contextList \\ addPR_{rl} &: resourceList \times resource \rightarrow resourceList \end{aligned}$$

$$\begin{aligned} addPR[\![(react\ proact, sn, pn, cid,& \\ (rn\ on\ vL))]\!]& = \begin{cases} react \cup \\ addPR[\![(proact, sn, pn, cid, (rn\ on\ vL))]\!] \end{cases} \end{aligned}$$

$$\begin{aligned} addPR[\![((sn'\ pL')sL, sn, pn, cid,& \\ (rn\ on\ vL))]\!]& = \begin{cases} \{(sn'\ pL')\} \cup \\ addPR[\![(sL, sn, pn, cid, (rn\ on\ vL))]\!] \\ \text{if } sn \neq sn' \\ \\ \{(sn'\ addPR_{pl}[\![(pL', pn, cid, (rn\ on\ vL))]\!])\} \\ \quad \cup\ sL \text{ otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} addPR[\![(sn'\ pL', sn, pn, cid,& \\ (rn\ on\ vL))]\!]& = \begin{cases} \{(sn'\ pL')\} \text{ if } sn \neq sn' \\ \\ \{(sn'\ addPR_{pl}[\![(pL', pn, cid, (rn\ on\ vL))]\!])\} \\ \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} addPRpl[\![((pn'\ cL')pL, pn, cid,& \\ (rn\ on\ vL))]\!]& = \begin{cases} \{(pn'\ cL')\} \cup \\ addPR_{pl}[\![(pL, pn, cid, (rn\ on\ vL))]\!] \\ \text{if } pn \neq pn' \\ \\ \{(pn'\ addPR_{cl}[\![(cL', cid, (rn\ on\ vL))]\!])\} \\ \cup pL \text{ otherwise} \end{cases} \end{aligned}$$

$$addPR_{pl}[\![(pn'\ cL', pn, cid, (rn\ on\ vL))]\!] = \begin{cases} \{(pn'\ cL')\} \text{ if } pn \neq pn' \\ \\ \{(pn'\ addPR_{cl}[\![(cL', cid, (rn\ on\ vL))]\!])\} \\ \text{otherwise} \end{cases}$$

$$addPR_{cl}[\![((cid'\ rL')cL, cid, (rn\ on\ vL))]\!] = \begin{cases} \{(cid'\ rL')\}\cup \\ addPR_{cl}[\![(cL, cid, (rn\ on\ vL))]\!] \\ \text{if } cid \neq cid' \\ \\ \{(cid'\ addPR_{rl}[\![(rL', (rn\ oL\ vL))]\!])\} \\ \cup\ cL \text{ otherwise} \end{cases}$$

$$addPR_{cl}[\![(cid'\ rL', cid, (rn\ on\ vL))]\!] = \begin{cases} \{(cid'\ rL')\} \\ \text{if } cid \neq cid' \\ \\ \{(cid'\ addPR_{rl}[\![rL', (rn\ on\ vL)]\!])\} \\ \quad \text{otherwise} \end{cases}$$

$$addPR_{rl}[\![((rn'\ on'\ vL')rL, (rn\ on\ vL))]\!] = \begin{cases} \{(rn'\ on'\ vL')\}\cup \\ \quad addPR_{rl}[\![(rL, (rn\ on\ vL))]\!] \\ \text{if } rn \neq rn' \\ \\ (rn'\ on'\ vL')rL \text{ otherwise} \end{cases}$$

$$addPR_{rl}[\![((rn'\ on'\ vL'), (rn\ on\ vL))]\!] = \begin{cases} \{(rn'\ on'\ vL')\} \cup \{(rn\ on\ vL)\} \\ \quad \text{if } rn \neq rn' \\ \\ \{(rn'\ on'\ vL')\} \text{ otherwise} \end{cases}$$

$$addPR[\![(\varepsilon, sn, pn, cid, (rn\ on\ vL))]\!] = \emptyset$$

A.2.3 Update

Semantics of $updRP$ (Update Reactive Policy)

$$\begin{aligned} updRP &: profile \times \mathrm{P} \times policy \rightarrow profile \\ updRP[\![(react\ proact,\ pn,\ pn\ cL)]\!] &= addRP[\![(remRP[\![(react\ proact,\ pn)]\!], pn\ cL)]\!] \end{aligned}$$

Semantics of $updRC$ (Update Reactive Context)

$$\begin{aligned} updRC &: profile \times \mathrm{P} \times \mathbb{N} \times context \rightarrow profile \\ updRC[\![(react\ proact,\ pn,\ cid,\ cid\ rL)]\!] &= addRC[\![(remRC[\![(react\ proact,\ pn,\ cid)]\!], \\ &\quad pn,\ cid\ rL)]\!] \end{aligned}$$

Semantics of $updRR$ (Update Reactive Resource)

$$\begin{aligned} updRR &: profile \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \times resource \rightarrow profile \\ updRR[\![(react\ proact, pn, cid, rn, (rn\ on\ vL))]\!] &= addRR[\![(remRR[\![(react\ proact, pn, cid, rn)]\!], \\ &\quad pn, cid, (rn\ on\ vL))]\!] \end{aligned}$$

Semantics of $updPS$ (Update Proactive Service)

$$\begin{aligned} updPS &: profile \times \mathrm{S} \times service \rightarrow profile \\ updPS[\![(react\ proact,\ sn,\ sn\ pL)]\!] &= addPS[\![(remPS[\![(react\ proact,\ sn)]\!], sn\ pL)]\!] \end{aligned}$$

Semantics of $updPP$ (Update Proactive Policy)

$$\begin{aligned} updPP &: profile \times \mathrm{S} \times \mathrm{P} \times policy \rightarrow profile \\ updPP[\![(react\ proact,\ sn,\ pn,\ pn\ cL)]\!] &= addPP[\![(remPP[\![(react\ proact,\ sn,\ pn)]\!], \\ &\quad sn,\ pn\ cL)]\!] \end{aligned}$$

Semantics of $updPC$ (Update Proactive Context)

$$\begin{aligned} updPC &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \times context \rightarrow profile \\ updPC[\![(react\ proact, sn, pn, cid, cid\ rL)]\!] &= addPC[\![(remPC[\![(react\ proact, sn, pn, cid)]\!], \\ &\quad sn, pn, cid\ rL)]\!] \end{aligned}$$

Semantics of $updPR$ (Update Proactive Resource)

$$\begin{aligned}
updPR &: profile \times \mathrm{S} \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \times resource \rightarrow profile \\
updPR[\![(react\ proact, sn, pn, cid, rn, & \\
(rn\ on\ vL))]\!] &= addPR[\![(\\
&\quad remPR[\![(react\ proact, sn, pn, cid, rn)]\!], \\
&\quad sn, pn, cid, (rn\ on\ vL))]\!]
\end{aligned}$$

A.3 $policyStatusList$ Adaptation

A.3.1 Remove

Semantics of $remPSP$ (Remove Policy Status Policy)

$$\begin{aligned}
remPSP &: policyStatusList \times \mathrm{P} \rightarrow policyStatusList \\
remPSP[\![((pn\ csL)psL,\ pn')]\!] &= \begin{cases} \{(pn\ csL)\} \cup remPSP[\![(psL,\ pn')]\!] \text{ if } pn \neq pn' \\ psL \text{ otherwise} \end{cases} \\
remPSP[\![(pn\ csL,\ pn')]\!] &= \begin{cases} \{(pn\ csL)\} \text{ if} pn \neq pn' \\ \emptyset \text{ otherwise} \end{cases} \\
remPSP[\![(\varepsilon,\ pn)]\!] &= \emptyset
\end{aligned}$$

Semantics of $remPSC$ (Remove Policy Status Context)

$$\begin{aligned}
remPSC &: policyStatusList \times \mathrm{P} \times \mathbb{N} \rightarrow policyStatusList \\
remPSC_{csl} &: contextStatusList \times \mathbb{N} \rightarrow contextStatusList \\
remPSC[\![((pn\ csL)psL,\ pn',\ cid')]\!] &= \begin{cases} \{(pn\ csL)\} \cup remPSC[\![(psL,\ pn',\ cid')]\!] \\ \quad \text{if } pn \neq pn' \\ \{(pn\ remPSC_{csl}[\![(csL, cid')]\!])\} \cup psL \\ \quad \text{otherwise} \end{cases} \\
remPSC[\![(pn\ csL,\ pn',\ cid')]\!] &= \begin{cases} \{(pn\ csL)\} \text{ if } pn \neq pn' \\ \{(pn\ remPSC_{csl}[\![(csL,\ cid')]\!])\} \text{ otherwise} \end{cases}
\end{aligned}$$

$$remPSC_{csl}[\![((cid\ rsL)csL,\ cid')]\!] = \begin{cases} \{(cid\ rsL)\} \cup remPSC_{csl}[\![(csL,\ cid')]\!] \\ \quad \text{if } cid \neq cid' \\ csL \text{ otherwise} \end{cases}$$

$$remPSC_{csl}[\![(cid\ rsL,\ cid')]\!] = \begin{cases} \{(cid\ rsL)\} \text{ if } cid \neq cid' \\ \emptyset \text{ otherwise} \end{cases}$$

$$remPSC[\![(\varepsilon,\ pn,\ cid)]\!] = \emptyset$$

Semantics of $remPSR$ (Remove Policy Status Resource)

$$\begin{aligned} remPSR &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow policyStatusList \\ remPSR_{csl} &: contextStatusList \times \mathbb{N} \times \mathrm{R} \rightarrow contextStatusList \\ remPSR_{rsl} &: resourceStatusList \times \mathrm{R} \rightarrow resourceStatusList \end{aligned}$$

$$remPSR[\![((pn\ csL)psL, pn', cid', rn')]\!] = \begin{cases} \{(pn\ csL)\} \cup remPSR[\![(psL, pn', cid', rn')]\!] \\ \text{if } pn \neq pn' \\ \\ \{(pn\ remPSR_{csl}[\![(csL, cid', rn')]\!])\} \\ \cup psL \text{ otherwise} \end{cases}$$

$$remPSR[\![(pn\ csL, pn', cid', rn')]\!] = \begin{cases} \{(pn\ csL)\} \text{ if } pn \neq pn' \\ \{(pn\ remPSR_{csl}[\![(csL, cid', rn')]\!])\} \text{ otherwise} \end{cases}$$

$$remPSR_{csl}[\![((cid\ rsL)csL, cid', rn')]\!] = \begin{cases} \{(cid\ rsL)\} \cup remPSR_{csl}[\![(csL, cid', rn')]\!] \\ \text{if } cid \neq cid' \\ \\ \{(cid\ remPSR_{rsl}[\![(rsL, rn')]\!])\} \\ \cup csL \text{ otherwise} \end{cases}$$

$$remPSR_{csl}[\![(cid\ rsL, cid', rn')]\!] = \begin{cases} \{(cid\ rsL)\} \text{ if } cid \neq cid' \\ \{(cid\ remPSR_{rsl}[\![(rsL, rn')]\!])\} \text{ otherwise} \end{cases}$$

$$remPSR_{rsl}[\![((rn', on', vL', b)rsL, rn')]\!] = \begin{cases} \{(rn', on', vL', b)\} \cup remPSR_{rsl}[\![(rsL, rn')]\!] \\ \text{if } rn \neq rn' \\ \\ rsL \text{ otherwise} \end{cases}$$

$$remPSR_{rsl}[\![((rn', on', vL', b), rn')]\!] = \begin{cases} \{(rn', on', vL', b)\} \text{ if } rn \neq rn' \\ \emptyset \text{ otherwise} \end{cases}$$

$$remPSR[\![(\varepsilon,\ pn, cid, rn)]\!] = \emptyset$$

A.3.2 Add

Semantics of $addPSP$ (Add Policy Status Policy)

$$\begin{aligned}
addPSP \ &: \ policyStatusList \times policy \rightarrow policyStatusList \\
addPSP[\![((pn\ csL)psL,\ pn'\ cL)]\!] &= \begin{cases} \{(pn\ csL)\} \cup addPSP[\![(psL,\ pn'\ cL)]\!] \text{ if } pn \neq pn' \\ (pn\ csL)psL \text{ otherwise} \end{cases} \\
addPSP[\![(pn\ csL,\ pn'\ cL)]\!] &= \begin{cases} \{(pn\ csL)\} \cup init[\![pn'\ cL]\!] \text{ if} pn \neq pn' \\ \{(pn\ csL)\} \text{ otherwise} \end{cases} \\
addPSP[\![(\varepsilon, pn\ cL)]\!] &= init[\![pn\ cL]\!]
\end{aligned}$$

Semantics of $addPSC$ (Add Policy Status Context)

$$\begin{aligned}
addPSC \ &: \ policyStatusList \times \mathrm{P} \times context \rightarrow policyStatusList \\
addPSC_{csl} \ &: \ contextStatusList \times context \rightarrow contextStatusList \\
addPSC[\![((pn\ csL)psL, pn', cid'\ rL')]\!] &= \begin{cases} \{(pn\ csL)\} \cup addPSC[\![(psL, pn', cid'\ rL')]\!] \\ \quad \text{if } pn \neq pn' \\ \{(pn\ addPSC_{csl}[\![(csL, cid'\ rL)]\!])\} \cup \ psL \\ \quad \text{if } pn = pn' \end{cases} \\
addPSC[\![(pn\ csL, pn', cid'\ rL')]\!] &= \begin{cases} \{(pn\ csL)\} \text{ if} pn \neq pn' \\ \{(pn\ addPSC_{csl}[\![(csL, cid'\ rL')]\!])\} \text{ otherwise} \end{cases} \\
addPSC_{csl}[\![((cid\ rsL)csL, cid'\ rL')]\!] &= \begin{cases} \{(cid\ rsL)\} \cup addPSC_{csl}[\![(csL, cid'\ rL')]\!] \\ \quad \text{if } cid \neq cid' \\ (cid\ rsL)csL \text{ otherwise} \end{cases} \\
addPSC_{csl}[\![(cid\ rsL, cid'\ rL')]\!] &= \begin{cases} \{(cid\ rsL)\} \cup init_{cl}[\![cid'\ rL']\!] \text{ if } cid \neq cid' \\ \{(cid\ rsL)\} \text{ otherwise} \end{cases} \\
addPSC[\![(\varepsilon, pn, cid\ rL)]\!] &= \emptyset
\end{aligned}$$

Semantics of $addPSR$ (Add Policy Status Resource)

$$\begin{aligned}
addPSR &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times resource \rightarrow policyStatusList \\
addPSR_{csl} &: contextStatusList \times \mathbb{N} \times resource \rightarrow contextStatusList \\
addPSR_{rsl} &: resourceStatusList \times resource \rightarrow resourceStatusList
\end{aligned}$$

$$\begin{aligned}
\begin{array}{r} addPSR[\![((pn\ csL)psL, pn',\\ cid', (rn'\ on'\ vL'))]\!] \end{array} &= \begin{cases} \{(pn\ csL)\} \cup addPSR[\![(psL, pn', cid', (rn'\ on'\ vL'))]\!] \\ \text{if } pn \neq pn' \\ \\ \{(pn\ addPSR_{csl}[\![(csL, cid', (rn'\ on'\ vL'))]\!])\} \cup\ psL \\ \text{if } pn = pn' \end{cases} \\
\begin{array}{r} addPSR[\![(pn\ csL, pn',\\ cid', (rn'\ on'\ vL'))]\!] \end{array} &= \begin{cases} \{(pn\ csL)\} \text{ if } pn \neq pn' \\ \\ \{(pn\ addPSR_{csl}[\![(csL, cid', (rn'\ on'\ vL'))]\!])\} \\ \text{otherwise} \end{cases} \\
\begin{array}{r} addPSR_{csl}[\![((cid\ rsL)csL,\\ cid', (rn'\ on'\ vL'))]\!] \end{array} &= \begin{cases} \{(cid\ rsL)\} \cup addPSR_{csl}[\![(csL, cid', (rn'\ on'\ vL'))]\!] \\ \text{if } cid \neq cid' \\ \\ \{(cid\ addPSR_{rsl}[\![(rsL, (rn'\ on'\ vL'))]\!])\}\ \cup\ csL \\ \text{otherwise} \end{cases} \\
\begin{array}{r} addPSR_{csl}[\![(cid\ rsL,\\ cid', (rn'\ on'\ vL'))]\!] \end{array} &= \begin{cases} \{(cid\ rsL)\} \text{ if } cid \neq cid' \\ \\ \{(cid\ addPSR_{rsl}[\![(rsL, (rn'\ on'\ vL'))]\!])\} \\ \quad \text{otherwise} \end{cases} \\
\begin{array}{r} addPSR_{rsl}[\![((rn, on, vL, b)rsL,\\ (rn'\ on'\ vL'))]\!] \end{array} &= \begin{cases} \{(rn, on, vL, b)\} \cup \\ addPSR_{rsl}[\![(rsL, (rn'\ on'\ vL'))]\!] \\ \text{if } rn \neq rn' \\ \\ (rn, on, vL, b)rsL \text{ otherwise} \end{cases} \\
\begin{array}{r} addPSR_{rsl}[\![((rn, on, vL, b),\\ (rn'\ on'\ vL'))]\!] \end{array} &= \begin{cases} \{(rn, on, v, b)\}\ \cup\ init_{rl}[\![rn'\ on'\ vL']\!] \\ \text{if } rn \neq rn' \\ \\ \{(rn, on, vL, b)\} \text{ otherwise} \end{cases} \\
addPSR[\![(\varepsilon, pn, cid, (rn\ on\ vL))]\!] &= \emptyset
\end{aligned}$$

A.3.3 Update

Semantics of *updPSP* (Update Policy Status Policy)

$$\begin{aligned} updPSP\ &:\ policyStatusList \times \mathrm{P} \times policy \rightarrow policyStatusList \\ updPSP[\![(psL,\ pn,\ pn\ cL)]\!] &= addPSP[\![(remPSP[\![(psL,\ pn)]\!],\ pn\ cL)]\!] \end{aligned}$$

Semantics of $updPSC$ (Update Policy Status Context)

$$\begin{aligned} updPSC &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times context \rightarrow policyStatusList \\ \begin{array}{r} updPSC[\![(psL, pn, cid, \\ cid\ rL)]\!] \end{array} &= addPSC[\![(remPSC[\![(psL,\ pn,\ cid)]\!],\ pn,\ cid\ rL)]\!] \end{aligned}$$

Semantics of $updPSR$ (Update Policy Status Resource)

$$\begin{aligned} updPSR &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \times resource \rightarrow policyStatusList \\ \begin{array}{r} updPSR[\![(psL, pn, cid, \\ (rn\ on\ vL))]\!] \end{array} &= addPSR[\![(remPSR[\![(psL, pn, cid, rn)]\!], pn, cid, (rn\ on\ vL))]\!] \end{aligned}$$

Appendix B

Metadata Encoding

B.1 Application Profile Schema Definition

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

    <xs:element name="APPLICATION_PROFILE">
        <xs:complexType>
            <xs:all>
                <xs:element ref="REACTIVE" minOccurs="1"/>
                <xs:element ref="PROACTIVE" minOccurs="1"/>
            </xs:all>
            <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>


    <xs:element name="REACTIVE">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="POLICY" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="frequency" type="xs:short" use="required"/>
        </xs:complexType>
    </xs:element>


    <xs:element name="PROACTIVE">
        <xs:complexType>
            <xs:sequence>
```

```
                <xs:element ref="SERVICE" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>


    <xs:element name="SERVICE">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="POLICY" minOccurs="1" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>


    <xs:element name="POLICY">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="CONTEXT" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>


    <xs:element name="CONTEXT">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="RESOURCE" minOccurs="1" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:positiveInteger" use="required"/>
        </xs:complexType>
    </xs:element>


    <xs:element name="RESOURCE">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="OPERATOR" minOccurs="1" maxOccurs="1"/>
                <xs:element ref="STATUS" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="OPERATOR">
        <xs:complexType>
            <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>


    <xs:element name="STATUS">
        <xs:complexType>
            <xs:attribute name="value" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

B.2 Utility Function Schema Definition

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

    <xs:element name="UTILITY_FUNCTION">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="ADD" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>


    <xs:element name="ADD">
        <xs:complexType>
            <xs:all>
                <xs:element ref="VALUE_OF" minOccurs="1"/>
                <xs:element ref="MULTIPLY_BY" minOccurs="1"/>
            </xs:all>
        </xs:complexType>
    </xs:element>


    <xs:element name="VALUE_OF">
        <xs:complexType>
            <xs:attribute name="expr" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>


    <xs:element name="MULTIPLY_BY">
        <xs:complexType>
            <xs:attribute name="rate" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:integer">
                        <xs:minInclusive value="0"/>
                        <xs:maxInclusive value="10"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

Bibliography

[Alternis S.A., 2000] Alternis S.A. (2000). Solutions for Location Data Mediation. http://www.alternis.com/.

[Angin et al., 1998] Angin, O., Campbell, A., Kounavis, M., and Liao, R. (1998). The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. *Personal Communications Magazine, Special Issue on Adapting to Network and Client Variability*, pages 32–44.

[ANSA, 1989] ANSA (1989). The Advanced Network Systems Architecture (ANSA). Reference manual, Architecture Project Management, Castle Hill, Cambridge, UK.

[Apparao et al., 1998] Apparao, V., Byrne, S., Champion, M., Isaacs, S., Jacobs, I., Hors, A. L., Nicol, G., Robie, J., Sutor, R., Wilson, C., and Wood, L. (1998). Document Object Model (DOM) Level 1 Specification. W3C Recommendation http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001, World Wide Web Consortium.

[Arnold et al., 1999] Arnold, K., O'Sullivan, B., Scheifler, R. W., Waldo, J., and Wollrath, A. (1999). *The Jini[tm] Specification.* Addison-Wesley.

[Asthana and Krzyzanowski, 1994] Asthana, A. and Krzyzanowski, M. C. P. (1994). An indoor wireless system for personalized shopping assistence. In *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications*, pages 69–74, Santa Cruz, California. IEEE Computer Society Press.

[Baker, 1997] Baker, S. (1997). *Corba Distributed Objects : Using Orbix.* Addison-Wesley.

[Bakker et al., 1999] Bakker, A., van Steen, M., and Tanenbaum, A. (1999). From Remote Objects to Physically Distributed Objects. In *Proc. 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 47–52, Cape Town, South Africa. IEEE Computer Society Press.

[Bennett et al., 1994] Bennett, F., Richardson, T., and Harter, A. (1994). Teleporting - making applications mobile. In *Proc. of the IEEE Workshop on Mobile Computing Systems and Applications*, pages 82–84, Santa Cruz, California. IEEE Computer Society Press.

[Binmore, 1992] Binmore, K. (1992). *Fun and Games: a text on game theory.* Lexington: D.C. Heath.

[Blair et al., 2001] Blair, G., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., and Saikoski, K. (2001). The Design and Implementation of OpenORB V2. *IEEE Distributed Systems Online Journal*, 2(6).

[Blair et al., 1998] Blair, G., Coulson, G., Robin, P., and Papathomas, M. (1998). An Architecture for Next Generation Middleware. In *Proc. of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 191–206, The Lake District, England, UK. Springer Verlag.

[Bray and Sturman, 2000] Bray, J. and Sturman, C. F. (2000). *Bluetooth: Connect Without Cables.* Prentice Hall.

[Bray et al., 1998] Bray, T., Paoli, J., and Sperberg-McQueen, C. M. (1998). Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium.

[Brown, 1998] Brown, P. (1998). Triggering information by context. *Personal Technologies*, 2(1):1–9.

[Campbell, 1997] Campbell, A. (1997). Mobiware: Qos-aware middleware for mobile multimedia communications. In *7^{th} IFIP International Conference on High Performance Networking*, White Plains, NY.

[Capra et al., 2002a] Capra, L., Blair, G. S., Mascolo, C., Emmerich, W., and Grace, P. (2002a). Exploiting Reflection in Mobile Computing Middleware. *ACM Mobile Computing and Communications Review*, 6(4):34–44.

[Capra et al., 2001a] Capra, L., Emmerich, W., and Mascolo, C. (2001a). Middleware for Mobile Computing: Awareness vs. Transparency (Position Summary). In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, page 142, Schloss Elmau, Germany.

[Capra et al., 2001b] Capra, L., Emmerich, W., and Mascolo, C. (2001b). Reflective Middleware Solutions for Context-Aware Applications. In *Proc. of REFLECTION 2001. The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 126–133, Kyoto, Japan.

[Capra et al., 2002b] Capra, L., Emmerich, W., and Mascolo, C. (2002b). A Micro-Economic Approach to Conflict Resolution in Mobile Computing. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 31–40, Charleston, South Carolina, USA. ACM Press.

[Capra et al., 2003] Capra, L., Emmerich, W., and Mascolo, C. (2003). CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering.* To appear in the November issue.

[Capra et al., 2001c] Capra, L., Mascolo, C., Zachariadis, S., and Emmerich, W. (2001c). Towards a Mobile Computing Middleware: a Synergy of Reflection and Mobile Code Techniques. In *In Proc. of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001)*, pages 148–154, Bologna, Italy.

[CellPoint, Inc., 2000] CellPoint, Inc. (2000). The CellPoint System. http://www.cellpt.com/thetechnology2.htm.

[Chalmers and Sloman, 1999a] Chalmers, D. and Sloman, M. (1999a). A Survey of Quality of Service in Mobile Computing Environments. *IEEE Communications Surveys*, 2(2):2–10.

[Chalmers and Sloman, 1999b] Chalmers, D. and Sloman, M. (1999b). QoS and Context Awareness for Mobile Computing. In *Workshop on Handheld Computing in the Field*, volume 1707 of *LNCS*, pages 380–382, Karlsruhe, Germany. Springer Verlag.

[Chalmers et al., 2001] Chalmers, D., Sloman, M., and Dulay, N. (2001). Map Adaptation for Users of Mobile Systems. In *Proceedings of the 10th International World Wide Web Conference (WWW-10)*, pages 735–744, Hong Kong.

[Clark and DeRose, 1999] Clark, J. and DeRose, S. (1999). XML Path Language (XPath). Technical Report http://www.w3.org/TR/xpath, World Wide Web Consortium.

[Coulson et al., 1992] Coulson, G., Blair, G. S., Davies, N., and Williams, N. (1992). Extensions to ANSA for Multimedia Computing. *Computer Networks and ISDN Systems*, 25(3):305–323.

[Cugola and Nitto, 2001] Cugola, G. and Nitto, E. D. (2001). Using a Publish/Subscribe Middleware to Support Mobile Computing. In *Proceedings of the Workshop on Middleware for Mobile Computing*, Heidelberg, Germany. In association with IFIP/ACM Middleware 2001 Conference.

[Dardenne et al., 1993] Dardenne, A., van Lamsweerde, A., and Fickas, S. (1993). Goal-Directed Requirements Acquisition. *Science of Computer Programming*, 20:3–50.

[Davies et al., 1999] Davies, N., Cheverst, K., Mitchell, K., and Friday, A. (1999). Caches in the Air: Disseminating Information in the Guide System. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, pages 11–19, New Orleans, US.

[Davies et al., 1998] Davies, N., Friday, A., Wade, S., and Blair, G. (1998). L2imbo: A Distributed Systems Platform for Mobile Computing. *ACM Mobile Networks and*

Applications (MONET), Special Issue on Protocols and Software Paradigms of Mobile Networks, 3(2):143–156.

[Dey et al., 1999] Dey, A., Futakawa, M., Salber, D., and Abowd, G. (1999). The Conference Assistant: Combining Context-Awareness with Wearable Computing. In *Proc. of the 3^{rd} International Symposium on wearable Computers (ISWC '99)*, pages 21–28, San Franfisco, California. IEEE Computer Society Press.

[Eisenstein et al., 2001] Eisenstein, J., Vanderonckt, J., and Puerta, A. (2001). Applying Model-Based Techniques to the Development of UIs for Mobile Computers. In *International Conference on Intelligent User Interfaces (IUI'01)*, pages 69–76, Santa Fe, New Mexico.

[Eliassen et al., 1999] Eliassen, F., Andersen, A., Blair, G. S., Costa, F., Coulson, G., Goebel, V., Hansen, O., Kristensen, T., Plagemann, T., Rafaelsen, H. O., Saikoski, K. B., and Yu, W. (1999). Next Generation Middleware: Requirements, Architecture and Prototypes. In *Proceedings of the 7^{th} IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 60–65. IEEE Computer Society Press.

[Emmerich, 1996] Emmerich, W. (1996). Tool Specification with GTSL. In *Proc. of the 8^{th} Int. Workshop on Software Specification and Design*, pages 26–35. IEEE Computer Society Press.

[Emmerich, 2000] Emmerich, W. (2000). *Engineering Distributed Objects.* John Wiley & Sons.

[Engels et al., 1992] Engels, G., Lewerentz, C., Nagl, M., Schäfer, W., and Schürr, A. (1992). Building Integrated Software Development Environments — Part 1: Tool Specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167.

[ExoLab, 2001] ExoLab (2001). OpenORB. http://openorb.exolab.org/openorb.html.

[Ferguson et al., 1993] Ferguson, D., Nikolaou, C., and Yemini, Y. (1993). An Economy for Managing Replicated Data in Autonomous Decentralised Systems. In *Proc. of International Symposium on Autonomous and Decentralised Systems*, pages 367–375, Los Alamitos, CA. IEEE Computer Society Press.

[Fickas and Feather, 1995] Fickas, S. and Feather, M. (1995). Requirements Monitoring in Dynamic Environments. In *Proc. of the 2^{nd} IEEE Int. Symposium on Requirements Engineering*, pages 140–147. IEEE Computer Society Press.

[Fritsch et al., 2000] Fritsch, D., Klinec, D., and Volz, S. (2000). NEXUS Positioning and Data Management Concepts for Location Aware Applications. In *Proceedings of the 2^{nd} International Symposium on Telegeoprocessing*, pages 171–184, Nice-Sophia-Antipolis, France.

[Georgiadis et al., 2002] Georgiadis, I., Magee, J., and Kramer, J. (2002). Self-Organising Software Architectures for Distributed Systems. In *Proceedings of the 1st Workshop on Self-healing Systems*, pages 33–38, Charleston, South Carolina.

[Guttman et al., 1999] Guttman, E., Perkins, C., Day, M., and Veizades, J. (1999). Service location protocol, version 2. http://www.ietf.org/rfc/rfc2608.txt. RFC 2608.

[Hall, 1996] Hall, C. (1996). *Building Client/Server Applications Using TUXEDO*. John Wiley & Son.

[Handorean and Roman, 2002] Handorean, R. and Roman, G.-C. (2002). Service Provision in Ad Hoc Networks. In *Coordination 2002*, volume 2315 of *LNCS*, pages 207–219, York, UK. Springer Verlag.

[Hanssen and Eliassen, 1999] Hanssen, Ø. and Eliassen, F. (1999). A Framework for Policy Bindings. In *Proceedings of International Symposium on Distributed Objects and Applications (DOA'99)*, pages 2–11, Edinburgh, Scotland. IEEE Computer Society Press.

[Held, 2000] Held, G. (2000). *Data Over Wireless Networks: Bluetooth, WAP, and Wireless Lans*. McGraw-Hill.

[Hudders, 1994] Hudders, E. (1994). *CICS: A Guide to Internal Structure*. John Wiley & Son.

[Hunter and Nuseibeh, 1998] Hunter, A. and Nuseibeh, B. (1998). Managing Inconsistent Specifications: Reasoning, Analysis, and Action. *ACM Trans. on Software Engineering and Methodology*, 7(4):335–367.

[Julien and Roman, 2002] Julien, C. and Roman, G.-C. (2002). Egocentric Context-Aware Programming in Ad Hoc Mobile Environments. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 21–30, Charleston, South Carolina. ACM Press.

[Kiczales et al., 1991] Kiczales, G., des Rivires, J., and Bobrow, D. (1991). The Art of the Metaobject Protocol. MIT Press.

[kObjects, 2002] kObjects (2002). kXML2. http://kxml.org.

[Kon et al., 2000a] Kon, F., Campbell, R., Mickunas, M., Nahrstedt, K., and Ballesteros, F. (2000a). 2k: A Distributed Operating System for Dynamic Heterogeneous Environments. In *9th IEEE International Symposium on High Performance Distributed Computing*, pages 201–210, Pittsburgh. IEEE Computer Society Press.

[Kon et al., 2000b] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., aes, L. M., and Cambpell, R. (2000b). Monitoring, Security, and Dynamic Configuration with the *dynamicTAO* Reflective ORB. In *International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, pages 121–143, New York. ACM/IFIP.

[Ledoux, 1999] Ledoux, T. (1999). OpenCorba: a Reflective Open Broker. In *Reflection'99*, volume 1616 of *LNCS*, pages 197–214, Saint-Malo, France. Springer.

[Leonhardt and Magee, 1996] Leonhardt, U. and Magee, J. (1996). Towards a General Location Service for Mobile Environments. In *Proceedings of the 3^{rd} IEEE Workshop on Services in Distributed and Networked Environments*, pages 43–50, Macau.

[Long et al., 1996] Long, S., Kooper, R., Abowd, G., and Atkenson, C. (1996). Rapid prototyping of mobile context-aware applications: the Cyberguide case study. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 97–107, White Plains, NY. ACM Press.

[Lupu and Sloman, 1999] Lupu, E. and Sloman, M. (1999). Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering*, 25(6):852–869.

[Maes, 1987] Maes, P. (1987). Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA '87*, pages 147–155, Orlando, Florida. ACM Sigplan Notices.

[Malone et al., 1988] Malone, T. W., Fikes, R. E., Grant, K. R., and Howard, M. T. (1988). Enterprise: A market-like task scheduler for distributed computing environments. In Huberman, B. A., editor, *The Ecology of Computation*, pages 177–205. North-Holland, Amsterdam.

[Mas-Colell et al., 1995] Mas-Colell, A., Whinston, M. D., and Green, J. R. (1995). *Microeconomic Theory*. Oxford University Press.

[Mascolo et al., 2002a] Mascolo, C., Capra, L., and Emmerich, W. (2002a). Middleware for mobile computing (a survey). In Gregori, E., Anastasi, G., and Basagni, S., editors, *Neworking 2002 Tutorial Papers*, volume 2497 of *LNCS*, pages 20–58. Springer.

[Mascolo et al., 2002b] Mascolo, C., Capra, L., Zachariadis, S., and Emmerich, W. (2002b). XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Int. Journal on Personal and Wireless Communications*, 21(1):77–103.

[McAffer, 1996] McAffer, J. (1996). Meta-Level Architecture Support for Distributed Objects. In Kiczales, G., editor, *Reflection 96*, pages 39–62, San Francisco, California. ACM Press.

[Milgrom, 1989] Milgrom, P. (1989). Auctions and Bidding: A Primer. *Journal of Economic Perspectives*, 3(3):3–22.

[Monson-Haefel, 2000] Monson-Haefel, R. (2000). *Enterprise Javabeans*. O'Reilly & Associates.

[Monson-Haefel et al., 2000] Monson-Haefel, R., Chappell, D. A., and Loukides, M. (2000). *Java Message Service*. O'Reilly & Associates.

[Murphy et al., 2001] Murphy, A. L., Picco, G. P., and Roman, G.-C. (2001). LIME: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 524–536, Mesa, AZ. IEEE Computer Society Press.

[Natarajan et al., 2000] Natarajan, V., Reich, S., and Vasudevan, B. (2000). *Programming With Visibroker : A Developer's Guide to Visibroker for Java*. John Wiley & Sons.

[Nentwich et al., 2002] Nentwich, C., Capra, L., Emmerich, W., and Finkelstein, A. (2002). xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology (TOIT)*, 2(2):151–185.

[Networks, 2000] Networks, E. B. R. A. (2000). ETSI HIPERLAN/2 Standard. http://portal.etsi.org/bran/kta/Hiperlan/hiperlan2.asp.

[Okamura et al., 1992] Okamura, H., Ishikawa, Y., and Tokoro, M. (1992). AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework. In *Workshop on New Models for Software Architecture*. ACM Sigplan.

[OMG, 1997] OMG (1997). CORBA Component Model. http://www.omg.org/cgi-bin/doc?orbos/97-06-12.

[Oracle Technology Network, 2000] Oracle Technology Network (2000). Oracle9i Application Server Wireless. http://technet.oracle.com/products/iaswe/content.html.

[Pitt and McNiff, 2001] Pitt, E. and McNiff, K. (2001). *Java.rmi : The Remote Method Invocation Guide*. Addison Wesley.

[Plagemann et al., 1999] Plagemann, T., Eliassen, F., Goebel, V., Kristensen, T., and Rafaelsen, H. (1999). Adaptive QoS Aware Binding of Persistent Objects. In *Proceedings of International Symposium on Distributed Objects and Applications (DOA'99)*, pages 306–317, Edinburgh, Scotland. IEEE Computer Society Press.

[Pope, 1998] Pope, A. (1998). *The Corba Reference Guide : Understanding the Common Object Request Broker Architecture*. Addison-Wesley.

[Psinaptic, 2001] Psinaptic (2001). JMatos. http://www.psinaptic.com/.

[Redbooks, 1999] Redbooks, I. (1999). *MQSeries Version 5.1 Administration and Programming Examples*. IBM Corporation.

[Robinson and Pawlowski, 1999] Robinson, W. N. and Pawlowski, S. D. (1999). Managing requirements inconsistency with development goal monitors. *IEEE Transactions on Software Engineering*, 25(6):816–835.

[Rogerson, 1997] Rogerson, D. (1997). *Inside COM*. Microsoft Press.

[Roman et al., 2002] Roman, G.-C., Julien, C., and Huang, Q. (2002). Network Abstractions for Context Aware Mobile Computing. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 363–373, Orlando, Florida. ACM Press.

[Román et al., 2002] Román, M., Hess, C. K., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K. (2002). Gaia: A Middleware Infrastructure to Enable Active Spaces. *IEEE Pervasive Computing*, pages 74–83.

[Román et al., 2001] Román, M., Kon, F., and Campbell, R. (2001). Reflective Middleware: From your Desk to your Hand. *IEEE Distributed Systems Online Journal*, 2(5).

[Román et al., 2000] Román, M., Mickunas, D., Kon, F., and Campbell, R. H. (2000). LegORB and Ubiquitous CORBA. In *IFIP/ACM Middleware 2000 - Workshop on Reflective Middleware*, IBM Palisades Executive Conference Center, NY.

[Rumbaugh et al., 1998] Rumbaugh, J., Jacobson, I., and Booch, G. (1998). *The Unified Modeling Language Reference Manual*. Addison Wesley.

[Sairamesh et al., 1995] Sairamesh, J., Ferguson, D., and Yemini, Y. (1995). An Approach to Pricing, Optimal Allocation and Quality of Service Provisioning in High-speed Packet Networks. In *Proc. of Conference on Computer Communications*, Boston, Massachusetts.

[Salber et al., 1999] Salber, D., Dey, A. K., and Abowd, G. D. (1999). The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*, pages 434–441.

[Salutation Consortium, 1999] Salutation Consortium (1999). Salutation. http://www.salutation.org/.

[Satyanarayanan, 1996] Satyanarayanan, M. (1996). Mobile Information Access. *IEEE Personal Communications*, 3(1):26–33.

[Satyanarayanan et al., 1990] Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. (1990). Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459.

[Schilit et al., 1994] Schilit, B., Adams, N., and Want, R. (1994). Context-Aware Computing Applications. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 85–90, Santa Cruz, CA. IEEE Computer Society Press.

[Schmidt et al., 1999] Schmidt, A., Beigl, M., and Gellersen, H.-W. (1999). There is more to Context than Location. *Computers and Graphics*, 23(6):893–901.

[Segall and Arnold, 1997] Segall, W. and Arnold, D. (1997). Elvin Has Left the Building: A Publish/Subscribe Notification Service with Quenching. In *Australian UNIX Users Group 97*, pages 373–380, Brisbane, Australia. ACM Press.

[SignalSoft, 2000] SignalSoft (2000). Wireless Location services. http://www.signalsoftcorp.com/.

[Smith, 1982] Smith, B. (1982). Reflection and Semantics in a Procedural Programming Language. Phd thesis, MIT.

[Softwired, 2002] Softwired (2002). iBus Mobile. http://www.softwired-inc.com/products/mobile/mobile.html.

[Sun Microsystem, 1997] Sun Microsystem, I. (1997). PersonalJava^{TM} Application Environment. http://java.sun.com/products/personaljava/.

[Sun Microsystem, 2000] Sun Microsystem, I. (2000). Java 2 Platform, Micro Edition. http://java.sun.com/j2me/.

[Terry et al., 1995] Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., and Hauser, C. (1995). Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, pages 172–183, Cooper Mountain, Colorado. ACM Press.

[UPnP Forum, 1998] UPnP Forum (1998). Universal Plug and Play. http://www.upnp.org/.

[van Lamsweerde et al., 1998] van Lamsweerde, A., Darimont, R., and Letier, E. (1998). Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926.

[van Steen et al., 1999] van Steen, M., Hauck, F., Homburg, P., and Tanenbaum, A. (1999). Globe: a Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78.

[Venkatasubramanian et al., 2001] Venkatasubramanian, N., Deshpande, M., Mahopatra, S., Gutierrez-Nolasco, S., and Wickramasuriya, J. (2001). Design and implementation of a Composable Reflective Middleware Framework. In *Proceedings of the IEEE International Conference on Distributed Computer Systems*, pages 644–653, Mesa, AZ. IEEE Computer Society Press.

[Venkatasubramanian and Talcott, 1995] Venkatasubramanian, N. and Talcott, C. (1995). Meta-architectures for resource management in open distributed systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 144–153, Ottawa, Ontario, Canada. ACM Press.

[Vickrey, 1961] Vickrey, W. (1961). Counterspeculation, auctions and competitive sealed tenders. *Journal of Finance*, 16(1):8–37.

[Waldo, 1998] Waldo, J. (1998). Javaspaces specification 1.0. Technical report, Sun Microsystems.

[Watanabe and Yonezawa, 1988] Watanabe, T. and Yonezawa, A. (1988). Reflection in an Object-Oriented Concurrent Language. In *OOPSLA 88*, volume 23, pages 306–415. ACM Press.

[Weicker, 1984] Weicker, R. P. (1984). Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030.

[Welling and Badrinath, 1998] Welling, G. and Badrinath, B. (1998). An Architecture for Exporting Environment Awareness to Mobile Computing. *IEEE Transactions on Software Engineering*, 24(5):391–400.

[Wyckoff et al., 1998] Wyckoff, P., McLaughry, S. W., Lehman, T. J., and Ford, D. A. (1998). T Spaces. *IBM Systems Journal*, 37(3):454–474.

[Yokote, 1992] Yokote, Y. (1992). The Apertos reflective operating system: The concept and its implementation. In *Proceedings of OOPSLA'92*, pages 414–434, Vancouver, British Columbia, Canada. ACM Press.

[Zachariadis et al., 2002] Zachariadis, S., Mascolo, C., and Emmerich, W. (2002). Exploiting Logical Mobility in Mobile Computing Middleware. In *Proceedings of IEEE Workshop on Mobile Team Work. Co-located with ICDCS02*, pages 385–386, Vienna, Austria.

[Zlotkin and Rosenschein, 1993] Zlotkin, G. and Rosenschein, J. S. (1993). A Domain Theory for Task Oriented negotiation. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 416–422, Chambery, France.

[Zlotkin and Rosenschein, 1996] Zlotkin, G. and Rosenschein, J. S. (1996). Mechanisms for Automated Negotiation in State Oriented Domains. *Journal of Artificial Intelligence Research*, 5:163–238.