Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
and
Universidad de Chile - Chile
2002

Reflection for Adaptable Mobile Code in Ubiquitous Computing

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Michaël Vernaillen

Advisor: Prof. Theo D'Hondt (VUB)
Co-promotors: Dr. José Piquer (Universidad de Chile)

Contents

1 Introduction **7**
- 1.1 Introduction . . . 7
- 1.2 Goals . . . 8
- 1.3 Note to the Reader . . . 8

2 Concepts **10**
- 2.1 Mobile Code . . . 10
  - 2.1.1 Mobility Mechanisms . . . 10
  - 2.1.2 Design Paradigms . . . 13
  - 2.1.3 Mobile Code Application Domains . . . 14
- 2.2 Reflection and Meta-programming . . . 16
  - 2.2.1 Principle . . . 16
- 2.3 Reflex . . . 17
  - 2.3.1 Overview . . . 17
  - 2.3.2 Architecture . . . 18
- 2.4 Reflex and Reference Management Policies . . . 18
  - 2.4.1 The Network Reference Policy . . . 18
  - 2.4.2 The Rebinding Policy . . . 19
- 2.5 Reflection and Dynamic Adaptability for Mobile Code . . . 19
- 2.6 Summary . . . 20

3 Mobile Environments and Adaptation **24**
- 3.1 Problems of Mobile Environments . . . 24
  - 3.1.1 Wireless Communication . . . 24
  - 3.1.2 Mobility . . . 25
  - 3.1.3 Portability . . . 25
- 3.2 The Need for Adaptability . . . 26
  - 3.2.1 Taxonomy of Adaptation Strategies . . . 26
  - 3.2.2 Overview of Systems that Offer Adaptability . . . 27
  - 3.2.3 Adaptation Mechanisms . . . 30
- 3.3 Summary . . . 31

4 Dynamic Adaptation of Migration Policies 32
4.1 Design . . . 32
4.1.1 Infrastructure Monitor . . . 33
4.1.2 Application Structure . . . 33
4.1.3 Metalevel Reasoning . . . 33
4.1.4 Specification of the MAC . . . 35
4.1.5 Running a Reflective Program . . . 36
4.2 Ubiquitous Computing . . . 36
4.2.1 Move by Copy . . . 36
4.2.2 Move by Copy with Synchronization . . . 37
4.2.3 Remote Reference . . . 40
4.3 Roaming Agents . . . 42
4.3.1 Switching from Move by Copy to Move by Copy . . . 43
4.3.2 Switching from Copy to Remote Reference . . . 43
4.3.3 Switching from Remote Reference to Remote Reference 43
4.3.4 Switching from Remote Reference to Move by Copy . 44
4.3.5 Multiple Agents Sharing the Same Resource and Garbage Collection . . . 45
4.3.6 A Concrete Example . . . 45
4.4 Summary . . . 46

5 A PIM Application 54
5.1 The Agenda Application . . . 54
5.1.1 Design . . . 54
5.2 Adaptation Scenarios . . . 55
5.3 Adaptable Types and Migration Policies . . . 56
5.4 Implementation of the Metalevel . . . 58
5.4.1 Migration Adaptors . . . 58
5.5 Network Disconnects and Synchronization . . . 60
5.6 The XML Specification . . . 63
5.7 Deployment . . . 63
5.8 Summary . . . 64

6 Conclusions 65
6.1 Achievements . . . 65
6.2 Limitations and Future Work . . . 66
6.3 Conclusion . . . 67

List of Figures

2.1 Computational Environment. 11
2.2 Data space management mechanisms. 21
2.3 The remote reference migration policy. 22
2.4 The rebinding migration policy. 23

3.1 Range of adaptation strategies. 26
3.2 Odyssey Client Architecture. 28

4.1 Design of the metalevel.
At run time, migration adaptors use information from the infrastructure monitor and other sources of information, and possibly instance filters are applied to determine whether and how a particular object should be migrated. 34
4.2 Class diagram of the metalevel entities. 35
4.3 Application structuring and metalevel reasoning.
XML is used to structure the application and to specify the migration adaptor for adaptable types. 35
4.4 Move by copy 37
4.5 Principle of the CopyWithSyncWrapper 39
4.6 Extension to our XML specification to support the synchronization protocol 40
4.7 Classes to implement the synchronization protocol. 41
4.8 Principle of the RemoteReferenceWrapper 41
4.9 Move by remote reference 42
4.10 Switching from move by copy to move by copy 44
4.11 Switching from copy to remote reference 49
4.12 Switching from remote reference to remote reference 50
4.13 Switching from remote reference to copy 51
4.14 Multiple agents sharing the same resource 52
4.15 Example of a roaming agent moving between 3 places, metalevel design. 53

5.1 UML class diagram of the agenda application 55
5.2 The three adaptation scenarios. 56

5.3 The adaptation strategy for fixed computers.
All the adaptable types are referenced remotely. 57
5.4 The adaptation strategy for small portable devices.
The agenda and the 5 relevant days are transported using the *copy with synchronization* policy. All the other days are referred remotely 57
5.5 The adaptation strategy for big portable computers.
The agenda and all the days are transported using *copy with synchronization*. 58
5.6 The Connector class to detect a disconnection. 61
5.7 Implementation of the disconnect operation. 62
5.8 The XML specification for specifying the MAC and *synchronizers* for the types in the agenda application. 63

Acknowledgments

First of all I would like to thank my co-promoter Eric Tanter for supporting me during my thesis period, for giving me some good ideas, and for answering all my questions. He was always prepared to help me even while he had a lot of work to do himself.

I would like to thank the people at AccessNova (the laboratory in the university where I worked) for giving me a comfortable working environment and helping me with my computer problem in the beginning of my thesis period.

I would also like to thank the other students of the EMOOSE program for the nice time we had together in Nantes. Although we had a lot work there, we also had a lot of fun, which was important to find the courage to continue the hard work. A special thank goes to Boris Mejias, for letting me borrow his appartement in Santiago. This way I didn't have to search for an appartement when I arrived, which probably saved me a lot of troubles and also time.

Finally, a big thank goes to all the organizers of this EMOOSE program, Annya Romanczuck, Jacques Noyé and Theo D'Hondt who made this wonderful program possible. I hope they will find the courage to continue there work so that many other generations of emoosers can follow.

Abstract

In perspective of ubiquitous computing and mobile computing, migrating entities need to be able to adapt to the state of the environment (network bandwidth, reliability, target host characteristics, etc.). In this work we show how the way migrating objects are referenced (reference, copy, etc.) can be adapted dynamically to the state of the environment.

To achieve this kind of adaptation, we use the reflective framework Reflex. Using Reflex, we can attach metaobjects to normal Java objects. Such a metaobject can trap the serialization process of its base object when it is about to migrate, and give control to other metalevel entities which then decide what migration policy (copy, reference, etc.) to use, by accessing different sources of information.

In this work, we design a metalevel library using this approach. We show how this metalevel library can transparently be plugged onto any application to dynamically adapt the way migrating parts of this application are referenced. Furthermore we design a reusable infrastructure to allow synchronization between parts of the application that migrate using the copy policy, and finally we extend this infrastructure to support the disconnect operation.

keywords: adaptation, adaptability, migration policy, reference management, mobile code, mobile agent, reflection, Reflex, Java, Ubiquitous Computing

Chapter 1

Introduction

1.1 Introduction

Network and computer technologies are evolving at high speed today. Portable devices such as PDA's, laptops, cellular phones, are being equipped with wireless interfaces, allowing networked communication, even when mobile.

Mobile computing constitutes a new paradigm of computing that is expected to revolutionize the way computers are used. Wireless networking greatly enhances the utility of carrying a computing device. It provides mobile users with versatile communications to other people and notification of important events. It also permits continuous access to other services and even resources of the land-based network. The combination of networking and mobility will gender new applications and services.

However these new technologies seem to be very promising, the technical challenges of this computing paradigm are non trivial. Wireless communications are characterized by low-bandwidth, disconnects and highly variable network conditions. Mobile devices are limited by resources (small batteries, not so powerful CPU's ...). Moreover these devices have very disparate characteristics, they interact through heterogeneous communication channels, and operated in non-static network topologies.

As a consequence of these constraints, the mechanism for mobile data access has to be adaptive, dynamically conforming the limitations of individual clients and their current environments. Software should be adaptable to react on changes in the environment, to make optimally use of the available resources.

Reflection [20] seems to be very attractive to use in the area of mobile computing, to adapt programs dynamically to the changes in the environment.

1.2 Goals

The main objective of this thesis is to achieve dynamic adaptability in mobile programs. To this end we use Reflex [33], a reflective system for Java [26] and written in Java. The Java programming language seems to be the best candidate for the heterogeneous world, as applications are now developed not knowing in advance on which operating system they will run.

In previous work [34], Reflex has been successfully used in order to specify the way objects referenced by a migrating object should be managed (reference, rebinding, etc.). It was made possible to statically specify which policy to use, whatever the type of the objects, achieving a nice degree of separation of concerns.

In this thesis we want to extend this system to make it able to switch dynamically between those migration policies, based on information gathered from the environment, and other sources of information. The idea is to create a metalevel architecture that reasons about upcoming migrations, using information from the environment (network characteristics, device resources, ...). This metalevel library can then be transparently plugged over any application to adapt the migration policies to information gathered from the different sources.

As an example, we want to develop a PIM[1] application, such as an agenda. The user of this kind of application typically owns a workstation at the office, a laptop at home, and probably a PDA and a cellular phone. To avoid replication of code and data, we can design a centralized solution, in which parts of the agenda application can be *migrated* from a main server to any of these devices.

When an application (or a collection of objects) migrates from one executing environment to another, some parts of the object graph could be left on the source environment and then be accessed remotely, depending on some parameters. Or on the contrary, it could be desirable that migration took more time, but then avoid remote access.

For the agenda application, when migrating from the application from the server to a PDA, such a reasoning could result in deciding to transfer for example only the data for the current day, instead of transferring everything.

1.3 Note to the Reader

This thesis is structured as following:

In chapter 1 we give a short introduction to our work, and highlight the main goals.

In chapter 2 we introduce the necessary concepts used in this thesis: mobile code, reflection, meta-programming and Reflex.

[1] Personal Information Management

Chapter 3 gives an overview of the problems current mobile environments are faced with, and we see how adaptation mechanisms can help to overcome those problems. We also describe some systems that offer adaptability.

Chapter 4 focuses on the dynamically switching between migration policies. We introduce the different metalevel entities needed to achieve the switching between the *remote reference* and the *move by copy* migration policies. Then we see how these metalevel entities are used in the world of *ubiquitous computing* and *roaming agents*.

In chapter 5 we apply this metalevel model to a concrete example. We start form a simple agenda application, and we see the different steps the application programmer has to follow to apply our library for dynamically switching between migration policies onto a base application.

In chapter 6 finally, we list all the achievements made in this work, we expose the limitations, and see how this work can be extended in the future.

Chapter 2

Concepts

In this chapter we introduce all the concepts that are used in this thesis. We start by defining the term mobile code, and we introduce the main concepts and technologies used in this area. Then we give an overview of some application domains where this technology can be used, and we highlight its main advantages. Next we give an introduction to reflection and metaprogramming, and we focus in particular on the Reflex [33] framework, a reflective framework for Java. Finally we expose the relation between reflection and the motivation of this thesis.

2.1 Mobile Code

Code mobility is not a new concept, it appears in many shapes. In the past many systems have been designed to move code among the different nodes of a network. Examples are Postscript [31] documents that are executed on the printer, Java applets used for animated and interactive web-pages.

In traditional distributed systems, processes reside in the same computational environment during their entire lifetime. Distributed systems with the mobile agent paradigm allow migration of *executing units* [1], an executing unit can be transferred to another computational environment, and resume its execution in the remote environment.

The most important aspects in the area of code mobility are: *mobility mechanisms*, *design paradigms*, and *application domains*. Note that the definitions given in this section are based on [14].

2.1.1 Mobility Mechanisms

Code and Execution State Mobility Executing units consist of a *code segment*, and a *state* composed of a *data space* and an *executing state*. The

[1] Executing units represent sequential flows of computation. Examples are single-threaded processes or an individual thread of multi-threaded process

data space is a set of references to resources that can be accessed. The execution state contains contains private data and control information related to the state of the executing unit, such as the call stack and the instruction pointer (see figure 2.1).

Current Mobile Code Systems offer two kinds of mobility, characterized by the executing unit that can be migrated:

- *strong mobility*: The ability of a mobile code system to allow migration of code **and** the execution state of an executing unit to a different computational environment.
- *weak mobility*: The ability of a mobile code system to allow code transfer across different computational environments. Code may be accompanied by some initialisation data, but no migration of execution state is involved.

Strong mobility can be realized by *migration* or *remote cloning*. For the migration mechanism, the execution of the executing unit is suspended, then the executing unit is transmitted to the destination computational environment, and finally resumed. The remote cloning mechanism creates a copy of an executing unit at a remote computational environment. The main difference between the two mechanisms is that in remote cloning, the original executing unit is not detached from its current computational environment.

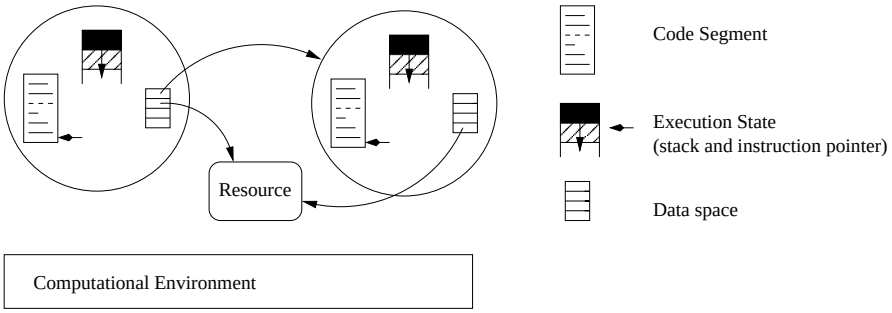

Figure 2.1: Computational Environment.

Data Space Management The executing unit references resources that are managed by the local computational environment. Upon migration of the executing unit, those references must be rearranged in such a way that the moved code has valid references in the new computational environment. The type of the resource determines the possible data *space management mechanisms*[2]. We can distinguish the following types of resources:

[2]We also refer to this mechanisms further in this work, as *migration policies* or *reference management policies*

- **Transferable resources** Transferable resources can be transported over the network. Moreover, transferable resources can be *free* or *fixed*. Fixed transferable resources could be transported over the network, but it is not the case. For example, it would be undesirable to migrate a huge database for performance reasons.

- **Non transferable resources** Resources of this type must stay in the local computational environment (e.g. a printer is not transferable).

The binding from a resource to an executing unit can be done in three forms according to [14]:

1. By *identifier*: This is the strongest form of binding. In this case, the executing unit must be bound at any time to an uniquely defined resource. This form of binding can be used for resources which cannot be substituted by some other equivalent resource. This means that the executing unit is interested in the *identity* of the resource.

2. By *value*: In this form of binding, the resource must be compliant with a given type at any moment, and its value can not change as a consequence of migration. This means that the executing unit is interested in the *content* of the resource.

3. By *type*: This is the weakest form of binding. In this case the executing unit needs, at any time, a resource with a specific type, no matter what its actual value or identity are. This kind of binding is useful to bind resources that are available on every computational environment, like libraries, or network devices.

Figure 2.2 on page 21 shows the possible data space management mechanisms. The executing unit that migrates is colored in grey. The figure shows for each data space management mechanism the source and the destination host before and after migration of the executing unit. We can distinguish the following types of data space management mechanisms:

- **move** : If the resource is a free transferable unit, then the resource can be transferred along with the executing unit to the destination computational environment. Moving away the resource can cause problems when other executing units refer to this resource. There are two solutions to this problem: either a network reference is used to bind the unit to the moved resource (see figure 2.2 (a) bottom), or the other bindings to the resource are removed (see figure 2.2 (a) top).

- **network reference** : If the resource is non transferable, a *remote reference* (see figure 2.2(b)) mechanism should be used. In this case the resource is not transferred, but when the unit has reached the destination executing environment, the binding is changed to reference the resource in the source computational environment.

- **copy** : In this case, a copy of the resource is transferred to the destination environment along with the executing unit (see figure 2.2 (d)). This type of data space mechanism is most usefull if the resource is bound by value, as the identity of the resource is not relevant.

- **rebinding** : In this case the binding is voided and reestablished to another resource on the target computational environment, after migration of the executing unit. A requirement for this kind of binding is that at the destination computational environment, there must be a resource of the same type then at the source computational environment. This is the most convenient mechanism for binding by type. (see figure 2.2 (d))

The data space mechanism that can be used for migration depends on the type of the resource, and the form of binding between the computational environment and the resource. Current Mobile Code Systems are limited in choosing between the different data space management mechanisms. The type of the resources, and the form of binding is implementation or system specific. We come back to this issue later, as this is the starting point for the motivation of this work.

2.1.2 Design Paradigms

Software architectures with similar characteristics can be represented by *design paradigms*, which define architectural abstractions and reference structures that may be instantiated into actual software architectures. To identify the design paradigms for distributed systems exploiting code mobility, we introduce the following *architectural components*:

- *Components*: The different parts that make up the software architecture. They can be further divided into *code components*, that encapsulate the know-how to perform a particular computation, *resource components* that represent data or devices used during computation, and *computational components*, that are active executors capable to carry out a computation.

- *Interactions*: Events that involve two or more components (e.g. exchange of messages between two components)

- *Sites*: Host for the components and support the execution of computational components

Client-Server In this paradigm, a computational component B offering a set of services is placed at site Sb. Resources and know-how needed for service execution are placed at site Sb as well. The client component A, located at site Sa, requests the execution of a service with an interaction with the server component

B. As a response B performs the requested service by executing the corresponding know-how and accessing the involved resources co-located with B.

Remote Evaluation In this paradigm, a computational component A, has the know-how to execute a task, but lacks the resources required, which are located on a different site. Thus the component A sends the know-how to the component B, which resides on the same site as the resource needed to complete the task. B then executes the code using the available resources, and sends the result back to A.

Code on Demand In this paradigm, component A, residing on site Sa, has the needed resources available, but the know-how to manipulate those resources is not available on Sa. Consequently, A interacts with a component B on another site Sb, by requesting the know-how that is located at Sb. B then delivers the know-how to A, that can subsequently executes it.

Mobile Agents The mobile agent paradigm is similar to the Remote Evaluation paradigm. A computational component A on site Sa, has the know-how to execute a task, but (some of) the resources are located at another site Sb. Thus, A *migrates* to Sb carrying the know-how and possibly some intermediate results. After A arrived at Sb, A can execute the task by using the needed resources available on Sb. The difference with the remote evaluation paradigm is that in the mobile agent paradigm, A does not presume a computational component B residing on the same site as the resources for executing the know-how. This paradigm involves the mobility of an *existing* component, while the Code on Demand and Remote Evaluation paradigms focus on the transfer between components.

2.1.3 Mobile Code Application Domains

Today applications that exploit code mobility are still very few, mainly because of the immaturity of the technology (mostly concerning security and performance). However the advantages of code mobility seem to be very appealing for some specific application domains. In this section we provide the user with some key benefits of mobile code, and an overview of mobile code applications suitable for exploiting those benefits.

Key Benefits of Mobile Code

Service Customization Conventional Client-Sever based distributed system provide a fixed set of services through a statically defined interface. It often occurs that the services offered do not satisfy the needs for unforeseen clients. In this case the server has to be upgraded, which increases its complexity without increasing its flexibility. The ability to execute code remotely increases the flexibility of the server, without affecting its size or complexity. In this case, the server provides very simple and low-level services that can be composed by the client to obtain customisable high-level functionality.

Deployment and Maintenance Mobile code supports the lasts phases of the development process. In current distributed settings the act of installing or rebuilding an application has to be performed locally and with human intervention.

Mobile code can be used to roam over the set of hosts in the network, analyse the local platform and perform the correct installation steps.

Autonomy Mobile code concepts and technology embody a notion of *autonomy*, which is a useful property for applications that use a heterogeneous communication infrastructures with different performance of the physical links (low-reliable, low-bandwidth networks). It is important to cope with those differences at design stage. In the Client-Server paradigm, a set of low level operations could be grouped into one client-server interaction. This service could execute *autonomously* and *independently* on the server without the need to connect to the node that send it, except for sending back the final result.

Fault Tolerance Autonomy of application programs has fault tolerance as a side-effect. In conventional client-server programs, the interleaving of client statements with server statements can lead to partial failures. The action from *migrating* code, and possibly sending back the results, is not immune from this problem. However, the action of executing code that contains a set of interactions that should otherwise take place across the network is immune from the partial problem.

Application Domains for Mobile Code

Distributed Information Retrieval This type of applications collect information matching some specified criteria from a set of information resources spread over the network. This is a wide application domain, for example the information to be retrieved might range from the list of all the publications of a given author to software configuration of hosts on a network. Mobile code could improve the efficiency, when the code that performs the information retrieval is migrated close to the information source.

Active Documents In this kind of applications, passive documents (like e-mail or web pages) are enhanced with the capability to executing programs that are somehow related with the document contents, enabling better presentation and interaction. Code mobility is fundamental for these applications since it enables the embedding of code and state into documents and supports the execution of the dynamic contents during document creation.

Remote Device Control and Configuration Those applications are used to control and configure a set of network devices and monitor their status. In the classical approach, monitoring is achieved by polling periodically the resource state. Configuration is performed using a set of services. This approach, based on the Client-Server paradigm, can lead to a number of problems. Code mobility could be used to design and implement monitoring components that are co-located with the devices being monitored and report events that represent the evolution of the device state. In addition, the shipment of management components to remote sites could improve both performance and flexibility [14].

Electronic Commerce Electronic commerce application allow users to perform business transactions through the network. The application environment is

composed of several independent and possibly competing business entities. A transaction may involve negotiation with remote entities and may require access to information that is continuously evolving. There is need to customize the behavior of the parties involved in order to match a particular negotiation protocol. It is also desirable to move application components close to the information relevant to the transaction.

2.2 Reflection and Meta-programming

Reflection is not a recent concept in computer science. It dates from Brian Smith's [25] work in the early 80s. It is a wide-ranging concept that has long been studied in philosophy and many different areas of science. It was introduced in computer science through artificial intelligence as it was considered as a property responsible, at least in part, for what is considered an *intelligent behavior*. But it has also been applied in the area of programming languages under the name of *computational reflection*. Patty Maes [20, 21] applied the idea of reflection to Object Oriented Programming [10]. The purpose of this section is to introduce the reader to the main concepts of reflection in object oriented languages and meta-programming.

2.2.1 Principle

Reflection can be defined in general as [25]:

> An entity's integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on and deals with its primary subject matter.

Bobrow et al.[4] noticed that there are two aspects of reflection, *introspection* and *intercession*. *Introspection* is the ability of a program to observe and therefore reason about its own state. *Intercession* is the ability for a program to modify its own interpretation and meaning.

A reflective system is generally composed of different levels. The first level, called *base level*, prescribes the tasks that must be carried out by the system. The second level, called *metalevel*, interprets the base level and describes how the previous tasks must be performed. Recursively, each metalevel i represents the base level for metalevel $i + 1$. This stack of interpreters is often referred to as a *reflective tower* [25]. The link between the base level and the metalevel is called the *metalink*[3]. *Meta-programming* is the activity of programming metalevel entities.

Each reflective computation can be separated in two logical aspects: computational flow context switching and meta-behavior. A computation starts with the computational flow in the base level; when the base entity begins action, such an action is trapped by the meta-entity and the flow raises at metalevel (*shift-up* action). Then the meta-entity completes its meta-computation, and when it allows the base-entity to perform the action, the computational flow goes back to the base level (*shift-down* action)

Two different kinds of reflection exist: *structural* and *behavioral* reflection [9]. The difference is that [11]:

[3]also known in the literature as *causal connection link*

Structural reflection requires the reification of entities used for building the system statically. Behavioral reflection goes further since it requires to reify entities used to perform the computation of the system (e.g. the execution stack). Thus these entities belong to the dynamic part of the system.

Reflective systems differ in the type of reflection they provide, as well as in the nature of the meta-entities. There are four recognized reflective models in this regard [6]:

the metaclass model (MCM). In this model, the reflective tower is realized by the instantiation link [8, 5]. The metaobject reifying a base-entity is its class, the metaobject reifying a metaobject is its metaclass, and so on. This main problem of this model is the difficulty of specializing the meta-behavior for a single instance, since any instance of a class has the same metaobject. Also, dynamically changing the behavior of an object implies substituting its metaclass, which can lead to inconsistencies and is not offered by all languages.

the metaobject model (MOM). In this model, the reflective tower is realized by the clientship relation [17]: separate entities handle intercession and introspection on each base-entity. With this approach it is simple to specialize the meta-behavior per object. The major drawback of the model is that a metaobject does not have access to the sender's identity when monitoring a message. It is the most used model, with applications in several areas.

the message-reification Model (MRM). In this model, meta-entities are special objects called *messages* which reify the actions that should be performed by the base-entities [12]. Every method call is reified into an object which is charged with its own management and exists only for the duration of the action it embodies. The major drawback of this model is the lack of information continuity, since it is impossible to store information among meta-computations.

the channel reification model (CRM). This model is an extension of the message reification model [3]. It is aimed to overcome some of its limits, in particular that of information continuity, while keeping its advantages.

2.3 Reflex

Reflex [33] is an open reflective extension of Java, mainly developed to overcome a part of the limitations of the Java Reflection API [29].

Since version 1.1 of the Java Development Kit [27], the Java reflective facilities have been successively extended. However those facilities have been restricted to introspection: for instance, Java enables programs to know the names of the methods in a given class, and to instantiate the given class with a given string name. On the other hand it does not enable to alter the program behavior.

2.3.1 Overview

Reflex is a system for behavioral reflection in Java based on the MetaObject Model (MOM) as described above. With Reflex, standard Java objects can be reflective, that is to say, they can have a metaobject attached to them. A metaobject is responsible for extending or modifying the semantics of a language mechanism

such as method invocation, object creation, serialization, ... In the case of method invocation, control flow shifts up to the metalevel when a method is invoked on a reflective object. Such a method call is first reified (made explicit as a first class entity), and then passed to the metaobject, which can analyze it and operate accordingly.

The implementation of Reflex relies on an existing library for structural reflection in Java, Javassist [7]. Javassist offers a high-level API to modify the structure of a class at load time, through bytecode transformation.

2.3.2 Architecture

Reflex is based on a generic *builder* that is in charge of transforming classes at load time in order to insert the necessary *hooks* for shifting control to the metalevel when needed. The generic builder is in fact responsible for installing the infrastructure for the required hooks. The actual transformation of bytecode to insert hooks is done by *transformers*, which are components implemented with Javassist.

To specify which types should be made reflective and which hooks should be installed, the *builder* of Reflex has a *specification table*. This table can be fed manually or through an XML specification file.

2.4 Reflex and Reference Management Policies

The aim of the developers of Reflex was to use Reflex to achieve flexibility in a variety of domains. In [34] Reflex has been successfully applied in the area of reference management and mobile object systems. Using Reflex, the the serialization process of objects in Java can be controlled. A metaobject that traps the serialization process is attached to a base object. Using this kind of metaobject, two different reference management policies were successfully implemented: one for the *network reference* policy and one for the *rebinding* policy. We explain here how these two policies have been implemented using Reflex.

2.4.1 The Network Reference Policy

The Java RMI [30] mechanism already provides a mean to achieve the *network reference* policy, but this mechanism is not flexible at all because the class of the remote object has to implement a sub interface of the *Remote Object* interface, and must extend another class of the RMI framework. Also the stubs and skeletons have to be generated by hand. So it is not possible to make a non-remote object remote dynamically. With Reflex and the metaobjects, the *remote reference* policy can be applied to any object, without any constraint upon its type. However the RMI mechanism is used for this purpose.

The idea is to transfer a blank object[4] instead of the huge object over the network, and then forward the method invocations to the local object. To achieve this, a special `Invoker` object is introduced, which is a generic remote RMI object. It is a method invoker able to invoke any method on any type of remote reflective object. Thus the metaobject of the remote blank object forwards all method invocations to

[4]A blank object is an object whose fields are all set to null, type-compatible with the resource on the source host

this `Invoker` (that resides on the same site as the local object), and this `Invoker` will in turn forward the method call to the base object.

This design is illustrated in figure 2.3 on page 22. On host A, a reflective resource (R) is controlled by a `SendProxy` metaobject. When the base object is first serialized for migration, the `SendProxy` metaobject takes the control, sets up the `Invoker`, and serializes itself with a reference to it (see figure 2.3). Since the `Invoker` is a remote RMI object, when the metaobject will be deserialized, it will have a reference to the server-side skeleton of the `Invoker`, and will thus be able to perform remote method invocation on it. Apart from setting up the `Invoker`, the metaobject also specifies that the base object should not be transmitted fully, a blank object should be created instead.

Therefore on destination site (site B in figure 2.3) is a blank object (of the same type) that has a metaobject of type `RemoteCall` that will forward any method invocation to the `Invoker` on the local site (site A in figure 2.3). The `Invoker` was initialised with a reference to the local resource, and can therefore forward the method invocation to this local base object. We will come back in more detail to this migration policy in chapter 4.

2.4.2 The Rebinding Policy

To achieve the *rebinding policy*, the hosts on the network must provide some way to get a resource to a local reference based on an identifier. This way the agent can rebind this resource when it arrives to another host, and unbind when it leaves. A simple *resource manager* is used that allows a program to publish an object as a local resource, associating a string identifier to it. An incoming agent can then query the *resource manager* by using the string to get a reference to a resource that was previously bound to this string. The design of the *rebinding policy* is show in figure 2.4 . The agent holds a reference to a dumb reflective object that acts as a proxy to a local resource published to the resource manager. This object is controlled by a metaobject that maintains the binding the local resource (found by querying the local resource manager) and traps all the method invocations on the dumb object. The metaobject forwards the trapped method invocations to the local resource. Upon migration the binding is cleared and reestablished after migration when first needed (this is called *lazy initialisation*)

2.5 Reflection and Dynamic Adaptability for Mobile Code

In 2.1.1 we exposed the relation between resources, executing environments and data space management mechanisms. We highlighted that in current Mobile Code Systems, the choice which migration policy to use, depends on the implementation of the system, and is thus fixed. This fixed choice however does not fit to the requirements of todays applications, as they have to cope with the limitations of mobile networks, and the constraints of mobile devices. For example, if we deal with a low-reliable network where a lot of disconnects can occur, it would be better not to use the *remote reference* policy. On the other hand, if the client device has only very low memory, it would be better not to use the *copy* migration policy. Applications might want to adapt the migration policy according to characteristics

from the environment. A first step to achieve this adaptation using reflection (in particular with the Reflex framework) was done in [34]. In this work, it was shown how the application programmer could implement different migration policies by attaching metaobjects to the transferable resources, in particular the *remote reference* policy and the *rebinding* policy could be used (see 2.3). However, the choice of which migration policy to use for which resource was still statically defined. The motivation of this thesis is to continue on this work, in particular investigate how the switching between different migration policies can be done dynamically, and apply the results to a concrete case.

2.6 Summary

In this chapter we introduced all the necessary concepts used in this dissertation.

We gave a short introduction to mobile code, we defined the main concepts that are used in this area, and finally we gave an overview of some domains where mobile agents are applicable.

Next we exposed the main ideas behind reflection and metaprogramming. We also introduced Reflex, a reflective framework for Java, and highlighted its main characteristics.

We ended by giving a motivation for this thesis: implement dynamically adaptable migration policies to overcome the limitations of todays mobile environments.

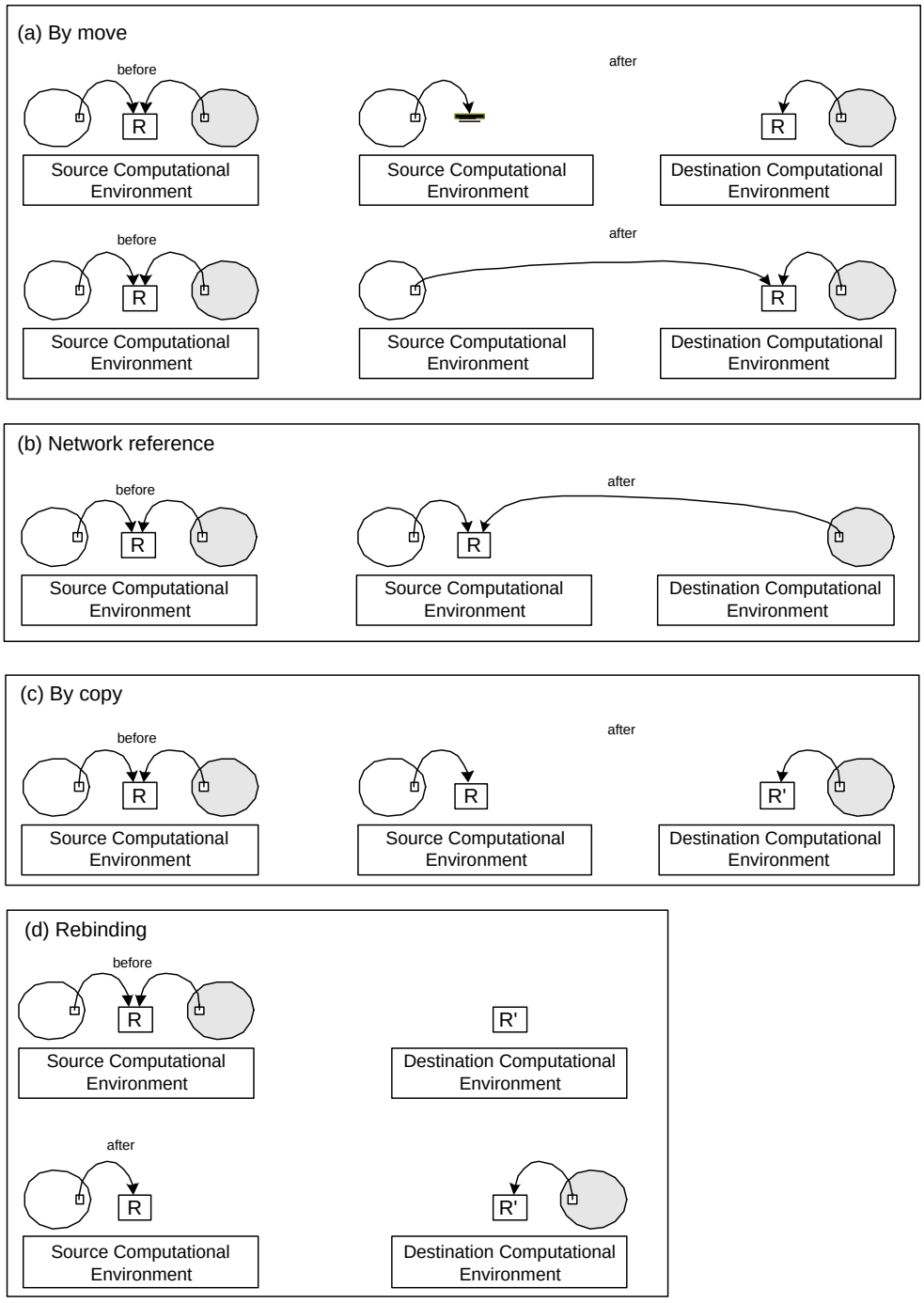

Figure 2.2: Data space management mechanisms.

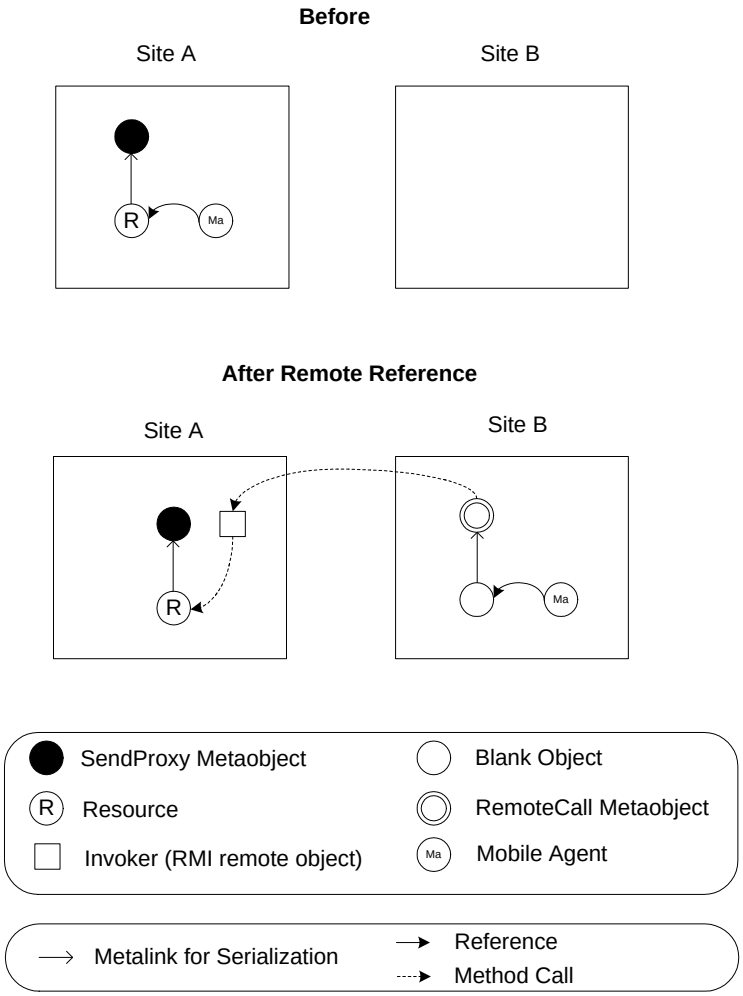

Figure 2.3: The remote reference migration policy.

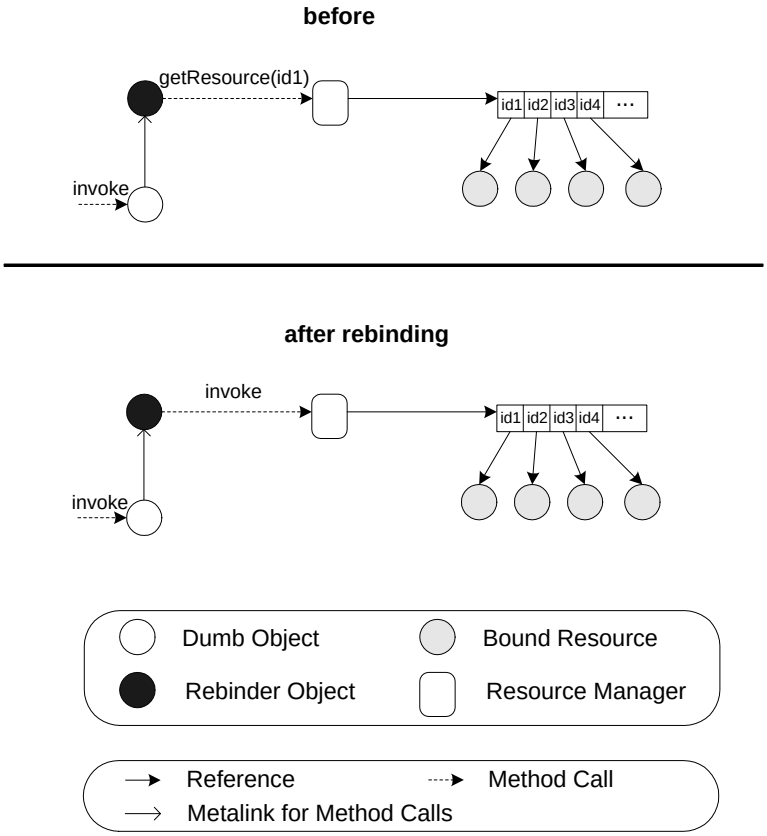

Figure 2.4: The rebinding migration policy.

Chapter 3

Mobile Environments and Adaptation

Mobile computing increases the interest of carrying mobile computing devices such as laptops, PDA's, cellular phones... Users can work, communicate, and use remote services. However the technical challenges that permit the use of mobile applications are not trivial. In this chapter we present the problems current mobile environments are faced with. We see how applications can be adapted to these problems, we give a classification of application adaptability strategies, and a state of the art of existing systems that offer adaptability.

3.1 Problems of Mobile Environments

3.1.1 Wireless Communication

Constraints on mobile environments are mainly due to the following aspects:

Disconnects Today's computers depend heavily on the network, and may cease to work correctly during network failures. Network failure is of greater concern to mobile computing design than traditional design, because wireless networks are very susceptible to disconnection.

The more autonomous a mobile computer, the better it can tolerate network disconnection. Some applications reduce network traffic by running entirely on the mobile device, rather then splitting the application and the user interface over the network. In environments with frequent disconnects, it is important that programs can run as *stand-alone* applications.

Low bandwidth When developing programs for mobile networks, one should take into account the low bandwidth of wireless networks. Current mobile technologies offer a bandwidth up to 2 Mbps (this is for UMTS[1]). However this bandwidth is shared among the different users sharing the same cell, so the speed depends on the size and the distribution of the population. The bandwidth also depends on whether the user is indoor or outdoor, walking, driving or standing etc.

[1] Universal Mobile Telecommunication System

High bandwidth variability Mobile computing also suffers from much greater variations in network bandwidth than traditional designs.

Heterogeneous networks Stationary computers stay connected to the same network, while mobile computers encounter more heterogeneous network connections. As they move to different places, they may experience different network qualities. In a meeting room for example, there may be better wireless equipment installed then in a hallway.

Also they may switch between interfaces when moving from indoors to outdoors. For example infrared can not be used outside because sunlight drowns out the signal.

Security Risks Precisely because it is so easy to connect to a wireless link, security of wireless communication can be compromised much easier than wired communication, especially when the transmission encompasses a large area.

3.1.2 Mobility

Users with mobile devices can move to other locations while connected to the network. For example, although a stationary computer can be configured statically to prefer the nearest server, a mobile computer needs a mechanism to determine which server to use.

The main problems introduced by mobility are:

- *Address Migration* As people move, their mobile computers will use different network addresses. Today's networks are not designed for dynamically changing addresses.
- *Location Dependent Information* For traditional computers that not move, information that depends on location is configured statically, such as printers, time zone... A challenge for mobile computing is to factor out this information and provide mechanisms to obtain configuration data dynamically, depending on the current location.

3.1.3 Portability

Today's desktop computers are not intended to be carried, so their design is liberal in their use of space, power, cabling. The design of mobile devices, on the contrary, should strive to properties as a wristwatch: small, lightweight, durable, water-resistant, and long battery life. Here follows a list of design issues caused by portability constraints:

- *Low power* Batteries should be light and small, and the life of the battery should be as long as possible.
- *Risks to Data* Portable devices are prone to physical damage, unauthorized access, or theft. Special security considerations can be taken: encryption of data,take copies that do not reside on the portable unit, etc..
- *Small user interfaces* The size constraint of mobile devices requires a small user interface. Systems with multiple windows open at the same time are impractical on small screens. Handwriting recognition can be used to replace many buttons.

- *Small storage capacity* Storage space on mobile devices is limited by physical size and power requirements.

3.2 The Need for Adaptability

In the previous section, we have highlighted the main problems of mobile computing. In spite of those problems, a mobile client can offer acceptable services through alertness and prompt reactions to changes in the environment. In other words, mobility requires support for adaptation as Katz summarized [16]:

> Mobility requires adaptability. By this we mean that systems must be location and situation aware, and must take advantage of this information to dynamically configure themselves in a distributed fashion.

Several systems have proposed a solution for achieving this adaptation. We present here a taxonomy of adaptation strategies that those systems use, and we give an overview of the three systems we investigated: *Odyssey*, *Coda* and *Sumatra*.

3.2.1 Taxonomy of Adaptation Strategies

The range of strategies for adaptation is delimited by two extremes, as shown in figure 3.1. At one extreme, adaptation is entirely the responsibility of individual applications. This *laissez-faire* approach avoids the need for system support, it lacks a central arbitrator to resolve incompatible resource demands of different applications and to enforce limits on resource usage. It also makes applications more difficult to write, and fails to amortize the development cost of support for adaptation.

The other extreme of *application-transparent adaptation* places entire responsibility for adaptation on the system. This approach is attractive because it is backward compatible without any modifications. The system provides the focal point for resource arbitration and control. The drawback of this approach is that there may be situations where the adaptation performed by the system is inadequate or even counterproductive.

Between these two extremes lies a spectrum of possibilities that we collectively refer to as *application-aware adaptation*. By supporting a collaborative partnership between applications and the system, this approach permits applications to determine how best to adapt, but preserves the ability of the system to monitor resources and enforce allocation decisions [24].

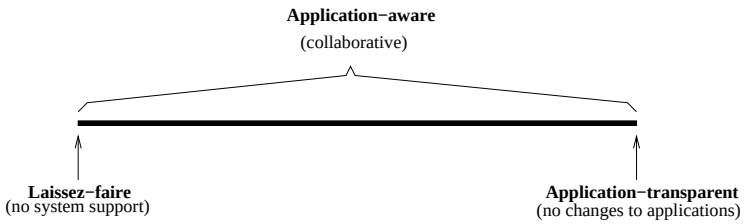

Figure 3.1: Range of adaptation strategies.

3.2.2 Overview of Systems that Offer Adaptability

Odyssey

Mobile hosts and therefor applications are expected to promptly react to changes in the environment and provide an acceptable quality of service despite these changes.

Limited resources on the mobile hosts suggest that they have to interact with remote servers for retrieving data, for getting local information, or querying databases.

The *Odyssey* [22] project aims to solve these problems at the operating system level. It provides tradeoffs between data quality and resource consumption. Such adaptation depends on system-provided choices and are customizable by applications. Odyssey is an example of an *application-aware* adaptation system, designed as a set of extensions to the NetBSD operating system. Applications can determine how to adapt by collaborating with the operating system. Odyssey's application-awareness mechanism is based on *fidelity* and *agility*.

Fidelity Fidelity is the degree to which data presented at the client for use to an application matches the reference copy at the server. Ideally, a data item at the client should be indistinguishable from that available on the server.

Fidelity has many dimensions, one well-known is consistency. Other dimensions of fidelity depend on the type of data in question. For example video data has at least two additional dimensions: frame rate and image quality of individual frames.

Agility Adaptive systems should be able to detected changes in the environment, and to react to those changes. Ideally, a mobile client should always have perfect knowledge of current resources available. In other words, there should be no time lag between a change in resources available and its detection. Further, if this change is sufficient to warrant modification of client behavior, that too should be accomplished without delay. Thus a key property of an adaptive system is the speed and accuracy with which it detects and responds to changes in resource availability. We call this property *agility* of the system. When changes are large, only highly agile systems can function effectively. In more stable environments, less agility can suffice. Agility is thus the property of a mobile system that determines the most turbulent environment in which it can function acceptably.

Agility is a complex property because of the need of managing various resources with various degrees of sensitivity. Sensitivity changes not only according to different kinds of resources but also according to applications. For example, one application might be sensitive to latency and bandwidth, while another application is sensitive to data accuracy.

Odyssey provides a framework into which diverse notions of data fidelity as well as agility components can easily be plugged. It allows concurrent applications to execute on a mobile host, and coordinates and controls resource usage of individual applications. The system monitors resource levels, notifies applications of relevant changes and enforces resource allocation decisions. On top of that, each application decides independently how best to adapt when notified. Finally, Odyssey tends to be minimal and limits the number of changes for extending an existing operating system.

In order to achieve the above *application-aware* adaptation and collaborative partnership, Odyssey (figure 3.2) is implemented as a new Virtual File Systems

(VSF) in NetBSD. Odyssey maintains type-specific knowledge through components called *wardens*. A warden encapsulates the necessary system-level support of a client to effectively manage a data type. Applications always contact servers transparently through wardens. Another Odyssey component, the *viceroy*, is in charge of globally coordinating the resource management. The viceroy is type-independent.

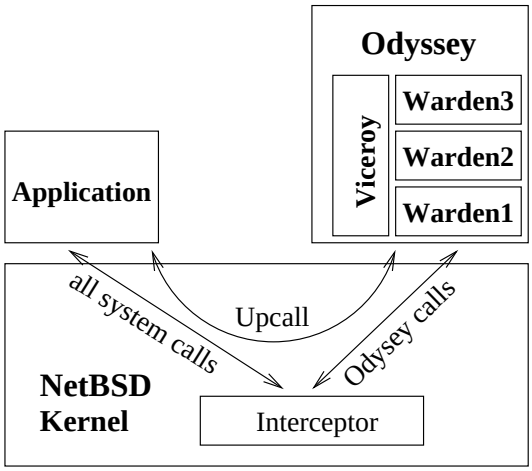

Figure 3.2: Odyssey Client Architecture.

An application can specify its resource expectations by giving a window of tolerance on the resource variation. The viceroy registers this request and notifies the application whenever the resource managed by Odyssey include network bandwidth and latency, disk cache space, CPU, battery power.

Odyssey has experimented with *application-aware* approach using applications such as a video player, a World-Wide Web browser, and a speech recognizer.

Coda

Replication is the key to availability of data in networks with mobile hosts. Replication however introduces its own problems, such as data access, consistency, and conflict resolution. Much work has been done to provide efficient data replication at file system level. Such mechanisms are based on file caching, grouping of related files, and consistency management using file and directory semantics. *Coda* [19, 18] is an example of such a file system, it uses optimistic replica control strategy to allow updates to cached data when disconnected. At each client, there is a component responsible for monitoring connectivity, resynchronizing cache state upon reconnection with servers, and detecting update conflicts. This component is called *Venus*. Once a conflict is detected on a file, Venus invokes an *application-specific resolver* (ASR) and lets it resolve the conflict. If the ASR succeeds, the conflict is transparent to the user; otherwise the conflict becomes visible and has to be repaired manually.

The importance of application assistance can be seen by considering the example of a calendar management program. Suppose an executive and her secretary both make appointments while the former is disconnected. Upon reconnection,

Venus detects that the file containing appointments is in conflict. But it has no knowledge of the format of the file contents, nor of whether there is really a scheduling conflict. Only code specific to the calendar program can tell, for instance, that appointments for an hour each at 8 am and 10 am on the same day pose no problem if they are in the executive's office, while those appointments are impossible to keep if they are in Nantes or Santiago.

Sumatra

Sumatra [23, 2] is an extension of the Java programming environment that supports adaptive mobile programs.

Sumatra is another example of an *application-aware* adaptation system, and it is based on the requirements of awareness and agility.

Awareness Resource-aware programs need to be able to monitor the availability and quality of the resources in the environment. Resource can be monitored either *on-demand* or *continuously*. Both kinds of monitoring are useful.

- **On-demand** This kind of monitoring is most useful in the following three situations:
  - If the resource is used infrequently, but is expensive to use. For example an application on a mobile host that uses a cell-modem to periodically scan incoming mail being held at the post-office machine.
  - If the availability of the resource in question changes infrequently. For example an application that chooses the location from which it monitors a process based on the amount of disk space available at that location
  - If the resource is expensive to monitor and the cost of monitoring outweighs the potential gains. For example an application that accesses large volumes of data over a very slow link
- **Continuous monitoring** This kind of monitoring is useful if the resource is frequently used or if the resource changes frequently or is cheap to monitor.

For continuous monitoring, the resource-monitoring interface should allow programs to register an application-specific filter which determines the granularity of the resource changes that should be reported.

Agility To achieve agility a mobile code language should provide mechanisms that allow programs to react quickly to events like revocation of bandwidth, memory, or qualitative changes in network connectivity.

In the design of *Sumatra* [23, 2], the Java language was not altered. *Sumatra* can run all Java programs without modification. All added functionality was provided by extending the Java class library and by modifying the Java interpreter, without affecting the virtual machine interface.

Sumatra adds four programming abstractions to Java: *object-groups*, *execution-engines*, *resource-monitoring* and *asynchronous events*. An object-group is a dynamically created group of objects. Objects can be added to or removed from object-groups. All objects within an object-group are treated as a unit for mobility-related operations. This allows the programmer to customize the granularity of

movement and to amortize the cost of moving and tracking individual objects. Object-groups also allow the programmer to control the life-time of objects. Objects that are included in an object-group continue to live on a host even after the thread that created them completes execution or migrate to another host. When an object-group is moved, all local references to the objects in the group are converted into *proxy references* which record the new location of the object. Method invocations on a proxy objects are translated into calls at the remote site.

An execution engine is the abstraction of a location in a distributed environment. In concrete terms it corresponds to an interpreter executing on a host. *Sumatra* allows object-groups to be moved between executing-engines.

The resource-monitoring support in Summatra allows programs to either query the level of resource availability or to control the extent to which various resources are used by the program itself as well as library code it is linked to. Both on-demand as well as continuous monitoring is supported.

In *Sumatra* asynchronous events are used to notify executing programs about current changes in their execution environment. These notifications can either come from the interpreter or from the external environment (the operating system, or some other administrative process). Asynchronous notifications are implemented as Unix signals. *Sumatra* allows applications to register handlers for a subset of Unix signals. Signals can be used to inform the application about urgent events. Using a handler the, the application can take appropriate action including moving away from the current execution state.

3.2.3 Adaptation Mechanisms

Replication Replication is the process of making copies, so that two or more objects can exchange updates of data. In mobile environments replication serves to offer *availability*. When a client is disconnected from a server for example, the client can continue working if it has local copies of the objects it needs from the server (*replicas*).

Binding The binding adaptation mechanism specifies how to bind an application object. Bindings can be remote or local. The binding policies to use depend on parameters such as network latency, available bandwidth, locality of the server object, and load of the server. Whenever those parameters become too poor in regard to the needed quality of service, another binding policy could be used in combination with the *compression* or *filtering* of data.

Filtering Filtering is intended to act directly on the data and object contents. It can either be used to remove a part of the data or to apply a transformation on it.

Filtering is strongly related to the application semantics as well as the type of data sent to the client. Let us look at some examples. In the case of a World Wide Web caching proxy, application specific filtering is usable to filter HTML pages. We can completely remove icons or images, or reduce their size. More generally, in the case of images or real-time color movies, a filtering strategy can reduce the size of the images, reduce the number of colors or even switch to black and white, or reduce the number of frames per second.

On small hosts, filtering allows to remove meaningless data such as icons, or other disk-space and bandwidth-consuming data such as Postscript or PDF files. On less restricted hosts, the filtering strategy can transform Mime Encoding, Postscript, or PDF into equivalent text versions.

Compression Compression algorithms do not change the data, they just perform at transport level in order to reduce the physical size of the message. Compression strategies are therefore more generic. They usually work well with any kind of data. However, for an optimal compression, it is sometimes necessary to take into account the data type.

Encryption Encryption strategies are very similar to compression strategies, in the way they are supported by the system. Encryption is the translation of data into a secret code. It is the most effective way to achieve data security. To read an encrypted file, you must have access to a secret key or password that enables you to decrypt it. Unencrypted data is called *plain text*; encrypted data is referred to as *cipher text*.

There are two main types of encryption: *asymmetric* encryption (also called public-key encryption) and *symmetric* encryption.

Tracking Environment Changes Adaptation mechanisms alone are insufficient to provide responsiveness to the environment changes. Both the system and the application need information about the environment to trigger efficient adaptation. Information can be retrieved through process that monitors changes in the environment. Applications should be able to subscribe to the environment monitor to know about the changes.

3.3 Summary

In this chapter, we first highlighted the problems of current mobile environments. We mentioned that mobile applications need adaptability to overcome those problems. We gave a taxonomy of systems that propose a solution for achieving this adaptability, and we reviewed the main characteristics of those systems. Finally we gave an overview of adaptability mechanisms that were used in those systems.

Chapter 4

Dynamic Adaptation of Migration Policies

The idea behind the switching of migration policies originated from the constraints current mobile systems are faced with. Consider the example where a user is running an application on his wireless device that accesses information on a remote server. Initially the *remote reference* migration policy could be used for communication between the wireless device (the client) and the server. But as wireless networks are very unreliable, the bandwidth between the client and the server could drop suddenly or the client could even become disconnected from the server. In this case it would be better to use the *copy* policy. This way computations can be done locally, and thus there is no network traffic between the client and the server. On the other hand, as the memory of portable devices is generally very low, the server could decide to switch to *remote reference*.

In previous work [34], Reflex has been successfully used in order to specify the way objects referenced by a migrating object should be managed (reference, rebinding, etc.). It was made possible to statically specify which policy to use, whatever the type of the objects, achieving a nice degree of separation of concerns. In this chapter, we extend this work. We first design an infrastructure to make it possible to switch dynamically between different migration policies, and we see how this infrastructure can be applied onto a base application. Next we illustrate how this switching between migration policies is done in the world of *ubiquitous computing* and *roaming agents* using this infrastructure. Finally we give a concrete example in the world of *roaming agents*.

4.1 Design

The concept of our model is the following: we attach a metaobject to the resource that is subject to migration. When the resource is about to migrate, the metaobject consults other resources that reasons about the upcoming migration in order to decide what migration policy to use. In this section we describe in detail how we designed this infrastructure that enables an application to switch dynamically between migration policies. We see how an application can be structured in order to migrate the different parts. Finally we highlight how our infrastructure can be

applied onto a base application using an XML specification.

4.1.1 Infrastructure Monitor

In this design we assume the existence of an Infrastructure Monitor that can be consulted to obtain information about the destination host (type of machine, memory available there, connection mode, etc...) and about the network connection between the current host and the destination one. This monitor is an important source of information for reasoning about an upcoming migration. Note that an implementation of this Infrastructure Monitor is beyond the scope of this work. We should look for current systems which have a similar infrastructure, that might be integrated into our work.

4.1.2 Application Structure

In order to migrate parts of an application, we have to structure it in some way. Our approach consists of making groups of objects according to their type. With regards to migration, each type is classified in one of the following groups:

- **transferable types**
  - **adaptable** objects of this type may migrate, depending on the result of metalevel reasoning.
  - **non-adaptable** objects of this type may migrate, but no metalevel reasoning is done.
- **non transferable types**: objects of this type can never migrate.

Using this approach for structuring an application, we can make type-based decisions upon migration.

4.1.3 Metalevel Reasoning

To preserve transparency, reasoning about upcoming migrations is done at the metalevel. The main idea behind our model is to control the serialization process of the object that is about to migrate. The serialization semantic[1] to use depends on the result of some metalevel reasoning about information from the *infrastructure monitor* and possibly other sources of information, and the application of some filters on the object that is about to migrate. Using different serialization semantics, we can achieve different migration policies. We give here an overview of the different types of metalevel entities used in our model (see also figures 4.1 and 4.2).

Serialize Metaobject Each adaptable type is associated with a *serialize metaobject*. This metaobject traps the serialization process of the object that is about to migrate. It first detects which kind of serialization is occurring (*backup* or *migration*) by performing type analysis of the stream object used (how this mechanism works is described in detail in [32]). If it is migration, then decision on how the base object will be serialized is delegated to the *migration adaptor*.

[1] Recall that the Java Serialization API [28] offers the possibility to specify an alternative object for serialization

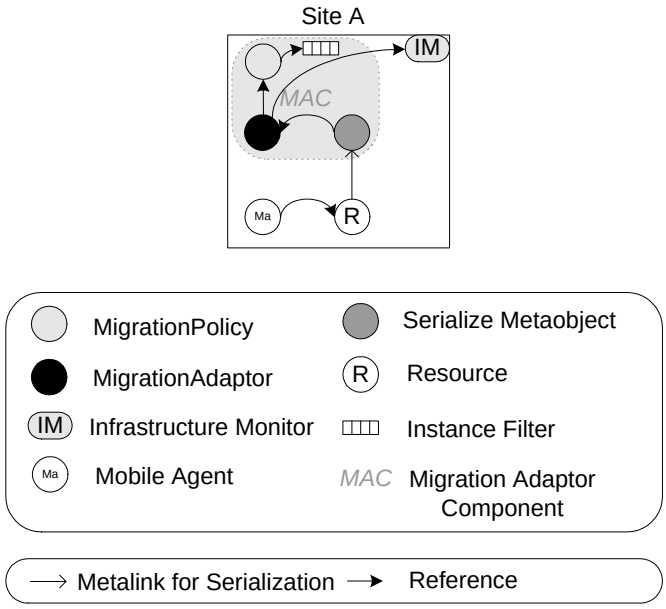

Figure 4.1: Design of the metalevel.
At run time, migration adaptors use information from the infrastructure monitor and other sources of information, and possibly instance filters are applied to determine whether and how a particular object should be migrated.

Migration Adaptor This object decides which migration policy will be used, by reasoning about information from the Infrastructure Monitor, the state of the base object (with base object we mean here the resource R from figure 4.2), and possibly other sources. The state of this object is then adapted to support the chosen migration policy: *remote reference* or *copy*. To implement this, we used the *state* design pattern [15]. Other migration policies can easily be added.

Instance Filters These objects are type-specific filters that encapsulate criteria to select candidates for migration among instances of a same type. They reason about the *state* of a particular instance, and decide how the object should be serialized. An instance filter can:

- **filter out certain instances**: The instance filter can decide, by accessing the state of the instance, if the instance should be migrated or not.
- **transform copies of instances**: If the filter decides that the instance may migrate, it can apply a transformation on the copy of the instance that will be transported. For example when using *copy* migration policy, a lightweight version of the object could be returned for serialization (some fields could be removed or changed).

Several *instance filters* can be composed into a tree-like structure using the *composite pattern*, and can then be combined by logical operators using the *visitor pattern* [15]. Figure 4.2 shows a UML [13] class diagram of our metalevel design to switch dynamically between migration policies.

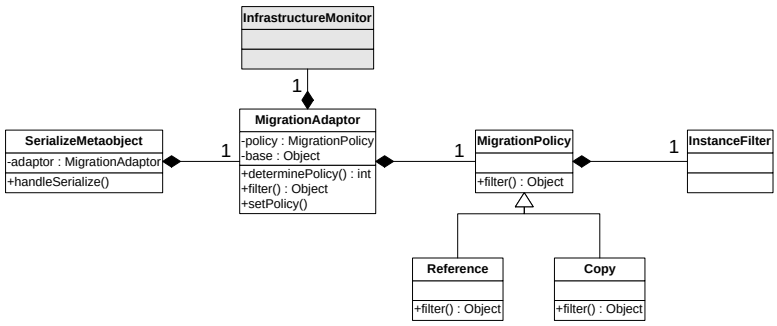

Figure 4.2: Class diagram of the metalevel entities.

In the following, we shall refer to this group of metalevel entities as the *Migration Adaptor Component* (MAC see figure 4.1).

4.1.4 Specification of the MAC

The entry point of the MAC, common to all adaptable types of the application, is the serialize metaobject. Each adaptable type is statically associated with a migration adaptor, using XML (see figure 4.3). The remaining components of the MAC, the migration policy and instance filters, are specified programmatically.

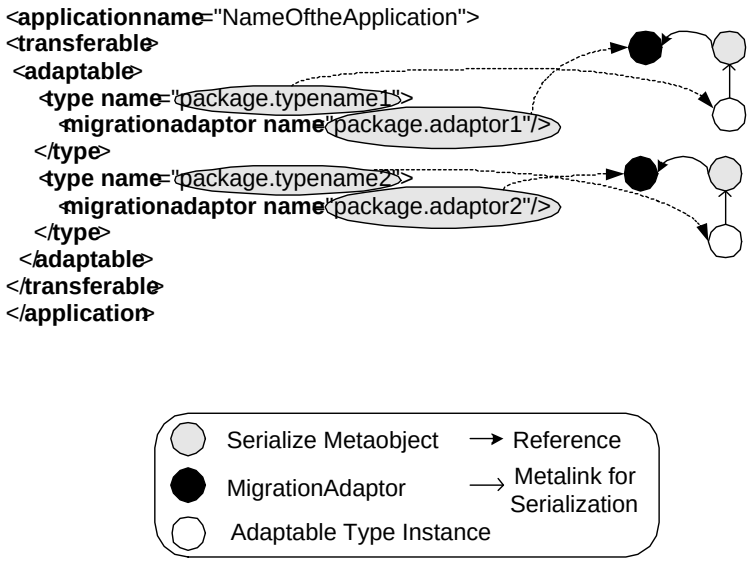

Figure 4.3: Application structuring and metalevel reasoning.
XML is used to structure the application and to specify the migration adaptor for adaptable types.

Note that from the information in this XML file we can generate the Reflex

specific XML file, which is used to fill the *specification table* like we exposed in 2.3.2.

4.1.5 Running a Reflective Program

To apply our metalevel library onto a base application, we wrote a program (called `Run`) that parses the XML specification for this application. This `Run` program is also responsible for generating the Reflex specific file XML (as we mentioned above in 4.1.4). Reflex then applies the class transformations to the adaptable types in the base application to make them reflective according to this information.

So to apply our reflective metalevel onto a base application, we simply use the `Run` program, with as arguments the name of the program to run and the location of the XML specification that contains the associations between the base - and metalevel types. This way we don't have to change anything to the base application when we want to apply our metalevel library onto it when running.

4.2 Ubiquitous Computing

Ubiquitous computing [1] names the third wave in computing, just now beginning. First were mainframes, each shared by lots of people. Now we are in the personal computing era, person and machine staring uneasily at each other across the desktop. Next comes ubiquitous computing, where one person has different computers. A person can have for example a fixed computer in his house, a portable computer for his work, and then maybe also a PDA or cellular phone. Typically this person wants to be able to access application data from all of these devices wherever he is and whenever he wants. To offer this way of computing, the application data is stored in a big central server. A user can consult and change this data from whatever device he wants.

In this section we see how we can apply our design from the previous section in the world of *ubiquitous computing* to switch dynamically between migration policies. We expose how we designed the metalevel for the *copy* and the *remote reference* migration policies in this world, and how the switching from one migration policy to another is done.

4.2.1 Move by Copy

In the move by *copy* migration policy, the resource that is subject to migration is copied and then transported to the destination host. Figure 4.4 shows our design of the metalevel in this case. A mobile agent *Ma* has a reference to a resource R. The agent *Ma* migrates from the server (A) to the client (B). When the resource R is about to be serialized, the `SerializeMetaobject` traps the serialization process of this resource. The `MigrationAdaptor` metaobject decides to use the *copy* migration policy (by accessing the Infrastructure Monitor and possibly other sources of information), and switches its state to the move by *copy* migration policy. This is done by setting the field `policy` of the `MigrationAdaptor` to the `Copy` policy. The `Copy` object finally returns the object to be serialized (R'). In its most simple form it will just return a copy of the original resource. Alternatively, if the `Copy` was initialized with an `InstanceFilter`, it passes the copy first to this `InstanceFilter`, and then returns this filtered copy as the object to be serialized.

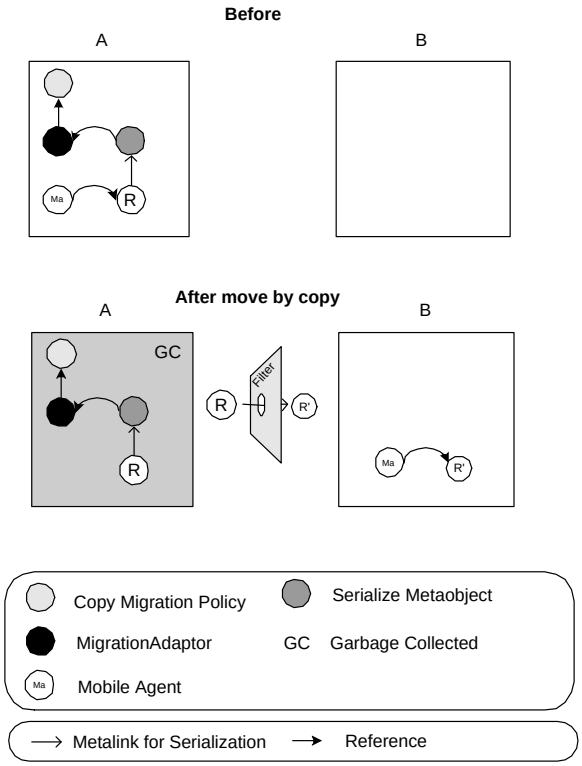

Figure 4.4: Move by copy

So summarized, the following steps are done when the agent *Ma* migrates from host A to host B using the move by *copy* migration policy for the resource R:

1. The `SerializeMetaobject` traps the serialization of R.
2. The `MigrationAdaptor` accesses the Infrastructure Monitor and decides to use the *copy* migration policy.
3. The `MigrationAdaptor` adapts its state to the chosen migration policy (by setting its policy field to `Copy`).
4. A copy of the resource is made in the `Copy` object.
5. The copy of the resource is passed to an `InstanceFilter`, which optionally filters the copy of the resource.
6. The filtered copy is returned for serialization.

The objects that remain on hosts A and B after migration of *Ma*, will finally be garbage collected, as they are no longer referenced by any objects.

4.2.2 Move by Copy with Synchronization

This is an extension of the *move by copy* policy we described above in 4.2.1. In the *move by copy* policy, the local copies on the destination host can be changed,

which results in different objects on the source host and the destination host. One might want to synchronize the unmodified original objects on the source host with the modified replicas on the destination host, in order to have an updated version on the source host.

CopyWithSyncWrapper, CopiesTable, and RemoteCopiesTable To be able to synchronize the objects on the source host with the copies on the destination host, we have to know which modified object on the destination host corresponds to which object on the source host. The solution we provide for this is the following (see also figure 4.5): Instead of just sending a copy of the object on the source host to the destination host, we send a `CopyWithSyncWrapper` object. This wrapper object wraps the copy of the object together with a unique ID for the object (which we retrieve using `System.identityHashCode(Object)`). On the source host, we have a hash-table (we call this hash-table the `RemoteCopiesTable`) with as key this unique ID and as value a reference to the object on the source host.

The `CopyWithSyncWrapper` has a `readresolve` method. This method allows a class to replace/resolve the object read from the stream before it is returned to the caller. By implementing the `readResolve` method, a class can directly control the types and instances of its own instances being deserialized [28]. The `readResolve` method in the `CopyWithSyncWrapper` returns the copy of the object upon deserialization on the destination host, and puts in another hash-table (we call this table the `CopiesTable`) the ID of the object (as the key) and a reference to the copy (as the value).

Thus on the source host, we have for each object that was transported by *copy with synchronization*, a reference to it in the `RemoteCopiesTable`. And we can retrieve this object using its unique ID. On the destination host, we have the `CopiesTable` which contains references to the copies of the objects that were transported using *copy with synchronization*. Those references can be retrieved using the same ID.

To synchronize the objects on the source host with the objects on the destination host, we just have to send the copies on the destination host together with their ID's to the source host. The original objects on the source host can then be retrieved using these ID's in the `RemoteCopiesTable` and be updated with the changes that where made to the copies on the destination host.

Synchronize Interface The protocol to use to update the original object on the source host can be different for each object type. We provide a `Synchronization` interface, which has a method `synchronize(Object old, Object new)`. This method specifies how the update from the `new` object to the `old` object should be done. Thus for each *transferable* type (*adaptable* or *non-adaptable*) that has to be synchronized, we have to implement this interface. We will refer to classes that implement this interface as *synchronizers* further in this work.

Binding a type to a synchronizer The server has to now in some way, for which type he has to use which *synchronizer*. To bind a type to a *synchronizer*, we extend the XML specification we used for specifying the MAC (like we explained in 4.1.4). We specify for each *transferable* type (*adaptable* or *non-adaptable*) which *synchronizer* should be used (see figure 4.6 on page 40).

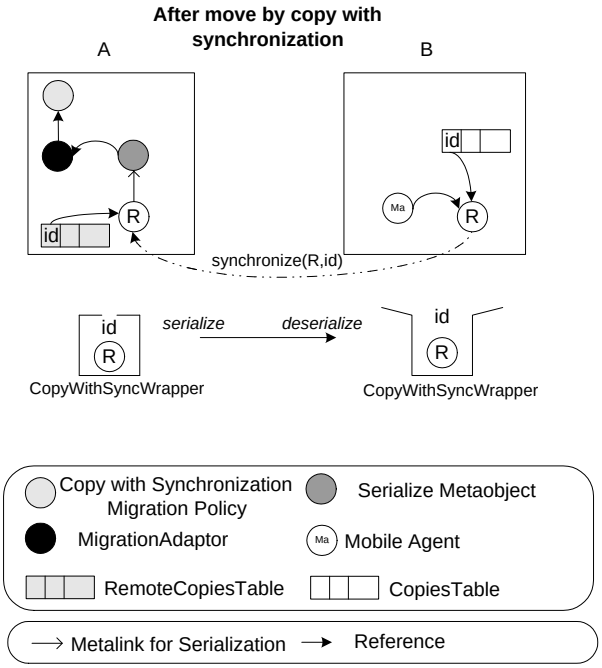

Figure 4.5: Principle of the CopyWithSyncWrapper

ApplicationServer and ApplicationClient To implement this synchronization mechanism, we introduce two new classes, one on the source host and one on the destination host: the `ApplicationServer` and the `ApplicationClient` (see also figure 4.7). The `ApplicationServer` is an RMI remote object, and we bind an instance of this class to the RMI registry at the source host. This object has a reference to the root of the Java application on the server. The `ApplicationClient` looks up the remote `ApplicationServer` instance in the RMI registry at the client side, and can get a reference the root of the application (`BaseApplication` in figure 4.7) that resides on the source host using the `getRootApplicationObject()` method. This method gets the root of the application from the source host, and transports it to the destination. Note that this is the starting point of the adaptation, the `SerializeMetaobject` traps the serialization process of the root of the application. The `SerializeMetaobject` will then give control to the `MigrationAdaptor`, which will decide what migration policy to use.

The `ApplicationServer` also has a reference to the `RemoteCopiesTable` (see figure 4.7), and has a method `synchronize(id, clientObject)`. This method is called from inside the `ApplicationClient`, when synchronization has to be done, for each object in its `RemoteCopiesTable`, with as arguments this object together with the hash-value (originally received from the source host). The `synchronize` method on the source host can then retrieve the original object using the hash-value received from the destination host, and synchronize this object on the source host with the object received from the destination host.

In chapter 5 we see a concrete example of this synchronization protocol.

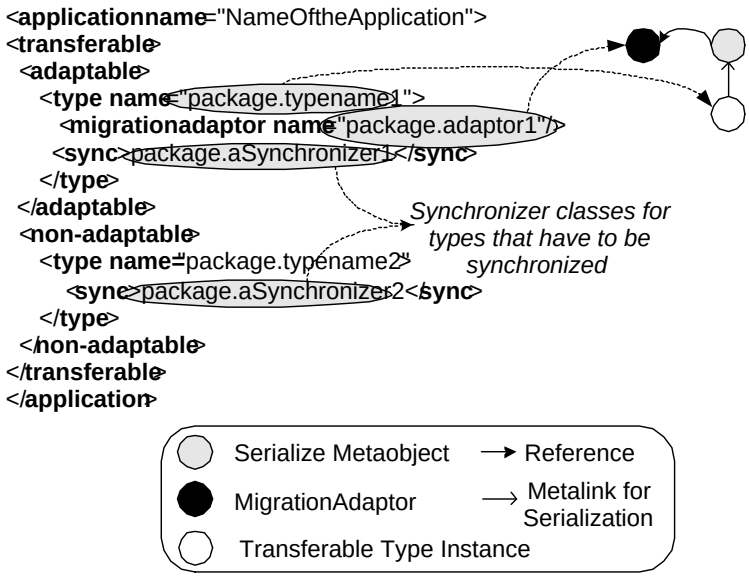

Figure 4.6: Extension to our XML specification to support the synchronization protocol

4.2.3 Remote Reference

In the *remote reference* migration policy, the resource that is subject to migration, stays on the source host, and references to it are done remotely from the destination host. Figure 4.9 shows our metalevel design to achieve this migration policy. In this figure, the agent *Ma* has a reference to a resource R. The agent wants to migrate from the server (A) to the client (B). When the resource R is about to be serialized, the `SerializeMetaobject` traps the serialization process of this resource. The `MigrationAdaptor` metaobject decides to use the *remote reference* migration policy (by accessing the Infrastructure Monitor and other sources of information), and switches its state to the *move by reference* migration policy. This is done by setting the field `policy` of the `MigrationAdaptor` to the `Reference` policy. The `Reference` object is responsible for setting up the remote infrastructure to achieve the *remote reference* migration policy. This object returns a `RemoteReferenceWrapper` object as the object that has to be serialized. The idea of this wrapper is the same as the wrapper we used to implement the synchronization protocol above in 4.2.2. This wrapper wraps the type of the class of the original resource R, and a `RemoteInvoker` (which is a remote RMI object, see figure 4.8). The `RemoteReferenceWrapper` has a `readResolve` method. The `readResolve` method in the `RemoteReferenceWrapper`, creates and returns a blank object of the same type as the original resource R (the wrapper contains the name of the type, and using the Java reflection API [29] we can instantiate an object of this type). The method also creates a `RemoteCaller` metaobject (on the blank object), that traps the method invocations that or done on the blank object, and forwards them to the remote resource via the `RemoteInvoker`.

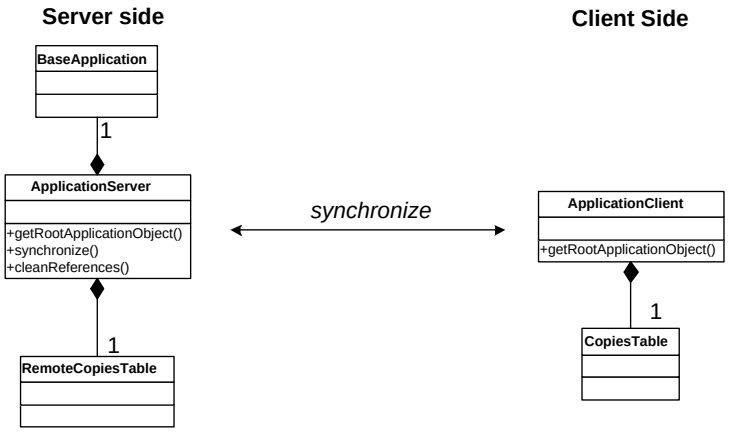

Figure 4.7: Classes to implement the synchronization protocol.

Objects that had a reference to the resource R on the source host, thus now have a reference to a type compatible blank object on the destination host.

This approach of achieving the *remote reference* migration policy is very effective, as we only transport the `RemoteInvoker` (actually only the RMI skeleton has to be transported), and the type of the original resource R.

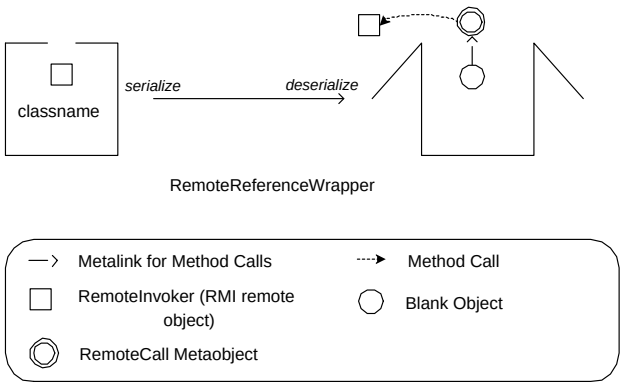

Figure 4.8: Principle of the RemoteReferenceWrapper

So summarized, the following steps are done when the agent *Ma* migrates from host A to host B using the *remote reference* policy for the resource R:

1. The `SerializeMetaobject` traps the serialization of R.
2. The `MigrationAdaptor` accesses the Infrastructure Monitor and possibly other sources of information, and decides to use the *reference* migration policy.
3. The `MigrationAdaptor` adapts its state to the chosen migration policy (by setting its `policy` field to `Reference`).

4. A `RemoteReferenceWrapper` is instantiated in the `Reference` object and returned as the object to serialize.

5. The `readResolve` method of the `RemoteReferenceWrapper`, returns a blank object (which is type-compatible with resource R) with a `RemoteCall` metaobject attached to it when an instance of the `RemoteReferenceWrapper` is deserialized. The `RemoteCall` metaobject traps method invocations that are done on the blank object, and forwards them to the remote resource R.

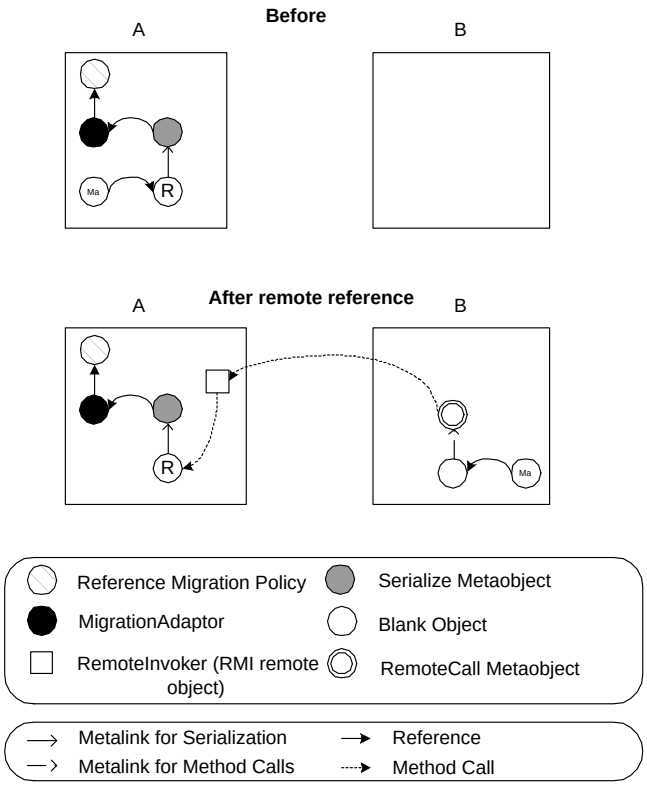

Figure 4.9: Move by remote reference

Extension to the remote reference migration policy The `RemoteInvoker` is an RMI remote object, and thus uses sockets for network communication. When the `RemoteInvoker` is created, we can specify the socket type that has to be used. We implemented successfully an example where *compression sockets* and *encryption sockets* were used.

4.3 Roaming Agents

In this section we see how we can apply our design from 4.1 in the domain of *roaming agents* for switching dynamically between migration policies. A roaming agent is

an agent that can move between a number of hosts, without any restrictions on the order in which the hosts are visited, or the number of times each host is visited. We consider here all the different situations that can occur when switching from one migration policy to another, in particular the switching between the *copy* and the *remote reference* migration policies. In the case where we have only two hosts, there is almost no difference with the situations we described above for *ubiquitous computing*. The only difference is that on the remote host, the adaptable resources have to be reflective also, as they can now migrate further to other hosts. We explain the different cases that can occur with three hosts, as this covers all the situations that can occur with more hosts.

4.3.1 Switching from Move by Copy to Move by Copy

This is the case (see figure 4.10) where an agent *Ma* with a reference to a resource (R) first moves from host A to host B, using the move by *copy* migration policy for the resource (R). Next the agent moves from host B to host C, and the resource R is again transported by *copy*. There is almost no difference with the move by *copy* policy described in 4.2.1. The only difference is that here, the resources on the destination hosts are also reflective, as the agent can move further to other hosts. Each time the resource is moved by *copy*, some `InstanceFilter` can be applied on it.

4.3.2 Switching from Copy to Remote Reference

In figure 4.11 on page 49 we see what happens when a resource is first moved by *copy* and then by *remote reference*. The mobile agent *Ma* has a reference to a resource R. When the agent moves from host A to host B, using move by *copy* for R, there is no difference with the move by *copy* policy described in 4.2.1. When the agent moves from host B to host C, we can see that the state from the `MigrationAdaptor` is changed from *copy* to *remote reference*. The process of achieving the *remote reference* policy is the same as described in 4.2.3.

Note that the resource on host C (see figure 4.11, after the agent *Ma* moved from A to B to) has two metaobjects: one for *serialization* (the `Serialize` metaobject) and one for trapping the *method invocations* (the `RemoteCall` metaobject).

4.3.3 Switching from Remote Reference to Remote Reference

The initial setting in figure 4.12 is the result of moving mobile agent *Ma* from host A to host B, using the *remote reference* strategy for the resource R. When the agent *Ma* then moves to host C, using the *remote reference* policy for R, there are two different options concerning how to move the resource:

1. We can create a new `RemoteInvoker` on host C, that forwards the method calls to the blank object on host B.
2. We can reuse the `RemoteInvoker` on host A, to forward the method calls to the original resource R.

We decided to implement the second option. In the first option, all the method invocations from host C on the resource R, would have to go through host B, which is an unnecessary overhead, as the original resource is on host A.

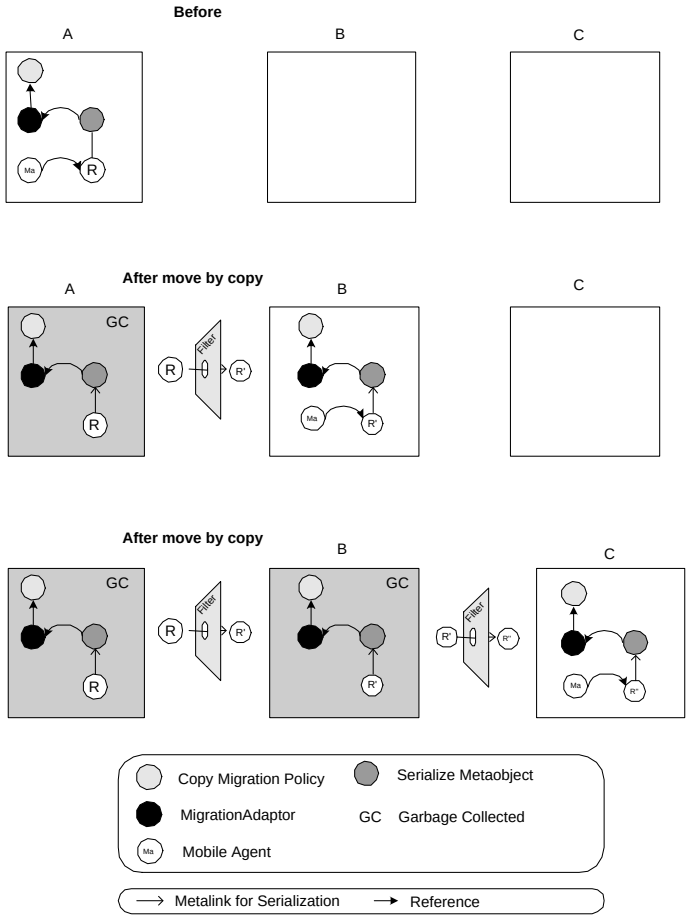

Figure 4.10: Switching from move by copy to move by copy

The `RemoteInvoker` on host A (that was used by the blank object on B to forward the method invocations to R) is reused by the blank object that is created on host C. So all the remote invocations from host C to resource R located on host A are done through the same `RemoteInvoker` as the one that was used on host B. All the objects that remain on host B after migration of *Ma*, are garbage collected, as they are no longer referenced by any object.

4.3.4 Switching from Remote Reference to Move by Copy

Here we describe the situation that occurs when first moving a resource by *remote reference*, and then by *copy* (see figure 4.13). The initial situation is the result of moving agent *Ma*, which has a reference to R, from host A to host B, using the *remote reference* migration policy for R. When *Ma* moves from host B to host C, using the move by *copy* policy for R, there are different options concerning how to move the resource:

1. Move a copy of the blank object (located on host B) to host C.
2. Get a real copy of the original resource R, and not of the blank object.

We decided to implemented the second option. In the first option, we don't really have the *copy* policy. If we send a copy of the blank object to the remote host, we actually still have a *remote reference* to resource R on host A, which is not really what we wanted. To implement the second option, we have to get the a copy of the original resource R from host A to host C. The only way to access the resource R is through the `RemoteInvoker`. Using the reflection API of Java it is possible to create a copy of R on host B, by instantiation a new object of the same type as R and then copying the fields of R (retrieved via the `RemoteInvoker`) to this type-compatible object.

The objects that remain on hosts A and B, will finally be garbage collected, as they are no longer referenced by any objects.

4.3.5 Multiple Agents Sharing the Same Resource and Garbage Collection

We explain here what happens when multiple agents share the same resource. Figure 4.14 on page 52 shows a situation where two agents (*Ma* and *Mb*) share the same resource. First agent *Mb* moves to host B. The *remote reference* migration policy is used for the resource R. When agent *Ma* moves then to another host C, the `RemoteInvoker` that *Mb* uses to reference R, is reused by mobile agent *Mb*. The distributed garbage collector of Java, takes care of the garbage collection of the `RemoteInvoker`. It uses a reference counting algorithm to remove the `RemoteInvoker` when it is not longer referenced.

4.3.6 A Concrete Example

To test our implementation of the dynamically switching of migration policies with roaming agents, we used the *EzAgent* [32] platform. This platform is a very primitive mobile agent platform for Java, based on RMI (it does not support inter-agent communication or any form of security). The main concepts in the EzAgent platform are *agents* and *places*. An agent is an active object, that spends its life in a so-called *agent-place*. Agents can be created in an agent-place, and can move between the different agent-places. An agent has a method called `on-arrival`, which is automatically executed when the agent arrives on an *agent-place*. Each agent also has an ID, this ID can be used to refer to the agent, and to move the agent to other locations.

To test our design and implementation, we created three places, and one agent that moves between those places. This agent has a reference to one resource, a vector. We made this vector an *adaptable type*. Thus when the vector is about to be serialized, the serialization process of the vector is trapped by its `SerializeMetaobject`. This metaobject has a simple `MigrationAdaptor`, which uses a random number generator to decide what migration policy to use when the vector is about to migrate.

When our agent arrives at a new place, he adds to the vector the name of this place. The migration policy used to migrate the resource is thus chosen randomly each time the agent moves. Below is the code that moves our agent between the

different agent-places (the agent-places have to be started from the command before running this code).

```
//lookup the places in the RMI registry
EzPlace place1 = (EzPlace)  Naming.lookup("//192.168.1.144/place1");
EzPlace place2 = (EzPlace)  Naming.lookup("//192.168.1.145/place2");
EzPlace place3 = (EzPlace)  Naming.lookup("//192.168.1.146/place3");

//create an agent with name agent1 on the first place
place1.createAgent("ezagent.TestAgent","agent1",  null);

// Move agent1 from place1 to place2
place1.moveAgent("agent1", "place2");

//Move agent1 from place2 to place3
place2.moveAgent("agent1", "place3");

//move agent back to place 2 (from place 3)
place3.moveAgent("agent1", "place2");
```

When we look at the above code, we can see that the agent is moved from place1 to place2, from place2 to place3, and finally from place3 back to place2.

Table 4.1 shows the transcript of each place during execution of the above program. Our random algorithm decided to use the following migration policies:

- move by copy from place1 to place2
- move by copy from place2 to place3
- move by reference from place3 to place2

When the agent arrives on place3, the vector in place1 only contains the element place1, as two times the *copy* policy was used when moving from place1 to place2, and from place2 to place3. After those two moves, the vector on place3 contains the elements place1, place2, place3. When the last move is done from place3 to place2, using the *network reference* policy, the vector stays on place3, and contains the elements place1, place2, place3, place2. Figure 4.15 on page 53 shows the structure of the metalevel when executing the above program. Below each host, we show the contents of the vector. We can see that place1 and place3 contain a copy of the vector after execution of the above program. The objects that remain in place1, and place2 (after moving the agent from place2 to place3) are garbage collected.

4.4 Summary

In this chapter we exposed how we designed the metalevel to achieve dynamically adaptable migration policies. The different metalevel entities in our design were introduced. We exposed the concept of *Migration Adaptor Component* (MAC) and showed how the main parts of this component can plugged transparently onto a base application using an XML specification. Two migration policies were implemented, *remote reference* and *copy*, as those are the most useful migration policies. We applied our metalevel design to the domain of ubiquitous computing and roaming

place1	place2	place3
Adding place1 to vector		
[place1]		
moving to place2		
switching to **copy policy**		
	agent arrived	
	[place1]	
	adding place2 to vector	
	[place1, place2]	
	moving to place 3	
	switching to **copy policy**	
		agent arrived
		[place1, place2]
		adding place3 to vector
		[place1, place2, place3]
		moving to place2
		switching to **reference policy**
	agent arrived	
	[place1, place2, place3]	
	adding place2 to vector	
	[place1, place2, place3, place2]	
[place1, place2]		[place1, place2, place3, place2]

Table 4.1: Example of a roaming agent moving between 3 places - transcript.

agents, and showed how the switching between *remote reference* and *copy* was achieved. Finally we showed that our design worked by implementing an example in the domain of roaming agents. We implemented a mobile agent which has a reference to a resource that moved between different locations, using the EzAgent platform. While our agent moved between the different places, we could successfully switch dynamically between different migration policies for the bound resource.

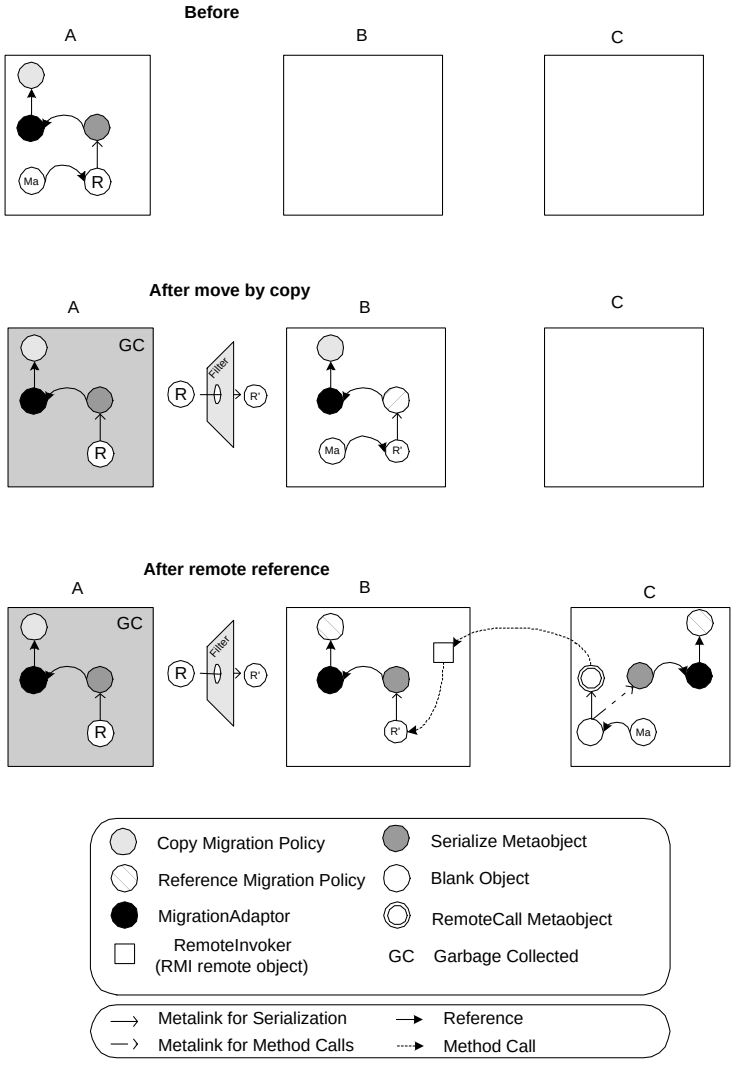

Figure 4.11: Switching from copy to remote reference

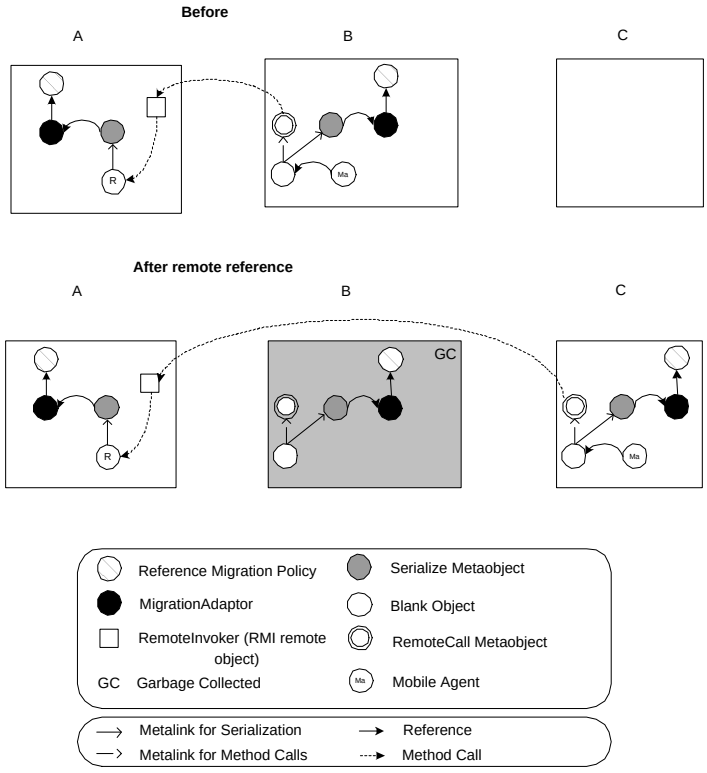

Figure 4.12: Switching from remote reference to remote reference

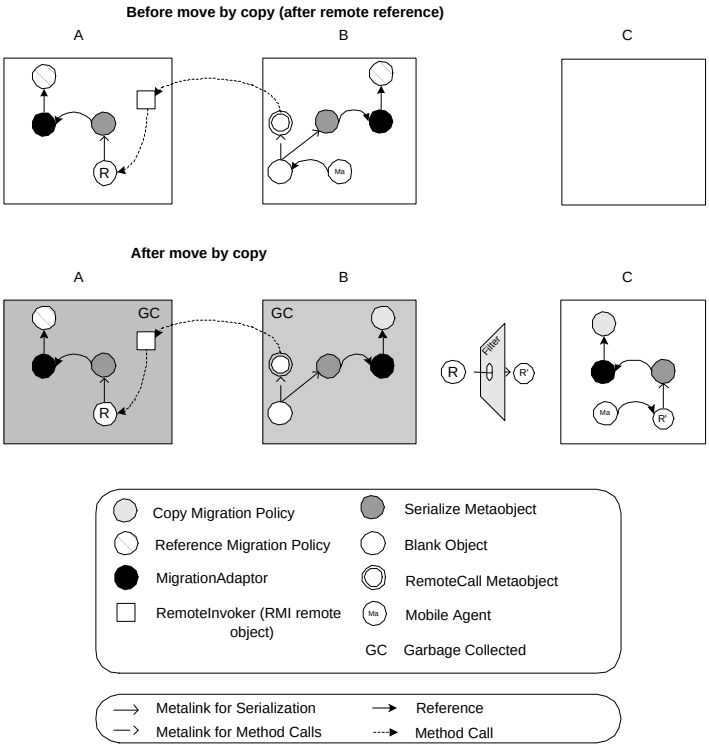

Figure 4.13: Switching from remote reference to copy

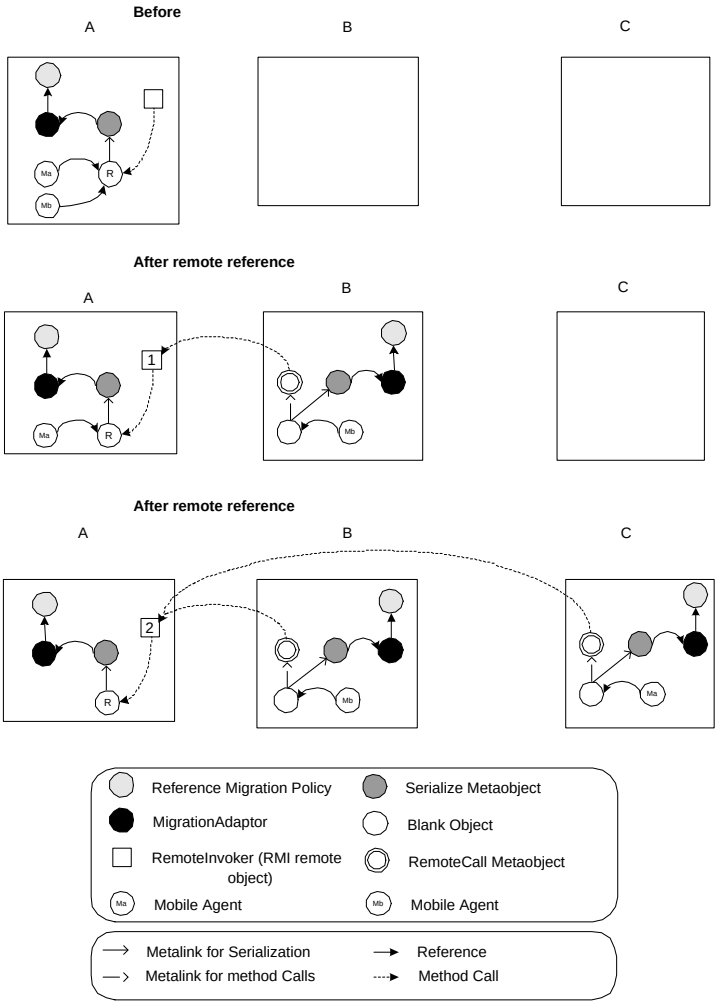

Figure 4.14: Multiple agents sharing the same resource

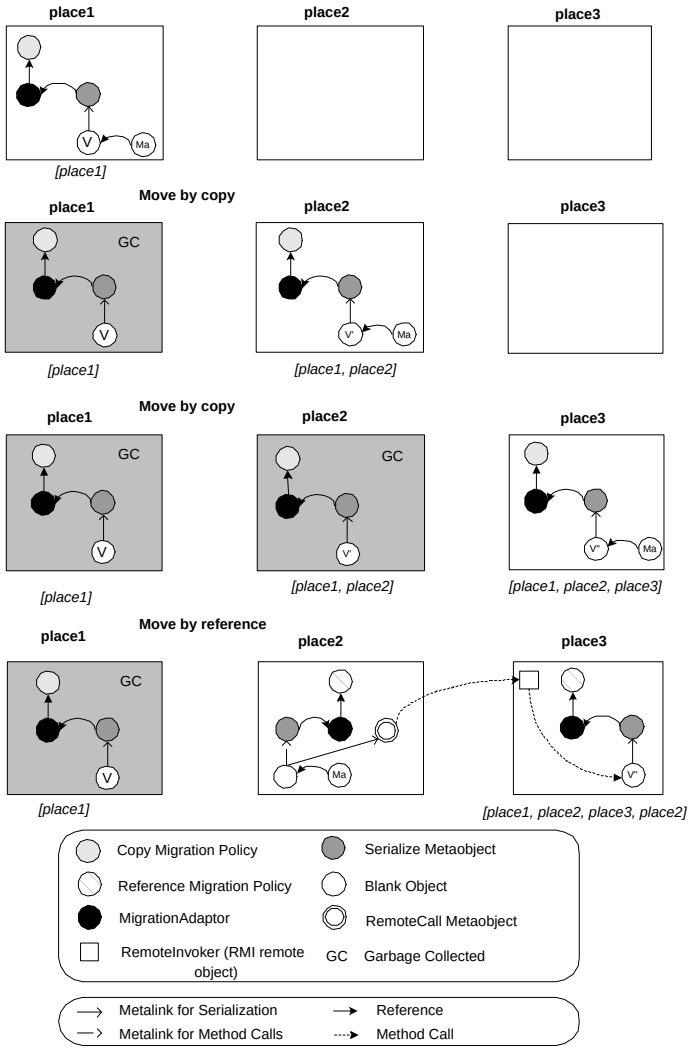

Figure 4.15: Example of a roaming agent moving between 3 places, metalevel design.

Chapter 5

A PIM Application

In this chapter we see how we can plug our metalevel model, that we exposed in the previous chapter, transparently onto any application. We start with a simple agenda application, and we show the different steps the application programmer has to follow to make the base application adapt to the changes in the environment when parts of it are migrating. The agenda application from which we start is a non-distributed application. After applying our metalevel library, the agenda becomes a distributed adaptable application useful in the world of ubiquitous computing, and this without making any changes to the base application.

We introduce different real-world scenarios to which our agenda application is adapted. Those scenarios are mainly based on the different types and characteristics of the client devices from which a person might want to access the agenda. The migration policies used for the transferable types in the agenda are adapted to the current scenario. We show which metalevel entities have to be implemented by the application programmer to achieve this adaptation, and how those entities can be transparently plugged onto to the base application using an XML specification.

Finally we expose how support for the disconnect operation can be added to the base agenda application, and how synchronization between objects on the client devices and the server can be done.

5.1 The Agenda Application

The agenda application we implemented is a simple application for organizing appointments. The user of this application can create, change and remove appointments from his agenda.

5.1.1 Design

The design of the application is very simple. It has the following objects (see also figure 5.1):

Appointment An appointment represents an appointment in the real world. It has the following fields:

- **Summary** a short description of the appointment

- **Description** a long description
- **Start** the start date and time
- **End** the end date and time

Day A day represents one day in a year. It is a collection that keeps the appointments for this day, and has methods for adding, retrieving, changing and removing appointments from the day.

Agenda The agenda is a collection which keeps instances of `Day`. The agenda has methods for adding, retrieving, changing and removing days and appointments.

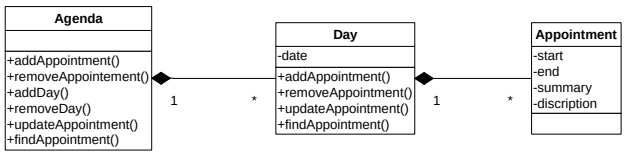

Figure 5.1: UML class diagram of the agenda application

5.2 Adaptation Scenarios

The agenda application, as it is now, is not useful at all in the world of ubiquitous computing. Typically a user of a digital agenda has several devices, like a fixed desktop computer, a portable computer, a PDA, and maybe even a cellular phone from which he wants to consult this agenda. To avoid replication of application data on each of the devices, we designed a centralized solution in which parts of the agenda application can be migrated from a central server to the different client devices.

The migration policy that is used to transport these migrating parts depends on the following scenarios (see also figure 5.2):

- **fixed computer** The user accesses the agenda from his office using a fixed desktop computer. As this kind of computer is connected through a wired network, we suppose that the network bandwidth is very high and reliable.
- **big portable computer** In this scenario, a big portable computer is used to consult the agenda. Such a computer usually has a lot of disk space and might be disconnected from the network.
- **a small portable** This kind of device usually has a small amount of memory, the network is unreliable and the bandwidth is low. An example of such a device can be a PDA or any other kind of hand-held computer. Typically such devices are often disconnected from the network. As we remarked in 3.1.2, those devices generally have low battery power. To save battery power, the devices might be disconnected from the wireless network, as being connected is very power consuming.

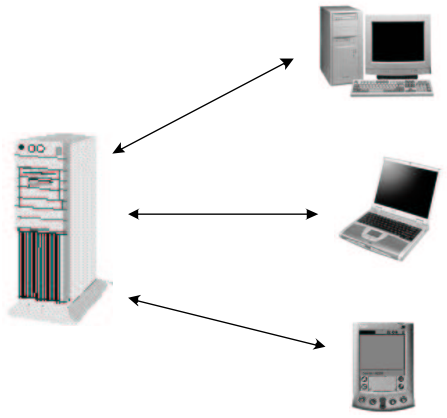

Figure 5.2: The three adaptation scenarios.

5.3 Adaptable Types and Migration Policies

Depending on one of the scenarios described above, we use different migration policies for the *adaptable types* of the application. We identify the following *adaptable types* and their migration policies in the agenda application:

- **Agenda** We use the *remote reference* migration policy if the client device is a fixed computer. In all the other cases we use the *copy with synchronization* policy.
- **Day**
  - If the client device is a big portable computer we use the *copy with synchronization* policy.
  - If the client device is a small portable, we use the *copy with synchronization* policy for the five relevant days[1]. In all the other cases we will use the *remote reference* policy. This way, if a disconnect occurs, the user can still access all the relevant days.

Note that the type `Appointment` is a *non-adaptable* type, as appointments can be transported also, but we don't specify a *MAC* for this type, which would make it able to switch dynamically between migration policies for this type. We could have chosen to make the `Appointment` type also *adaptable* but we prefer to keep this example as simple as possible to show that the idea is working. Types that are not *adaptable* are always transported by *copy*, as this is the default mechanism used by RMI [30].

If the `Agenda` type is referred remotely, then all access is done remotely (see figure 5.3). On the other hand, if the `Agenda` is transported by *copy*, then the non-relevant days are referenced remotely in the case of small portable devices (see figure 5.4), and in the case of a big portable device all the days are transported by *copy* (see figure 5.5). Thus if an instance of `Day` is transported by *copy*, all the instances of `Appointment` in that day are also transported by *copy*.

[1] The 5 relevant days are the current day and the four days after the current day

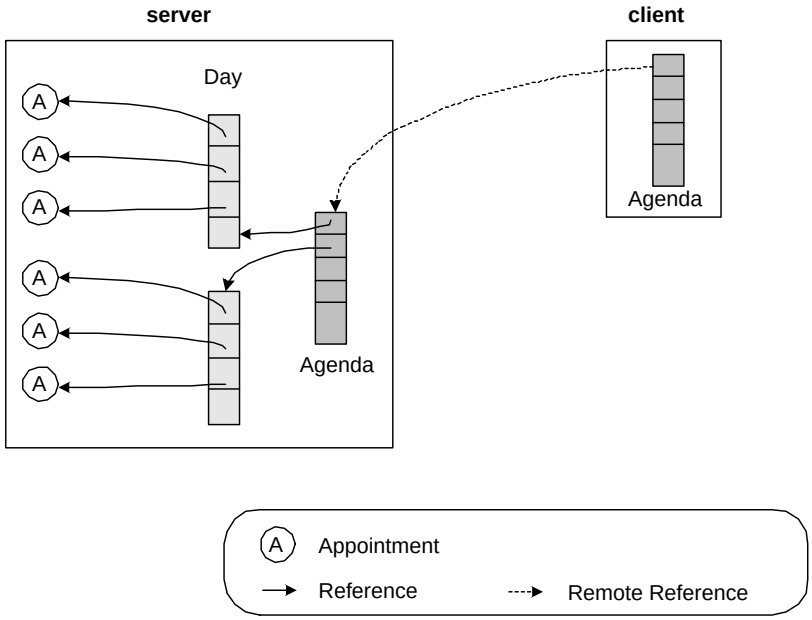

Figure 5.3: The adaptation strategy for fixed computers.
All the adaptable types are referenced remotely.

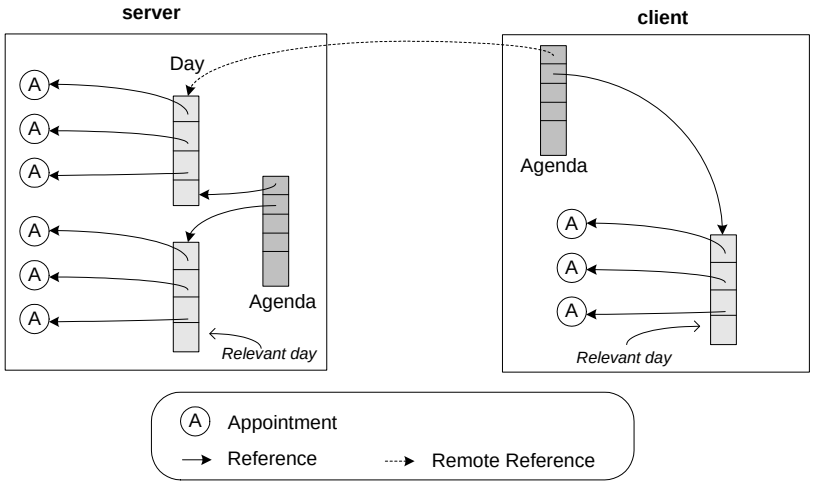

Figure 5.4: The adaptation strategy for small portable devices.
The agenda and the 5 relevant days are transported using the *copy with synchronization* policy. All the other days are referred remotely .

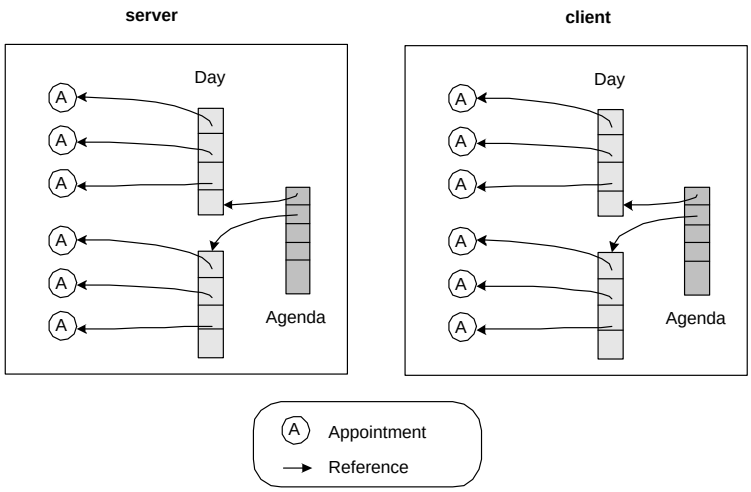

Figure 5.5: The adaptation strategy for big portable computers.
The agenda and all the days are transported using *copy with synchronization*.

5.4 Implementation of the Metalevel

5.4.1 Migration Adaptors

The application programmer has to implement a *migration adaptor* for each *adaptable type*. Several *adaptable types* can also share the same *migration adaptor*, but this is generally not the case, as each *adaptable type* probably uses different criteria to switch to another migration policy. The application programmer can create a new *migration adaptor* by sub-classing a class we provide, the `DefaultMigrationAdaptor`. This class has an abstract method `determinePolicy()` which has to be implemented in its subclasses. This method contains the code which determines the migration policy to use for its base object. It can reason about the state of its base object and information gathered from the Infrastructure Monitor or other sources of information, to make the decision.

To switch to another migration policy, the `DefaultMigrationAdaptor` has a method `setPolicy(aMigrationPolicy)`. This method just sets the `policy` field of the migration adaptor to `aMigrationPolicy` (we used the *state pattern* [15] here).

For the agenda application we created two migration adaptors: `DayMigrationAdaptor` and `AgendaMigrationAdaptor`. Those migration adaptors access the InfrastructureMonitor to determine the scenario we are in. Note that in our current implementation this Infrastructure Monitor is just a dumb object, a good implementation of this object is beyond the scope of this work.

AgendaMigrationAdaptor This *migration adaptor* just checks the type of the client device by accessing the Infrastructure Monitor and adapts the migration policy accordingly. Here is the pseudo code of its `determinePolicy()` method:

```
void determinePolicy()
{
  String clientDeviceType = InfrastructureMonitor.getType();
```

```
  if(type.equals("fixed"))
    setPolicy(new Reference());
  else
    setPolicy(new CopyWithSync());
}
```

As you can see from the above code, in the case of a fixed computer, the *remote reference* migration policy is used. In all other cases, we use the *copy with synchronization* migration policy.

DayMigrationAdaptor The `determinePolicy()` method of this migration adaptor does not only access the Infrastructure Monitor to determine the current scenario, but also reasons about the state of its base object. Here its base object is an instance of `Day`. In the case of a small portable device, we use the *copy* policy if the day is one of the five relevant days. Otherwise we use the *remote reference* policy. For fixed computers, we always use the *remote reference* policy, and for big portable computers, we always use the *copy with synchronization* policy when migrating instances of `Day`.

Here is the pseudo code of the `determiniPolicy()` method of the `DayMigrationAdaptor`:

```
void determinePolicy()
{
  String clientDeviceType = InfrastructureMonitor.getType();

  if( clientDeviceType.equals("fixed"))
    setPolicy(new Reference());

  if( clientDeviceType.equals("big_portable"))
    setPolicy(new CopyWithSync());

  if( clientDeviceType.equals("small_portable"))
  {
      Day currentDay = Calendar.getToday();
      Day inFourDays  = currentDay + 4;
      Day baseDay = (Day)this.getBase();

      //check if the days is a relevant day
      if(baseDay >= currentDay && baseDay <= inFourDays)
      {
        setPolicy(new CopyWithSync());
      }
      else
      {
        setPolicy(new Reference());
      }
  }
}
```

5.5 Network Disconnects and Synchronization

As the networks for wireless devices are generally very unreliable, a lot of disconnects can occur. Also a person might temporary want to disconnect his portable computer from the network when for example moving to another room, or to save battery power. Or he might go to a place where there is no network connection available, and want to continue there the work he was doing when he was online.

We successfully designed a reusable infrastructure for the agenda application to support this disconnect operation, without changing anything to the base application.

Detecting Disconnects The first problem to support the disconnect operation, is how to detect a disconnect without changing anything to the base application. To achieve this, we extend the two classes introduced in the synchronization protocol in 4.2.2: the `ApplicationServer` and the `ApplicationClient`. Like we noted in 4.2.2, the `ApplicationClient` gets a reference to the `ApplicationServer` by looking up the `ApplicationServer` in the RMI registry. Recall that the `ApplicationClient` can get a reference to the root of the application object graph (in the agenda application, this is the `Agenda` instance), using the `getRootApplicationObject()` method provided by the `ApplicationServer`.

To detect a disconnection, the `ApplicationClient` starts a thread (called `Connector`, see figure 5.6) which checks every X seconds if the `ApplicationServer` is still bound in the registry. This checking is done using the `Naming.lookup()` call. If this call does not respond after Y seconds, a disconnect has been detected. The user of the agenda application can then only access the appointments of the days that where transported by *copy*. He can make changes to the copies that are locally available, but this results in inconsistencies between the objects on the server and the objects on the client. Therefore the objects on the server have to be synchronized with the objects on the client upon reconnection.

The objects that were referred remotely are of-course not accessible during a disconnection. In the agenda application for example, when a disconnect occurs, the appointments in the days that were referred remotely are not visible to the user. Currently we don't notify the user about this, for the user it is just as there were no appointments during those days. However it might be possible to notify the user through the user interface, by for example putting the inaccessible days in a red color.

Synchronization A reconnection is detected by the `Connector` when the `Naming.lookup()` call returns before Y seconds have been passed. In order to do the synchronization, we use the mechanism described in 4.2.2.

In the case of the agenda application, we had to implement two *synchronizers* (recall from 4.2.2 that a *synchronizer* implements that `Synchronization` interface), one for each *adaptable type*: one for the `Agenda` type, one for the `Day` type.

Here is the implementation of the synchronize method for the `Agenda` synchronization class:

```
public void synchronize(Object oldObject, Object newObject)
{
    Agenda oldAgenda = (Agenda)oldObject;
```

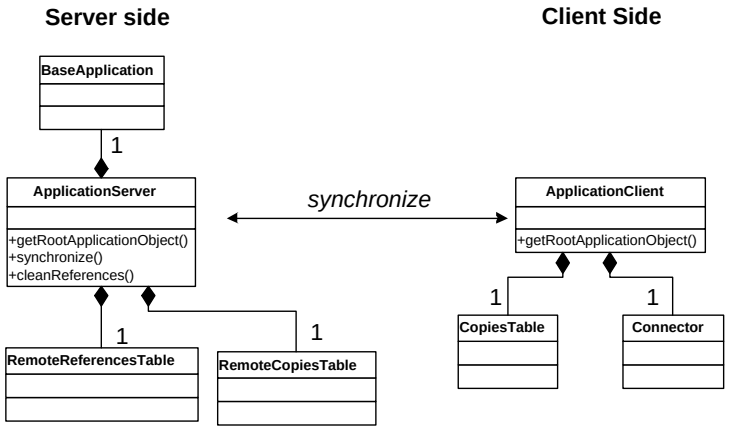

Figure 5.6: The Connector class to detect a disconnection.

```
        Agenda newAgenda = (Agenda)newObject;
        Iterator it = newAgenda.iterator();

        while(it.hasNext())
            {
                Day currentDay = (Day)it.next();
                oldAgenda.removeDay(currentDay);
                oldAgenda.addDay(currentDay);
            }
}
```

We iterate over all the days in the `Agenda` received from the client, and for each `Day` that way find in this `Agenda`, we replace that day on the server with the day in the `Agenda` from the client. The *synchronizer* or the `Day` type is very similar, we iterate over the day, and replace all the `Appointments`.

Remote References upon Disconnection When the client device is disconnected, the objects that were referenced remotely are of course not accessible. For implementing the *remote reference* migration policy (see 4.2.3) we used RMI [30] as the underlying protocol. When the client is connected, there are (remote) references to all the stubs on the server of `RemoteInvoker` (recall from 4.2.3 that the `RemoteInvoker` is an RMI object used to implement the *Remote Reference* migration policy). When the client disconnects those reference are gone. The distributed garbage collector of Java will notice this on some moment, and will garbage collect all the stubs. So when the client reconnects those stubs will be gone, and the remote reference will not be valid any longer.

We have to prevent the garbage collection of the objects that are referenced remotely to be able to implement the disconnect operation. The only way to do this is to keep an extra reference to all the instances of `RemoteInvoker` (in fact the stubs at the server side). Therefore we keep a table (`RemoteReferenceTable`

in figure 5.7) in the `ApplicationServer` which keeps references to those objects. Every time a new `RemoteInvoker` is created, we add a reference in this table to this new instance. Thus when a client reconnects, the `RemoteInvoker` will be still alive, and the remote reference will work. Of course this table has to be emptied when the client shuts down the application, so that all the `RemoteInvoker` instances will be garbage collected.

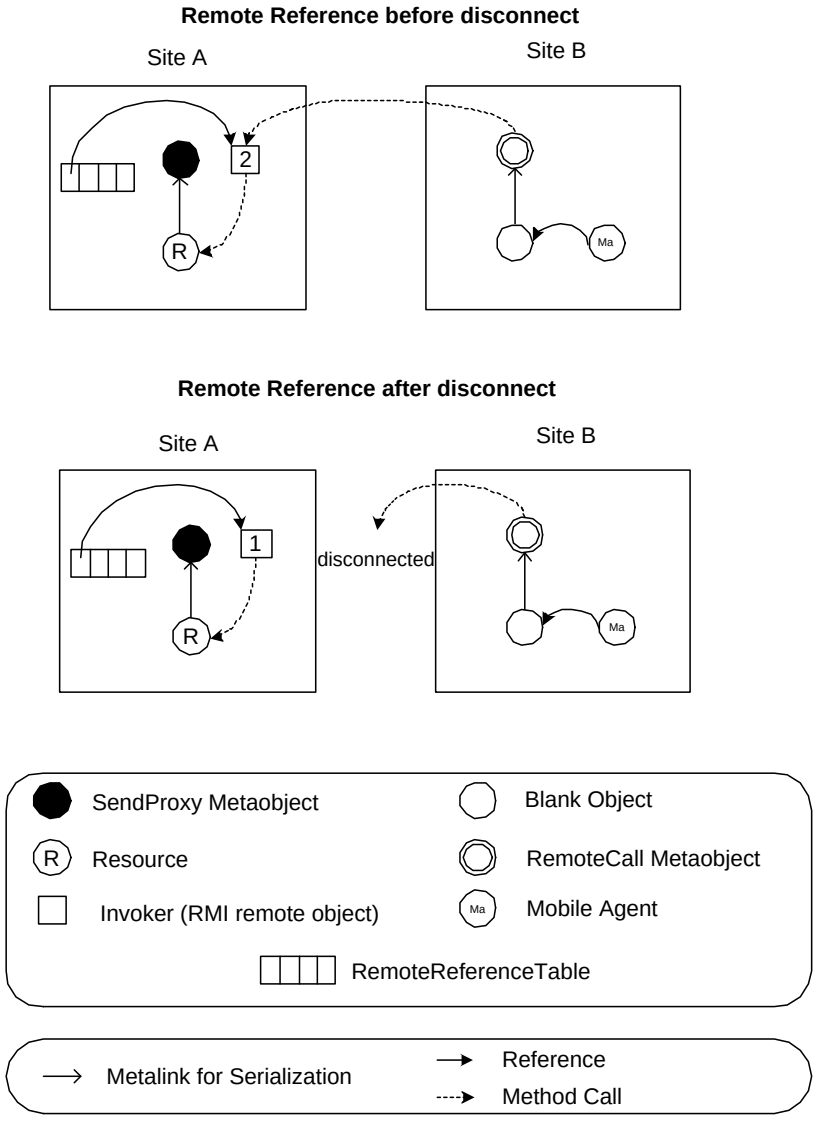

Figure 5.7: Implementation of the disconnect operation.

5.6 The XML Specification

To associate the metalevel entities with the base level, and to associate the *synchronizers* with the corresponding types, we use an XML specification (like we explained in 4.2.2). Figure 5.8 shows this complete specification for the agenda application. As we can see from this figure, we associate the type `Agenda` with the `AgendaMigrationAdaptor`, and the type `Day` with the `DayMigrationAdaptor`. We can also see that the `Agenda` uses the `AgendaSynchronizer`, and the `Day` the `DaySynchronizer` to synchronize between client and server objects.

```
<application name="pim agenda application">
 <transferable>
  <adaptable>
   <type name="pim.Agenda">
      <migrationadapter>"pim.migrationadaptors.AgendaMigrationAdaptor"</ migrationadapter>
      <sync>"pim.sync.AgendaSynchronizer"</ sync>
   </type>
   <type name="pim.Day">
      <migrationadapter>"policies.testing.DayMigrationAdaptor"</ migrationadapter>
      <sync>"pim.sync.DaySynchronizer"</ sync>
   </type>
  </adaptable>
 </transferable>
</application>
```

Figure 5.8: The XML specification for specifying the MAC and *synchronizers* for the types in the agenda application.

5.7 Deployment

To apply our metalevel model described in chapter 4 onto a base application (in our example the agenda application), we had to follow the following steps:

- Identify the *adaptable types* of the base application.
- Implement the `MigrationAdaptors` for the identified *adaptable types*.
- Implement the `Synchronization` interface for the types that have to be synchronized.
- Create the XML specification to specify which `MigrationAdaptor` and *synchronizer* to use for which type.
- Bootstrap the agenda application with the `ApplicationClient` and `ApplicationServer` classes.

We tested our implementation on two fixed computers, using Linux as operating system and JDK 1.3 [27]. One computer acted as server, and one as client. On the server an instance of the `ApplicationServer` is initialized with an instance of `Agenda` as the root application object.

The `ApplicationClient` resides on the client computer. This program gets the agenda root object from the server, by calling the `getRootApplicationObject`. This method gets the root of the agenda application (the `Agenda` instance) from the server, and transports it to the client. This is the starting point of our adaptation. The `SerializeMetaobject` of the `Agenda` type, will trap the serialization process of the `Agenda`. Depending on the current scenario, like described above, the *remote reference* or *copy with synchronization* migration policy will be used. When the `Agenda` is serialized, the serialization process of the instances of `Day` and `Appointment` that the `Agenda` contains, will also be trapped. The type `Day` is an *adaptable type*, so the serialization semantic will depend on the current scenario we are in, like for the `Agenda` type. For the `Appointment` type, the *copy* migration policy will always be used, as we decided to make this type not adaptable. Both, the `ApplicationServer` and `ApplicationClient` programs, have to be run with the reflective metalevel attached to it. We can do this using our `Run` program (see 4.1.5), which parses the XML specification of the agenda application, and applies our metalevel library onto it.

5.8 Summary

In this chapter we exposed how we can apply our metalevel library transparently onto any base application, to make the base application adaptable. In particular, we explained how we can dynamically adapt migration policies for the parts of the base application that are subject to migration. We introduced three different scenarios to which those migration policies were adapted: *fixed computers* with good network connection, *big portable computers* with a lot of memory, and *small portable computers* (like PDA's) with an unreliable network, small memories, and a lot of disconnects.

We also explained how the disconnect operation can be implemented without making any changes to the base application, and how synchronization between the client and the server can be achieved transparently.

Summarized the following steps have to be taken to add adaptability to a base application using our framework:

- Identification of *adaptable types*.
- Implementing the `MigrationAdaptors` for those types.
- Create *synchronizers* for the types that have to be synchronized.
- Create an XML specification to bind types with *migration adaptors* and *synchronizers*.
- Bootstrap the application with the provided classes (`ApplicationClient` and `ApplicationServer`).
- Run the client and server program with the `Run` class we provided to make them reflective.

Note that we don't have to make any changes to original code of the base application to apply our framework. Very few extra application specific code has to be written, and all this code is factored out very well.

Chapter 6

Conclusions

6.1 Achievements

The main goal of this thesis was to achieve dynamic adaptable migration policies in mobile programs, using the reflective system Reflex. The following achievements were made in this work:

- We created a metalevel library to be able to dynamically switch between different migration policies when parts of an application are migrating. In particular the *remote reference* and the *copy* migration policies are supported. To make the decision to switch to another migration policy, different sources are accessed: the Infrastructure Monitor and the state of the objects that are about to migrate. All this could be done at the metalevel only.
- This metalevel library has been applied in the world of *roaming agents* and *ubiquitous computing*, but can be used in any mobile code system.
- We created an infrastructure to support the disconnect operation, and synchronization between objects on a server and replications of those objects on a client.
- Our library can be plugged transparently onto any base application. This means that we don't have to make any changes at all to the code of the base application. Only few straightforward extra code is needed, and all this code has been factored out very well. Recall that the application programmer should create *migration adaptors*, *synchronizers*, and the two classes to bootstrap the base application (`ApplicationClient` and `ApplicationServer`) by implementing interfaces we provided.
- To validate our work, we implemented a simple agenda application, to which we applied successfully our metalevel library and our infrastructure to support the disconnect operation and synchronization. We introduced three real-world scenarios to which the migration policies for migrating parts of the agenda were adapted.

6.2 Limitations and Future Work

Infrastructure Monitor This is the source of information we used to make decisions upon migration. In the current implementation this is just a dumb object. This object should be able to give information about the network connection, and the device to which parts of the application migrate. For example the type of the device (PDA, fixed computer, ...), available memory, ... We should look for existing systems that offer a similar infrastructure, and try to integrate it in our work.

Shutting Down the Client Application when Off-line A remaining issue is that of supporting client application shutdowns. When the client device is disconnected, it is not possible to shut down the client application without losing all the changes made to local copies on the client. If changes are made to objects that are local on the client device, all the changes will be gone after shutting down the application, as no synchronization with the server can be done because the client device is off-line. The local copies on the client device should be saved on disk before closing the application, and synchronized with the server when restarting the application and connected.

Conflicting Replicas It can be that one person makes changes to remote referenced appointments in the agenda when connected from his fixed computer, while another person makes changes to the local copies of the same appointments from another device (for example from a PDA). When the PDA reconnects, the synchronization with the server will be done. However, the synchronization protocol will not detect that the objects on the server were changed by the first person from his fixed computer. This means that the changes made by this person will be overwritten by the changes made by the second person from the PDA upon synchronization. This might not exactly be what we want, for example the changes made from the fixed computer might have preference over the changes made from the PDA. A system should be developed to detect and resolve those conflicts.

Note that in our case (the agenda application), we focused on the scenario where one person owns the different devices, and thus he only accesses the agenda application from one device at a time. Therefore no concurrent sessions are possible, and the problem of conflicting replicas doesn't exist.

Scalability Tests Currently we only tested our work with two fixed computers, both using Linux as operation system, with JDK 1.3 installed. We should test our framework more intensively. We could for example use other devices with a Java Virtual Machine installed (for example a PDA) and with other types of network connections (for example a more unreliable wireless connection). Also the example agenda application we implemented had a very simple application object graph (an `Agenda` which contains `Days`, and `Days` that contain `Appointments`). Some analysis and tests should be done to see how our approach can be scaled to applications with a more complex object graph.

Application Partitioning In our work, we decided to partition the application, based on the types of the objects. We could declare types to be *adaptable* using an XML specification. Other types of application partitioning should be investigated. J-Orchestra (see [35]) for example, partitions the application using a

classification algorithm, that can detect if a class is *anchored* (this are system classes that have native methods) or *mobile* (classes that are not anchored). J-Orchestra also provides the user a simple Graphical User Interface to allow specify its custom partitioning.

A similar user interface could be build for our system to generate the XML specification.

Static Definition of Adaptable Types We use an XML specification to specify the *adaptable types*. This means that all the instances of one type are reflective when this type is declared as *adaptable* in the XML specification. However we can always change or remove the metaobjects of an instance at runtime to stop adapting it.

Performance It is obvious that we loose performance by having an extra layer at the metalevel. But this performance loss is negligible in distributed computing, as real performance problems generally come from network latency.

For implementing the disconnect operation, we keep for every `RemoteInvoker` instance a reference in a table, in order to not garbage collect these instances when the client disconnects. This of-course is quite costly in terms of memory usage. Another solution could be to create only one instance of `RemoteInvoker` for all objects that are referenced remotely. But then we have to pass for each method invocation from the client to the server an extra ID, so that the `RemoteInvoker` knows to which object he has to forward the method call. We also need an extra table on the server to retrieve the resource that corresponds to this ID. This solution is maybe better in terms of memory usage, but for each method call we need to send an ID over the network (which means more network traffic) and we need an extra lookup in the table to retrieve the resource on which the method has to be invoked. Another disadvantage of this solution is that the socket type has to be the same for each object type, as we only have one `RemoteInvoker` instance. This means that we can not use compression sockets for one object type, and normal or encryption sockets for another object type.

6.3 Conclusion

In this work we successfully created a metalevel library that can add dynamically adaptable migration policies to a base application, using the reflective framework Reflex. This metalevel library can transparently be plugged onto any base application, without having to make any changes to the base application.

We showed the design of this metalevel library in the world of *ubiquitous computing* and *roaming agents*. We introduced all the metalevel entities used in this library, and we explained how this library can transparently be plugged onto a base application using an XML specification.

Finally we saw, in a case study, the different steps to transform a non-distributed base application into a distributed application with dynamically adaptable migration policies by applying our metalevel library onto it. We also exposed how synchronization could be done between the client and the server, and how the disconnect operation could be supported.

Bibliography

[1] Ubiquitous computing, http://www.ubiq.com/.

[2] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 111–130. Springer, April 1997.

[3] M. Ancona, W. Cazzola, G. Dodero, and V. Gianuzzi. Channel Reification: a Reflective Model for Distributed Computation. In *Proceedings of IEEE International Performance Computing, and Communication Conference (IPCCC'98)*, pages 32–36. IEEE, February 1998.

[4] D. G. Bobrow, R. G. Gabriel, and J. L. White. CLOS in Context – The Shape of the Design Space. In *Object Oriented Programming – The CLOS Perspective*. MIT Press, 1993.

[5] J. P. Briot and P. Cointe. Programming with Explicit Metaclasses in SmallTalk-80. In *Proceedings of OOPSLA'89*, volume 24 of *Sigplan Notices*, pages 419–431. ACM, October 1989.

[6] W. Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98), 12th European Conference on Object-Oriented Programming (ECOOP'98)*, 1998.

[7] S. Chiba. Load-time structural reflection in java. In E. Bertino, editor, *ECOOP 2000 - Object Oriented Programming + 14th European Conference*, number 1850 in Lecture notes in computer science, pages 313–336, Sophia Antipolis and Cannes, France, june 2000. ECOOP 2000, Springer-Verlag.

[8] P. Cointe. MetaClasses are first class objects: the ObjVLisp model. In *Proceedings of OOPSLA'87*, volume 22 of *Sigplan Notices*. ACM, October 1987.

[9] P. Cointe. A tutorial introduction to metaclass architecture as provided by by class oriented languages. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 592–608, Tokyo, Japan, November 1988. Tokyo and Springer-Verlag.

[10] Iain Craig. *The Interpretation of Object-Oriented Programming Languages*. Springer Verlag, 2001.

[11] J. Malenfant F.-N. Demers and M. Jacques. A Tutorial on Behavioral Reflection and its Implementation. In *Proceedings of Reflection '96*, 1996.

[12] J. Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of OOPSLA'89*, volume 24 of *Sigplan Notices*, pages 317–326. ACM, October 1989.

[13] M. Fowler. *UML Distilled.* Object Technology Series. Addison-Wesley, 1997.

[14] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. In *IEEE Transactions on Software Engineering*, volume 24(5), May 1998.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley professional computing series, 1995.

[16] R. H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1:6–17, 1994.

[17] G. Kiczales, J. Des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[18] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[19] M. Satyanarayana Lilly B. Mummert, Maria R. Ebling. Exploiting weak connectivity for mobile file acces. Technical report, School of Computer Science Carnegie Mellon University, 1996.

[20] P. Maes. *Computational Reflection.* PhD thesis, Vrije Universiteit Brussel, 1987.

[21] P. Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA'87 Proceedings*, pages 147–155, Orlando, Florida, 1987.

[22] Brian D. Noble, Morgan Price, and Mahadev Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. Technical Report CS-95-119, 1995.

[23] Mudumbai Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware Mobile Programs. In *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA, 1997.

[24] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Symposium on Principles of Distributed Computing*, pages 1–7, 1996.

[25] B. C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, January 1984.

[26] Sun Microsystems, Inc. The Java Language Specification, 1996.

[27] Sun Microsystems, Inc. The Java Development Kit. `http://java.sun.com/products/jdk/`, 1999.

[28] Sun Microsystems, Inc. Object Serialization. `http://java.sun.com/products/jdk/1.3/docs/guide/serialization/`, 2000.

[29] Sun Microsystems, Inc. Reflection API Documentation. `http://java.sun.com/products/jdk/1.3/docs/guide/reflection/`, 2000.

[30] Sun Microsystems, Inc. Remote Method Invocation. `http://java.sun.com/products/jdk/1.3/docs/guide/rmi/`, 2000.

[31] Ado Adobe Systems. Postscript language reference manual.

[32] Eric Tanter. Reflex, a reflective system for java – application to flexible resource management in mobile object systems, msc thesis, August 2000.

[33] Eric Tanter, Noury Bouraqadi, and Jacques Noyé. Reflex – Towards an Open Reflective Extension of Java. In *Proceedings of the Third International Conference on Metalevel Architectures and Advanced Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 25–43, Kyoto, Japan, September 2001. Springer-Verlag.

[34] Eric Tanter and José Piquer. Managing references upon object migration: Applying separation of concerns. In *Proceedings of theXXI International Conference of the Chilean Computer Science Society (SCCC 2001)*. IEEE Computer Science Press, November 2001.

[35] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. Technical report, Center for Experimental Research in Computer Science (CERCS),College of Computing Georgia Institute of Technology, Atlanta, June 2002.