Department of Computer Science
University College London
University of London

Adapting Mobile Systems Using Logical Mobility Primitives

Stefanos Zachariadis
s.zachariadis@cs.ucl.ac.uk

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

May 2005

Abstract

Mobile computing devices, such as personal digital assistants and mobile phones, are becoming increasingly popular, smaller, more capable and even fashionable personal items. Combined with the recent advent of wireless networking techniques, users are equipped with mobile devices of significant computational abilities, which are able to wirelessly access information by dynamically connecting to many different networks. Despite the *ubiquity* of mobile devices, mobile systems are built using monolithic architectures, use a small set of predefined interaction paradigms and do not exploit or adapt to the dynamicity of their local or remote context.

Applications deployed on mobile devices face considerable challenges posed by their changing surroundings. One of the main peculiarities of mobile devices is heterogeneity, which may occur in software, hardware and network protocols. Mobile systems may carry a large number of different applications, use different operating systems and middleware and, often, have more than one network interface. A further challenge is their considerable variation in the computational resources available, such as battery power, CPU speed, network bandwidth and volatile and persistent memory. Moreover, mobile computing systems are *highly dynamic systems*, in terms of their surroundings, implying that the requirements for applications deployed on a mobile device are a moving target. Changes in the requirements (such as integration with a new service) may require changes to the application. Consequently, these changes may mean that the application behaviour needs to *adapt*.

This thesis argues that the potential of the ubiquity of mobile devices cannot be realised using *static* and monolithic architectures, as mobile systems need to be able to adapt to accommodate changes to their environment. It investigates the use of three technologies to offer adaptation to mobile devices: Logical mobility techniques, component systems and middleware technologies. More specifically, this thesis presents the SATIN (System Adaptation Targeting Integrated Networks) component metamodel, a lightweight local component metamodel that offers the flexible use of logical mobility primitives. The metamodel is instantiated to build the SATIN middleware system, a component-based mobile computing middleware that uses the mobility primitives exported by the metamodel to reconfigure itself and applications running on top of it.

The suitability of SATIN for the creation of adaptable mobile systems is demonstrated, by using it to implement and evaluate a number of applications showing different aspects of adaptation. Moreover, existing projects are reengineered to run as SATIN components, showing the flexibility of the approach and the advantages gained over the originals.

Acknowledgements

My advisors, Cecilia Mascolo and Wolfgang Emmerich; starting from my undergraduate project, on to my research assistantship and throughout my PhD, their help, support and friendship was and is invaluable.

Stephen Hailes, Licia Capra, Peter Kirstein, Lionel Sacks and Saleem Bhatti; for advice, help, ideas and encouragement in producing this work.

Martin Ijaha; for developing a mobile code toolkit.

My examiners, Gordon Blair and George Roussos; thank you.

UCL and the Department of Computer Science; for making my stay here very enjoyable.

There are many wonderful people that, in one way or another, have helped me complete this work.

Νίκο και Νίτσα, Licia, Cecilia, Γιώργο, Νίκη, Ειρήνη, Τόμη, Στάθη, Νίκο Β., Δημήτρη, Ήβη, Ντίνα, Νίκο Κ., Αλέξη, Εύη, Δανάη, Θέμη, Richard, Luca, Nima, Mirco, Suzy, Andrea, James, Clovis, Daniele, Rami, Genaina, Σπύρο, Όλγα, Φέη, Νικολέττα, Christian, Daniel, Ελίνα, Δάφνη, Δέσποινα, Μαρίνα, Torsten, Carina, Σωκράτη, Costin, Vladimir, Manish, Andrea, Gian Pietro, Στράτο, Maike, Mohamed, Μαριλίζα, Ρένα; many, many, many thanks.

This work was kindly sponsored by EPSRC grant GR/R70460.

στους γονείς μου.

Contents

1 Introduction **1**

1.1 Hypothesis Statement and Contributions 3

1.1.1 Contributions 4

1.1.2 Related Topics 6

1.2 Thesis Outline 6

2 Motivation **9**

2.1 Motivating Examples 9

2.1.1 Case Study: Mobile Application Development and Deployment in Practice 10

2.1.2 Case Study: Application Deployment and Update in a Mobile Environment 11

2.2 Concerning Adaptation 12

2.2.1 Research Perspective 13

2.2.2 In Conclusion: Adapting Mobile Systems 17

2.3 Assumptions 18

3 Logical Mobility Primitives for Mobile Computing **21**

3.1 Physical Mobility and Mobile Networking Paradigms 21

3.2 Defining Logical Mobility 22

3.2.1 Introduction to Code Mobility 23
3.2.2 Paradigms of Code Mobility 25
3.2.3 Application Domains 27
3.2.4 Summary: Benefits of Logical Mobility for Mobile Reconfiguration . 29
3.3 Conceptualising the Use of Logical Mobility over Physical Mobility 30
3.3.1 Requirements 31
3.3.2 The Logical Mobility Unit 32
3.3.3 A Framework for Logical Mobility 35
3.3.4 Transferring Logical Mobility Units 46
3.4 Related Work 50
3.5 Summary 51

4 The SATIN Component Meta Model for Mobile Adaptation 53

4.1 Models, Metamodels and Instances 53
4.2 Components, Distribution and Collocation 55
4.3 A Component Model for Mobile Adaptation 57
4.3.1 Component Model Overview 57
4.3.2 Components 58
4.3.3 Components and Containers 60
4.3.4 Distribution and Logical Mobility 60
4.3.5 Component Lifecycle 62
4.4 Static Semantics 63
4.5 Concrete Semantics 64
4.6 Discussion 67
4.7 Related Work 68
4.7.1 In-process Component Models 68

4.7.2 Distributed Component Models 71
4.8 Summary 73

5 The SATIN Mobile Computing Middleware System 75

5.1 The SATIN Middleware System 76
5.1.1 Middleware System Overview 76
5.1.2 Required Attributes 77
5.1.3 Middleware Services 77
5.1.4 Application Components 81
5.1.5 An Object-Oriented Attribute System 81
5.2 Discussion 82
5.3 Related Work 84
5.3.1 Logical Mobility for Application Reconfiguration 84
5.3.2 Middleware Reconfiguration 87
5.4 Summary 88

6 Implementation and Evaluation 91

6.1 On Evaluating SATIN 91
6.2 Quantitative Evaluation: Implementation 93
6.3 Quantitative and Qualitative Evaluation: Applications 95
6.3.1 The SATIN Program Launcher 95
6.3.2 The SATIN Music Player 99
6.3.3 The SATIN Scripting Framework 101
6.3.4 Code Fragments 101
6.3.5 Summary 103
6.4 Qualitative Evaluation: Q-CAD 103

6.4.1 Case Study 105

6.4.2 The Q-CAD Model 107

6.4.3 Q-CAD Architecture 117

6.4.4 Summary 120

6.5 Qualitative Evaluation: ZION 121

6.6 Discussion 123

6.7 Summary 124

7 Conclusion and Future Work **127**

7.1 Contributions 128

7.2 Critical Evaluation 129

7.3 Future Work 129

7.4 Further Dissemination 132

7.5 Conclusions 133

A Q-CAD Metadata Encoding **135**

A.1 Application Profile 135

A.2 Utility Function 138

B Programming With SATIN **141**

B.1 Transferring Components 141

B.2 Advertising, Discovering and Requesting Components 145

References **151**

List of Figures

2.1 (a) Deploying applications (ring tones and games) on mobile phones. (b) Deploying and maintaining applications from peers: Dotted lines represent certificate downloads. Solid lines represent update downloads. 12

2.2 A Typical View of a Mobile System. 16

2.3 A taxonomy for mobile adaptation (adapted from [Satyanarayanan, 1996]). 17

3.1 A mobile code system, showing the transfer of a unit from Host B to Host A. Adapted from [Picco, 1998c]. 24

3.2 The Logical Mobility Unit. 32

3.3 A Logical Mobility Framework. 35

3.4 The specification of the Trust & Security Layer. 37

3.5 A state machine representing the Trust & Security Layer. 37

3.6 The specification of the Serialisation & Deserialisation Engine. 38

3.7 A state machine representing the Serialisation & Deserialisation Engine. . . 38

3.8 The specification of the Controller. 39

3.9 A state machine representing the Controller. 39

3.10 The specification of the Sender/Receiver. 40

3.11 A state machine representing the Sender/Receiver Controller. 41

3.12 A state machine representing the Sender. 41

3.13 A state machine representing the Receiver. 42

3.14 The specification of the API. 42

3.15 The specification of an application receiving an LMU. 43

3.16 A state machine representing an application receiving an LMU. 43

3.17 Safety and liveness properties for the framework. 44

3.18 A state machine representing the safety property `SERIALISETRUSTED`. . . . 44

3.19 A state machine representing the safety property `SENDSERIALISED`. 44

3.20 A state machine representing the safety property `INSPECTSERIALISED`. . . . 45

3.21 A state machine representing the safety property `DEPLOYACCEPTED`. 45

3.22 The full specification of the framework. 46

3.23 A composition of two instances of the logical mobility framework. 46

4.1 The relationship between objects, models, metamodels and meta meta models. 54

4.2 The SATIN metamodel. 57

4.3 A list of suggested attributes for the SATIN component model. 59

5.1 The architecture of the SATIN middleware system. 76

5.2 The SATIN middleware system advertisement and discovery framework. . . . 79

5.3 The SATIN middleware system attributes architecture. 81

6.1 Details on the SATIN implementation. 94

6.2 The SATIN Program Launcher. 96

6.3 The SATIN Program Launcher: Showing what components are advertised on all networks, including those of the local host. 96

6.4 The SATIN Program Launcher: Component “STN:TESTAPP” was installed from a remote host and is displayed by the Launcher. 97

6.5 The SATIN Launcher as a collection of components. 97

6.6 The testing configuration. The dotted line represents the wireless network. The solid line represents the wired ethernet. 98

6.7 The SATIN Music Player. 99

6.8 The SATIN Music Player: Playing a remote stream. 100

6.9 The SATIN Shell. 100

6.10 Q-CAD in the SATIN middleware system. 104

6.11 Service Discovery Scenarios. 106

6.12 Example of Resource Descriptor. 108

6.13 Application Profile - Example of Proactive Encoding. 110

6.14 Application Profile - Example of Reactive Encoding. 112

6.15 Example of a Utility Function. 113

6.16 3-Step Resource Discovery Protocol. 114

6.17 Proactive and Reactive Discovery Protocol Examples. 116

6.18 A High-Level Overview of the Q-CAD Architecture. 117

6.19 The Q-CAD Architecture. 118

6.20 The ZION Architecture. Taken from [Chatterjee et al., 2004]. 122

Chapter 1

Introduction

According to Mark Squires (Nokia), it took 15 years for the television to reach a critical mass of 50 million users, but it took the mobile phone industry only 18 months to sell 50 million handsets in Europe alone. In recent years, mobile devices such as mobile phones, Personal Digital Assistants (PDAs), laptop computers etc, are becoming increasingly popular, leading to a further and rapid decentralisation of computing, with devices becoming more capable, cheaper, mobile and even fashionable personal items. The recent advances in wireless networking (UMTS, Bluetooth [Mettala, 1999], 802.11 [V. Hayes, 1996] etc.), combined with the popularity of mobile devices, allow users to carry sophisticated computing environments which facilitate access to local and remote information *on the move*.

Mobile computing introduces radical new usage paradigms. Whereas a desktop computer user would typically run an application (such as a word processor) for long periods of time, a mobile user would usually run applications for short periods but frequently (e.g. to check the daily schedule), using different input and output mechanisms (e.g. input styli and small resolution screens). In traditional computing systems, application developers can often assume that their software will be executed by a powerful machine, which is always or easily connected to a centralised network using a high bandwidth link. Similarly, traditional distributed systems are usually composed of powerful nodes, interconnected using high bandwidth links over a fixed topology. Mobility breaks this *static* model, as mobile devices are considerably less powerful in terms of computational resources available, such as CPU speed, network bandwidth and volatile and persistent memory. Moreover, the mobile network topology is considerably more *dynamic*, as nodes may come and go freely and frequently, dynamically aggregating into various hybrid, independent and even incompatible (Infrared and Bluetooth for example) networks. Although devices are becoming increasingly more capable, they will, for the foreseeable future, lag behind their fixed counterparts with respect to their available resources (especially power supplies and network connectivity, which will be somewhat intermittent).

Mobile computing systems can be *highly heterogeneous*. This heterogeneity occurs in software, hardware and network protocols. Mobile devices carry a large number of different applications, use different operating systems and middleware and often have more than one network interface. Moreover, mobile applications are exposed to a *highly dynamic environment*, in terms of their local and remote context. Consequently, the requirements for mobile systems are a moving target. Changes in the requirements (such as integration with a new service) may require changes to the application. Consequently, these changes may mean that the application behaviour needs to *adapt*.

The current state-of-practise for developing software for mobile systems offers little flexibility to accommodate such heterogeneity and variation. Thus, application developers have to decide at design time what possible uses their applications may have; the applications do not change or adapt once they are deployed on a mobile host. In fact, mobile applications are currently developed in monolithic architectures, which are more suitable for a fixed execution context rather than a dynamic, mobile one.

Consider the recent exit of Sony from the global PDA market, despite commanding a significant percentage of this market [Kort and Dulaney, 2004]. There were various reasons given for this move. Sony stated [PalmInfocenter, 2004] that

> "they view wireless communications features as a key pillar to their business strategy and that they plan to continue their collaboration with Sony Ericsson."

This statement translates to that although Sony are exiting the traditional PDA market, they are going to continue shipping mobile computers in the form of smartphones, which are essentially PDAs with cellular networking connectivity. This shows that networking connectivity and access to information is pivotal in the mobile device market.

IDC claimed [Solheim, 2004] that the reason for Sony's exit is the saturation of the market. In particular, it was claimed that

> "the PDA really has not been able to redefine itself from the PDA of the 1990s to the PDA of 2000."

This statement, although vague, merits some analysis. Personal Digital Assistants were pioneered by Apple Computer in 1993, when the Newton Messagepad was launched. A portable general purpose computer, the Newton had the ability to connect to a desktop computer in order to install more applications or to synchronise changes to shared data (such as a to-do list) between applications on the PDA and the desktop computer. There have been various different PDAs developed and shipped ever since. A large percentage of them, including various Newton models, have had at least one form of networking

connectivity, either wired (Ethernet, serial, modem, etc.) or wireless (infrared, Bluetooth, Wi-Fi, etc.). Even though PDAs became increasingly capable, the basic mode of operation remained the same: Although the potential for interaction with their environment is great, the industry still sees PDAs as autonomous black box devices that rarely interact with each other; the operating systems that ship with them do not offer any interoperability primitives higher than a networking stack. As such, application programmers who want to develop mobile applications that communicate with each other, have to directly program on the protocol stack of the network operating system, tackling the issues introduced by mobility such as heterogeneity and dynamicity – a procedure which is tedious and error prone. Consequently, interaction with their environment and peers is either not considered or is very constrained. Connectivity is usually limited to data synchronisation at a desktop computer. Installing an application is usually also difficult: It typically involves locating the application on a fixed computer, connecting the PDA to that computer and transferring it. Moreover, applications are usually monolithic, composed of a single large file making little use of libraries; thus maintenance and updating of an application is difficult.

This thesis argues that more flexible solutions are required that empower systems to automatically adapt to changes in the environment and to the needs of the users. Power [Power, 1990] postulated more than a decade ago that it is common in distributed systems that

> "when something unanticipated happens in the environment, such as changing user requirements and/or resources, the goals may appear to change. When this occurs the system lends itself to the biological metaphor in that the system entities and their relationships need to self-organise in order to accommodate the new requirements."

Along those lines, this work considers a *self-organising* or *adaptive* system as a system that is able to change to accommodate changes to its requirements. As a highly dynamic system, a mobile system encounters, by definition, changes to its requirements; The thesis therefore argues that mobile systems can benefit from the use of primitives for adaptation as they can, in principle, allow for accommodation to changes to their requirements.

1.1 Hypothesis Statement and Contributions

> *Mobile computing systems are an instance of highly dynamic systems. Mobile applications execute in resource constrained environments with potentially high fluctuations in network connectivity. Moreover, as the devices are physically mobile, their environment constantly changes. Changes to the environment and execution context can dictate changes to the system requirements. Monolithic applications which feature no interaction with their environment and are*

thus primarily designed to be executed in a static context are not representative of the potential of the ubiquity of new mobile devices and networks and new design principles have to be investigated. The hypothesis of this thesis is that mobile systems engineering can benefit from the availability and use of adaptation primitives, by allowing them to dynamically mutate to accommodate new requirements, dictated by the changing context. Given this, this thesis argues that a component model that offers the use of logical mobility primitives as a first class citizen, can be successfully exploited to offer adaptation and reorganisation to mobile systems.

1.1.1 Contributions

Specifically, the contributions of this thesis are the following:

Logical Mobility in a Mobile Environment. Logical Mobility is defined as the ability to send part of an application or even migrate a complete process from one processing environment (or host), to another. This has been further classified into paradigms in [Fuggetta et al., 1998]. Although the use of logical mobility has been well established in legacy distributed systems [Anderson et al., 2002, Sun Microsystems, 1998c, OMG, 1995], there has been little investigation on the use of these primitives in a mobile environment with the peculiarities that this entails (heterogeneity, ad-hoc networking, etc.). This thesis argues that logical mobility can assist in building adaptive mobile systems, as it can allow devices to dynamically acquire new functionality and to better utilise the limited available resources. The sending and receiving, deploying as well as identifying and grouping elements of logical mobility are investigated and handled.

Component Model for Adaptive Mobile Systems. Component-based development argues for the decomposition of a system into a set of interacting components with well defined interfaces. Components promote decomposition and reusability of software. There are numerous component models already developed and discussed in the literature [Hamilton, 1997, Rogerson, 1997, Monson-Haefel, 2000, OMG, 1997], offering various services such as transactions and concurrency control and which have been used to represent systems as a collection of either local or remote components. This thesis presents the SATIN (System Adaptation Targeting Integrated Networks) component model, which is a lightweight component model for mobile systems that offers migrational services using logical mobility primitives. The thesis demonstrates how SATIN addresses issues introduced by mobility, heterogeneity and monolithism in particular, and how it can be used to build mobile systems that can adapt.

Middleware Design and Implementation. In distributed systems, middleware has been traditionally used to handle issues that arise from distributed application de-

velopment, such as heterogeneity, fault tolerance and persistence, in a more general and systematic way. A middleware system is layered between the network operating system and the application and provides abstractions and primitives for remote evaluation, transactions etc., hiding the complexity of the underlying system from the application developer. More recently, there have been a number of middleware systems developed that target mobile devices [Mascolo et al., 2002a]. Some of these attempt to make use of logical mobility primitives to gain some of the advantages inherent in this mechanism. However, the use of logical mobility has been very limited, as middleware systems that employ it either take the "black box" route, by using logical mobility primitives internally and not exposing them to applications, or, alternatively, they expose a very limited subset. The transparency of usage in the first approach may be suitable in fixed distributed systems, in which the execution context is essentially static. It has been argued, however, that transparency, or the complete abstraction of the issues of mobility in mobile middleware systems is disadvantageous to mobile applications [Capra et al., 2001, Eliassen et al., 1999]. Moreover, the exposure of a limited subset of logical mobility primitives as taken by the second approach, has led to the development of a plethora of mobile computing middleware systems [Cugola and Picco, 2002a, Murphy et al., 2001, Weinsberg and Ben-Shaul, 2002], each one solving a particular and thus limited problem, as these systems are not *general* enough to flexibly export logical mobility primitives to applications.

The thesis describes the design and implementation of the SATIN middleware system as an instantiation of the SATIN component model, in which the system and applications deployed on top of it are a collection of SATIN components. The middleware uses logical mobility primitives to reconfigure itself but also exposes the flexible use of any logical mobility paradigm to applications running on top of it. As such, the middleware system is general and flexible enough to offer the solutions described by other approaches that use logical mobility primitives.

Evaluation of Results. The thesis describes the implementation and evaluation of SATIN, by designing, developing and testing a number of applications on top of it, some of which are open source projects that have been converted to SATIN components. The applications, both the developed and the converted, demonstrate different aspects of adaptation. The flexibility and applicability of the SATIN approach is illustrated, by showing how it allows those applications to dynamically reorganise. Moreover, the development of two large scale systems, Q-CAD and ZION, using the model and respectively is illustrated and evaluated. The results show that SATIN is very customisable, easy to use by application developers and that converting existing projects to run on top of SATIN is straightforward.

1.1.2 Related Topics

Although related to the topic, the following are considered outside the scope of this thesis:

Security and Trust. Security in a mobile distributed system is a complex issue, particularly when runtime adaptability is considered. There are a number of issues involved, including how to guarantee that code received will behave as advertised, how to trust a remote host, which may be offering a security certificate or to which code or data is sent. Although this thesis discusses this issue from an abstract point of view, it does not claim to solve it; the author is aware, however, of other projects [Kon et al., 2000, Capra, 2004] addressing these points.

Context Inspection and Reflection. This work assumes the ability to inspect and reason about the local context (such as remaining battery power and memory available) and to expose this information to applications, allowing them to make intelligent choices about adaptation. Although related, this mechanism is not investigated further, as other groups [Capra et al., 2002, Cheverst et al., 2000, Schilit et al., 1994] are researching this. This issue is further discussed in Chapter 6.

Paradigm Evaluation. The use of Logical Mobility has been classified into a set of paradigms in [Fuggetta et al., 1998]. This is further described in Chapter 3. Depending on a number of factors either in the environment or in the status of the device (battery left, networking bandwidth, etc.), a particular paradigm may perform better than another one. The definition of what is better, an investigation on how to evaluate the suitability of the various paradigms and a methodology to choose particular ones are considered outside the scope of this work. Note that there are other projects [Grassi and Mirandola, 2002, Baldi and Picco, 1998] that are researching this issue.

Constraining Adaptation. It may be desirable to constrain or bound the adaptation of a system for various reasons, such as to guarantee a certain quality of service. This aspect is not addressed in this thesis. Note, however, that there are other approaches [Capra et al., 2003, Georgiadis et al., 2002, Mikic-Rakic and Medvidovic, 2002, Parlavantzas et al., 2000] that investigate this.

1.2 Thesis Outline

The remainder of this thesis is structured so as to answer particular research questions, with each chapter building on the answers provided by the previous ones. In particular, this thesis is structured as follows:

Chapter 2 motivates the reader on the novelty, applicability and benefits that this work gives in developing mobile systems. By presenting two industrial examples of mobile applications and giving more background information into this area of research, it answers the questions *what is mobile adaptation* and *why is it important.*

Chapter 3 answers the question *how can a mobile system adapt.* It introduces and conceptually discusses logical mobility and its application on mobile computing. It also discusses and formalises a conceptual platform that allows the flexible use of logical mobility primitives.

Chapter 4 answers the question *how can a system be engineered for mobile adaptation*, by introducing and discussing the SATIN component model for mobile computing. The model is formalised as a a Meta Object Facility [OMG, 2000]-compliant extension of the UML [OMG, 2003] meta model. The chapter shows how the model supports adaptation by encapsulating and offering the platform described in Chapter 3.

Chapter 5 answers the question *how can the* SATIN *component model be instantiated into an adaptable system*, by discussing a realisation of the component model as a middleware system. It describes how the SATIN middleware system is represented as a collection of interoperable dynamic components and how it handles mobility and reaction to context changes. The chapter also describes how reorganisational functionality and the use of logical mobility primitives are offered to SATIN applications.

Chapter 6 evaluates this approach, by discussing the SATIN implementation, illustrating the development of various applications, some of which are conversions of existing projects, to run under SATIN. The chapter also describes the development of two large scale systems using SATIN.

Chapter 7 concludes the thesis by summarising and evaluating the contributions of SATIN to mobile computing research and explores directions for future work.

Chapter 2

Motivation

In order to address the dynamicity inherent in the mobile environment, mobile systems need to be able to accommodate changes to their requirements through *adaptation*. This thesis investigates the use of logical mobility and components to support the construction of adaptive mobile systems.

A number of issues need to be considered when constructing adaptive systems. *What is adaptation? How is adaptation quantified? Who adapts?* and *What triggers the adaptation process?* are some of the questions that need to be answered.

This chapter starts by presenting two examples of industrial mobile systems. It outlines their features, presents their limitations and tries to motivate on the advantages that an adaptive approach using components and logical mobility would provide. The chapter continues by defining adaptation and concludes by stating the assumptions this thesis is founded on.

2.1 Motivating Examples

This section presents two industrial examples, which are part of the motivation of this work. An overview of their functionality is given, followed by a discussion on their limitations and a description on how an adaptive approach based on components, together with the systematic use of logical mobility primitives, can help in overcoming them.

2.1.1 Case Study: Mobile Application Development and Deployment in Practice

PalmOS [PalmSource, 2004b] is the most widely used operating system for Portable Digital Assistants (PDAs); it powers more than 30 million devices worldwide, including mobile phones, GPS receivers, PDAs and sub notebooks. As an example, a popular device running PalmOS has 64 megabytes of RAM (which is used both as storage and heap memory), Bluetooth, infrared and 802.11b wireless networking and wired (serial) networking interfaces, as well as a 400MHz ARM processor. The current version of PalmOS allows for the creation of event driven, single-threaded applications. All files (applications and data) are stored in main memory. Developers compile an application into a single Palm Resource File (PRC) and application data can be stored in Palm Databases (PDBs). The operating system allows for limited use of libraries. Applications are identified by a unique 4 byte identifier, the Creator ID. Developers register Creator IDs for each individual application with the operating system vendor. A PalmOS device usually ships with personal information management (PIM) software installed. Installing new applications requires either locating a desktop computer and performing the installation there or having the application sent by another device directly, a procedure which is not automated.

This model has various disadvantages: there is very little code sharing between applications running on the same device. There is no middleware providing higher level interoperability and communication primitives for applications running on different devices. Applications are monolithic, composed of a single PRC, which makes it impossible to update part of an application. Finally, the procedure needed to install third party applications is difficult.

Palm-based computers can be utilised in both a nomadic and ad hoc networking settings. The potential for interaction with their environment is great, however PalmOS does not provide any primitives for this. The result is that PalmOS based PDAs are still seen as stand-alone independent devices, which interact mainly with a desktop computer to synchronise changes to shared data - interaction with their environment and peers is either not considered or is very limited. Thus, although *physically mobile*, they are *logically static* systems.

A component based approach using logical mobility primitives would have several advantages:

- Decomposition of applications as interoperable components allows for updating of individual parts of an application, rather than replacing the application completely.
- Componentisation promotes code reusability, preserving the limited resources of mobile devices.
- Logical mobility primitives facilitate the transfer of components existing on any host

that is in reach, in a peer to peer fashion. This makes application installation and update more autonomic and therefore more scalable.

- A component model can provide interaction and communication primitives between components higher than a protocol stack.

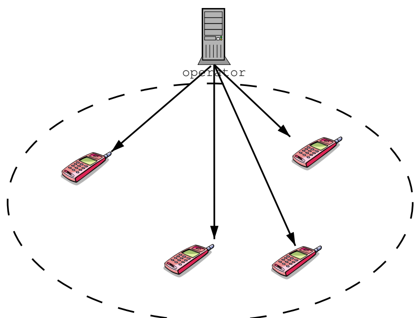
Please note that in other, less popular PDA operating systems, such as Windows CE-based environments and Linux, the use of components is more prevalent, especially by parts of the operating system. However, most of the problems outlined above still exist, as those devices also do not interact with services available in their environment and applications are usually monolithic and static.

2.1.2 Case Study: Application Deployment and Update in a Mobile Environment

Consider mobile phones equipped with a Bluetooth chip. The cellular and Bluetooth connectivity allow these devices to form spontaneous ad hoc networks with other peers in reach, while being connected to a centralised network at the same time. The two different networks, however, exhibit different characteristics; the ad hoc connectivity is highly fluctuating in terms of bandwidth and availability of peers, but is free of charge. On the other hand, cellular connectivity is usually expensive and restricted to areas of network coverage.

Interaction between current mobile phone applications is limited; they do not to react to events and changes in their context, even though the Bluetooth adaptor could be potentially used to monitor the environment for changes in peer and service availability. There have been some approaches to promote interoperability between applications running on mobile phones allowing for data synchronisation, such as SyncML-based approaches [SyncML, 2000], or PalmOS HotSync [PalmSource, 2004a]; however, these only address limited aspects of interoperability, as they only target synchronisation of shared information between multiple devices. The ad hoc connectivity via Bluetooth remains largely underutilised – it is used mainly to exchange contacts and play multiplayer games.

Installing new applications on mobile phones and updating existing ones is currently difficult. In fact, the only popular updates of mobile phone software are the download of ring tones and games. The source of the download is usually the network operator, or another centralised agency. The cellular bandwidth, which is expensive for both the user and the operator, is used for the transfer. Figure 2.1(a) shows how this is done. Even though the phones can communicate with each other directly using Bluetooth, they download the application from the network operator using the cellular network. Figure 2.1(b) shows an adaptive approach to the same problem: The devices discover and download the updates from each other, when feasible, and only need to interact with the network operator to



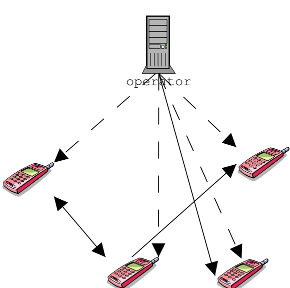


Figure 2.1: (a) Deploying applications (ring tones and games) on mobile phones. (b) Deploying and maintaining applications from peers: Dotted lines represent certificate downloads. Solid lines represent update downloads.

get a certificate of authenticity – thus verifying that the code they have downloaded from a peer is what they want it to be.

Logical mobility for mobile application deployment and maintenance can offer significant advantages over current systems. In particular, it could:

- Facilitate transparent discovery and retrieval of new functionality that is needed by a mobile application. For example, a media player would be able to download a codec when needing to play a file encoded in a new format.
- Allow for transparent maintenance and update of an existing application from peers. A trusted host from the centralised network could be used to verify the authenticity of the updates.
- Enable removal of functionality when infrequently used. The functionality could be transparently retrieved from peers or a centralised host when needed again.

Summarising the examples outlined above, the monolithic nature of current mobile computing systems contributes to their rigidity: Mobile software is deployed once and is very rarely updated. Moreover, mobile systems lack a generalised infrastructure or middleware system to support interaction with their environment and adaptation to its changes.

2.2 Concerning Adaptation

This work argues that the potential of the ubiquity of mobile devices cannot be realised with static applications – applications that do not *adapt* to changes to their context. Thus, this thesis investigates the principles needed to offer reorganisational abilities to mobile systems.

Adaptation is a general concept, which has been defined in a variety of ways. [Capra, 2003] regards adaptation to be

> *the ability of the application to alter and reconfigure itself as a result of (i.e., in reaction to) context changes* [..] *to deliver the same service in different ways when requested in different contexts and at different points in time.*

[Katz, 1994] postulates that adaptation means

> *that systems must be location and situation-aware, and must take advantage of this information to dynamically configure themselves in a distributed fashion.*

[Satyanarayanan, 1996], highlights the importance of mobile adaptation, in arguing that

> *mobility exacerbates the tension between autonomy and interdependence that is characteristic of all distributed systems. To function successfully, mobile elements must be adaptive.*

This thesis refines the above definitions and considers adaptation to be the following:

> *Adaptation is the process by which a system can dynamically acquire or discard functionality.*

Given this definition, a number of questions arise regarding mobile adaptation. These are examined in the following section.

2.2.1 Research Perspective

What triggers the adaptation process?

Mobile adaptation is always a *reactive* process. It is an *action* that results as a consequence of a particular *event* or group of events. Events that can trigger adaptation are classified into the following categories:

Changes to the Environment. Mobile adaptation may be triggered by an external environmental factor. When a mobile device discovers a new remote service, for example, the system may have to adapt, acquiring the functionality needed to interact with the service. Alternatively, a system that accesses remote data, by replicating

them locally, may switch to a remote access mechanism, when connected to a high bandwidth and inexpensive network. The functionality to remotely access the data may need to be dynamically acquired.

Changes to the Local Context. Mobile adaptation may also be triggered by changes to the local context. Consider a system that is using strong encryption to communicate with other hosts. If the system is running out of battery, it may be desirable to switch to weak encryption, which is computationally less expensive, allowing the system to throttle down the CPU, thus conserving more power. The weak encryption functionality may need to be dynamically acquired [1].

User Action. Finally, mobile adaptation may be triggered by a user action. For example, the user may wish to download and use some software. Alternatively, adaptation may be *influenced* by the user. Consider the example of switching to weak encryption when running out of battery, given above. The user may want to override this action, because they may consider the communication too important to be only weakly encrypted. As such, adaptation must *not be completely transparent to the application.*

As a result, an adaptive system must be able to:

- **Monitor changes to its local context**. The local context is defined as the dynamic characteristics of the mobile device, such as remaining battery power, remaining volatile memory, enabled networking interfaces etc. Changes to the local context may trigger the adaptation mechanism; moreover the context of the device at a particular instant in time (the current status of the device) may influence the reorganisation process.

- **Monitor changes to its environment**. This work defines changes to the environment to be changes to remote host, service and resource availability as well as changes to the network configuration and topology.

- **Allow user or application input to the adaptation process**. The principle of transparency, which has long been advocated in middleware systems, has been argued [Capra et al., 2001, Conti et al., 2004] to be disadvantageous in mobile systems. In particular, it is argued [Kon et al., 2002] that in some cases, applications can gain significant advantages by examining the dynamic state of the underlying system. Thus, an adaptive mobile system must allow the application and, by extension, the end user, to influence the adaptation process.

[1] It is acknowledged that because of the low battery charge, the functionality may not be downloaded.

What is functionality?

Adaptation was defined above as the process by which functionality can be dynamically acquired or discarded. Before continuing, the concept of functionality must be defined.

> *Functionality is any part of an application or system that can influence its overall behaviour, by allowing a class of operations to be performed.*

Functionality may be represented as binary code, compiled for a particular platform, bytecode compiled for a virtual machine, or even interpreted textual scripts.

With the recent advent of virtual machines and scripting languages, the difference between system data and functionality is blurred. In essence, functionality allows a system to perform *a class of new operations*, the actual results of which vary based on different *inputs*. Data, on the other hand, act as input to the functional aspects of the system, allowing it to perform a pre-defined set of operations on the input data or based on the input data. Consider, for example, an audio player application. A collection of classes that decode a new audio format into a format the audio player can process and play back is functionality. An audio file encoded into a particular format is data for the audio player application, which can decode it, randomly seek into its data stream and play it.

The following is required for a system to support dynamically acquiring or discarding functionality:

Encapsulating and Describing Functionality. In an adaptive system, functionality must be encapsulated into identifiable *units*. This allows for reasoning about the current abilities of a system and for deducing what unit or units need to be acquired to adapt the system in a particular way. Given the heterogeneity of the target environment, these units need to be described to allow reasoning about their requirements, either hardware (as a unit may be implemented for a particular platform), or software-based (as a unit may depend on the existence of a particular library). The description of the unit also needs to contain information on what it does and how it can be used (such as which interfaces it implements).

Moving and Deploying Functionality. Once functionality has been encapsulated into units, the system must be able to serialise and deserialise them, allowing for transferring them between nodes. Moreover, the units must be able to be dynamically deployable into a running system. As such, the system needs to be *engineered for adaptation.*

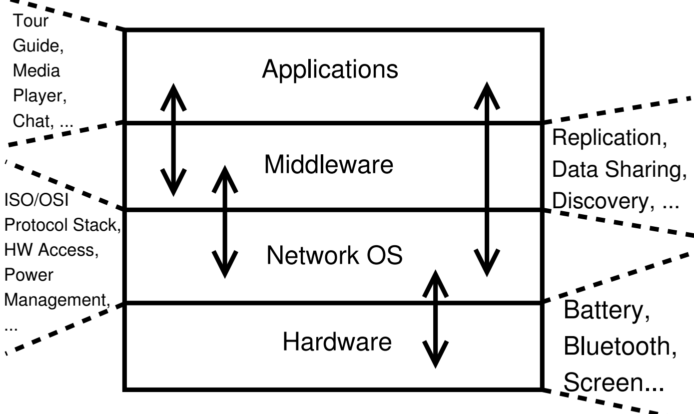


Figure 2.2: A Typical View of a Mobile System.

Who adapts?

A mobile system can be represented as a hierarchy of layers. The lower layer represents the hardware of the device. This includes all the hardware aspects of a system, such as networking interfaces, battery, processor, etc. Typically, some hardware such as persistent memory cards and networking interfaces can be dynamically added and removed, via the use of various expansion slots. The hardware is managed by the network operating system, which provides various hardware abstraction APIs, as well as a TCP/IP protocol stack. Built on top of the operating system is usually a middleware system, which provides higher level primitives such as data-sharing, or advertising and discovery of other services and hosts. A number of applications interact with both the middleware system and the network operating system to provide particular functionality. Examples include media players, tour guides etc. This is shown in Figure 2.2.

As this thesis investigates mobile adaptation, it is important to define which layer in the above description is allowed to adapt. There are various approaches that address adaptation in the middleware layer, such as ReMMoC [Grace et al., 2003b] and UIC [Roman et al., 2001]. Others take an application-centric view to adaptation, such as CARISMA [Capra et al., 2003] and Gravity [Cervantes and Hall, 2004]. This thesis argues that in a pervasive environment, adaptation may need to occur in both application and middleware layers. The services that the middleware system provides, such as service advertising and discovery, may need to adapt in order to allow interoperability with other middleware systems. This can allow, for example, a device to suitably adapt in order to bind to a Jini [Arnold et al., 1999, Hashman and Knudsen, 2001] registry when in the reach of a Jini network. Similarly, an application may need to dynamically acquire functionality to perform a particular task, such as to decode an audio file. The approach that this work proposes is able to address both application and middleware level adaptation.

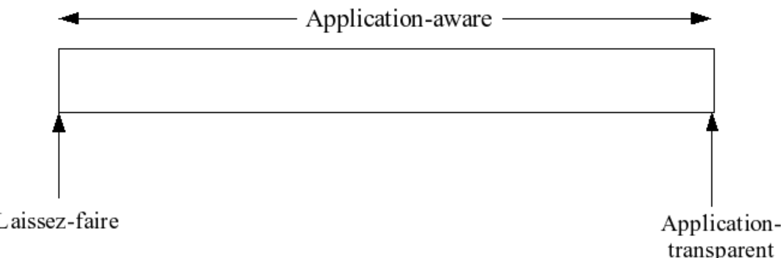


Figure 2.3: A taxonomy for mobile adaptation (adapted from [Satyanarayanan, 1996]).
.

How to Adapt?

Assuming that events have been emitted that should trigger an adaptation process, it is important to decide on how the middleware system or the application will adapt. There are various issues that require investigation, including conflicting requests for adaptation (for example, triggered by the same set of events, two different applications may request two different and conflicting ways to adapt), constraining adaptation to meet a certain set of Quality of Service requirements etc.

[Satyanarayanan, 1996] defines a taxonomy for adaptation, which is outlined in Figure 2.3. On the one end of the spectrum is *laissez-faire* adaptation. Systems that offer laissez-faire adaptation provide the mechanisms needed to adapt but lack a central arbitrator that encapsulates the decision logic behind the adaptation process. This is delegated to the application, which is fully *aware* of the adaptation process. On the other end there is *application-transparent* adaptation, where the application is not aware of the adaptation process; the latter happens internally by the underlying system. In between, there are various types of *application-aware* adaptation, where the application can influence the central arbitrator that handles the adaptation process.

This work focuses on providing a *laissez-faire* adaptable system. As such, the system does not provide the decision logic on *how* to adapt, but provides a structured way to *engineer* a system for adaptation and exports the primitives needed to adapt. Chapters 4 and 5 further discuss this issue. The reason for this is that a *laissez-faire* adaptable system can provide full translucency on the adaptation process. On the other hand, various decision logic layers can be used in conjunction with this system, to arbitrate the adaptation process for applications that require it. The thesis examines one such mechanism in Chapter 6.

2.2.2 In Conclusion: Adapting Mobile Systems

This thesis argues that, in order to take advantage of the pervasiveness of mobile devices, mobile systems need to be able to adapt. To that end, it argues for *engineering* mobile

systems in *modular architectures*, where each module or *unit* encapsulates functionality, as opposed to monolithic architectures, examples of which were summarised in Section 2.1. Moreover, this work argues for requiring *runtime support* that allows for dynamically acquiring or discarding functionality to and from the running system – an *adaptive system* that offers *primitives for adaptation* to mobile applications.

2.3 Assumptions

In developing adaptive mobile systems, there are many issues to be considered, including communications and service discovery. This thesis makes the following assumptions:

1. The work assumes the existence of a communication layer that allows for device coordination and for sending bytes from one host to another. The thesis is not concerned with the particular communication paradigm that the layer provides, whether it is synchronous or asynchronous, point to point or multihop infrastructure-based or peer to peer, etc. However, the communication paradigm must be suitable for the mobile setting, tackling issues such as frequent disconnections, low bandwidth etc. As such, a number of layers would be suitable, such as JXTA [Sun Microsystems, Inc., 2001b], JavaSpaces [Waldo, 1998], Lime [Murphy et al., 2001], etc.

2. Adaptation was defined above as a reactive process to context changes. This thesis is not interested in the actual sensing and encoding of changes to context. The work does provide, on the other hand, a general mechanism which can be used to support different context encodings (see Chapters 4, 5 and 6).

3. Similarly, in describing units of functionality, this work is not concerned with the actual encoding semantics of the various metadata. Again, this thesis does provide a generalised mechanism that can be used to support different encodings and ontologies (see Chapter 4).

4. The thesis assumes the existence of one or more different mechanisms to advertise and discover hosts, services and resources in a mobile environment. Although the work does provide an adaptable mechanism and abstractions to do service advertising and discovery in multiple ways (see Chapter 5), it is not concerned with the actual implementation details of those. Indeed, the implementations of the abstractions that are provided are simple, comprised of multicast and centralised publish/subscribe systems (see Chapter 6). However, more sophisticated approaches such as Jini [Arnold et al., 1999], JMatos [Hashman and Knudsen, 2001], Universal Plug n' Play [UPnP Forum, 1998] etc. could be supported by the infrastructure provided. This is further discussed in Chapters 5 and 6.

5. It is conceivable that adaptation may lead to a system that fails to meet a set of requirements concerning quality of service, or may not even be usable, as some essential functionality could have been discarded. Moreover, it may be the case that two or more applications behave in a conflicting way when reacting to a change in context. As mentioned in Section 1.1, this thesis is not concerned with constraining the potential adaptation of a system. As such, the system presented in this work offers *laissez-faire* adaptation. This is further discussed in Chapters 4 and 5. An *application-aware* mechanism allowing the application to influence a central arbitrator on how to adapt, built on top of the system this thesis presents, is discussed in Chapter 6.

6. This work and in particular Chapter 3, assumes the use of the object oriented paradigm when engineering systems. As such, when discussing code, the notion of a class will be used and the state of that code will be represented by an instance of a class. It is expected that other styles can be encapsulated inside one or more classes. Code and state are further discussed in the following chapter.

Chapter 3

Logical Mobility Primitives for Mobile Computing

As outlined in the previous chapter, this thesis investigates *adaptation* in mobile computing systems, from both system and application perspectives. Furthermore, the previous chapter defined adaptation as the dynamic modification of the functionality of a system, with functionality being any part of an application or system that can influence its overall behaviour, by allowing a class of operations to be performed.

This thesis conceptually splits distributed systems into two layers: the physical layer, which refers to the hardware of the nodes participating in the system, and the logical layer, which refers to the software that runs on the hardware layer of each node.

This chapter discusses mobility at both physical and logical layers of a distributed system. In particular, it briefly discusses mobile networking paradigms and refines the concept of *logical mobility*, which this work uses to encapsulate, transfer and deploy functionality to mobile systems. Thus, it discusses the relation between logical mobility and *code mobility* and its applicability to physically mobile systems. It identifies mobile application domains that can benefit from its use and presents a conceptual model for a logical mobility system for mobile devices.

3.1 Physical Mobility and Mobile Networking Paradigms

Physical Mobility refers to the ability of a node to change its physical location. A *mobile node* can encounter various types of networking connectivity. As such, there are different paradigms of mobile networking, which differ on the cost, quality, availability and routing strategies provided by the networking infrastructure. These are outlined below.

Nomadic. This paradigm describes temporal fixed network connectivity, which is only available in particular locations. Connectivity is therefore transient and lasts for as long as the device stays in the particular location. Thus, applications cannot be connected to the network and be physically mobile at the same time. Examples of this type of mobility include wired networks like Ethernet and dial up networks. It offers users fast and usually inexpensive access to a fixed network (the Internet or an intranet), with routing handled by the network infrastructure.

Base Station Mobility. In base station mobility, a node is connected to a fixed network (such as the Internet) while still being physically mobile within a region where there is coverage. Exploiting wireless links to connect to a fixed infrastructure network, base station mobility offers an error prone, low-bandwidth and potentially continuous connection. Examples include using GSM/GPRS phones or Wi-Fi enabled PDAs. This type of connection is usually expensive: users pay either for the time they are connected (e.g., GSM) or for the the data that they exchange (e.g., GSM/GPRS). Routing is provided by the infrastructure of the network.

Ad Hoc. The final type of mobility considered is Ad-Hoc networking, where connectivity relies on no fixed infrastructure, and hosts usually employ wireless links to form temporal and highly dynamic networks, allowing them to connect to other hosts that are currently *in reach*. In some types of Ad-Hoc networks, (some of) the participating nodes act as routers, allowing for multi-hop connectivity. The networks formed are thus very dynamic and rapidly changing. The communication link is usually of relatively low speed and error prone. However, as there is no centralised provider, connectivity is essentially free, costing mobile users only in terms of local resources, such as battery power and CPU cycles.

This thesis argues that given the current trend in mobile technologies, most mobile devices will be using a number of mobile networking paradigms, depending on the situation. This will allow a user to browse the web on the go (base station mobility), exchange files with other peers in reach (Ad-Hoc) and connect to a company network to access documents, when on site (nomadic connectivity).

3.2 Defining Logical Mobility

Logical Mobility refers to the ability to change the configuration of the logical layer of a distributed system, by *transferring* logical units between nodes. Informally, this thesis defines logical mobility as:

the transfer of any part of a software system from one node to another.

Logical Mobility has been argued [Roman et al., 2000] to have great potential for engineering mobile systems, a potential that has not yet been realised. This thesis claims that logical mobility can be used to adapt mobile systems, as it can *encapsulate functionality* which can be dynamically added to a running system. Logical mobility is usually offered using *Code Mobility* techniques. The next section describes what code mobility is, its relationship to logical mobility and outlines its usage paradigms.

3.2.1 Introduction to Code Mobility

Code mobility has been defined [Carzaniga et al., 1997] as '*the capability to dynamically change the bindings between code fragments and the location where they are executed*'. More informally, code mobility can be defined as the ability to move code between nodes in the network. It has been argued [Fuggetta et al., 1998] that code mobility is a technology that can be used to engineer configurable, scalable and customisable large scale distributed systems, by allowing code to migrate and bind to different nodes of the running system. Mobile code systems usually define a *code unit* as a conceptual or realised abstraction that encapsulates a form of code. A code unit is the *minimal unit of transfer* or *unit of mobility.*

Whereas code mobility specifically refers to the transfer of code between nodes, logical mobility builds on this notion and refers to the reconfiguration of systems by moving any part of the logical layer between nodes. Logical Mobility is usually offered using code mobility techniques to transfer information, including binary code, compiled for a specific architecture, interpreted textual scripts, bytecode compiled for a virtual platform, such as the Java Virtual Machine (JVM), but also application data such as profiles. In this context, data are defined to be anything that cannot be directly executed by the underlying platform.

Figure 3.1, adapted from [Picco, 1998c], shows an outline of two systems using mobile code. The core operating system is built on the hardware layer, and provides abstractions to access the hardware and basic services such as memory management. The networking aspects of the operating system are built on top of this; they provide basic networking services, such as a TCP/IP protocol stack. Layered on the network operating system is the Processing Environment (PE). The processing environment is a *container* which allows *code units* [1] to run; it provides a set of primitives (the extent of which varies between platforms) to allow for code migration, or even for access to local resources. Any coordination between various units as well as between the units and the rest of the system happens at this layer. A Processing Environment usually acts as a sandbox, restricting the access of a unit to protect from malicious code. Figure 3.1 shows a unit transferred

[1] [Picco, 1998c], on which this diagram is based, uses the term 'component' instead of 'code unit'. However, 'component' is defined and used differently in this thesis and hence the term 'code unit' is used here.

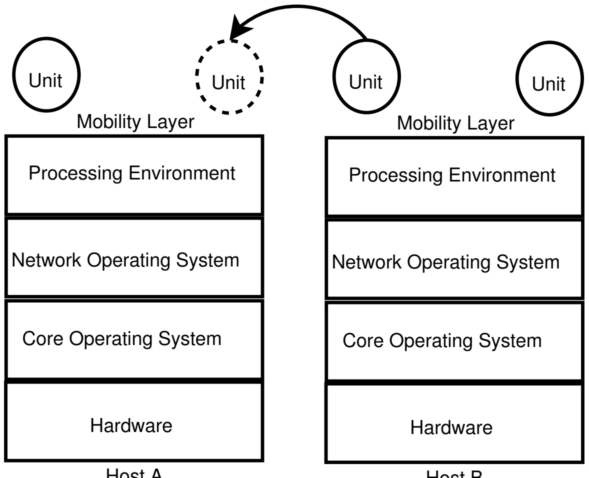


Figure 3.1: A mobile code system, showing the transfer of a unit from Host B to Host A. Adapted from [Picco, 1998c].

from one node to another.

A code unit is composed of the following:

- A Code Segment, which represents a sequential flow of computational statements.
- A Data Space, which represents a set of references to resources that the code segment is using.
- An Execution State, which represents the control information of a running code segment (program counter, register contents etc.).
- Resources, which are references to entities representing data and devices that the code unit accesses.

Code mobility can be further refined as the transfer of the code segment and potentially the execution state from one processing environment to another. Depending on whether the resources are transfered or can be recreated on the recipient node, the data space of a migrating unit is either transfered, modified or recreated upon deployment on the recipient node.

There are two manifestations of code mobility: *weak* mobility, where a code unit transfered cannot include execution state information and *strong* mobility, where this is possible. Strong mobility allows for a process or thread to suspend execution, move to another host and resume execution from the exact point where it stopped on the originating host. At a conceptual level, the migration of a thread or process using a strong mobility mechanism can be completely hidden from the application programmer. A weak mobility mechanism

can approximate the operation of a strong mobility one, provided that the application programmer is aware of the migration process; the programmer would need to explicitly save any data that are needed to resume execution at the recipient host before the transfer takes place and to use the data to resume appropriately after the transfer process is completed. By definition, weak mobility platforms consume less computational resources than strong mobility ones.

Both strong and weak code mobility can be realised in two ways: *migration* and *cloning*. When a code unit migrates, it is *moved* from the source host to the recipient. After the operation is complete, the unit does not exist anymore in the original host. Cloning, on the other hand, allows for a copy to exist in both the source and the recipient hosts.

There are a number of issues which are raised when using code mobility, in terms of *heterogeneity*, *binding*, *security* and *dependability*. Figure 3.1 assumes that hosts participating in the system have compatible architectures. More specifically, it assumes that they have compatible hardware, operating system and processing environment layers. Virtualisation techniques, such as the Java Virtual Machine [Lindholm and Yellin, 1999], have been proposed and used to solve the problem of heterogeneous platforms, by introducing a virtual platform which can be realised on top of existing ones. Another issue is that binding a code unit to an already running system is not trivial. The code unit, may define itself using namespaces that are already in use in the recipient node. A code unit may also pose a security threat to the recipient node; it can be a virus for example. Finally, the code unit may depend on the existence of software functionality (libraries, etc) or resources (files, devices, etc) which may not exist on the recipient node.

3.2.2 Paradigms of Code Mobility

Code mobility and, by extension, logical mobility (whether weak or strong, using either migration or remote cloning), may be employed using a set of paradigms [Fuggetta et al., 1998] described by the *code pushing* and *code pulling* models. Code pushing refers to sending a code unit from one processing environment to another. Code pulling, on the other hand, refers to retrieving a code unit and deploying it to the local processing environment. The interaction between two nodes in a mobile code system differs mainly with respect to the node that requests the transfer of the code unit and to the one that transfers it. As such, the use of code mobility has been classified using the following paradigms:

Client - Server Interactions

Client - Server (CS) dictates the execution of a unit in a server, triggered by the request of a client, which may receive the result of that execution. The most common example of

this paradigm is the use of Remote Procedure Calls (RPCs). The client pushes the request to the server, which contains the code unit to be executed. There is no actual transfer of any code unit in this paradigm.

Client - Server interactions are mostly used in traditional distributed systems, such as RMI [Sun Microsystems, 1998c] and CORBA [OMG, 1995] and not on mobile code systems as they do not involve moving code. It is presented here for completion.

Remote Evaluation

In the Remote Evaluation (REV) paradigm, a host *pushes* a particular code unit to a remote processing environment. If accepted at the destination, the unit is deployed and executed there. A result may or may not be needed by the source host, depending on the application.

This paradigm is popular with distributed computation projects, which operate by using the divide and conquer algorithmical paradigm to break large computational challenges into smaller, more manageable problems and distribute those to nodes on the internet. The results are then sent back to the server orchestrating the computation, which can recompose the answer to the original challenge. Examples of these projects include SETI@home [The Seti At Home Project, 1997] and Distributed.NET [The Distributed.net Project, 1995].

Code on Demand

In the Code on Demand (COD) paradigm, a host requests and subsequently *pulls* a code unit from another node. This is an example of dynamic code update, whereby a host or application can update its libraries and available codebase at runtime.

Many examples of COD have recently emerged, due to the popularity of Java, its built-in class loading mechanism and its object serialisation framework [Sun Microsystems, 1998b]. Java web browser applets and Jini [Waldo, 1999] are examples of the use of this paradigm.

Mobile Agents

A Mobile Agent [Wong et al., 1999] (MA) is an autonomous code unit. It is injected into the network, to perform some tasks on behalf of a user or an application. The agent can potentially migrate from one processing environment to another. Typically, a mobile agent system requires infrastructure that supports strong mobility, to allow for suspending at one node and resuming at another. In platforms supporting weak mobility only, an agent can

be modelled using the remote evaluation paradigm to migrate the agent from one node to the next; the application programmer needs to be aware, however, of the migration process, in order to gather and store the information needed to resume execution at the next node.

There are a number of Mobile Agent platforms, such as Aglets [Lange and Oshima, 1998] and μCode [Picco, 1998b]. There are also some mobile middleware systems that employ them - For example LIME [Murphy et al., 2001] uses μCode to offer data-sharing in a mobile environment, using Mobile Agents. There has also been extensive work in the use of Mobile Agents in network management [Bieszczad et al., 1998, Baldi et al., 1997, Gavalas et al., 1999].

3.2.3 Application Domains

This section outlines a non-exhaustive collection of mobile application domains that can benefit from the use of logical mobility primitives. This has been previously discussed in [Zachariadis et al., 2002].

Limited Resources and Dynamic Update

The traditional goal of Human Computer Interaction design, that is, making computers easy to use, becomes increasingly important in a ubiquitous computing setting, as users are now potentially interacting very frequently with an increasing number of different devices. In particular, the use of mobile computing devices in a ubiquitous setting should be transparent to the user, avoiding exposure to the complex installation and (re)configuration procedures that are needed to make a device perform a new task, such as binding to a remote service. Preloading all the functionality that may be needed throughout the lifetime of a device is difficult, for two reasons: Mobile devices usually have limited resources, especially memory or storage; hence it is expensive to store all the code that may be needed a priori on the device. Moreover, it is improbable or even impossible to predict the functionality that will be needed by a general purpose computing device which is exposed to a such a dynamic environment.

Code on demand can be effectively used to download functionality from resources in the environment. This would allow media players, for example, to transparently download new codecs when needed from any resources in reach. Moreover, least used functionality can be discarded when a system has little remaining memory, only to be re-acquired from the environment when needed again. A mixed networking setting, where a device is connected ad hoc via short range radio network and to a central ISP via a cellular network, for example, would allow a device to use the inexpensive ad hoc networking bandwidth to download software and the cellular bandwidth to verify the authenticity of

that software from a trusted third party. Similarly, code on demand primitives can be used to transparently update the software running on a device; the device would need to discover that a peer is running a newer version of the software than the one that is available locally and transparently download the update [2]. This can help device manufacturers and software developers to keep the software running on mobile devices up to date.

Location-Based Services

There have been a number of approaches [Abowd et al., 1997, Cheverst et al., 2000] offering services to applications based on computational facilities or sensors that are available in the current context of a device. However, most of these approaches assume that the device is equipped a priori with the software functionality that is needed to interact with the services; for example to parse any data received from a sensor. Logical mobility primitives can be used to overcome this limitation.

Similarly to how Java Applets adapt a web browser based on a virtual location (the Internet address that serves the applets), logical mobility and code on demand in particular can adapt the functionality of a mobile system based on its current physical location. For example, upon entering a movie theatre, a mobile device can be transparently updated with a graphical user interface and the functionality needed to allow the user to order movie tickets and food. Location-based reconfigurability would also be beneficial at a system level. For example, upon connecting to a Jini [Waldo, 1999] network, a system can use code on demand to receive the functionality needed to participate in the Jini Federation, allowing applications to print on a Jini-based printer.

Active Networking.

Logical Mobility can be employed in routing packets between nodes, either on the same or on different networks. This *active networking* approach can result in the introduction of more flexibility over traditional routing techniques, which may be useful in the dynamic environment imposed by mobile computing. Consider, for example, multimedia transmission. In times of contention and limited bandwidth, a transcoder could be shipped in one of the nodes of the network, to recode the stream to a lower bitrate, thus requiring less bandwidth.

Similarly, logical mobility and mobile agents in particular can be used to offer adaptable routing in disaster area scenarios. In these cases, mobile computers are usually scattered over a particular area, large or small, which either lacks a network infrastructure or it is unusable, as it may be overloaded. The nodes can communicate by employing mobile

[2] There are various issues here, regarding compatibility with older versions. Versioning is discussed in Chapter 4.

agents to migrate between the various devices and flexibly disseminate information. The agents can intelligently choose which node to migrate next to, based on context information gathered on each node.

Limiting Connectivity Costs

As wireless network connectivity is expensive, traditional usage of a network (e.g. web browsing) from a mobile device is often not practical. This is also because the input and output mechanisms of the device may make the process difficult (e.g. pen input and a small screen). For tasks that can be automated, using web services for example, logical mobility and mobile agents in particular can provide a more efficient solution. For example, an electronic shopping application can encapsulate information about the item that the user wants to buy (maximum price, etc.) in an agent and then migrate the agent to a processing environment provided by another host in the environment or even the ISP of the user. The agent would then use the presumably inexpensive networking connectivity provided by that environment to find the best product matching the description of the user, contacting him/her to confirm the purchase. A similar approach has been detailed in [Keegan and O'Hare, 2003].

Exploiting Computational Resources

Logical mobility primitives (remote evaluation in particular) can be used by applications to distribute expensive computations to powerful hosts that are reachable. This can effectively allow applications to appear to operate faster. Similar approaches have been used in static distributed systems; For example, in [The Seti At Home Project, 1997, The Distributed.net Project, 1995], this paradigm is used to break problems down into more manageable parts and remotely evaluate those on various nodes. Related approaches have also be used in mobile systems. For example, in [Cugola and Picco, 2002a], a shared virtual data structure is defined, which is distributed amongst various mobile nodes. Remote evaluation is used to distribute operations on the shared data to the nodes that host it locally.

3.2.4 Summary: Benefits of Logical Mobility for Mobile Reconfiguration

Given the application domains outlined above, the benefits of logical mobility for mobile systems, with regards to adaptation in particular, can be summarised to the following interrelated points:

- Logical mobility allows applications to update their codebase, hence acquiring new

functionality.

- Logical mobility may permit interoperability with remote applications and environments, which have not been envisioned at design time.
- Logical mobility potentially achieves the efficient use of peer resources, as computationally expensive calculations can be offloaded to the environment.
- Logical mobility facilitates the efficient use of local resources, as infrequently used functionality can be removed to free some of the limited memory that mobile devices are equipped with. The functionality may potentially be retrieved later when needed.
- Logical Mobility primitives can be used to encapsulate, request and transfer functionality between nodes; hence it is a tool that can be used to create adaptable systems.
- By allowing functionality to be retrieved locally, Logical Mobility allows for autonomous operation instead of relying on an externally provided service.

3.3 Conceptualising the Use of Logical Mobility over Physical Mobility

Having outlined the benefits of logical mobility for adaptable mobile computing systems, this section presents a conceptual framework for logical mobility targeting mobile systems. The system is general enough as to not be tied to any particular application or paradigm. The framework presented is built on the notion of *weak mobility*. Given that a weak mobility model can approximate a strong mobility one if the programmer of the application is aware of the mobility process, the framework is built on a weak mobility model, because:

- In order to *adapt* a mobile system by dynamically transferring functionality between two nodes, it is not necessary to transfer the state of that functionality in the transmitting host.
- It has already been argued [Capra et al., 2001, Mascolo et al., 2002a] that mobile computing programming should not be transparent but reflective, exposing the issues introduced by mobility to the programmer, rather than abstracting away from them. As such, making the programmer aware of the mobility process is not a disadvantage, but, practically, a requirement.
- The implementation of a strong mobility model requires, by definition, more resources than that of a weak mobility one, making it less suitable for the constrained computational resources of mobile devices.

This section does not discuss in great detail the Client/Server paradigm defined above. Client/Server interactions are not directly beneficial for purposes of mobile adaptation, as they do not involve the transfer of functionality. On the other hand, the examples given in Section 3.2.3 showed that Code On Demand, Remote Evaluation and Mobile Agents, are all useful in providing the necessary flexibility needed for adaptive mobile systems. As such, the thesis argues that a framework is needed that offers the flexible use of Logical Mobility primitives by mobile applications.

This section continues by listing a number of requirements and assumptions and continues with a detailed description of the framework offered.

3.3.1 Requirements

A number of requirements were identified in designing a conceptual framework that offers the use of logical mobility, targeting adaptable and physically mobile systems. These are discussed below.

A Lightweight System. The framework produced must be lightweight and geared for devices with limited computational capabilities. The framework achieves this, by being based on a *weak mobility* mechanism and by being modular to allow for implementations that only realise the modules that are needed. This is also discussed in the implementation of that framework in Chapter 6.

A Symmetric System. The framework produced must be symmetric, allowing both sending and receiving functionality. The reasons for this are similar to the arguments between traditional client/server and peer to peer systems. A symmetric framework allows for potentially building a large scale peer to peer network of offered resources and functionality.

A Flexible System. The flexible use of logical mobility requires the ability to use any logical mobility paradigm described above, by any application. This implies the ability to encapsulate modern programming language abstractions such as objects and classes into the unit of mobility. The framework achieves flexibility by providing a general API, allowing applications to represent, send and receive multiple aspects of functionality atomically, as a single unit of transfer (more on this in Section 3.3.2). It also does not pose any limitations on which paradigm should be used.

A Secure System. Security is one of the fundamental issues with mobile code and, by extension, with logical mobility. Although Section 1.1 claimed that addressing security issues is not an explicit goal, the framework presented here recognises and conceptually addresses this issue, by incorporating trust and authentication mechanisms into the system. Moreover, conflicts, or the possibility of unintentionally overwriting existing functionality is also addressed.

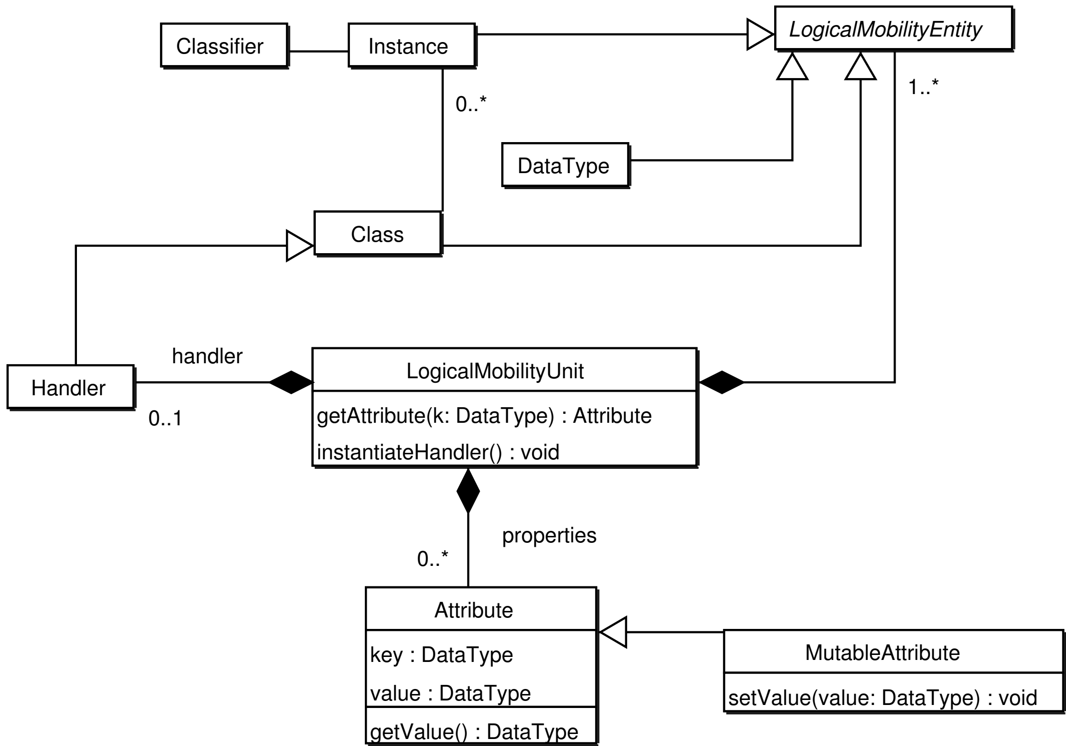


Figure 3.2: The Logical Mobility Unit.

A Mobile System. A framework for logical mobility for mobile systems must address the peculiarities that its environment entails. Mobile distributed systems are composed of many different resource constrained nodes, which exhibit heterogeneity at many levels: They have many different software packages installed, are built on many different middleware systems and are implemented on different hardware. Moreover, networking connectivity is erratic, and disconnection is a frequent event. As such, the system presented discusses issues of heterogeneity and assumes a communication layer that allows for both synchronous and asynchronous communication.

Having defined a number of requirements for a logical mobility framework for mobile computing, this section continues with its description. It is composed of various conceptual layers. The remainder of the section starts by describing a container that is used to encapsulate aspects of logical mobility; it then builds on that to define all the layers and aspects of the framework.

3.3.2 The Logical Mobility Unit

Figure 3.2 presents a conceptual encapsulation of logical mobility as a Meta Object Facility [OMG, 2000]-compliant extension of the UML [OMG, 2003] meta model version 1.5.

It builds upon and extends the concepts of Classifier, Class, Instance and DataType [3].

The diagram defines three aspects of Logical Mobility: Classes, Instances and DataTypes; the latter is defined as a bitstream that is not directly executable by the underlying architecture. As such, the framework specifically addresses the transfer of classes, instances and data as aspects of logical mobility. The *Logical Mobility Entity* (LME) is defined as an abstract generalisation of a Class, Instance or Data. Consequently, an instantiation of an LME represents an aspect of the logical layer of a system.

The *Logical Mobility Unit* (LMU) is defined as the minimal unit of transfer in this framework. An LMU is a container, that can encapsulate various constructs and representations of code and data. As such, an LMU is, in part, a composition of an arbitrary number of LMEs. This allows an LMU to contain anything from a single class to a collection of classes, instances and data. The LMU provides operations that permit inspection of contents. This allows a recipient to inspect an LMU before using it.

The LMU can potentially encapsulate a *Handler* class. The Handler can be instantiated and the resulting object used by the recipient to deploy and manipulate the contents of the LMU. This can allow sender-customised deployment and binding. The Handler concept and name is taken from [Picco, 1998a]. Handlers and deployment in general are further discussed in the next paragraphs.

An LMU also encapsulates a set of *attributes*, called the *properties* of the LMU. An attribute is a tuple containing a *key* and a *value* and the properties of the LMU map each key to its associated value. As such, a reference to an attribute encapsulated in the LMU can be obtained by identifying its key. Attributes represent the metadata of the LMU. Attributes can be either mutable or immutable. The number and type of attributes is not fixed. The properties are used to describe the LMU they are associated with. For example, logical (software) or physical (hardware) dependencies, digital signatures and even end-user textual descriptions can be expressed as attributes. As such, they can be used to express the heterogeneity of the target environment. For example, an LMU that contains Java classes may specify that it requires a Java Virtual Machine that implements version 2 of the appropriate specification as an attribute. An ontology for attribute keys and values is not defined at this stage.

Mutable attributes are useful because they allow for storing the state of the Logical Mobility Entities separately to their logic. This allows, in principle, to update the logic of a logical mobility entity, while maintaining its state. This is useful in many scenarios; for example in self-updating mobile agents.

The Object Constraint Language [OMG, 2003] is used to encode static semantics of the architecture described in Figure 3.2. The constraints are outlined below:

[3] Note that Classifier, Class and DataType are taken from UML Core. Instance is taken from UML Common Behaviour. [OMG, 2003].

Constraint 1:

```
context LogicalMobilityUnit
def: hasAttribute( k : DataType ) : Boolean =
                                   self.properties->exists( key = k )
```

Constraint 1 is an auxiliary function that checks whether a Logical Mobility Unit has an attribute with key `k`.

Constraint 2:

```
context LogicalMobilityUnit
inv: self.properties->forAll ( a1 , a2 | a1.key <> a2.key )
```

Constraint 2 prescribes that each key in the Logical Mobility Unit properties is unique.

Constraint 3:

```
context LogicalMogilityUnit::getAttribute( k : DataType ) : Attribute
pre : hasAttribute( k ) = true
post : result = self.properties -> select( key = k)
```

Constraint 3 prescribes that if an attribute requested is defined in the Logical Mobility Unit, then that attribute is returned to the caller.

Constraint 4:

```
context LogicalMogilityUnit::getAttribute( k : DataType ) : Attribute
pre : hasAttribute( k ) = false
post : result = null
```

Constraint 4 prescribes that if the attribute requested for is not defined in the Logical Mobility Unit, then null is returned to the caller.

Constraint 5:

```
context LogicalMogilityUnit::instantiateHandler : void
inv : self.handler->notEmpty()
```

Constraint 5 prescribes that a handler can only be instantiated to deploy the LMU if the LMU includes one.

Constraints 2, 3 and 4 essentially dictate that each attribute is uniquely identified by its key in the context of the properties of a Logical Mobility Unit.

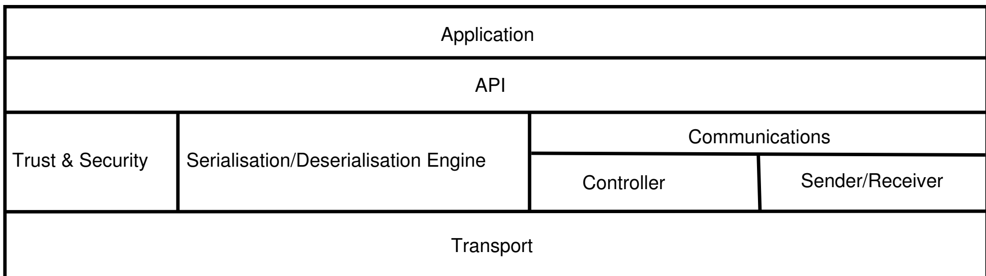


Figure 3.3: A Logical Mobility Framework.

The LMU and its contents can be serialised and deserialised. As such, using logical mobility techniques is equivalent to composing the LMU, serialising it, transferring it, deserialising it and deploying it, as well as triggering this sequence of operations. The next section describes a framework that allows this.

3.3.3 A Framework for Logical Mobility

Figure 3.3 outlines a framework for the use of logical mobility techniques by mobile systems, as a collection of conceptual layers, built on top of the network operating system. The following paragraphs describe each layer in detail. The operations of the framework is modelled as a collection of interacting concurrent processes, using the Finite State Processes (FSP) process algebra [Magee and Kramer, 1999], with each layer represented by a process. A process algebra was chosen over alternatives such as axiomatic and denotational models because of the more powerful model of concurrency that an algebra provided, which allows for clear definition of the various states of each process and of the system as a whole. FSP was chosen in particular for reasons of familiarity and tool support. For reasons of clarity, the process algebra for each layer is also visualised as a state machine. The process algebra allows us to express safety and liveness properties on the framework, and verify that it operates correctly, while allowing callers to use any logical mobility paradigm.

Note that this framework is *abstract* and general purpose; not all of the layers have actually been realised. For implementation details, see Chapter 6.

The Transport Layer.

Although not strictly part of the architecture provided, the system is layered on top of the networking primitives that the operating system provides. As such, the *transport layer* is responsible for negotiating, establishing and maintaining a communication channel between two nodes. This includes detecting and identifying hosts that are currently in

reach and potentially negotiating compression and encryption techniques between the two nodes. Implementations of the transport layer may provide both synchronous and asynchronous communication primitives. No assumptions are made about the mode of communication. Failures in the transport layer may result in data packets not successfully sent or received; These failures are discussed in the layers that concern them below.

The Trust & Security Layer.

There are many aspects to consider when trying to establish a secure mobile computing environment that uses logical mobility. In particular, the following may be required:

- *Privacy* of communications. As such, data exchanged between two nodes should not be able to be meaningfully intercepted, tampered with, or repeated to perform a replay attack. This implies some form of encryption and is considered the responsibility of the *transport layer*.
- *System integrity* against malicious LMUs. This implies that the system should be protected against LMUs that attempt to damage the system (e.g. viruses and trojans), or, more generally, that fail to performed as advertised. This implies that the system must support inspection of incoming LMUs before accepting and incorporating them into the system.
- A level of *trust* should be maintained between the various nodes. In a fixed network distributed system with a limited number of nodes, a number of assumptions can be made with respect to trust; more often than not, all nodes in the system are assumed to be trustworthy. This assumption cannot be made in a mobile computing system, as the system is orders of magnitude more dynamic, with hosts forming rapidly changing ad-hoc networks. Maintaining trust allows the system to avoid sending sensitive data to hosts which are not trusted.

System integrity and trust are the responsibility of the *trust & security* layer, of which there can be various realisations that employ different trust models, digital signatures and trusted third party-based verifications, heuristic virus scanning, or even logic based techniques such as proof carrying code [Necula, 1997] to offer various levels of security.

Figure 3.4 shows the specification of the Trust and Security Layer in FSP, while Figure 3.5 shows a state machine generated from it. It is represented as the `TRUSTANDSEC` process, which may either *inspect* an incoming LMU (thus trying to maintain system integrity) or *examine* the host to which an LMU is to be sent (thus implementing a trust mechanism). The result of the inspection (represented by `INSPECTION`) is either *accepted*, which denotes that the LMU is not malicious and that it behaves as advertised, or *rejected* otherwise. The result of the examination (represented by `EXAMINATION`) is either *trusted*, or *mistrusted*.

```
TRUSTANDSEC = ( inspect -> INSPECTION
               |examine -> EXAMINATION ),

EXAMINATION = ( trusted -> TRUSTANDSEC
               |mistrusted -> TRUSTANDSEC ),

INSPECTION = ( accepted -> TRUSTANDSEC
              |rejected-> TRUSTANDSEC ).
```

Figure 3.4: The specification of the Trust & Security Layer.

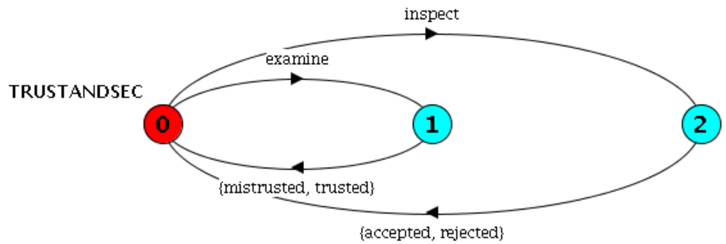


Figure 3.5: A state machine representing the Trust & Security Layer.

The exact semantics of accepted, rejected, trusted and mistrusted depend on the particular realisation.

Note that different levels of privacy, protection and trust may be required, depending on the actual system realisation. Moreover, as mentioned in Section 1.1, this layer is not implemented (see Chapter 6).

The Serialisation & Deserialisation Engine.

The *serialisation & deserialisation* engine is responsible for converting an LMU into a bitstream and vice versa. Different implementations may use different encodings to write and read the stream. When deserialising a bitstream, the engine is responsible for instantiating an LMU with the contents of the bitstream into a processing environment where it can be inspected by the trust & security layer. Deserialisation may fail if an element in the LMU has references which cannot be restored in the recipient node or if, because of a failure in the transport layer, the bitstream was not successfully received. Upon deserialisation, the elements of the LMU are checked for conflicts with elements already in the system. Essentially, conflicts may occur if elements of the LMU define themselves using names that are already in use in the recipient node. If a conflict is detected, implementations of the serialisation & deserialisation engine may either reject the LMU, or try to resolve the conflict, by loading it, for example, into a private namespace. This is only partially implemented (see Chapter 6).

```
SERDESERENGINE = ( deserialise -> DESERIALISING
                  |serialise -> SERIALISING),

DESERIALISING = ( deserialised ->  CONFLICTCHECK
                 |deserFailed -> SERDESERENGINE ),

CONFLICTCHECK = ( conflict -> RESOLVECONFLICT
                 |noConflict -> deserSuccess -> SERDESERENGINE ),

RESOLVECONFLICT = ( deserReject -> deserFailed -> SERDESERENGINE
                   |conflictResolved -> deserSuccess -> SERDESERENGINE ),

SERIALISING = ( serSuccess -> SERDESERENGINE
               |serFailed -> SERDESERENGINE ).
```

Figure 3.6: The specification of the Serialisation & Deserialisation Engine.

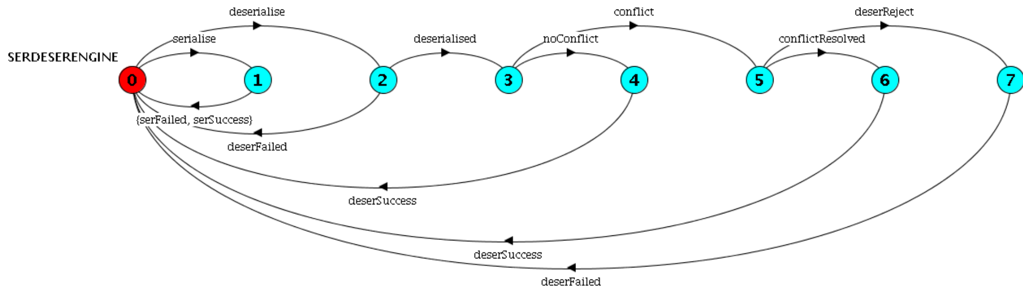


Figure 3.7: A state machine representing the Serialisation & Deserialisation Engine.

Figure 3.6 shows the specification of the serialisation & deserialisation engine, represented by process `SERDESERENGINE`. The process can either *deserialise* an incoming LMU or *serialise* an outgoing one. In the former scenario, represented by `DESERIALISING`, the process can either fail (*deserFailed*), if, for example, the incoming bitstream was invalid (due to a transport layer failure) and could not be read or references contained in the LMU could not be restored, or succeed (*deserialised*). In the latter case, the serialisation & deserialisation engine checks whether the contents of the LMU conflict with the running system (`CONFLICTCHECK`). If a conflict is not detected (*noConflict*), then the deserialisation process is successfully completed (*deserSuccess*). If a conflict is detected, then the engine may try to resolve it (`RESOLVECONFLICT`), by, for example, loading the contents in a private namespace. If the conflict resolution process is successful (*conflictResolved*), then the deserialisation process is successfully completed (*deserSuccess*). Otherwise, the LMU is rejected (*deserReject*) and the deserialisation process fails (*deserFailed*).

When serialising an outgoing LMU (`SERIALISING`), the process may either successfully complete (*serSuccess*) or fail (*serFailed*). A reason for failure is, for instance, that the contents of the LMU contained non serialisable references, such as a reference to a hardware

```
CONTROLLER = ( controllerStart -> ON ),

ON = ( sendRequest -> ON
      |receiveRequest -> REQUESTRECEIVED
      |controllerStop -> CONTROLLER ),

REQUESTRECEIVED = ( acceptRequest -> ON
                   |rejectRequest -> ON).
```

Figure 3.8: The specification of the Controller.

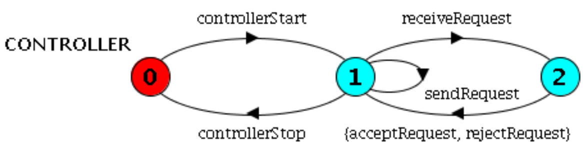


Figure 3.9: A state machine representing the Controller.

resource that does not exist on the recipient host.

Figure 3.7 shows a state machine generated from the process algebra specification.

The Communications Layer.

The communications layer builds on the basic primitives provided by the transport layer for sending and receiving LMUs. In particular, it is composed of two different modules, the *controller* and the *sender/receiver*, both of which are described below.

The Controller. The *controller* implements an application layer Client/Server protocol that allows hosts to *request* the composition and transfer of a particular LMU from a remote host. The protocol encapsulating the request is considered to be specific to the implementation, but it is expected that the request message will be based on LMU attributes. Thus, the controller allows remote hosts to *pull* logical mobility units.

Note that the protocol implemented by the controller is *asynchronous*; a request is *non blocking* and an LMU requested may be retrieved at a later stage. Moreover, a request that has been denied simply results in the requested LMU not being sent - no other information is generated and the requesting host is not notified of the failure. This is shown in Figures 3.8 and 3.9, where sending and receiving an LMU are modelled as two independent actions. This thesis considers potential failure to be typical of the dynamicity of a mobile distributed system - as such, failure is not an exception, rather it is a frequent event that the application programmer (or a middleware system built around this framework) must be aware of.

```
SENDERRECEIVERCONTROL = ( srStart -> ON ),
ON = ( receiveLMU -> ON
      |sendLMU -> ON
      |srStop -> SENDERRECEIVERCONTROL ).

RECEIVER = ( receiveLMU -> deserialise -> DESERIALISATION
            |srStop -> RECEIVER ),

DESERIALISATION = ( deserSuccess -> inspect -> INSPECTION
                   |deserFailed -> RECEIVER ),

INSPECTION = ( accepted -> deployLMU -> deployed -> RECEIVER
              |rejected -> RECEIVER ).

SENDER = ( sendLMU -> examine -> EXAMINATION
          |srStop -> SENDER ),

EXAMINATION = ( trusted -> serialise -> SERIALISING
               |mistrusted -> SENDER ),

SERIALISING = ( serSuccess -> lmuSend -> SENDER
               |serFailed -> SENDER ).

||SENDERRECEIVER = ( SENDERRECEIVERCONTROL || SENDER || RECEIVER ).
```

Figure 3.10: The specification of the Sender/Receiver.

Figure 3.8 shows a process algebra specification for the controller, represented by process `CONTROLLER`. Initially the controller is not active - this means that no requests can be sent and received. The controller can be activated (*controllerStart*) and an active controller (`ON`) can either receive a request (*receiveRequest*), send a request (*sendRequest*) or be deactivated (*controllerStop*). Notice that this implies that application programmers using this framework should be made explicitly aware of the fact that their requests may fail and that no reply is expected when sending a request, as explained above. When a request is received (`REQUESTRECEIVED`), it can be either rejected (*rejectRequest*) or accepted (*acceptRequest*). An accepted request implies that an LMU will be composed and sent (*sendLMU* in the sender/receiver - see below). Note that failures in the transport layer may result in a request not being successfully sent or received. Realisations of the framework may notify the caller about these failures. Figure 3.9 shows a state machine generated from the process algebra.

The Sender/Receiver. Using the infrastructure provided by the serialisation & deserialisation engine and the trust & security layer, the *sender/receiver* allows for sending and receiving LMUs.

Figure 3.10 shows a process algebra representation of the sender/receiver as process

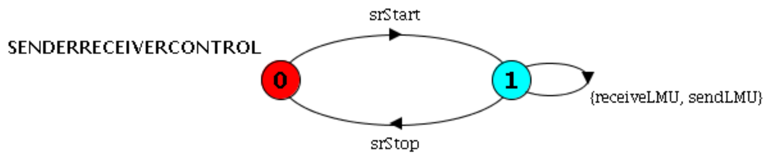


Figure 3.11: A state machine representing the Sender/Receiver Controller.

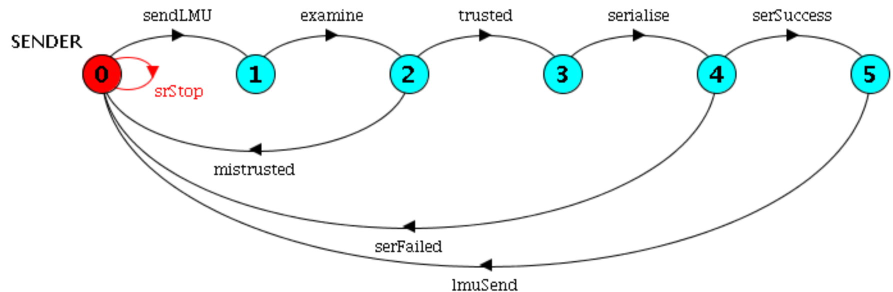


Figure 3.12: A state machine representing the Sender.

SENDERRECEIVER. SENDERRECEIVER is modelled as a concurrent composition of processes SENDER, RECEIVER and SENDERRECEIVERCONTROL. The latter is responsible for enabling and disabling the functionality of sending and receiving LMUs, while SENDER and RECEIVER are responsible for sending and receiving LMUs respectively. Initially, the sender/receiver is inactive, meaning that no LMUs can be sent or received. It can be activated (*srStart*), and an active sender/receiver (ON) can either receive an LMU (*receiveLMU*), send an LMU (*sendLMU*) or be deactivated (*srStop*). When receiving an LMU, the serialisation & deserialisation engine is used to *deserialise* the incoming bitstream into an LMU. The deserialisation process (DESERIALISATION), can either succeed (*deserSuccess*) or fail (*deserFailed*). In the former scenario, the sender/receiver uses the trust & security layer to *inspect* the deserialised LMU (INSPECTION) for malicious elements. This can either result in rejecting the LMU (*rejected*), or accepting it (*accepted*) and passing it to the application for deployment (*deployLMU*).

When sending an LMU (*sendLMU*) the recipient host is first examined (*examine*) by the trust & security layer, to see whether the local host trusts it to send it information. The result of this process (EXAMINATION), is that the host is either *trusted* or *mistrusted*. If the host is trusted, then the serialisation & deserialisation engine attempts to *serialise* the LMU. As mentioned above,the serialisation process (SERIALISING) can either result in success (*serSuccess*), allowing the LMU to be sent (*lmuSend*), or in failure (*serFailed*).

Note that failures in the transport layer may result in the LMU not being successfully sent or received. In this case, deserialisation process in the receiving host, as performed by the serialisation and deserialisation engine, will fail.

Figures 3.11, 3.12 and 3.13 show the state machines generated from the process algebra

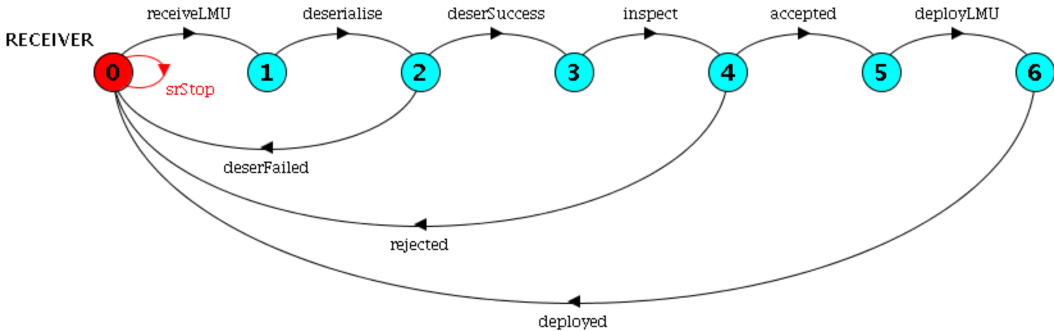


Figure 3.13: A state machine representing the Receiver.

```
||API = ( SENDERRECEIVERCONTROL || SENDER || RECEIVER
         || TRUSTANDSEC || SERDESERENGINE ||CONTROLLER).
```

Figure 3.14: The specification of the API.

in Figure 3.10. Note that the state machine generated from the full composition of the `SENDER`, `RECEIVER` and `SENDERRECEIVERCONTROL` processes is composed of 43 different states and is, therefore, not shown, as it is too complex for visual representation.

Note that LMUs can be sent independently of the controller - i.e. an LMU can be sent without the host having requested it. The recipient does, however, have the option of rejecting an incoming LMU. As such, when modelled as concurrent processes, the *sendLMU* action is available independently of whether a request has been received and accepted. This allows for the operation of the Mobile Agent and Remote Evaluation paradigms, as will be shown in Section 3.3.4.

Both the *controller* and the *sender/receiver* can be realised as concurrent threads which can be started and stopped by the Application Programmer Interface. By allowing this to happen, we allow implementations to stop monitoring for requests, conserving resources, such as battery, and catering for the eventuality of network disconnection.

The Application Programmers Interface.

The Application Programmers Interface (API) builds on the functionality provided by the lower layers and provides primitives that an application can use to create and send an LMU, to request an LMU to be received as well as to start and stop the controller and the sender/receiver. Algebraically, the functionality exposed by the API can be represented as a *concurrent composition* of the above, shown in Figure 3.14. Note that the state machine generated from this is composed of 183 different states and is therefore not shown.

```
APPLICATION = ( deployLMU -> DEPLOYLMU ),

DEPLOYLMU = ( lmuPartialAccept -> deployed -> APPLICATION
            |lmuAccept -> deployed -> APPLICATION
            |lmuInstantiateHandler -> deployed -> APPLICATION
            |lmureject -> deployed -> APPLICATION ).
```

Figure 3.15: The specification of an application receiving an LMU.

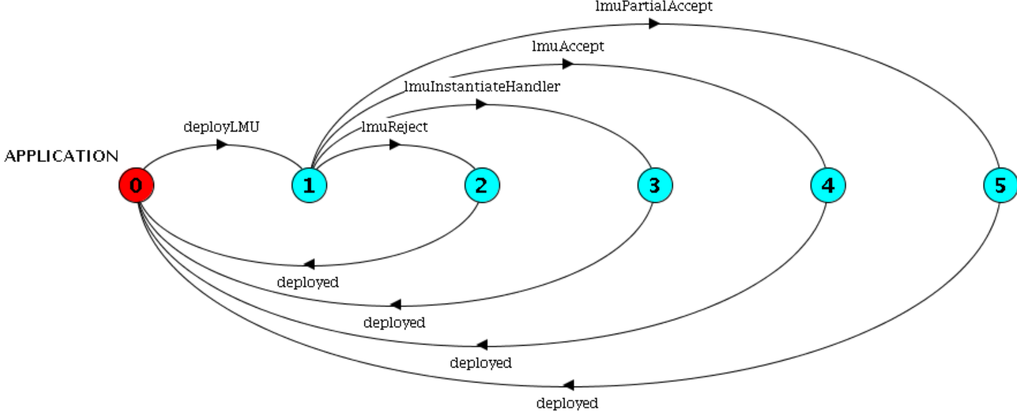


Figure 3.16: A state machine representing an application receiving an LMU.

The Application Layer.

Applications built using this framework are part of the *application layer*. When the communication layer receives an LMU which is successfully deserialised and inspected, it passes it on to an application for deployment. Note that, in this context, an application may represent any software abstraction that uses this framework for logical mobility; as such, applications can range from end-user applications to aspects of the system. This chapter does not detail how different LMUs can be passed to different applications; this is discussed in the following chapter.

Figure 3.15 shows the process algebra specification for an application (`APPLICATION`) deploying an LMU. When an LMU is passed to an application for deployment (*deployLMU*), it is inspected (shown in `DEPLOYLMU`). The results of the inspection can be the following:

- *Partial Acceptance*. Some aspects of the LMU are accepted, while others are rejected. This is represented by *lmuPartialAccept*.
- *Full Acceptance*. All the contents of the LMU are accepted by the application. This is represented by *lmuAccept*.
- *Instantiation of the Handler*. The application may not know how to deploy the LMU

```
property SERIALISETRUSTED = ( trusted -> serialise -> SERIALISETRUSTED ).
property SENDSERIALISED = ( serSuccess -> lmuSend  -> SENDSERIALISED ).
property INSPECTDESERIALISED = ( deserSuccess -> inspect ->INSPECTDESERIALISED ).
property DEPLOYACCEPTED = ( accepted -> deployLMU -> DEPLOYACCEPTED ).

progress SENDANDRECEIVELMUS = { sendLMU, receiveLMU }
progress SENDANDRECEIVEREQUESTS = { sendRequest, sendRequest }
```

Figure 3.17: Safety and liveness properties for the framework.

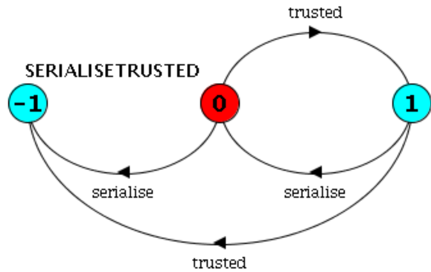


Figure 3.18: A state machine representing the safety property `SERIALISETRUSTED`.

received. If the LMU contains a *Handler*, then the latter can be instantiated to take care of the deployment. This is represented by *lmuInstantiateHandler*.

- *Rejection*. The LMU may also be rejected by the application. There can be many reasons for this - the application may, for example, have no need for the contents of the LMU. This is represented by *lmuReject*.

Figure 3.16 shows a state machine generated by the process algebra of the application.

The full framework can be represented as a concurrent composition of all the layers, as shown in Figure 3.22. The state machine generated is composed of 201 states and cannot be shown. `SERIALISETRUSTED`, `SENDSERIALISED`, `INSPECTDESERIALISED`, `DEPLOYACCEPTED` are the *safety properties* for this framework, which, along with the liveness properties, are expressed in Figure 3.17. In particular:

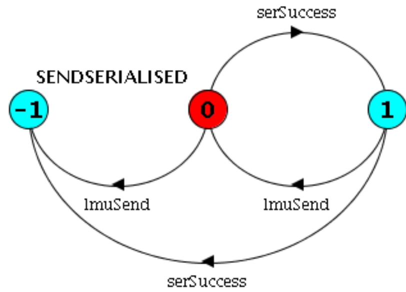


Figure 3.19: A state machine representing the safety property `SENDSERIALISED`.

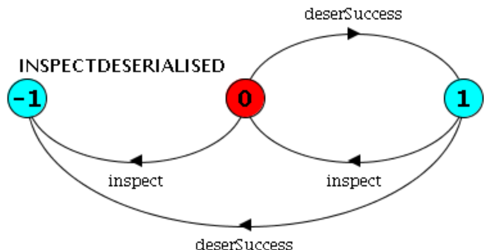


Figure 3.20: A state machine representing the safety property `INSPECTSERIALISED`.

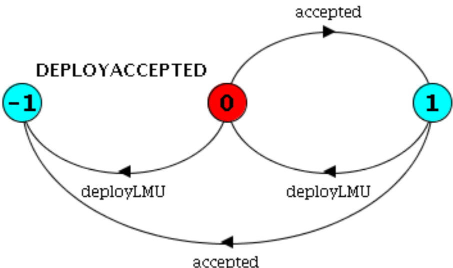


Figure 3.21: A state machine representing the safety property `DEPLOYACCEPTED`.

- `SERIALISETRUSTED` is a safety property that ensures that an LMU will only be serialised to be sent (action *serialise*) if the recipient host is trusted (action *trusted*) to receive it. It is visualised in Figure 3.18.

- `SENDSERIALISED` is a safety property that ensures that an LMU will only be sent (action *lmuSend*) if it has been successfully serialised (action *serSuccess*). It is visualised in Figure 3.19.

- `INSPECTDESERIALISED` is a safety property that ensures that an LMU will only be inspected for security reasons (action *inspect*) if it has been successfully deserialised (action *deserSuccess*). It is visualised in Figure 3.20.

- `DEPLOYACCEPTED` is a safety property that ensures that an LMU will only be deployed (action *deployLMU*) if it is accepted (action *accepted*) by the inspection process. It is visualised in Figure 3.21.

- `SENDANDRECEIVELMUS` is a liveness property that ensures that, given an infinite length of time, the framework will be able to send and receive an infinite number of LMUs (actions *sendLMU* and *receiveLMU*), avoiding deadlocks. In other words, that both sending and receiving will be happening an infinite number of times.

- `SENDANDRECEIVEREQUESTS` is a liveness property that ensures that, given an infinite length of time, the framework will be able to send and receive an infinite number of requests for LMUs (actions *sendRequest* and *receiveRequest*), avoiding deadlocks.

```
||FRAMEWORK = ( SENDERRECEIVERCONTROL || SENDER || RECEIVER
              || TRUSTANDSEC || SERDESERENGINE || CONTROLLER
              || APPLICATION || SERIALISETRUSTED || SENDSERIALISED
              || INSPECTDESERIALISED || DEPLOYACCEPTED ).
```

Figure 3.22: The full specification of the framework.

```
||TWOINSTANCES = ( a:FRAMEWORK || b:FRAMEWORK )
                  /{a.sendRequest/b.receiveRequest, a.receiveLMU/b.lmuSend,
                    b.sendRequest/a.receiveRequest, b.receiveLMU/a.lmuSend}.
```

Figure 3.23: A composition of two instances of the logical mobility framework.

The framework was found to satisfy all safety and liveness properties, by using the FSP model checking tool, LTSA [Magee and Kramer, 1999]. The next section describes the framework and shows how it can be used to offer all the paradigms discussed in 3.2.2.

3.3.4 Transferring Logical Mobility Units

The following paragraphs demonstrate the generality and applicability of this framework, by showing how it can be used by applications to employ the logical mobility paradigms outlined in Section 3.2.2. In particular, this section shows how Code On Demand, Remote Evaluation and Mobile Agents can be mapped onto a sequence of actions on the model of the framework. To illustrate this, two instances of the framework, A & B, are composed. The composition is shown in Figure 3.23. The `/` notation renames actions. This results, for example, in `b.receiveRequest` to be renamed to `a.sendRequest`. This results in modelling that when `A` sends a request, `B` receives it.

Code On Demand

The use of Code on Demand is equivalent to sending a request via the controller and getting the code requested by the sender/receiver. In the following trace, node A requests and receives an LMU from node B:

Step Number	Action	Description
0	`a.srStart`	starts the sender/receiver on node A
1	`a.controllerStart`	starts the controller on node A
2	`b.srStart`	starts the sender/receiver on node B
3	`b.controllerStart`	starts the controller on node B
4	`a.sendRequest`	A sends a request for the code required
5	`b.acceptRequest`	B accepts the request
6	`b.sendLMU`	B packs and tries to send the LMU
7	`b.examine`	B inspects the target node (A) to see whether it is trusted
8	`b.trusted`	B finds that A is trusted
9	`b.serialise`	B tries to serialise the LMU
10	`b.serSuccess`	B successfully serialises the LMU
11	`a.receiveLMU`	B sends the LMU / A receives it
12	`a.deserialise`	A deserialises the LMU
13	`a.deserialised`	the LMU is deserialised and checked for conflicts
14	`a.conflict`	a conflict is detected
15	`a.conflictResolved`	conflict is resolved
16	`a.deserSuccess`	deserialisation process is successfully completed
17	`a.inspect`	LMU is inspected for security
18	`a.accepted`	it is accepted into the system
19	`a.deployLMU`	LMU is passed on to the application for deployment
20	`a.lmuAccept`	the application fully accepts it
21	`a.deployed`	LMU is successfully deployed on A

Remote Evaluation

The use of Remote Evaluation is equivalent to sending the LMU via the sender/receiver. The recipient host may decline the LMU. In the following trace, node A sends an LMU to node B.

Step Number	Action	Description
0	`a.srStart`	starts the sender/receiver on node A
1	`b.srStart`	starts the sender/receiver on node B
2	`a.sendLMU`	A packs and tries to send the LMU to B
3	`a.examine`	A inspects the target node (B) to see whether it is trusted
4	`a.trusted`	A finds that B is trusted
5	`a.serialise`	A tries to serialise the LMU
6	`a.serSuccess`	A successfully serialises the LMU
7	`b.receiveLMU`	A sends the LMU / B receives it
8	`b.deserialise`	B deserialises the LMU
9	`b.deserialised`	the LMU is deserialised and checked for conflicts
10	`b.noConflict`	no conflict was found
11	`b.deserSuccess`	deserialisation process is successfully completed
12	`b.inspect`	LMU is inspected for security
13	`b.accepted`	it is accepted into the system
14	`b.deployLMU`	LMU is passed on to the application for deployment
15	`b.lmuPartialAccept`	the application partially accepts it (i.e. parts of the LMU are discarded)
16	`b.deployed`	the LMU is successfully deployed on B

Note that this framework does not directly address the issue that an application using Remote Evaluation may request a reply based on the execution of the LMU sent. The request may be stored in the properties of the LMU. The reply sent is considered to be an application level issue.

Mobile Agents

The use of Mobile Agents is equivalent to sending an LMU with a Handler, responsible for activating a thread representing the agent on the recipient host. In the following trace, node A sends an agent to node B.

Step Number	Action	Description
0	`a.srStart`	starts the sender/receiver on node A
1	`b.srStart`	starts the sender/receiver on node B
2	`a.sendLMU`	A packs and tries to send the LMU to B
3	`a.examine`	A inspects the target node (B) to see whether it is trusted
4	`a.trusted`	A finds that B is trusted
5	`a.serialise`	A tries to serialise the LMU
6	`a.serSuccess`	A successfully serialises the LMU
7	`b.receiveLMU`	A sends the LMU / B receives it
8	`b.deserialise`	B deserialises the LMU
9	`b.deserialised`	the LMU is deserialised and checked for conflicts
10	`b.noConflict`	no conflict was found
11	`b.deserSuccess`	deserialisation process is successfully completed
12	`b.inspect`	LMU is inspected for security
13	`b.accepted`	it is accepted into the system
14	`b.deployLMU`	LMU is passed on to the application for deployment
15	`b.lmuInstantiateHandler`	application instantiates the handler of the LMU
16	`b.deployed`	the LMU is successfully deployed on B
17		handler starts a thread representing the agent.

The agent can then use API of the framework to migrate itself to another host.

It is important to note that the framework offers the ability to reject an incoming LMU at many different stages. As such, an LMU can be rejected if deserialisation fails, if it is malicious, or if it creates an unresolvable conflict in the system. Moreover, finer-grained control is given to the application programmer, who may inspect the contents of the LMU before accepting it or rejecting it, partially or fully. Similarly, the process of sending an LMU can fail on two stages: if the target host is not trusted or if the serialisation process fails. Finally, realisations of the serialisation & deserialisation engine may decline serialising an LMU if it contains data that should not be shared (for legal reasons, for example) or cannot be shared (as the data can contain non serialisable elements).

Please note that this framework does not differentiate between migration and remote cloning. This is considered the responsibility of the application that creates and packs the LMU. More specifically, to achieve migration, the contents of the LMU must be deleted from the host framework after sending.

3.4 Related Work

Code mobility and, by extension, logical mobility, is a general concept that has been extensively used in the industry and examined by the research community to achieve various aims, from the download of ringtones in the mobile phone industry, to the distribution of system updates in operating systems. It is not the aim of this thesis to provide an exhaustive and general discussion on the use of logical mobility, as it is primarily used as a tool to achieve a particular solution. This section discusses related work on the conceptualisation of logical mobility and illustrates its use in a mobile code toolkit, related to the framework described above. Logical mobility in mobile distributed systems, which is the primary focus of this thesis, is usually offered in terms of a middleware system and its use is considered outside the scope of this chapter - it is discussed in detail in Chapter 5.

[Fuggetta et al., 1998] provide a conceptual framework for reasoning about code mobility, on which this chapter is based. They discuss code mobility and its applications, define a mobile code system, as shown in Figure 3.1, and examine the paradigms of code mobility as discussed in Section 3.2.2. They also provide a survey of mobile code toolkits and outline some application domains for the use of mobile code.

There has also been research in evaluating the performance of logical mobility. [Grassi and Mirandola, 2002] describe a UML-based methodology for performance analysis of logically mobile software architectures. UML sequence and collaboration diagrams, are annotated with mobility-related stereotypes, allowing the developer to model the code migration aspects of the system. The diagrams are then annotated with probabilities and cost information, and a performance model of the application is obtained, allowing the designer to evaluate the choices made. [Grassi and Mirandola, 2003] build on this work to target mobile computing and allow the developer to build a Markov model, the solution of which can be used to find the best code mobility-based adaptation strategy for a given execution context.

[Roman et al., 2000] discuss engineering software for mobile computing. As part of this discussion, they identify some of the benefits of Logical Mobility and define the unit of mobility.

In [Picco et al., 2001], the notion of *location* in a Mobile Unity [Roman et al., 1997] program is used to model the various paradigms of transferring of code between nodes. Although similar to what presented in this chapter, the main difference is that was described before was a framework for moving logical mobility elements, rather than only the move of those elements.

[Popa et al., 2004] describe a technique called *code collection*, which is a mechanism for predictively loading and discarding code units on a node, based on their frequency of use. Realisations of the framework presented in this chapter can potentially use this technique

for efficiency.

Work that is more related to what has been presented in this chapter is μCode [Picco, 1998b]. μCode is a lightweight Java library which provides a minimal set of primitives allowing code mobility. μCode was specifically designed to provide programmers with a set of primitives to move Java Classes and Objects between hosts. The unit of mobility in μCode is the *group*. A group is a collection of classes and objects defined by the programmer. The group also contains two special classes, the *group handler* and the *root* class. The group handler is used to instantiate an object which is utilised to unpack and manipulate the contents of the group. The root class can be used to provide additional information on the group, for example, information on how to spawn a new thread of execution at the destination. Note that the handler and the root class can actually be the same class and it is not necessary for them to be included in the group.

The destination of a μCode group is the MuServer. The MuServer provides the runtime support for the μCode platform. A MuServer can create a μCode group and can receive groups sent over the network.

The framework presented in this chapter shares μCode's objective to offer a very lightweight set of primitives to support code mobility. Its non-obtrusiveness allows it to be easily integrated with various middleware systems, and its small footprint makes it suitable for mobile middleware. There are various differences between the way μCode and the framework presented in this chapter are designed. There is no concept of a *root* class in an LMU, as this functionality can be encapsulated using the handler class or the LMU attributes. In tests conducted with μCode, the root class was found to be redundant. Moreover, there is no concept of an attribute in an μCode group which relies on the Java virtual machine to handle issues of heterogeneity. Another difference is that the functionality of migrating a group from one host to another is offered by the methods of the group object, which uses the functionality offered by the rest of the framework to perform the transfer. In contrast, in the framework presented, this functionality is exported by the communications layer, via the API. It is argued that the reason for having these methods on the group object itself is that μCode is primarily geared for transferring mobile agents. Its primary use has been on a system that used mobile agents [Picco et al., 1999] and it offers methods that specifically deal with threads. Finally, the μCode architecture is monolithic and does not address issues of security and trust.

3.5 Summary

It has been the purpose of this chapter to describe mobile systems conceptually in two layers, the physical and the logical one, and to discuss the concept of mobility in each. In particular, the chapter discussed in detail the concept of Logical Mobility (manifested

using code mobility techniques) as a mechanism that can be used to dynamically add functionality and, thus, adapt a mobile system. More specifically, the concepts of logical and code mobility were detailed including how they can be used. The chapter also identified mobile application domains that show the clear benefits of the use of these techniques.

The chapter defined a conceptual framework that offers the use of logical mobility techniques as a collection of loosely coupled hierarchical layers, which can be used to send and receive classes, objects and data by applications. Those layers were further refined, by representing them using process algebra and describing the framework as a concurrent composition of these processes. The chapter showed how the framework is general enough to be used to offer the use of any logical mobility paradigm by applications.

As such, the framework described in this chapter can be used to offer the *systematic* use of logical mobility by applications. This will be demonstrated in the next two chapters, where the framework will be offered as part of a component model and then instantiated in terms of a middleware system.

Chapter 4

The SATIN Component Meta Model for Mobile Adaptation

The previous chapter discussed the concept of *mobility* in distributed systems and in particular the concept of *logical mobility*, its suitability for mobile systems and how it can be used to create adaptive systems. Furthermore, it detailed a conceptual framework that can offer the flexible use of logical mobility techniques in mobile systems.

But how should a mobile application or system be engineered in order to take advantage of the functionality provided by the logical mobility framework? How should it use it to adapt to accommodate changes to its requirements? How is the logical mobility framework made available to applications? How can the functionality of a system be represented and described, so that the symmetry of the logical mobility framework can be used to identify, request, send and deploy functionality, represented by logical mobility units?

This chapter addresses these questions by presenting the SATIN (System Adaptation Targeting Integrated Networks) component meta-model, a lightweight collocated model that offers the use of logical mobility to applications as a first class citizen, by encapsulating and offering the logical mobility framework discussed in Chapter 3. The work that this chapter discusses has been outlined in [Zachariadis et al., 2004] and is described here in detail.

4.1 Models, Metamodels and Instances

Before beginning this chapter, this section defines the terminology that will be used in the rest of this thesis about component models. Figure 4.1 presents what is generally accepted as a modelling framework for object systems. Level 0, the Object level, hosts entities which

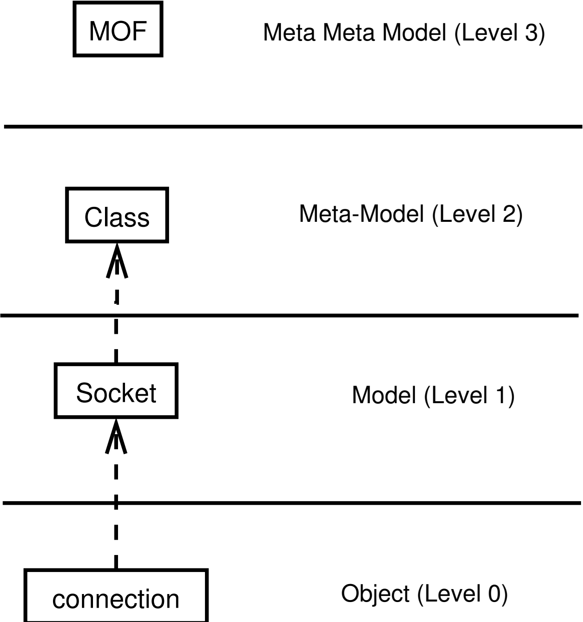


Figure 4.1: The relationship between objects, models, metamodels and meta meta models.

are responsible for holding information about the current state of the system. Figure 4.1 shows an *Object* named `connection` in level 0. Items in the information level reify or instantiate entities in level 1, the type level. The type level defines the behaviour of items in the object level. `Connection` is an instantiation of a `Socket`. Similarly, level 2, the meta-model level, defines how entities in level 1 behave. Figure 4.1 shows that `Socket` is a reification of a *Class*. Finally, level 3, the meta meta model level, states how to define entities in level 2. Figure 4.1 shows that a *class* is defined in the Meta Object Facility [OMG, 2000]. As such, in Figure 4.1, `connection` is an object that is an instance of `Socket` which is a `Class` which is defined in the Meta Object Facility.

There is general confusion in the literature about the terminology used for component-based systems. In particular, it is very frequent that the concepts of component models and component meta-models are used interchangeably. This thesis is concerned with the abstractions defined in levels 0, 1 and 2 of Figure 4.1. As such, this chapter presents the SATIN component metamodel for adaptive mobile systems (level 2), Chapter 5 realises the metamodel as a middleware system (level 1) and Chapter 6 presents its implementation (level 0). However, unless otherwise specified, the term component model as opposed to component metamodel will be used in the remainder of this work, as often happens in the description of other existing systems.

4.2 Components, Distribution and Collocation

Although component based systems are widely used in business client/server type applications [OMG, 1997, Sun Microsystems, 1998a], as well as in desktop systems [Granroth, 2000, Eddon and Eddon, 1998, Rogerson, 1997, The GNOME Project, 2001], their use in mobile devices is very limited. This can be attributed to many factors. In particular, the use of components impairs a computational, memory and storage [1] cost over traditional monolithic systems; until recently, mobile devices had limited resources that could not accommodate this cost. With devices becoming increasingly more capable, as shown in Section 2.1.1, this barrier has been effectively lifted and mobile devices can support lightweight component models. However, as Section 4.7 shows, a lot of existing component models are too heavyweight.

Section 2.1 showed some of the limitations of current approaches; it was claimed that an adaptive approach based on the use of logical mobility primitives and structured in terms of a component model can be used to overcome these limitations, namely that mobile systems are monolithic, failing to interact with their environment and to adapt to its changes. It was further claimed that the use of logical mobility combined with a component-based approach to structure a mobile system offers the following benefits in the target area of this thesis:

- Components break the monolithic structures that currently prevail in mobile systems by promoting the decomposition of applications and systems into a collection of interacting components. The advantages of modular over monolithic software architectures are well understood in the literature and will not be discussed in great detail here. What is of particular interest for this thesis, is the code reusability that components provide (thus reducing the resources required), the separation of interfaces and implementation, and also the potential for late binding, or the ability of an application to select which component to use for a particular task at runtime.
- Components logically structure a system into distinct *units of functionality* which are composed together to form the system. As such, a component provides an abstraction higher than the one defined in Section 3.3.2 to represent aspects of the logical layer of a system. Consequently, components can provide a coarse grained guide on how a system can adapt, by dynamically adding and deleting components.

There are two major types of component models: distributed and local. In a *distributed component model*, like the CORBA Component Model [OMG, 1997], components are distributed amongst different and, potentially, heterogeneous nodes and are interconnected

[1] Depending on the actual system, a component-based approach can lead to a decreased storage cost, because of code (component) reusability.

using networking references. In a *local* or *in-process component model*, such as JavaBeans [Hamilton, 1997], components are collocated in the same address space on a single node, interconnected using local references.

It would seem at first that distributed component models are suitable for mobile devices, since they already address issues of heterogeneity, which are inherent in mobile computing. A comparison between distribution and collocation for object systems can be found in [Emmerich, 2000]. Using the arguments presented there, it is claimed that distributed component models are not suitable for mobile adaptation in a dynamic environment, for the following reasons:

Size: Mobile devices have very limited resources, compared to desktop computers. Distributed component model implementations usually require large amounts of memory and significant CPU power to deliver functionality such as transactions, persistence and concurrency control, which are often not essential in a mobile setting. These primitives can be provided at a higher level (i.e. built using the component model), if needed.

Remote References: A reference to a component in a local, shared memory system, is usually a pointer, which is a lightweight data structure. In distributed systems however, the reference is usually a more substantial data structure, that encodes location and security information. The process of calling a method in a distributed object involves marshaling and unmarshaling both request and reply. Most distributed component model implementations assume a continuous network connection with a high bandwidth and low latency to deliver synchronous remote procedure calls. On the other hand, mobile devices usually have intermittent network connectivity at low bandwidth and high latency. Invalidating those assumptions usually implies invalidating the remote component reference. As such, network references are often unsuitable for mobile applications, not providing for system autonomy when invalidated.

Complexity: Distributed component models usually assume a client / server architecture, with a predictable number of clients accessing one or more application servers. A system is seen as a collection of components distributed in a predictable number of potentially heterogeneous devices. The physical mobility and temporal nature of the networking connectivity of mobile devices, as outlined in Section 3.1, dictates that the devices form highly dynamic networks which may even be completely structureless (ad hoc). Even when the latter is not the case, mobile devices form significantly less predictable topologies than distributed systems in fixed networks. Given this, mobile applications are hardly comparable to standard distributed systems, in terms of structure and complexity.

The next section presents the SATIN component model, which is a collocated component

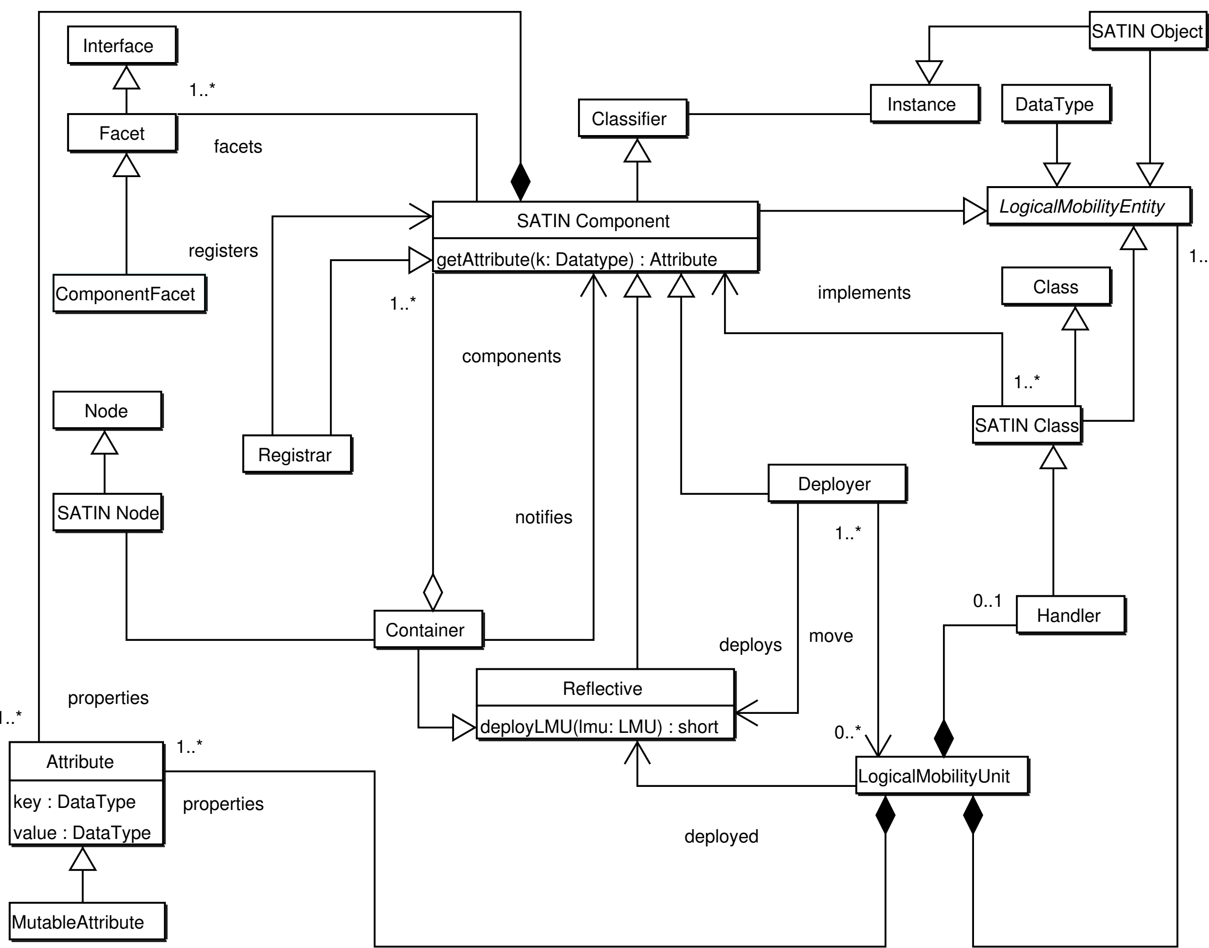


Figure 4.2: The SATIN metamodel.

model for mobile adaptive systems, that offers the use of logical mobility primitives as a first class citizen. Systems engineered using SATIN can leverage the logical mobility platform presented in Chapter 3 to adapt to changes to their requirements.

4.3 A Component Model for Mobile Adaptation

4.3.1 Component Model Overview

The SATIN component model is a *local*, or *in process*, *reflective* component model, targeting mobile devices, that uses logical mobility primitives to provide distribution services and offers the flexible use of those primitives to applications; Instead of relying on the invocation of remote services via the network, SATIN components are collocated on the same address space. The model supports the remote cloning of components between hosts, providing

for system autonomy when network connectivity is missing or is unreliable. As such, an instance of SATIN is represented as a collection of local components, interconnected using local references and well defined interfaces, deployed on a single host. The model also offers support for *structural reflection* on a component level; as such, an application can inspect at runtime which components are available locally and can select one to perform a particular task, or it can dynamically change the configuration of the system by adding or removing components.

The SATIN component model, as shown in Figure 4.2, is a Meta Object Facility [OMG, 2000]-compliant extension of the UML [OMG, 2003] meta model, version 1.5. It builds upon and extends the concepts of Classifier, Node, Class, Interface, DataType and Instance [2]. An extension of the metamodel was preferred to using the UML extension mechanism, because the concepts introduced by SATIN are too radical to be supported by the latter. The most novel aspect of the model is the way in which it offers distribution services to local components, allowing instances to dynamically send and receive components at runtime. This section continues by describing the model in detail.

4.3.2 Components

A SATIN *component* encapsulates particular functionality, such as, for instance, a user interface, an advertising mechanism, a service, an audio codec or a compression library. SATIN components separate interfaces and implementations. A component can implement one or more interfaces, called *facets*, with each facet offering any number of operations. Each facet is immutable. A component implementation is achieved by one or several SATIN classes.

Component Metadata

Although the SATIN component model is a local one, it is used to represent a largely heterogeneous set of devices and architectures. As such, the SATIN component abstraction must be rich enough to describe components that may be deployed over a large number of platforms. To this end, parallels are drawn with the Debian Project's [Murdock, 1994] `.deb` packaging system. Debian is an operating system the packages of which are deployed over twelve different hardware architectures and different operating system kernels – a Debian system may run on the Linux, Hurd, NetBSD or FreeBSD kernels; it is composed of hundreds of different installable packages, most of which have various inter-dependencies, to create a complete system. The Debian package format uses metadata to describe the heterogeneity of these platforms. SATIN follows a similar approach, by using attributes to

[2]Note that Node, Classifier, Class, DataType and Interface are taken from UML Core. Instance is taken from UML Common Behaviour. [OMG, 2003]

Key	Description
ID	Component Identifier
DESC	Description
SIZE	Installed size
DEP	Dependencies
ARCH	Machine Architecture Required (i386,JVM,etc.)
VER	Implementation Version
FACETS	Facets Provided
SIG	Digital Signature

Figure 4.3: A list of suggested attributes for the SATIN component model.

describe a component.

Similarly to the logical mobility unit attributes (see Section 3.3.2), a SATIN component *attribute* is a tuple containing a *key* and a *value*. The set of all attributes of a component are the *properties* of the component, which map each key to the value associated with it. A set of attribute keys, or an ontology for both keys and values, are not defined at the metamodel stage. It is suggested that developers use an `ID` attribute, that acts as a component identifier, similar to the PalmOS Creator ID (see Section 2.1.1) and a VER attribute, which denotes the version of the component implementation. As such, a component implementation can be uniquely identified using the `ID` and `VER` attributes. Moreover, this allows for differentiating between different versions of a component implementation. It is also suggested that each component will have a `DEP` attribute, which expresses the dependencies of the particular component to other ones. Figure 4.3 shows an example of the keys of the properties of a particular component and a description of their meaning. Note that the metamodel only defines the attribute as an entity, but does not prescribe requirements on particular attributes. This is done by any reifications of the metamodel. The attributes presented in Figure 4.3 are shown here as an example, for reasons of clarity.

Attributes can be mutable. The purpose of having mutable attributes is that they allow a component implementation to save its state externally to its logic. This allows, in principle, to update a component while maintaining its state.

Each SATIN component implements at least one facet, the `component` facet. The purpose of the component facet is to allow callers to reason about the component and its attributes. As such, it permits access to the properties of the component, by retrieving, adding, removing and modifying attributes. The component facet also contains a *constructor*, which is used to initialise the component into the system and a *destructor*, which is used to remove the component from the system (see below). Finally, the component facet allows for enabling or disabling a component (see Section 4.3.5).

4.3.3 Components and Containers

The central component of every instance of SATIN is the *container* component. A container is a component specialisation that acts as a registry of components installed on an instance of SATIN. As such, a reference to each component is available via the container.

The container component implements a specialisation of the component facet that exports functionality allowing for querying for components that match a given set of attributes. Moreover, it permits the registration of listeners (represented by components that implement the `ComponentListener` facet) to be notified when components matching a set of attributes given by the listener are added or removed. This allows the system to react to changes in local component availability. For example, media player applications can be notified when components implementing the `AUDIOFORMAT` facet are deployed in the system. Thus, queries for components that satisfy a set of attributes can be performed.

The container can dynamically add or delete components to and from the system. Registration and de-registration of components is delegated to one or more *registrars*. A registrar, which is a component implementing a specialisation of the component facet, is responsible for loading the component, validating its dependencies and adding it to the registry. When removing a component, a registrar is responsible for checking that the removal of the particular component will not invalidate the dependencies of others and then calling its destructor. Different registrars can have different policies on loading and removing components (from different sources, for example) and verifying that the dependencies are satisfied. For example, implementations of the container and registrar can keep track of how often components are used - this frequency based approach can be used to drop least used components when the system runs out of memory. Moreover, implementations of the registrar can emit events to notify interested listeners on component registration failures. Finally, registrar implementations may offer *atomic* registration and removal of groups of components.

The use of the container allows for introspecting the status of the platform, as callers of the container facet can reason about the current availability of functionality, encapsulated in components. Combined with the use of registrars to allow dynamic addition and removal of components, the SATIN container offers structural reflection at the component level, that is the ability to reason about the components in the system.

4.3.4 Distribution and Logical Mobility

A system built using SATIN can reconfigure itself by means of the logical mobility primitives provided by the framework described in the previous chapter. Distribution is not built into the components themselves, as SATIN is a local component model, but it is provided by the model as a service; SATIN instances can, in fact, dynamically send and receive

components. This functionality is provided using Logical Mobility Entities and Units, as well as Deployer and Reflective components. Their relationship is outlined in Figure 4.2 and discussed below.

The Logical Mobility Entity (LME) abstraction, as defined in Section 3.3.2 is extended to a generalisation of a class, an instance, data or a SATIN component. As such, a SATIN Logical Mobility Unit (LMU) is a container which is able to store arbitrary numbers of classes, instances, data and components. Consequently, an LMU can be used to encapsulate various granularities of logical mobility from an individual class to a collection of components.

When the deployment of logical mobility units was discussed in the previous chapter, it was not specified how different LMUs are deployed to different applications. In terms of the SATIN component model, an LMU is always deployed in a *Reflective* component. A Reflective component is a component specialisation that can be *adapted* at runtime by receiving LMUs from the SATIN migration services. By definition, the container is always a reflective component, as it can receive and host new components at runtime.

A SATIN LMU has two required attributes; `TARG`, which specifies the intended target node and `LTARG`, which specifies the logical target or reflective component in the host specified by `TARG` that the LMU is going to be deployed to. For example, when deploying a component into the system, the value of the `LTARG` attribute points to the instance of the container in that system. As such, `TARG` is referred to as the physical destination of the LMU, whereas `LTARG` is the logical destination.

A SATIN application cannot send an LMU directly. The functionality of sending, receiving and deploying LMUs is abstracted and handled by the *Deployer*. The Deployer is a SATIN component specialisation that manages requesting, creating, sending, receiving and deploying LMUs to the appropriate reflective components. In particular, the Deployer is an encapsulation of the logical mobility platform defined in Section 3.3. A Deployer is directly accessible to any application through the container.

A Deployer will reject any request to send LMUs that do not specify a logical and a physical destination. Otherwise, it is responsible for serialising and sending the LMU to the Deployer component instance located at the physical destination. When receiving an LMU, the Deployer uses the container to verify that the component identified by the logical destination of the LMU exists in the local SATIN instance and that it is a reflective component. The LMU is then moved to its logical destination, which has the option of inspecting the contents before deployment; by using the methods exported by the LMU, a reflective component can access the properties and contents of the LMU before accepting it. Thus, a reflective component behaves as specified by the `APPLICATION` process in Section 3.3. As such, the inspection can result either in *full acceptance*, which means that the contents of the LMU are accepted in their entirety; *partial acceptance*, which means that

parts of the LMU are accepted and others discarded; *rejection*, which means that the LMU is rejected and dropped; *handler instantiation*, which means that the reflective component instantiates the Handler, encapsulated in the LMU, to perform the deployment. The result is determined by the reflective component, based on the contents of the LMU.

As specified by the `CONTROLLER` process in Section 3.3, a Deployer also listens for requests and sends requests for LMUs from/to other hosts. A request message is formed using the properties of the LMU requested. The details of the request message are left to instantiations of the metamodel.

4.3.5 Component Lifecycle

SATIN supports a very simple and lightweight component lifecycle; extensions to the model may augment it with further functionality. As such, when a component is passed on to the container for registration, by loading it from disk, using a Deployer, etc., the container delegates registration to a registrar component. If there are more than one registrars available, which registrar is chosen is left to the implementation. The registrar is responsible for checking that the dependencies of the component are satisfied, instantiating the component using its constructor and adding it to the registry. Note that the component facet prescribes a single constructor. An instantiated component can use the container facet to get references to any other components that it may require. A component deployed and instantiated in the container may be in one of the following states: `ENABLED` or `DISABLED`. The semantics of those depend on the component implementation. The initial state also depends on the constructor of the component. The functionality needed to manipulate the state of the component is exported by the component facet.

SATIN does not distinguish between multiple instances of the same component. This is further discussed in Section 4.6. When removing a component, a registrar is responsible for verifying that the removal of the component does not break any dependencies in the system, disabling it, calling the destructor of the component and then removing it from the registry. Similarly to component registration, if there are more than one registrar component available, the semantics of which registrar to chose to perform the removal process are left to the implementation. Instantiations of the metamodel may choose to associate the registrar that registered the component with the component itself. In this way, the same registrar can be automatically called to remove the component when requested. Finally, the semantics of the verification that the removal of the component does not leave the system in an inconsistent state are also left to the implementation.

4.4 Static Semantics

This section further refines the SATIN metamodel presented above, by using the Object Constraint Language [OMG, 2003] to encode static semantics on the architecture presented in Figure 4.2. Note that the expressions presented before built upon and extend those presented in Section 3.3.2.

Constraint 1:

```
context LogicalMobilityUnit
inv:  hasAttribute( LTARG ) = true
```

Constraint 1 denotes that a Logical Mobility Unit must have a logical destination defined.

Constraint 2:

```
context LogicalMobilityUnit
inv:  hasAttribute( TARG ) = true
```

Constraint 2 denotes that a Logical Mobility Unit must have a physical destination defined.

Constraint 3:

```
context Component
def:  hasAttribute( k :  DataType ) :  Boolean =
                                        self.properties->exists( key = k )
```

Constraint 3 is an auxiliary function that checks whether a SATIN component has an attribute with key `k`.

Constraint 4:

```
context Component::getAttribute( k :  DataType ) :  Attribute
pre :  hasAttribute( k ) = true
post :  result = self.properties -> select( key = k)
```

Constraint 4 prescribes that if an attribute requested is defined in the component, then that attribute is returned to the caller.

Constraint 5:

```
context Component::getAttribute( k :  DataType ) :  Attribute
pre :  hasAttribute( k ) = false
post :  result = null
```

Constraint 5 prescribes that if the attribute requested for is not defined in the component, then null is returned to the caller. Constraints 3, 4 and 5 essentially dictate that each attribute is uniquely identified by its key in the context of the properties of a component.

Constraint 6:

```
context Container
inv :  self.components->select( oclIsTypeOf( Deployer ) )->size() >= 1
```

Constraint 6 prescribes that each instance of the container has at least one deployer registered with it.

Constraint 7:

```
context Container
inv :  self.components->select( oclIsTypeOf( Registrar ) )->size() >= 1
```

Constraint 7 prescribes that each instance of the container has at least one registrar registered with it.

Constraint 8:

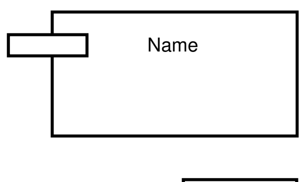
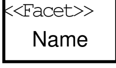
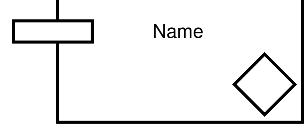
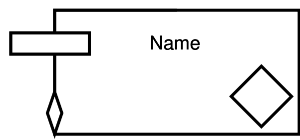
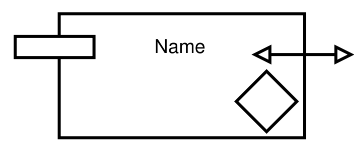
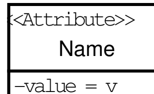
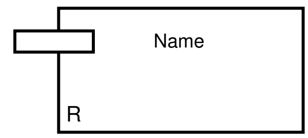
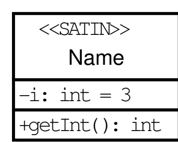
```
context Component
inv :  self.facets->select( oclIsTypeOf( ComponentFacet ) )->size() = 1
```

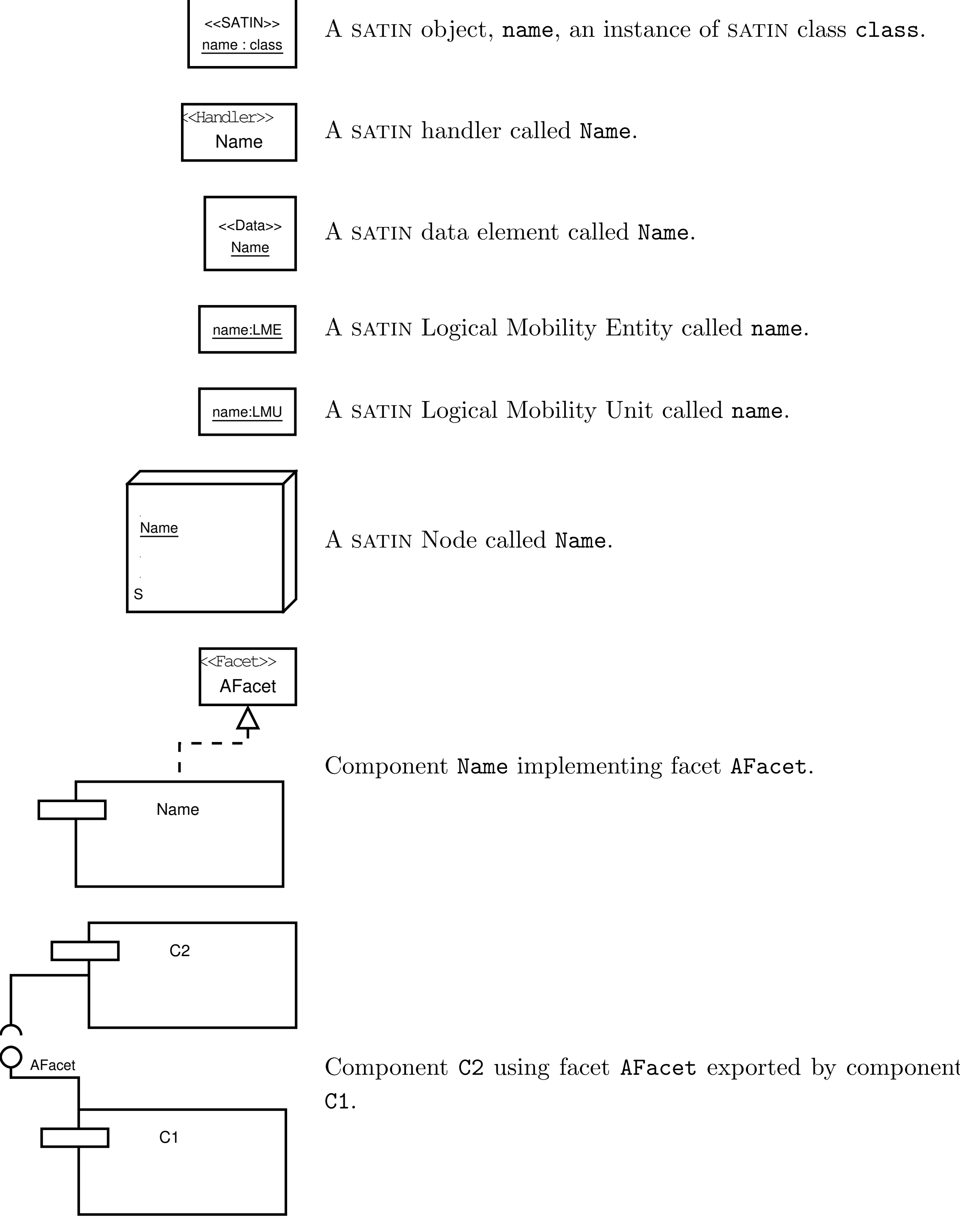
Constraint 9 prescribes that each component instance implements at a specialisation of the component facet.

4.5 Concrete Semantics

This section presents a notation for specifying models that realise the SATIN meta component model. This notation is used to describe various realisations of the metamodel further on.

Note that as the SATIN meta model extends the UML meta model, only the concepts that are new or differ from those defined in the UML meta model are presented here. The reader is referred to [OMG, 2003] for completeness.

Notation	Description
	A SATIN Component called `Name`.
	A SATIN facet called `Name`.
	A reflective SATIN component called `Name`.
	A SATIN container called `Name`. Note that it is also a reflective component.
	A SATIN deployer called `Name`. Note that it is also a reflective component.
	A SATIN Attribute with key `Name` and value `v`.
	A SATIN registrar component called `Name`.
	A SATIN class called `Name`, with a private integer, `i` the initial value of which is 3, and a public method called `getInt` that returns an integer.



A SATIN object, `name`, an instance of SATIN class `class`.

A SATIN handler called `Name`.

A SATIN data element called `Name`.

A SATIN Logical Mobility Entity called `name`.

A SATIN Logical Mobility Unit called `name`.

A SATIN Node called `Name`.

Component `Name` implementing facet `AFacet`.

Component `C2` using facet `AFacet` exported by component `C1`.

4.6 Discussion

The rationale behind the design of the SATIN component model is to create a very lightweight component model that offers the flexible use of logical mobility primitives. SATIN should be easy to integrate into existing projects and easy to use to engineer new systems. This will be evaluated in Chapters 5 and 6.

SATIN can be used to build very modular systems. An instance of SATIN can be static, for example, if it does not have any registrars or any deployer components installed. Although this is not particularly novel, it should be noted that SATIN is flexible enough to allow it. Moreover, the SATIN component model allows for using logical mobility techniques as a *computational primitive*; In fact, not only components, but individual classes and instances can be sent and received. This is further discussed in the next chapter. SATIN can be easily extended to accommodate remote components as well, by having two containers; one that encapsulates local components and one that holds references to remote ones; this is discussed in the concluding chapter. It is also important to note that SATIN does not make any type of client/server type distinctions - a SATIN instance can be fully symmetric, allowing for both sending and receiving LMUs. To this end, SATIN makes a number of design choices, to promote ease of use, flexibility and limited demand on resources. These are discussed in the following paragraphs.

SATIN does not directly support or dictate the use of component frameworks [Szyperski, 1999] or component sub-composition. Component frameworks refer to the interaction of a collection of components by a common set of interfaces. Although this is not directly supported, extensions to the model could accommodate for it, using the properties of the components and potentially a specialisation of the container. Component frameworks can also be used to constrain adaptation via various rules.

The SATIN component model does not define a binary level interoperability standard, like the vtable data structure of COM, the intermediate language of .NET or the bytecode of the Java Virtual Machine. Similarly, it does not define a language in which to specify the facets. The component metadata, however, can be used to describe the particular requirements of each component implementation with regards to binary level interoperability. This issue is left to the implementors; the author used the Java Virtual Machine bytecode and Java interfaces for interoperability and facet specification in the SATIN implementation, as will be shown in the following chapters.

Another issue is related to circular or conflicting dependencies. This is a problem with most dependency-based packaging formats, like `.deb` of the Debian project or the RedHat Package Manager (`RPM`) [Bailey, 2000]. It is resolved by the package, or, in this case, component implementors.

Moreover, the SATIN component model does not differentiate between the availability of

a component in the container and different instances of the same component. Depending on the actual component, multiple instances may be needed to let applications perform different tasks. Alternatively, a single instance may be able to provide the functionality needed to accommodate all applications. As such, SATIN does not differentiate between availability of a component and instances of the component at the component metamodel level, but delegates this and the functionality required to instantiate multiple instances of the component to the component implementor. Extensions to the model can use specialisations of the container to support the direct management of multiple instances. SATIN provides a minimal and generic API and component lifecycle via the component facet. This approach allows deployment of realisations on a wide range of resource constrained devices. Instantiations can extend the component facet to offer more specific and complex functionality.

A binary format for serialised components or LMUs is not defined. This is, again, left to the implementation.

Continuing, the SATIN component model does not directly support abstract components (which cannot be instantiated), as the objective of abstract components can also be achieved using facets.

Finally, it should be noted that the only non-functional properties of a system that SATIN provides for, are the functionality encapsulated by a Deployer and by the Container. Other properties, such as transactions, or a generic event system are not directly addressed by SATIN but can be built on top of it and expressed as SATIN components; components that require this functionality can express that requirement in their properties.

4.7 Related Work

The use of components in either static or mobile systems is not a novel idea and has been employed in many systems. The novel aspect of the SATIN component model is the way in which logical mobility is offered in a lightweight component model aimed for mobile devices. As such, it is not the purpose of this section to provide an exhaustive review of the available component models. Given this, the section summarises a number of local and distributed component models that bear some similarities to the SATIN component model.

4.7.1 In-process Component Models

JavaBeans [Hamilton, 1997] are a component model technology introduced by SUN Microsystems in Java 1.1. A *Bean* is a Java class that that follows certain conventions: It

is augmented with metadata described by properties, it can emit and listen to events and is packaged into a Java archive (.jar), along with all the Java classes that it uses. Each property can be either read only, write only or read-write. Beans are mainly used for graphical user interface development. The JavaBeans model is similar to SATIN in the way that each component is described with attributes.

The Component Object Model (COM) [Rogerson, 1997] is a component model introduced by Microsoft, that facilitates binary compatibility by defining a binary-level interoperability standard, encapsulated by the vtable data structure. As such, COM components can be developed in different languages. COM defines the Microsoft Interface Definition Language (MIDL), that is used to express COM interfaces, which are immutable. COM also uses attributes to define components. Similarly to the component facet of SATIN, each COM component implements the *IUnknown* interface, which also allows for checking whether a component supports a given interface. COM allows for multiple instances of each component. It also has a centralised repository of components called the *registry*. Each COM component is identified by a globally unique identifier (`GUID`), similarly to the `ID` attribute that was suggested in Section 4.3.2 for SATIN components. COM also addresses the issue of component versioning. Finally, COM provides for distributed components via the DCOM extension, which will be discussed later on.

OpenCOM [Clarke et al., 2001] is a lightweight component model based on Microsoft COM, the purpose of which is to implement in a popular component model the reflective and adaptation capabilities identified in [Blair et al., 1998]. In particular, OpenCOM builds on a subset of COM and adds support for pre and post function call interception. It also makes explicit the interdependencies among the components (similarly to SATIN) and adds mechanism-level support for system reconfiguration and mutual exclusion locks to serialise operations that modify the configuration of the components. OpenCOM provides a component that is available in every instance of OpenCOM, called *IOpenCOM*; this is similar to the SATIN container, in that it acts as a repository of available component types. Unlike SATIN, OpenCOM differentiates between component availability and different component instances, by having a more complex component lifecycle (implemented by the *ILifeCycle* interface). Both COM and, consequently, OpenCOM allow for the dynamic configuration of components; components which are no longer used can be deleted. This can be built into SATIN using a container and registrar specialisation. OpenCOM version 2 [Coulson et al., 2004] is not based on Microsoft COM like the original version, but can be built on different kernels depending on the target platform. OpenCOM uses the concept of pluggable loading and binding frameworks. Loading encapsulates actions which in SATIN are taken care of by the SATIN container. The binding framework is responsible for distributing references to components.

COM and OpenCOM are similar to SATIN, in that they provide a lightweight component model with support for reconfiguration; however, both support a number of non-functional aspects by default. Moreover, SATIN focuses on directly supporting reconfiguration via

the *systematic* and *symmetric* use of logical mobility techniques which are built into the component model. Although COM and OpenCOM allow for the dynamic addition of components, they do not provide the rich component deployment abstractions that are built into the SATIN metamodel. COM and OpenCOM explicitly define component connections, which are bindings between components. Those connections are established by the underlying infrastructure. Although SATIN components can explicitly express their dependencies via their attributes, the SATIN framework does not automatically give references of the dependencies of a component - this implies that components manually request from the container for references to individual components, when needed. Although this requires more work for the component implementor, it is by its nature more lightweight, as component references are only distributed when requested by the component implementation. To emulate the behaviour of OpenCOM in this respect, a customised component lifecycle and registrar would be needed. Moreover, OpenCOM is superior to SATIN with respect to the locking abilities that it provides for component references; to emulate this behaviour with SATIN, modifications of the container and registrar abstractions would be needed. OpenCOM, unlike SATIN does not provide logical mobility primitives as first class citizens into the model.

Beanome [Cervantes and Hall, 2002] is a component model for the Open Services Gateway Initiative (OSGi) Framework [The OSGi Alliance, 1999]. OSGi is a commercial framework for the Java platform, that allows service providers to deliver services to consumer devices attached to a residential network and to manage those devices remotely. Each OSGi service has an interface and is implemented by a *bundle*, which is the minimal and only unit of transfer in the OSGi framework . Each bundle is encapsulated in a Java archive and described using metadata. The OSGi platform also defines a number of standard services. Beanome builds on OSGi by defining a lightweight collocated component model which is used to compose applications. The main limitation of Beanome/OSGi in relation to SATIN is that they define a strict client/server architecture. In comparison, SATIN instances are symmetric, being able to both send and receive LMUs.

Gravity [Cervantes and Hall, 2004, Hall and Cervantes, 2003] is another in-process component model built on top of OSGi, that allows for reconfiguration of user-oriented applications, by focusing on the dynamic availability of components. Similarly to SATIN, it assumes that components may come and go at runtime. Like Beanome, its main limitation is that it makes a clear distinction between clients and servers and only supports the reconfiguration of a Gravity client as dictated by a server, that acts as the central arbitrator to the reconfiguration process.

The Dynamically Programmable and Reconfigurable Software (DPRS) architecture [Roman and Islam, 2004] discusses a design for dynamic programmable and reconfigurable systems. The notion of a Micro-Building Block (MBB) is defined, which is, essentially, a minimal component that inputs data in the form of tuples, performs some action on it and outputs a result in tuples. The state of the MBB is stored as tuples in a

state storage area, that is provided by the system. Collections of related MBBs are represented by the Domain abstraction. Systems are defined as a collection of Domains, and can be represented using a graph of MBBs. Actions on the services are defined either by using interpreted statements that lead to a deterministic traversal of the tree, or compiled statements, which are themselves represented using an MBB. Changing the behaviour of a system consists of either replacing an MBB, or modifying the interpreted statements. An MBB is similar to a SATIN component that implements a single facet with a single method. An MBB can store its state externally to its logic, as can a SATIN component by using mutable attributes. The main difference to SATIN is that the latter defines a way to dynamically reconfigure the system by sending and receiving components, via the use of the Deployer and Reflective components. The DPRS architecture does not define any logical mobility mechanism. Finally, the DPRS implementation is rather heavyweight, as witnessed in its testing procedure that requires multi-GHz machines, whereas Chapter 6 shows that SATIN is lightweight.

There are a number of other collocated component model systems, such as Bonobo [The GNOME Project, 2001], KParts [Granroth, 2000] and XPCOM [The Mozilla Foundation, 2003]. These will not be discussed, as they are not directly related to physical or logical mobility.

4.7.2 Distributed Component Models

There have also been a number of distributed component models researched and implemented in industry. This section will outline three of the most widely used distributed component models, as well as some that have been specifically engineered for mobility.

Enterprise Java Beans (EJB) [Sun Microsystems, 1998a] is a distributed component model introduced by Sun Microsystems as part of the Java 2 Enterprise Edition [Sun Microsystems, Inc., 2001a] framework. The EJB model is based on the JavaBeans framework and makes a strict separation between clients and servers. It is a resource-demanding framework, that offers transactions, threading & process control, security and other non functional properties to application designers. The EJB framework defines the notion of a *container*, that SATIN borrows. The EJB container hosts components and offers lifecycle operations as well as the additional services such as transactions etc.

Microsoft DCOM [Eddon and Eddon, 1998] extends COM to support communication between components residing on different nodes, by providing a runtime that is used to marshal and unmarshal requests and replies.

The CORBA Component Model [OMG, 1997] builds on the CORBA distributed object model [OMG, 1995]. It defines a CORBA component as a new CORBA meta type. A

CORBA component is defined by a series of attributes. Components implement (or *provide*) particular interfaces called *facets* and connect to other components via the use of *receptacles*. Facets are specified in the Interface Definition Language (IDL). The SATIN metamodel borrows the term *facets* from the CORBA Component Model. All CORBA components implement the *Navigation* facet, which is similar to the component facet in SATIN or the IUnknown interface in COM. The CORBA component model provides a number of non-functional properties, such as an event model and persistence.

P2PComp [Ferscha et al., 2004] is a lightweight service-oriented component model for mobile devices, built using the OSGi framework. P2PComp allows for location independent synchronous and asynchronous communication between components. Components can provide services and can migrate between nodes. P2PComp offers complete location transparency; as such, programmers cannot know if a method call is executed locally or remotely.

PCOM [Becker et al., 2004] is a distributed component model for pervasive computing. Built on top of BASE [Becker and Schiele, 2003], a middleware system that allows for dynamically selecting communication protocol stacks, PCOM allows for designing application as a collection of potentially distributed components, which make their dependencies explicit. If those dependencies are invalidated, PCOM can attempt to automatically adapt by detecting alternatives according to various strategies.

FarGo[Holder et al., 1999] is a component system that provides dynamic distributed application layout support and a monitoring service that enables applications to register and react to specific system events. Dynamic application layout permits distributing the logical layer of an application at runtime. FarGo, which is implemented as an extension of Java, provides component mobility, allowing components to be attached to the same address space, or conversely, detached into different address spaces. The basic unit of mobility in FarGo is the *complet*. A complet, analogous to an application component, is a collection of objects with a FarGo application being typically comprised of a collection of complets. FarGo allows for complet migration. When a complet moves from one host to another, all complet references are updated so that they remain valid. Note that FarGo is geared for legacy and static component model systems.

FarGo-DA [Weinsberg and Ben-Shaul, 2002] is a mobility-related extension of FarGo, providing a mobile framework for resource-constrained devices that allows disconnected operations. FarGo-DA works on the assumption that mobile devices are offered services by particular servers in their environments. As such, FarGo-DA extends the complet to a Disconnected Aware complet. When a DA complet is disconnected, it has a number of options to allow the remote reference to remain valid. These options include complet cloning, replacing the reference and more. The implementation of FarGo-DA relies on the Java Remote Method Invocation [Sun Microsystems, 1998c] (RMI) framework. As such, it uses code on demand and client server interactions to allow mobile devices partial access

to remote services, even when disconnected. The SATIN component metamodel provides a much more general use of logical mobility primitives, and focuses on the reconfiguration of autonomous hosts.

In [Mikic-Rakic and Medvidovic, 2002], a software architecture [Shaw and Garlan, 1996] - based distributed component model is proposed, that has the ability to update the components that constitute applications engineered using it. Using the concepts of components, which describe the logic and state of the system, connectors, which are responsible for interconnecting local and remote components, and configurations, which define topologies of components and connectors, the approach requires pre-loading of the software architecture skeleton (or meta-level configuration) on all hosts where the component-based application is to be deployed. Main differences with SATIN are that SATIN does not provide a formal software architecture description, other than exposing component interdependencies, although something more formal could be built using the component properties. On the other hand, SATIN offers more fine grained use of Logical Mobility built into the model (whereas the approach in [Mikic-Rakic and Medvidovic, 2002] can only send and receive components) and allows for reflection and late binding, without requiring that any architecture description be preloaded on any node.

4.8 Summary

This chapter builds on Chapter 3, by encapsulating the logical mobility platform described in Section 3.3 and offering its systematic use as part of SATIN, a flexible and lightweight collocated component model. As such, this chapter defines a model that can be used to build adaptable mobile systems, by representing them as a collection of interacting components that can be dynamically added and removed.

Compared to related work, this approach offers the flexible use of logical mobility primitives to applications. It offers a symmetric model, which does not make distinctions between clients and servers. Moreover, as it focuses on adaptation via the reconfiguration of collocated components, it allows the system to function autonomously, even in the event of network unavailability and disconnection.

The next chapter realises the SATIN meta model into a component-based middleware system for adaptive mobile applications.

Chapter 5

The SATIN Mobile Computing Middleware System

The previous chapter described the design of a lightweight component meta model for mobile systems, that offers the flexible and symmetric use of logical mobility primitives. It was claimed that the SATIN component meta model can be used to build adaptable mobile computing systems and applications.

This chapter instantiates the SATIN component meta model presented, to build the SATIN middleware system, a lightweight, adaptable and component-based middleware system, that offers the flexible use of logical mobility primitives to applications. The SATIN middleware system is *one possible realisation* of the component meta model into a full middleware. It is used to illustrate and evaluate how the metamodel can be extended and instanced to realise a mobile computing middleware system. The SATIN middleware system is mapped to the type layer of Figure 4.1.

The chapter presents the middleware system in detail. It illustrates how the component meta model is realised to offer various middleware services and how these services are useful for mobile adaptation. Moreover, it shows how the system itself and applications developed over it are adaptable, via the flexible use of logical mobility primitives that the deployer provides. Finally, the SATIN middleware system is compared to other mobile middleware systems that employ the use of logical mobility techniques.

It is important to note that from the perspective of the component meta model, the services that the SATIN middleware system offers are simple component realisations. This chapter demonstrates how the component model described in Chapter 4 can be instantiated to suit particular needs and illustrates one possible mobile middleware system built using it. As such, it is part of the evaluation of this work, as it shows how SATIN can be instantiated to build a component based middleware system but it is also one of the contributions of

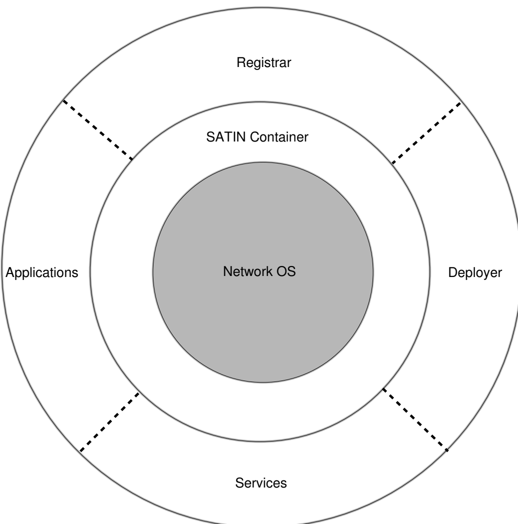


Figure 5.1: The architecture of the SATIN middleware system.

the thesis. Aspects of the work discussed in the following sections have been outlined in [Zachariadis et al., 2004, Zachariadis et al., 2003, Zachariadis and Mascolo, 2003] and are presented here in detail.

5.1 The SATIN Middleware System

The middleware itself and all applications developed over it are represented as a collection of SATIN components registered with the the system and implementing various facets. The next sections describe the middleware system in detail.

5.1.1 Middleware System Overview

Figure 5.1 provides a high level overview of the SATIN mobile computing middleware system. The system is built on top of the network operating system and provides an instance of the SATIN container, as defined in Chapter 4, which is the central aspect of every instance of the middleware system. Registered with the container are all the components that are part of the system. This includes application components (such as a media player application), libraries (such as audio codecs) and system services (such as any registrars, deployers service advertising and discovery components, etc.). All components make their dependencies explicit through their properties, as described in Section 4.3.2.

The circular notation used in Figure 5.1 denotes that, from the point of view of the container, all other components available to the system are *equal*. Even though components

may build complex dependency graphs expressed via their properties, they are all components implementing various facets, which can be added and removed at runtime. This allows the middleware system itself to adapt.

5.1.2 Required Attributes

As hinted in Section 4.3.2, the SATIN middleware system requires that component implementations have the `ID`, `VER`, `FACETS` and `DEP` attributes set, defined in Figure 4.3.

The value of `ID` is a string, which is used to uniquely identify a component in a single instance of SATIN. There are two types of identifiers: *globally* unique and *locally* unique ones. Globally unique identifiers are identifiers which have been registered in a centralised database, similar to the PalmOS Creator ID. Locally unique identifiers, which are prefixed with "`LCL:`", are identifiers which have not been registered globally - as such, even though they are unique on a particular instance of the middleware, they may be used differently in other instances. The use of unique identifiers allows for easily identifying a component; local identifiers allow for uniquely identifying a component on a local scale, while global identifiers extends the scope of identification to a global scale. The disadvantage of using globally unique identifiers is that they rely on a centralised database.

The value of the `DEP` attribute expresses the dependencies of a component, as a sequence of `ID` values, representing the components that it depends on.

The value of `FACETS` expresses the facets that the component provides.

Finally, the value of `VER` represents the version of the implementation of the component. As such, the values of `ID` and `VER` are used to uniquely identify a component realisation.

5.1.3 Middleware Services

The SATIN middleware system offers a number of services to components that use it. The services themselves, as shown in Figure 5.1, are seen as regular components built on top of the container by the middleware system. As such, they can be dynamically added and removed. The following paragraphs describe the services that the SATIN middleware system offers, and how those relate to mobile adaptation.

The Container and the Registrar

The SATIN middleware system Container & Registrar support basic transactional registering of components. In particular, when an LMU is passed on to the Container, the registration of the components the LMU contains is an atomic operation. More specifically,

the registrar checks whether all the components to be installed satisfy their dependencies and if and only if that is the case, it registers them. If one of the components fails to have its dependencies satisfied, then the whole group is rejected.

The container and registrar implementations of the SATIN middleware system also support the registration and availability of multiple versions of the same component using the `VER` attribute. Unless the version number is specified, then the latest version of the component is always returned when a component is asked for. Finally, the middleware container keeps track of how often components are requested. This can be used to discard least used functionality that does not break any dependencies, when the system has scarce resources.

Reasoning About the Underlying System

As detailed in Section 2.2.1, a system can adapt as a result of any of the following: A specific user action, a change to the remote context or a change to the local context.

In order to allow for monitoring changes to the local context, the SATIN middleware system provides a `LocalHost` component, which is responsible for reasoning about the current state of the local machine. Other components in the system can query `LocalHost` to get information about the state of the battery, the network, etc., and to be notified if there are any changes to them. Note that this component has not been fully implemented, as local context inspection is considered to be out of the scope of this thesis (see Section 1.1). As such, the current implementation of `LocalHost` simply allows other components to get the current IP address(es) of the host.

Advertisement and Discovery Framework

Similarly, a system can adapt as a result of changes to the environment. In general, one of the pivotal requirements of mobile and adaptable pervasive computing, is the ability to reason about the environment. The environment is defined as the network of devices that can, at a specific point in time, communicate with each other. The devices can be both mobile and stationary - with the presence of mobile devices, however, the environment can be rapidly changing. In order to adapt, a mobile system needs to be able to detect changes to its environment. As the device itself is also part of that environment, it also needs to advertise its presence. A mobile device, however, may be able to connect to different types of networks, either concurrently or at different times, with different networking interfaces. There also are many different ways to do advertising and discovery. Imposing a particular advertisement and discovery mechanism can hinder interoperability with other systems, making assumptions about the network, the nodes and environment, which may be violated at some later stage or simply not be optimal in a future setting - something which is likely to happen, given the dynamicity of the target area of this thesis.

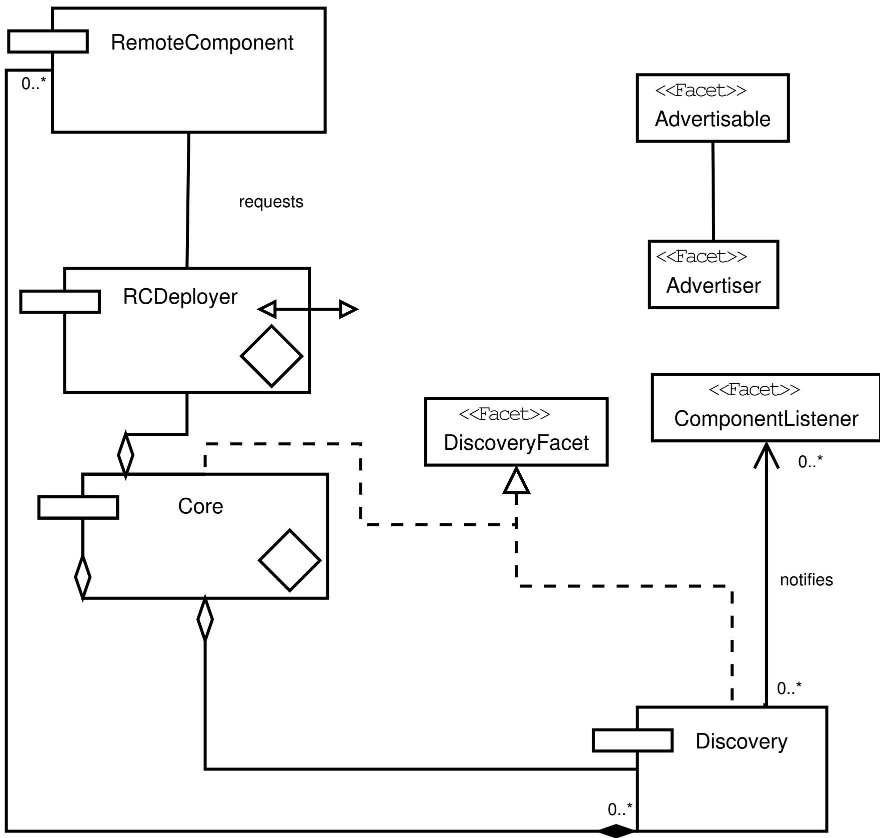


Figure 5.2: The SATIN middleware system advertisement and discovery framework.

From the point of view of SATIN, the ability to reason about the environment is translated into the ability to discover components currently in reach and to advertise the components installed in the local system. This is achieved via the use of *Remote* and *Discovery* components, as well as *Advertiser*, *Advertisable*, *DiscoveryFacet* and *ComponentListener* facets. This is a *realisation* of the SATIN meta component model and is outlined in Figure 5.2, which uses the notation defined in Section 4.5. It is described in detail below.

Components that wish to advertise their presence in the environment must implement the `Advertisable` facet. Examples of advertisable components include codec repositories, services, etc. The `Advertisable` facet exports a method that returns a message that is used for advertising; thus, the advertising message allows the Advertisable component to express information that it requires advertised.

An advertising technique is represented by an *Advertiser* component, which is a component implementing the `Advertiser` facet. An advertiser component is responsible for accepting the message of advertisable components, potentially transforming it into another format and using it to advertise them. An advertiser allows components that wish to be advertised

to register themselves with it to be advertised. The combination of component availability notification and advertiser registration, allows an advertisable component to register with the container to be notified when specific advertisers are added to the system. The advertisable component can then register to be advertised by them. Moreover, an advertisable component can express that it requires a particular advertiser in its dependencies. Thus, the semantics of the advertisable message are not defined and depend on the advertisable component and on the advertising technique (i.e. the advertiser component) used. Note that a component can implement both the `Advertiser` and the `Advertisable` facets. This allows for the advertising of advertising techniques; in this way, for example, the existence of a multicast advertising group can be advertised using a broadcast advertiser. Combined with the use of logical mobility primitives, this allows a host to dynamically acquire a different advertising and discovery mechanism, for a network that was just detected. For example, upon approaching a Jini network [Waldo, 1999], a node can request and download the components that are needed to advertise to, and use functionality from, the network.

Similarly, discovery techniques are encapsulated by *Discovery* Components, which implement the `DiscoveryFacet` facet. There can be any number of discovery components installed in a system. A discovery component acts as registry of advertisable components located remotely. The middleware system defines the `RemoteComponent`, which is used to represent components, which have been found remotely. A remote component is an immutable component that cannot directly export any functionality to local components. It only exports methods needed to access its properties, location and advertising message. Hence, Discovery components act as a collector of Remote component references, which can be added and removed dynamically, as they are discovered. Discovery components emit events representing the availability of remote components. Local components can register a `ComponentListener` with a discovery component, to be notified when components satisfying a given set of attributes are located. `ComponentListener` is represented as a SATIN *facet*. The SATIN middleware system Deployer (called `RCDeployer` in Figure 5.2) can be used to request advertisable components from other hosts, by requesting the appropriate `RemoteComponent`, as found and encapsulated by a `Discovery` component.

Assume, for example, the presence of an Advertiser component, which broadcasts the advertising message of registered advertisable components at regular intervals and that this Advertiser requires that the advertising message is written using an XML language. Given this, let us further assume the existence of a component that implements a File Transfer Protocol (FTP) server and that the value of its `ID` attribute is `FTP`. The FTP server component implements the `Advertisable` facet and defines
`<port>21<port><anonymous/>`
as its advertising message, which can be translated as listening on port 21 and allowing for anonymous access. The advertiser can transform this message into

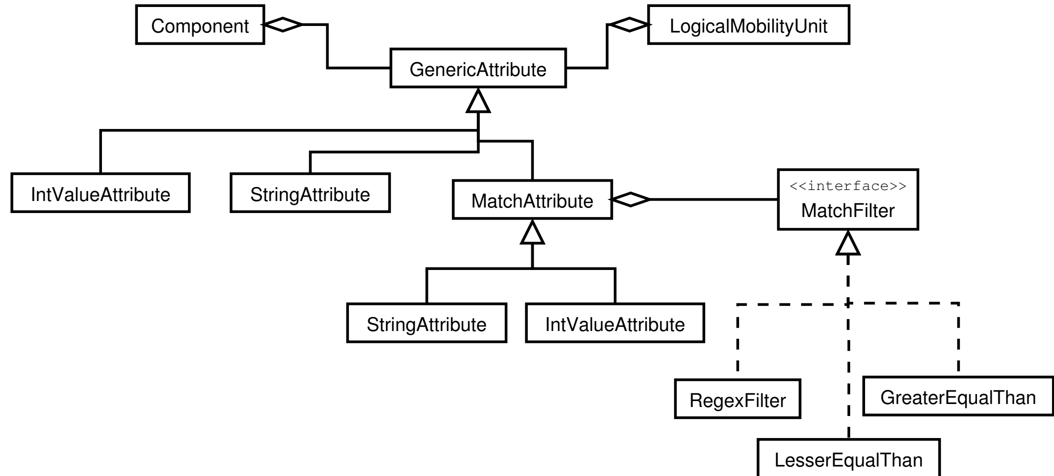


Figure 5.3: The SATIN middleware system attributes architecture.

```
<component id=''FTP'', version=''1''>
   <port>21</port>
   <anonymous/>
</component>
```

thus adding metadata on the advertising message before broadcasting to any hosts in reach.

Given the similarities between the container and a discovery component, the container of the SATIN middleware system (called `Core` in Figure 5.2) also realises the *DiscoveryFacet*, as it "discovers" components located and registered locally.

5.1.4 Application Components

The SATIN middleware system defines an `APPLICATION` facet. This facet is implemented by components which represent the main class of an application. As such, the `APPLICATION` facet exports functionality to start, stop, suspend and resume the application.

5.1.5 An Object-Oriented Attribute System

The use of attributes is prevalent in the SATIN component model and middleware system. Their importance is significant, as they are used to form *querying templates*, used to locate components either locally (by querying the container) or remotely (by querying an advertiser). Locating a component is fundamentally related to mobile adaptation from the point of view of this work, as components are used to encapsulate functionality using

which a system can adapt.

As such, the SATIN middleware system extends the attribute concept outlined in Figure 4.2 and described in Section 4.3.2, by defining an object-oriented attribute architecture, a high level overview of which is shown in Figure 5.3. The top-level attribute is a `GenericAttribute`. A GenericAttribute uses object references to represent the attribute key and value encapsulated. Components (represented in Figure 5.3 using an abstract class) and Logical Mobility Units instantiate a `GenericAttribute` to encapsulate each of their attributes. Various specialisations of a generic attribute are defined. In particular, Figure 5.3 shows an `IntValueAttribute`, which is an attribute the value of which is an integer, and a `StringAttribute`, the key and value of which are textual strings.

More importantly, a `MatchAttribute` is defined as a specialisation of a generic attribute that is immutable. Its purpose is to offer customisable comparator semantics with generic attributes. To this end, a match attribute encapsulates a `MatchFilter`, which is an interface implementations of which perform the actual comparison between the value contained in a generic attribute and the one contained in the match attribute. Figure 5.3 illustrates three classes implementing MatchFilter as an example. `RegexFilter` uses regular expressions, whereas `LesserEqualThan` and `GreaterEqualThan` use arithmetic comparisons to compare the values of attributes. As such, when registering to be notified when components that satisfy a set of attributes become available, programmers can use MatchAttributes to customise the evaluation of the satisfiability process.

Generic Attributes and, hence, Match Attributes are extensible by application developers using the SATIN middleware system. Moreover, as they are SATIN classes, they can be migrated between nodes using the primitives offered by the Deployer. Finally, Match Attributes can be used by developers to query for the existence of both local and remote components, by introspecting the container or a discovery service, respectively.

5.2 Discussion

The SATIN middleware system was designed as a realisation of the SATIN component model detailed in the previous chapter. The aim is to show that the component model can be used to build a lightweight middleware system that can systematically offer the flexible use of logical mobility primitives to applications built using it, so that they can adapt to context changes.

As mentioned above, all middleware services, libraries and applications, apart from the container, are on the same conceptual level; as such, they are all SATIN components, implementing the component facet and are offered the same services. This has a number of repercussions, which are discussed in the following paragraphs.

SATIN allows the middleware system and applications to mutate without any constraints. As mentioned in Section 1.1, constraining adaptation is not one of the considerations of this thesis. However, adaptation can lead to situations where the mutation of the system leaves it at an undesirable state - for example, the deployer and registrar may be removed, thus making the system logically static. A simplistic solution to this issue would be to modify the container to blacklist a specified number of components, disallowing their removal. This, would not address, however, limiting the mutation of any reflective component other than the container. A more general solution would be to extend the reflective component with a rule-based mechanism to define rules as to how a component is allowed to adapt; alternatively, component frameworks with strict rules on their configuration could be used.

It should also be noted that even though the SATIN middleware system provides for requesting components to be received and deployed locally, it does not have a systematic way to request finer grained Logical Mobility Elements, such as instances and classes. Even though sending any type of LME is provided for by the middleware system, it is assumed that the functionality to describe, advertise and, thus, allow other nodes to request finer grained LMEs will be either provided as a service, encapsulated as a component on top of the middleware, or that components that allow the request of these LME instances will implement their own mechanisms and advertise them appropriately.

Moreover, it should be noted that even though match attributes can be used to provided customisable attribute comparison on a *per attribute* basis, on the whole, each attribute in the properties set is evaluated individually and all of them must be successfully evaluated (*and* semantics) for the comparison operation on the property set to be successful. More complex semantics (such as *or* or *xor* semantics), compound and flow control statements are not provided for. Providing for these would require using a logic-based evaluation language and extending the properties abstraction to implement it.

Finally, note that although a comprehensive and adaptable advertising and discovery mechanism is provided, it is not the purpose of this thesis to examine the merits of this mechanism. The discussion of the advertising and discovery framework was given to show how the SATIN component model can be realised to offer middleware services and how its inherent modularity and use of logical mobility techniques can be used to build an adaptive service discovery and advertising framework in a middleware system.

The modular nature of the system permits for building abstractions on top of the system that address the issues described above. Components that require this functionality can then express it as a requirement in their dependencies.

5.3 Related Work

The purpose of this section is not to provide a comprehensive survey of mobile computing middleware systems. To that end, the reader is refered to [Mascolo et al., 2002a]. Moreover, as mentioned above, it is not the purpose of this thesis to examine advertising and discovery protocol interoperability. The reader is directed to [Bromberg and Issarny, 2004] for a discussion of this matter.

This section critically discusses related approaches that support aspects of adaptation - either by employing the use of logical mobility techniques, or by using a component system. These approaches are outlined, followed by a comparison to the SATIN middleware system.

5.3.1 Logical Mobility for Application Reconfiguration

Sun Microsystems' Jini[Arnold et al., 1999] technology is a distributed networking system, which allows devices to enter a federation and offer services to other devices or utilise services already offered. Jini, based on the Java programming language, exploits the inherent code mobility capabilities of that language to allow devices to locate and use services offered. This distributed system of services is called a *djinn* in Jini terminology. A service in a Jini system is an object, or a collection of objects, which can be utilised by other devices in the Jini federation. Examples include device drivers, time services, etc. Essentially, any object or collection of objects that performs a certain task or controls a device can become a Jini service. The Jini architecture relies on the operation of servers called the lookup services. Lookup services are centralised indexes where objects advertise their services and clients can search for particular ones and receive the object reference needed to communicate with the service provider.

When a client wishes to utilise a service offered in a djinn, it must query the registrar for the service needed, based on a service template, which is comprised of a set of optional information, such as the service ID. The lookup process returns the matches that satisfy the query. Note that the client can query multiple lookup services, to select the service which is more suitable to the task. If a service is found, the client downloads, using the Code on Demand paradigm, all the classes that are needed to use the service, using the Java object serialisation framework [Sun Microsystems, 1998b]. Jini provides support for service leasing, distributed events and transactions. Internally, Jini uses Java Remote Method Invocation (RMI) [Sun Microsystems, 1998c], TCP and UDP sockets.

Jini technology can be used to provide context-aware mobile services in a centralised environment. It is not suitable for rapidly changing ad-hoc networks. One of the major hurdles in the acceptance of Jini in mobile systems, is that the reference implementation offered by Sun is very resource demanding. Psinaptic has developed a version of Jini called

JMatos[Hashman and Knudsen, 2001], which is specifically geared for mobile devices. Occupying around 100KB of storage on the device, JMatos does not use the RMI technology, and does not implement a proxy-based lookup service, resulting in a more efficient fully compliant Jini implementation.

In contrast to Jini, SATIN uses logical mobility to adapt autonomous systems. Potentially, Jini-like systems can be used using the logical mobility primitives provided by the SATIN middleware system.

Linda in a Mobile Environment (Lime) [Murphy et al., 2001, Picco et al., 1999] is a Java-based middleware system, which allows the development of applications which exhibit logical and/or physical mobility characteristics, by providing a coordination model based on Linda tuple spaces. Lime is primarily geared for ad-hoc networks, although it is not limited to such configurations only.

Linda is a communication model for concurrent processes, which was developed at Yale University in the mid-1980s. Linda processes communicate using a shared repository of tuples, the tuple space. A tuple is an elementary data structure, a sequence of typed parameters, such as ("foo",1), and represents information being communicated. A tuple, `t`, can be deposited to the tuple space using the `out(t)` operation, and can be retrieved using the `in(p)` operation, where `p` is a pattern matching the tuple returned. If no tuple currently matches the pattern, the requesting thread is blocked, until the pattern is matched. If more than one tuples match the given pattern, the one returned is selected non-deterministically. `in(p)` removes the resulting tuple from the tuple space. The model includes an operation to read a tuple from the tuple space without removing it, `rd`. The tuple space can be accessed concurrently by threads and processes, the resulting model providing spatial and temporal decoupling.

Lime exploits the decoupled nature of tuple spaces to provide coordination primitives and information sharing for mobile components. The unit of mobility in Lime is a mobile agent, with mobile hosts acting as simple containers for those agents. Special scenarios of this configuration are stationary agents, which are also supported by Lime. An agent can contact other agents if they reside on the same host or if the nodes that host the agents are in reach. Each agent can have a set of tuple spaces, which are identified by their name. The tuple spaces are bound to the agents and, as the agent migrates, the tuple spaces migrate as well. The agent can choose whether to share a tuple space it owns. Lime makes all shared tuple spaces with the same name transparently appear as a single tuple space. Lime also allows agents to react to changes in context, by defining the reaction primitive. A reaction `r(c,p)` specifies a code fragment, `c`, that is executed when the tuple matching pattern `p` is found in the tuple space.

In effect, Lime provides application developers with a data-sharing middleware geared for ad-hoc networks, employing the mobile agent paradigm, unlike SATIN which does not

provide any data-sharing middleware. The primitives that it provides however and in particular the use of mobile agents, can be used to provide similar functionality.

PeerWare[Cugola and Picco, 2002b, Cugola and Picco, 2002a] is a mobile computing middleware system that offers peer-to-peer communication, event subscription and a shared data space to applications.

The PeerWare system is based around the concept of a Global Virtual Data Structure (GVDS). A GVDS is a communication and coordination meta-model for mobile environments. It is essentially a generalisation of the Lime coordination model. A GVDS provides a global data space that is created dynamically by the the local data spaces of each peer in range; thus it is virtual, as it does not exist on any host as a single entity. The GVDS meta-model does not specify how the GVDS is structured, leaving this issue to the implementation.

PeerWare makes a sharp distinction between operations that can be performed on the local data structure and on the GVDS. PeerWare exploits logical mobility, by considering the execution of an action on the GVDS, as a distributed execution of the action on the data structures of the connected peers. As such, it uses Remote Evaluation to offer distributed execution to a shared and mobile data structure.

[Kangas and Oening, 1999] use COD techniques to allow adding virtual objects into the real world view of the mobile user, using computers to project them to the user's sensory systems. The core idea behind this project is that physical objects are connected to virtual ones. A physical object is equipped with an active tag; when a mobile user locates the tag, the system can request the code that represents the virtual object (the code is contained in the tag) and execute it. Upon receiving the code, the virtual object can interact with the tag and hence the physical object, via a communication link established between them.

The Software Dock [Hall et al., 1999] is a mobile agent-based software deployment architecture. It makes a strict separation between software producers and software consumers and uses a distributed event service and mobile agents to distribute, deploy and maintain software from the producers to the consumers.

XMIDDLE [Mascolo et al., 2002b] is a mobile computing middleware system that allows mobile applications to share XML documents. It uses replication to support operations on disconnected data and can reconcile changes to the documents upon reconnection. XMIDDLE hosts can dynamically decide which reconciliation protocol to use; XMIDDLE employs the use of code on demand primitives to allow a host to download the protocol if it is not available locally.

Compared to the work discussed above, the SATIN middleware system is a more general purpose system in that it does not limit how the logical mobility techniques that are provided will be used. Thus, they can be used to adapt both middleware services (such as

advertising and discovery) and applications. As SATIN takes a finer grained approach to logical mobility, allowing components to send and receive individual instances and classes as well as complete components, it can be used to implement the solutions of previous approaches, but its use and applicability is much more general. The approaches above use limited aspects of logical mobility to solve particular problems. For example, Jini uses COD to offer services to devices, Lime uses mobile agents to support data sharing and PeerWare uses REV to distribute computations on shared data. The generality of SATIN can be exploited, allowing existing middleware systems such as Lime or XMIDDLE to be implemented on top of SATIN, as a collection of SATIN components. This would allow, for example, an XMIDDLE application to express in its properties that it depends on the XMIDDLE components. This would add a further layer of adaptability to XMIDDLE, by allowing XMIDDLE-based applications to use the SATIN middleware system to dynamically request the XMIDDLE components when needed.

5.3.2 Middleware Reconfiguration

OpenORBv2 [Clarke et al., 2001, Blair et al., 2001, Blair et al., 2002] is a reflective component-based middleware system which is built using OpenCOM and is compatible with the OMG CORBA [OMG, 1995] distributed programming environment. OpenORBv2 is structured as a layered collection of component frameworks. The system can reconfigure itself, by dynamically changing the structure of the component frameworks, thus specialising the system to support different functionality.

ReMMoC [Grace et al., 2003b, Grace et al., 2003a] is a reflective and component based middleware platform, which can dynamically reconfigure the service discovery and binding protocol of a mobile device. The purpose of ReMMoC is to use reflection to overcome *middleware heterogeneity*. To that end, ReMMoC is built using OpenCOM (described in Section 4.7) and is structured using component frameworks, to decouple the application from the actual service that it may request. ReMMoC defines abstractions of binding and discovery protocols that the application can program against; the ReMMoC infrastructure maps a request against this abstraction to a concrete implementation at runtime. ReMMoC employs a service discovery and a binding component framework, which are configured by plugging in different service discovery protocols and binding type implementations respectively. The service discovery mechanism is quite similar to the one that SATIN provides. As such, in SATIN, applications program using the advertiser, discovery and advertisable component abstractions, which in turn are handled by their concrete implementations. ReMMoC leverages the Web Services Description Language (WSDL) [Chinnici et al., 2003] to describe and request services, thus hiding from the developer the problem of middleware heterogeneity. The main differences with SATIN stem from the fact that the SATIN middleware is based on the SATIN component model, whereas ReMMoC is based on OpenCOM. As such, SATIN offers the use of logical mobility prim-

itives for reconfiguration built into the component model, whereas ReMMoC does not address this issue. Furthermore, due to the differences in focus, the SATIN middleware system allows for both applications and middleware level services to dynamically adapt, as opposed to ReMMoC, which focuses on the latter. Lastly, the SATIN middleware system, unlike ReMMoC, offers the flexible use of logical mobility primitives to applications. Unlike ReMMoC, however, the SATIN middleware system does not use WSDL to describe services that may be offered by components. This implies that a component cannot depend on an abstract service, but on the attributes that the implementation of that service exports. This constrain can be lifted by using the properties abstraction of the component and WSDL to describe the service that the component offers.

The Universal Interoperable Core (UIC) [Roman et al., 2001] is a generic request broker that defines a skeleton of abstract components which have to be specialised to the particular properties of each middleware platform the device wishes to interact with. It mainly focuses on synchronous communication paradigms and is thus not particularly suited to physically mobile devices.

Unlike the approaches outlined above, the SATIN middleware system offers the dynamic reconfiguration of system-level *and* application-level reflective components via the systematic, symmetric and general use of logical mobility primitives. The middleware system formalises and offers the flexible use of logical mobility as a first class citizen. As such, it can adapt itself and its applications, not only by reconfiguring the components that already exist on the mobile device, but also by allowing new components to be retrieved and dynamically instantiated - it thus offers structural reflection at the component level (as inherited by the SATIN component model), but also the ability to dynamically add and discard components obtained from different sources.

5.4 Summary

This chapter described the SATIN mobile computing middleware system, as an instantiation of the SATIN component model, which was described in the previous chapter. The general adaptability and flexibility through logical mobility allows SATIN-based applications to heal and mutate according to context, which they can monitor, making them suitable for mobile computing. Moreover, the complete componentisation of all system aspects, including advertising and discovery, makes SATIN demonstrably suitable for roaming. The collocation of SATIN components allows a system to be autonomous; As SATIN focuses on the reconfiguration of local components, the middleware allows for applications to function in the event of disconnection from remote hosts - This is particularly important, given the dynamicity of the network connectivity of mobile devices. Moreover, SATIN allows for devices to both send and receive LMUs; By not making any distinction between server and client, SATIN allows for the potential creation of a large peer to peer network of offered

functionality.

The next chapter presents the implementation and evaluation of this approach through testing and application conversion and development. It also reports on usability.

Chapter 6

Implementation and Evaluation

The previous chapter discussed SATIN, a mobile middleware system that extends and instantiates the component model detailed in Chapter 4, which, in turn, encapsulates and offers the use of the logical mobility platform described in Chapter 3.

In this chapter, the details of the implementation of the SATIN middleware system are illustrated. The implementation was used to develop a number of applications. some of which are adaptations of pre-existing software that run over the middleware. The suitability of SATIN for mobile adaptation is evaluated, by illustrating how each application adapts and how this is made possible through the use of SATIN. The applicability of SATIN to adaptive systems is also shown, by describing, in detail, the development of Q-CAD [Capra et al., 2005], a mobile QoS aware framework for resource discovery, that is built using SATIN. Q-CAD provides a decision logic on *how to adapt*, based on the use of application profiles and utility functions. Moreover, the use of the SATIN implementation in the ZION [Chatterjee et al., 2004] project, a security framework for pervasive computing, is discussed. Finally, the performance of the SATIN implementation on mobile devices is evaluated, by measuring its memory overhead and time needed to adapt.

6.1 On Evaluating SATIN

The methodology used to evaluate SATIN follows the established combination of qualitative and quantitative evaluation and also uses the taxonomy identified in [Zelkowitz and Walace, 1997], to reflect upon how to validate this work. As such, a combination of replicated experiments, dynamic analysis, case studies, assertions and comparison with legacy data is used for the purpose of the evaluation.

Thus, this chapter evaluates SATIN using the following criteria:

Feasibility. How feasible are the metamodel abstractions, their instantiation and their actual implementation for the constraints of mobility? To answer this question, this chapter begins by defining a class of devices which will be specifically targeted and continues by detailing the implementation. It is shown that it is lightweight enough to run on mobile devices despite the added functionality, and performs numerous benchmarks with it to show this. To demonstrate that it is lightweight, the SATIN memory overhead as well as the time needed to adapt are measured. Moreover, the size of the implementation is compared with other approaches in this area of research.

Utility and Completeness. Can complete systems be built using the abstractions defined? Are they sufficient to allow for engineering systems that exhibit system level as well as application level adaptation? To answer these questions, this chapter details the design of numerous systems and applications using SATIN and shows how they address specific issues identified in the case studies presented in Section 2.1. In particular, this chapter presents the design of Q-CAD in detail, in order to show a complete adaptable resource discovery system built using the abstraction defined in this work. Q-CAD also verifies the claim made in chapter 2, that SATIN is flexible enough to allow for building a decision logic for adaptation on top of it.

Usability and Complexity. How easy is it to program using SATIN? How easily can existing projects be converted to run under SATIN? Are the abstractions and design provided by SATIN usable by third parties? To answer these questions, this chapter details the conversion of existing open source applications into SATIN components, showing that only minimal changes were needed. The componentised pre-existing applications gain functionality via the use of the SATIN migration primitives, while incurring minimal overhead; In particular, they can be dynamically shipped between nodes and added and removed into a running SATIN system. Code fragments are also given. Finally, the chapter includes a usability study, where a group of postgraduate students in networking built a pervasive security system using SATIN. The system is outlined, and a report on their experience is given.

It was decided not to use simulation, as it was argued that it would not be beneficial in evaluating this work. The reason for this is that the work behind SATIN does not propose any new networking protocols the behaviour of which would be advantageous to simulate. Rather, this thesis presents a new approach to engineer mobile systems. As the approach itself is inherently modular, the details of the implementation of each individual module or component (which can be replaced) is not deemed important to simulate or benchmark extensively against.

6.2 Quantitative Evaluation: Implementation

On implementing SATIN, some design choices had to be made regarding the class of mobile device that the implementation would be targeting. Considering the rate at which hardware changes, this may be considered immaterial; however, by defining an actual target, quantifiable assertions about speed and resource requirements can be made.

With these statements in mind, the class of devices that the implementation presented in this chapter, are PDAs. More specifically, the target was one of the most popular PDAs of 2001, the Compaq iPAQ H3600. It sports a 206MHz ARM CPU and 64 MB of RAM (which is also used as storage). An 802.11b wireless adaptor was also connected to it.

This device was chosen because the hardware that it offers is very mediocre with todays standards. In fact, entry-level PDAs are significantly more powerful, and cellphones are starting to become equivalent in functionality.

SATIN has been implemented using Java 2 Micro Edition (Connected Device Configuration (CDC), Personal Profile) [Sun Microsystems, 2000]. There are many reasons behind this choice: Java, and in particular Java 2 Micro Edition, is a portable language and virtual machine for mobile devices. The virtual machine and the Java bytecode is used for binary-level interoperability between, components, as discussed in Chapter 4 and also to realise a portable processing environment, as discussed in Chapter 3. The CDC and personal profile were specifically chosen because they allow the use of the Java Object Serialisation Framework [Sun Microsystems, 1998b] and Reflection API [Sun Microsystems, 1998d] for the deployer implementation; this enables the dynamic sending and receiving of Java classes and objects. SUN Microsystems claims that the class of devices that the CDC targets feature 32 bit CPUs and more than 5MB of RAM; this matches our intended target.

A mobile code toolkit, MiToolkit [Ijaha, 2004], developed by an undergraduate student under the co-supervision of the author, is used to realise the abstract platform for logical mobility described in Section 3.3. As such, MiToolkit is a library that can be used to send and receive Java-based LMUs. In particular, MiToolkit realises the *Controller* and the *Serialisation & Deserialisation Engine* of the abstract logical mobility platform defined in Chapter 3. It does not implement The *Trust & Security Layer*. MiToolkit is encapsulated in the SATIN deployer implementation. In the current implementation, all LMUs are sent uncompressed. Although initial development and experimentation was carried out with μCode [Picco, 1998b], MiToolkit was developed and is used because MiToolkit addresses the drawbacks of μCode, outlined in Section 3.4. In particular, MiToolkit is modular and is not geared towards mobile agents in any way.

The Serialisation and Deserialisation Engine of MiToolkit enables basic class name conflict resolution. A conflict is detected if a class encapsulated in an incoming LMU has the

Item	Size (in bytes)	Physical Source Lines of Code
MiToolkit	89,072	800
Metamodel	13,607	233
Implementation (Model)	47,650	925
Advertising and Discovery Framework	6,668	118
Publish Subscribe	22,797	574
Multicast	22,161	418
Total	201,955	3,068

Figure 6.1: Details on the SATIN implementation.

same canonical name of a class that already exists in the recipient instance of SATIN. As such, when a conflict is detected, the conflicting class is loaded into a private namespace. MiToolkit allows for one private namespace for each node that sends code to the local host. Classes in the private namespace are accessible from migrated instances originating from the host to which the private namespace is assigned. The conflicting class is rejected if it also conflicts with the contents of the respective private namespace. Note that this scenario requires the same host to have sent two different versions of the same class - a possibility that is not considered likely. MiToolkit differentiates between different hosts using a random integer generated at startup.

MiToolkit also supports dynamic calculation all the classes that a particular class, and by extension, component, references, ignoring those that are available by default on all SATIN nodes, such as the basic SATIN classes and the CDC/Personal Profile class library. This mechanism was found to fail in complex components during testing. Moreover, the mechanism would not support components that reference resources other than classes, such as data files. As such, a facet was developed, named `NeedsPacking`. `NeedsPacking` is implemented by components that want to customise the way they are encapsulated in an LMU. As such, before sending a component that implements `NeedsPacking`, the deployer calls a method exported by the facet that allows the component to have access to the LMU it is going to be encapsulated in, thus permitting it to add any logical mobility element.

Figure 6.2 gives details on the size of the SATIN implementation. Note that the size figure presented represents the *uncompressed* size, which is against the typical Java tradition. The Physical Source Lines of Code were calculated using SLOCCount [Wheeler, 2004] and the number does not include comments or blank lines. Note that the implementation of SATIN includes numerous MatchFilters. These numbers compare favourably to other related projects. For example, OpenCOM [Clarke et al., 2001] requires 28,160 or 18,432 bytes for its ARM and x86 implementation respectively, versus the 13,607 bytes that the SATIN metamodel requires [1]. μCode [Picco, 1998b] requires 86,946 bytes, versus the

[1]It is recognised that the SATIN metamodel has the benefit of being able to use the Personal Profile class library. However, the use of the library is very minimal, consisting mainly of the Socket interface and few data structures, such as Hashtables.

89,072 bytes that MiToolkit requires, despite the added modularity that MiToolkit offers. The complete SATIN implementation requires 201,955 bytes, versus 609,700 bytes that, for instance, Lime [Murphy et al., 2001] requires.

Starting the middleware system up and registering the deployer takes 1,846 milliseconds, on a Pentium II 266MHz machine with 64 megabytes of RAM. At that time, the middleware objects require 113872 bytes of heap memory. Note that the numbers presented here and in the rest of the chapter are with debugging and logging enabled. The SATIN implementation has not been optimised yet. It is expected that an optimised implementation, without debugging will show significant improvements. The numbers were obtained by using a combination of the UNIX `pmap` and `time` commands, as well as the Java `Runtime.getRuntime().totalMemory()` and `Runtime.getRuntime().freeMemory()` programming statements, while having control of the Java garbage collector. This practice is suggested by Sun Microsystems in [Wilson and Kesselman, 2000].

A simple XML [Bray et al., 1998a] based language was used to write the advertising messages of advertisable components. It is claimed that XML is useful in a pervasive computing environment with multiple service discovery techniques, as it offers a structured way to communicate information that can be easily transformed into other formats. In particular, the KXML2 [kObjects, 2002] parser was used to allow applications to parse advertising messages. KXML2 was chosen because it is lightweight and supports Java 2 Micro Edition. KXML2 occupies a further 21145 bytes as a compressed jar file.

SATIN was used to implement a number of applications. These are outlined below.

6.3 Quantitative and Qualitative Evaluation: Applications

6.3.1 The SATIN Program Launcher

Inspired by the problems discussed in Section 2.1, this application is a Dynamic Program Manager, or Launcher, for mobile devices. It is similar to the PalmOS Launcher, in that its basic purpose is to display and launch applications that are registered with the container. The applications installed are shown as buttons, with the component identifiers as labels. The Launcher also manages and controls all components installed. As mentioned in Chapter 5, applications are components that implement the `Application` facet. As such, the program launcher registers itself with the container, to be notified when a component implementing the `Application` facet (i.e., a new application) is registered, in order to automatically redraw its interface to accommodate for it.

The dynamic program launcher offers the following services: Using the deployer, it can

Figure 6.2: The SATIN Program Launcher.

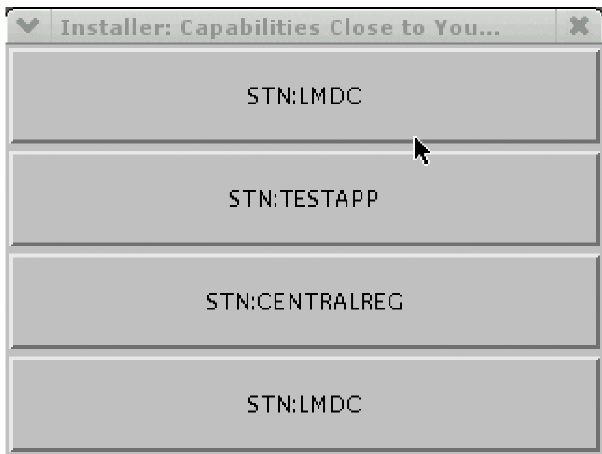


Figure 6.3: The SATIN Program Launcher: Showing what components are advertised on all networks, including those of the local host.

install any component from any discoverable source (i.e., through any discovery service). Figure 6.3 shows the Launcher displaying the components that are currently advertised by hosts in reach. Figure 6.4 shows a component with identifier `STN:TESTAPP` after it was installed remotely. Using the same mechanism, it can update the components installed in the system, either transparently or as a result of a user command. An implementation of the container that monitors the usage (by counting how often each component is requested) of the components installed was deployed: If the device running the Launcher runs out of memory, it can discard unused components based on their frequency of use.

The application caters for the scenario presented in Section 2.1.1: Mobile devices roam through a dynamic context, able to transparently update their libraries and install new applications available in their current environment. Moreover, it presents an alternative to the scenario illustrated in Section 2.1.2: All applications are componentised, which

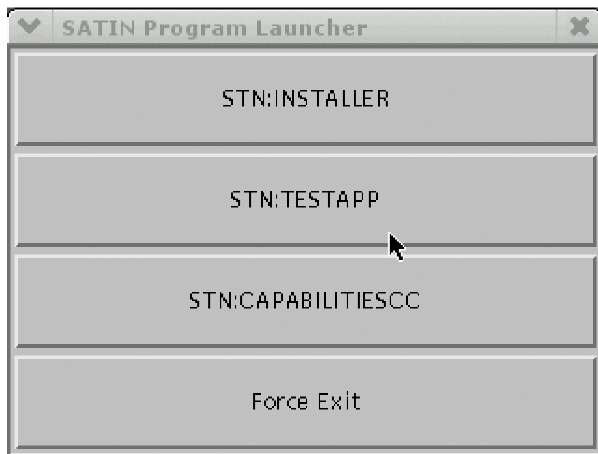


Figure 6.4: The SATIN Program Launcher: Component "STN:TESTAPP" was installed from a remote host and is displayed by the Launcher.

Component Identifier	Description	Depends On
STN:LAUNCHER	The SATIN Application Launcher	STN:UPDATER
STN:CAPABILITIESCC	Manages installed components	STN:INSTALLER
STN:CAPABILITYREADER	Loads components from disk	-
STN:INSTALLER	Uses the Deployer and any discovery services to request and install components from a remote location	STN:COMPONENTREADER
STN:UPDATER	The component that is used to update the system	-

Figure 6.5: The SATIN Launcher as a collection of components.

facilitates maintenance.

The components of the launcher occupy 39749 bytes as an uncompressed jar file and were written in 778 physical source lines of code. The launcher is fully componentised; as such the updating mechanism and the remote install mechanism are encapsulated as separate components. Figure 6.5 shows the full full set of components of the launcher.

The application was tested with three devices: a PDA equipped with an 802.11b card in ad hoc mode, a laptop equipped with an 802.11b card (again in ad hoc mode) and a Fast Ethernet card, and a desktop with a Fast Ethernet card. The PDA was equipped with a 200MHz StrongARM processor and 64 megabytes of RAM (32 megabytes of which are used for storage), the laptop was equipped with a 733MHz Pentium III and 256 megabytes of RAM and the desktop was equipped with a 266MHz Pentium II and 64 megabytes of RAM. All three machines were running various flavours of Linux. The PDA was running

Figure 6.6: The testing configuration. The dotted line represents the wireless network. The solid line represents the wired ethernet.

a beta version of Java, with no Just In Time (JIT) compilation.

The laptop could communicate with both the desktop and the PDA, whereas the PDA and the desktop could only communicate with the laptop. This is shown in Figure 6.6.

The laptop and PDA used the multicast advertising and discovery service to communicate over the wireless network, whereas the laptop and the desktop used the centralised advertising and discovery services over Ethernet. When testing, the desktop was advertising the availability of version 2 of a component with identifier STN:TESTAPP, version 1 of which was installed on the PDA. STN:TESTAPP is a simple, one class SATIN component, requiring 1257 bytes. The laptop installed version 1 of the component from the PDA and updated it to version 2 from the desktop. The PDA then discovered the availability of version 2 on the laptop and updated its copy. The table below shows the Java heap memory usage and the startup time for the Launcher on the PDA, the time it took for STN:TESTAPP to be installed from the PDA to the laptop, the time it took for the laptop to update STN:TESTAPP to version 2 from the desktop and the time it took for the PDA to update to version 2 from the laptop.

Startup Time on PDA	21673 ms
Memory Usage on PDA	1155474 bytes
Time to install component from PDA to Laptop	1998 ms
Update time from Desktop to Laptop	1452 ms
Update time from Laptop to PDA	2063 ms

The results obtained above show that the system implementation is reasonably lightweight. The large time difference between the tests when the PDA was involved (installation time from the PDA to the laptop and update time from the laptop to the PDA) and when it was not (update time from the desktop to the laptop) is attributed to the fact that the PDA runs a beta version of an interpreted JVM and to the nature of the wireless network.

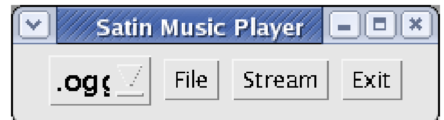


Figure 6.7: The SATIN Music Player.

The time difference between installing from the PDA to the laptop and updating from the laptop to the PDA is attributed to the fact that the PDA discovery component had to discover the updated version of the component in reach. The large startup time on the PDA is attributed to the interpreted JVM - note, however, that the middleware only needs to start up once - applications can then be dynamically added to the running instance.

6.3.2 The SATIN Music Player

A simple music player was also implemented using SATIN. Components that implement audio codecs must implement the AUDIOFORMAT facet. As such, the Music Player uses the notification service to be notified whenever a component that provides this facet is registered. Moreover, it uses the deployer and the discovery components to download any codecs that are found remotely. The application itself occupies 6,568 bytes as an uncompressed jar file and was written in 133 physical source lines of code.

JOrbis [JCraft, 2001] [2], an open source Ogg Vorbis [The Xiph.org Foundation, 1998] implementation, was also adapted to run as a SATIN audio codec component. Ogg Vorbis is a royalty and patent free lossy compressed audio codec. Both the music player and the componentised audio codec can be dynamically sent and deployed on SATIN nodes. The application is automatically notified when a codec component is found and adapts its interface accordingly. The JOrbis component occupies 169978 bytes as an uncompressed Java archive and is composed of 50 classes. Please note that the Music Player application is a Java 2 Standard Edition application and not a Micro Edition application. This is denoted in the component attributes. Java 2 Standard Edition was used for this application, because there are very few open implementations of the Java Mobile Media API [Sun Microsystems, Inc., 2003] for the Connected Device Configuration of Java 2 Micro Edition. JOrbis occupies 169,454 bytes as an uncompressed jar file.

The component abstraction for JOrbis occupies 1,371 bytes as an uncompressed Java archive. It was written in 60 physical source lines of code.

A number of tests were conducted using the Ogg Vorbis codec component. The overhead of playing an audio stream via the componentised codec versus the JOrbis stand-alone implementation was measured. Moreover, the time needed to send the componentised

[2] The version of JOrbis used was 0.0.14.

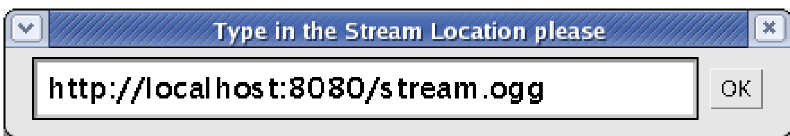


Figure 6.8: The SATIN Music Player: Playing a remote stream.

```
-=Initialising the Container=-
-=Container (ID=STN:CONTAINER,FACETS=Discovery,VER=1)
  initialised=-
-=Creating Self=-
-=Registering Self (ID=STN:SHELL)=-
-=This is SATIN version 0.8=-
-=Running on Linux 2.6.5-1.358 / i386=-
-=Hostname: hamsalad.cs.ucl.ac.uk=-
-=Java 1.4.2_04 / Sun Microsystems Inc.=-
-=A reference to the container will be made available via the
  object reference container=-
-=Starting the BeanShell...=-
BeanShell 2.0b1.1 - by Pat Niemeyer (pat@pat.net)
bsh % Component c=container.getComponent("STN:SHELL");
```

Figure 6.9: The SATIN Shell.

codec from one node to another was measured. The figures are presented below.

Component-Based Overhead (includes SATIN)	19000 bytes
Time to Send LMU:	2682 ms
Time to Deploy LMU	860 ms

The tests were run using the desktop and laptop described above. They show that the overhead (which includes the full SATIN middleware running) is relatively small. Moreover, the time needed to send and deploy the LMU containing the componentised Ogg Vorbis codec is quite small. The sending of an LMU took considerably more time than its deployment. The reason for this is that MiToolkit needs to compute all the classes that the component uses, in order to send the complete set.

The Music Player demonstrates an application that uses the container to listen to the arrival of new components, then adapting its interface and functionality to reflect the arrival of a new component. It also demonstrates reaction to context changes, as the application monitors the discovery services for new codec components and schedules them for download as soon as they appear. The operation is transparent to the end user.

6.3.3 The SATIN Scripting Framework

BeanShell [Niemeyer, 1997] [3], an open source Java source interpreter and scripting mechanism, was adapted to run as a SATIN component. This allows SATIN components to use scripts and to be scripted. A "Shell" application was created for SATIN using the BeanShell, which allows developers to manipulate the container and its contents by typing Java statements at runtime. The Shell component and BeanShell encapsulation require 10,028 bytes as an uncompressed jar file. They were written in 221 physical source lines of code. BeanShell, on the other hand, requires 551079 bytes as an uncompressed jar file and is composed of 131 classes. Figure 6.9 shows sample output from the shell. The last line, in particular, shows how to get a reference of a component from the container.

A number of tests were performed using the BeanShell component, on the laptop and desktop described above. The overhead of using the BeanShell through the SATIN Shell versus stand-alone BeanShell as well as the time taken to send and deploy an LMU containing the SATIN Shell and BeanShell were measured. The figures are presented below.

Component-Based Overhead (includes SATIN)	20728 bytes
Time to Send LMU:	6771 ms
Time to Deploy LMU	1058 ms

The figures confirm the result obtained when testing the Ogg Vorbis codec component. The memory overhead of the componentised BeanShell is, again, quite small. Moreover, it takes significantly more time to send an LMU, rather than to deploy it. It can be concluded that calculating the classes that a component references and serialising them is more expensive than deserialisation and deployment. The BeanShell component is significantly more complex than the components previously described, judging from the size and number of classes. This can explain why calculating the referenced classes and serialising them takes significantly longer than for the other components described in the previous sections.

The SATIN shell and BeanShell components demonstrate how a library is added into the system, promoting reusability between components and adapting the services that the middleware offers. Moreover, the scripting framework can be expressed as a component dependency, for components (such as the shell) that require it and can be registered dynamically, when needed.

6.3.4 Code Fragments

This section includes some actual code fragments from the applications described above. For more details, please refer to the programming guide in Appendix B.

[3] The version of BeanShell used was 2.0b1.

The following following shows how to initialise SATIN and register a Deployer and a discovery service with the container:

```
new Core(); //initialise the container
//get a reference to it
Container container = Container.getContainer();

//get a reference to the registrar
RegistrarFacet registrar = container.getDefaultRegistrar();

DeployerFacet d = new MiToolkitDeployer(); //Initialise the deployer
registrar.registerComponent((Component)d); //register it with the middleware
d.setEnabled(true); //enable it

CentralDiscovery disc=new CentralDiscovery(); //initialise the discovery component
registrar.registerComponent(disc);  //register it
```

The following fragment shows how to initialise and send the Ogg Vorbis codec, to a node with IP address 192.168.0.1:

```
ComponentFacet c=new OggVorbisCodec();
registrar.registerComponent(c);

//gets a reference to the deployer, using its identifier
DeployerFacet d=container.getComponent("STN:MITKDEP'');

/* The following statement creates a new LMU,
  * with target "192.168.0.1" and
  * defines its destination as "STN:CONTAINER",
  * which is the container of the recipient host.
  *
  * This effectively defines that the LMU should be sent to the
  * container of 192.168.0.1.
  *
  * The container will try to register it.
  */
LMU lmu=new HashLMU("192.168.0.1","STN:CONTAINER");
lmu.addComponent(c); //adds the component to the LMU.

d.send(lmu);//sends the LMU
```

The following fragment shows requesting to be notified when a component with the value of the BITRATE attribute greater than or equal to 32 and the value of VER equal to 1 is found. This specification matches the Ogg Vorbis codec encapsulation.

```
Hashtable template=new Hashtable();
template.put("BITRATE",new MatchAttribute("BITRATE",new Integer(32),
    new GreaterEqualThanFilter()));
template.put("VER",new GenericAttribute("VER",new Integer(1),true));

//adds a listener for a component based on that template
container.addListener((ComponentListener)this,template);
```

6.3.5 Summary

The implementation of the SATIN middleware system and the applications confirm that SATIN is reasonably lightweight, despite the offered features and added flexibility. In particular, it was shown that memory overhead is minimal, dynamically deploying components is speedy, and execution is quick on our target platform.

The BeanShell adaptation shows how functionality can be added into the system, and the dynamic launcher and music player show how applications can monitor their local and remote context in terms of component availability, and how they can adapt and mutate in reaction to changes.

Adapting JOrbis and BeanShell to run as SATIN components required little programming. Componentising JOrbis and BeanShell, or any other application for that matter, adds a thin layer of indirection to the software, the component abstraction and facets in particular. The memory overhead measured for this was minimal, as shown above.

The component-based JOrbis and BeanShell use the SATIN primitives to gain *mobility*. As such they can be transfered between nodes and export functionality to other SATIN components.

This chapter continues with a report and analysis of two large scale systems that were built using SATIN.

6.4 Qualitative Evaluation: Q-CAD

When discussing adaptation in Section 2.2, the notions of *laissez-faire*, *application-aware* and *application-transparent* adaptation were defined. A *laissez-faire* adaptable system provides the mechanism for adaptation, but does not provide the logic on how to adapt. In an *application-transparent* system, the application is not aware of the adaptation process and cannot influence it; it happens internally by the underlying system. In between these two extreme cases is *application-aware* adaptation, where the application can influence a central arbitrator provided by the system that decides on *how* to adapt.

Chapter 2 stated that the system that this thesis provides is a *laissez-faire* adaptation system. Indeed, the SATIN component model enables mobile systems to adapt and the SATIN middleware system allows them to reason about their context. However the decision on how to adapt rests with the application. This section addresses this limitation, by building Q-CAD, a Quality of Service (QoS) and Context Aware resource Discovery framework, that uses utility functions and application profiles to decide on how to adapt. Q-CAD is an instance of the SATIN metamodel and it is at the same conceptual level as the SATIN middleware system. In fact, the components provided by the latter are reused by Q-CAD. Q-CAD itself is designed as a collection of SATIN components and applications that require it can express that requirement in their attributes.

Q-CAD can be used to allow applications to *bind* to remote resources. Q-CAD remote resources are *services* provided by remote providers, *sensors* from which an application may get data, and SATIN *components* located remotely and that can be downloaded and deployed on the local host.

Each application encodes the way context should influence the discovery of, and the binding to, resources in an *application profile*; Q-CAD uses this information to reduce the resources available in the current context to a subset of 'plausible' ones. Each application also encodes the QoS needs of the user into a *utility function* that Q-CAD applies to select the most suitable resource among the plausible ones (i.e., the one that maximises the utility of the user). To cater for the varying execution context and non-functional requirements of the user, both the application profile and utility function can be changed dynamically.

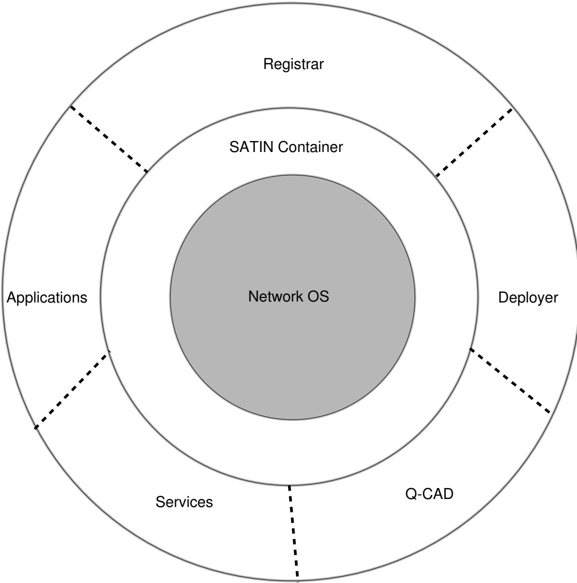


Figure 6.10: Q-CAD in the SATIN middleware system.

From the point of view of the SATIN middleware system, a Q-CAD realisation is simply

a collection of interdependent components built and registered with the SATIN container. It is a service that is provided to applications which may express their dependency on it. This is shown in Figure 6.10. Q-CAD can be thought of as an arbitrator, that decides on how the system will adapt based on the preferences of applications.

Note that Q-CAD is currently being implemented, but the details and benchmarks of that implementation will not be included, Rather, from the point of view of the evaluation of SATIN, the detailing of the *design* of Q-CAD demonstrates the utility and completeness of SATIN because:

- It shows that the abstractions that SATIN provides are useful and suitable for building a large-scale adaptable framework for resource discovery, targeting mobile systems.
- It demonstrates that the SATIN metamodel can be used to build a decision logic for mobile adaptation, as claimed in Chapter 2.

The section continues by introducing a running example, to better illustrate what Q-CAD tries to achieve.

6.4.1 Case Study

In order to explain how Q-CAD works, this section sketches an example of a pervasive computing application, highlighting different cases of resource discovery where both QoS and context awareness are needed. This is followed by an outline of the major goals of Q-CAD and a summary of its assumptions.

Consider, for example, the case of Alice going on holiday to New York. Figure 6.11 illustrates two examples of resource discovery: in the first scenario, Alice wishes to print the pictures she has taken with her digital camera. In order to do so, she has to discover and select a photo development service provider, amongst the many available. Different parameters may influence this choice: for example, location of the provider (Alice may prefer a provider located close to her hotel, as to be able to collect the prints conveniently), cost of the service, quality of the prints, and so on. In the second scenario, a number of sensors[4] have been deployed in the most prominent locations of the city, providing tourist information to other devices in proximity. While on a bus tour, Alice may use her PDA to dynamically discover and bind to these sensors; however, different tourist companies may have deployed their own, providing different quality/amount of information for different prices to potential customers. The non-functional requirements of Alice must thus be used to decide what sensor to bind to amongst the discovered ones. Context must be taken into account too, as, for example, audio information may be preferred to both audio and

[4]In the remainder of this section, very small devices with limited capabilities and resources (e.g., actuators and sensors) are referred to as *sensors*.

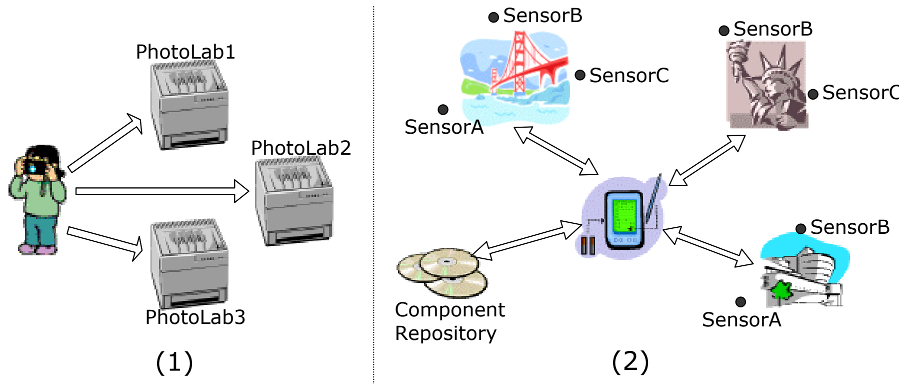


Figure 6.11: Service Discovery Scenarios.

video when running out of battery. Moreover, the processing of the data may require components that are not installed on Alice's PDA, so that dynamic discovery, download and deployment of components from available repositories may be part of the procedure too. Although fictitious, this example is not too far from reality, as shown by the Urban Tapestry project [The Urban Tapestry Project, 2002].

The first scenario is referred to as *proactive discovery*, as the discovery and binding to a service provider is a consequence of an explicit request of the user to locate such a service. The second scenario is an instance of *reactive discovery* instead, as the discovery and binding to sensors and component repositories is a result of context changes. Although discovery is triggered by different events in the two scenarios (user action in the first scenario, context change in the second scenario), both types of discovery demand a similar behaviour from the discovery framework: locating and binding to a resource (whether it is a service provider, a sensor, or a component) that is best suited in the current context (*context-awareness*) and according to the current non-functional requirements of the user (*QoS-awareness*).

The Q-CAD discovery model for pervasive computing applications aims to:

1. provide applications with a means to explicitly state the context conditions of interest to their user (i.e., context awareness);
2. provide applications with a means to explicitly state the non-functional requirements of their user (i.e., QoS awareness);
3. develop a resource discovery, selection and adaptation protocol that takes the preferences of the user into account (both in terms of context and of QoS needs).

Q-CAD builds on the following assumptions: first, the existence of a shared ontology to refer to context elements and conditions, resource names and characteristics, and non-

functional requirements; second, the integration with an existing discovery protocol for mobile ad-hoc networks on which Q-CAD relies to route advertisements and queries. The SATIN component model is used to represent applications and the framework itself.

6.4.2 The Q-CAD Model

Q-CAD achieves context and QoS awareness by means of *application profiles* and *utility functions* respectively, that is, metadata that represent the preferences of the user. Both application profiles and utility functions are encoded using XML [Bray et al., 1998b]; although the concrete languages used are not fundamental to the model, examples are illustrated in XML, rather than using an abstract syntax, to ease presentation. The complete XML Schema [Fallside, 2000] specifications are presented in Appendix A.

Q-CAD Resources, Descriptors and Binding

Central to the Q-CAD model is the notion of a *resource*. Before detailing what information is encoded in application profiles and utility functions, the concepts of a resource, *binding* to a resource and that of a *resource descriptor* are defined.

The resources that the Q-CAD model considers are: *services* provided by remote providers, *sensors* from which an application may get data, and *components* located remotely and that can be downloaded and deployed on the local host. These resources are referred to as *remote resources*, to distinguish them from those local to a device (e.g., battery, memory, CPU, etc.).

Remote resources are uniquely identified by means of an addressable naming scheme that is resolved by the underlying communication framework. The namespace used can be local or global, although it is expected that, in practice, a combination of the two will be used. For example, considering a device with both a cellular and an ad-hoc Bluetooth interface, a global naming scheme can be used in the cellular interface, while a local one can be used in the Bluetooth interface. Thus *binding* to a resource (i.e., the last step of the resource discovery and selection process) is defined as the association of the selected remote resource to a SATIN component that is local to the device and that can interact with it. A SATIN component that can be associated with a remote resource must implement the `BindTarget` facet. A remote resource could itself be a component: in this case, binding refers to downloading and deploying the component to the local container.

Each remote resource is encapsulated as a SATIN advertisable component and is associated with a static specification, or *resource descriptor*, that characterises the resource by means of the properties of the component. The descriptor is encapsulated in the advertisable component's advertising message and attributes. Figure 6.12 illustrates an example of a

```
(ID, QCAD:displayVideo)
(type, component)
(code, display800600.jar)
(resolution, 800x600)
(version, 2.1)
(platform, JVM2)
(size, 70KB)
(cost, $10)
(memory, 2)
(battery, 4)
```

Figure 6.12: Example of Resource Descriptor.

remote resource descriptor, for a component that displays video at a resolution of 800x600; information about the component implementation follows (e.g., version number, platform required, size of the component, etc.). In addition, the descriptor contains information that can be used to assess the quality of the resource itself; this includes, for example, estimates of local resources (e.g., battery and memory) consumption. Assuming that these estimates vary in a range $[0, 10]$, then the descriptor in Figure 6.12 says that the component consumes much more energy than memory. Note that these values do not aim to be precise estimates of actual consumptions; rather, they aim to enable comparisons of resources of the same type (e.g., services, components, etc.). It is assumed that the estimates have not been maliciously altered. The resource descriptor tuples are mapped to the properties of the advertisable component. As will be detailed in the following sections, this information is crucial to performing QoS-aware resource discovery.

Note that, in Figure 6.12, the `ID` attribute is prefixed with `QCAD:` to denote that this component is part of the Q-CAD framework.

Application Profiles

Application profiles specify how the user or application wishes the context to influence the discovery of remote resources, both in proactive and in reactive situations.

Proactive Discovery. For each remote resource the application may be willing to bind to, the proactive encoding of its profile contains an association between the resource name (tag `<BIND_RESOURCE>`) and the context conditions that must hold for the binding to be enabled (tag `<REMOTE_CONTEXT>`). For example, the encoding shown in Figure 6.13 states that only printing service providers that give customers at least 100MB of disk space should be considered during the discovery. This condition acts as a filter over the possibly high number of providers of the same service. Only one context configuration (tag

<REMOTE_CONTEXT id="1">), containing only one condition (tag <CONDITION>) is specified. More generally, multiple contexts can be associated to the same binding resource, and more conditions may be associated to the same context; for example, another condition could be to consider only service providers with utilisation load below a certain threshold. The semantics of this encoding are the following: the binding to the remote resource is enabled if and only if *at least* one of the context configurations is enabled (*or* semantics); a context configuration is enabled if and only if *all* the conditions associated to it hold (*and* semantics). If more than one service provider passes the filtering, the actual provider to bind to will be selected using the application's utility function (see Section 6.4.2).

As Figure 6.13 shows, the proactive part of the application profile supports richer encodings than the one illustrated so far. The additional information provides further support for dynamic adaptation to context. In particular, the application may specify in which contexts (initial tags <LOCAL_CONTEXT> and <REMOTE_CONTEXT>) the discovery and binding process should be enabled; for example, it may be forbidden when running out of battery (local condition), or when the quality of the network connection is too unstable (remote condition). In the above example, these contexts are not specified, thus indicating no pre-condition to the discovery and binding process.

Once a remote service provider has been discovered and selected, the application has to decide how to interact with it, as different behaviours/protocols may be available. *Binding* is the last step of the resource discovery process; it associates the remote service provider to the SATIN component that implements the desired behaviour/protocol. Such a component should be selected out of a list of desirable ones (tag <ADAPT_COMPONENT>); the choice depends on the following information, that is attached to each of these components: local context (tag <LOCAL_CONTEXT>), remote context (tag <REMOTE_CONTEXT>), and application preferences (tag <ATTRIBUTES>). For example, the encoding of Figure 6.13 dictates that pictures should be uploaded to the provider site using a component that supports an encryption protocol (<ATTRIBUTE key="protocol" op="equals" value="encryptedUpload"/>) when battery permits, while using a simple, plaintext upload when battery is low (<ATTRIBUTE key="protocol" op="equals" value="plaintextUpload"/>). Application preferences can be evaluated by comparing the values of the attributes listed in the profile (i.e., <ATTRIBUTE key=.../>) with those that appear in the component properties. The operators are mapped to SATIN Match Filters. More generally, in the <ADAPT> part of the application profile specification the following may be found: nothing, indicating that whatever component is able to support interaction with the remote resource will be used; one single component, without any context associated, thus requiring exactly a component that satisfies the given attributes (e.g., a component that implements an encrypted upload, regardless of context) to be used; finally, a list of components with associated attributes and contexts. If multiple components match the criteria given, the utility function will be used to select the one that best satisfies the QoS needs of the user (see Section 6.4.2). Note that the chosen

```
<PROACTIVE id="1">
    <LOCAL_CONTEXT/>

    <REMOTE_CONTEXT/>

    <BIND>
        <BIND_RESOURCE name="printPicture">
            <REMOTE_CONTEXT id="1">
                <CONDITION name="diskSpace" op="greaterThan" value="100MB"/>
            </REMOTE_CONTEXT>
        </BIND_RESOURCE>
    </BIND>

    <ADAPT>
        <ADAPT_COMPONENT id="1">
            <LOCAL_CONTEXT id="2">
                <CONDITION name="battery" op="greaterThan" value="30%"/>
            </LOCAL_CONTEXT>

            <REMOTE_CONTEXT/>

            <ATTRIBUTES>
                <ATTRIBUTE key="protocol" op="equals" value="encryptedUpload"/>
            </ATTRIBUTES>
        </ADAPT_COMPONENT>

        <ADAPT_COMPONENT id="2">
            <LOCAL_CONTEXT id="3">
                <CONDITION name="battery" op="lessThan" value="30%"/>
            </LOCAL_CONTEXT>

            <REMOTE_CONTEXT/>

            <ATTRIBUTES>
                <ATTRIBUTE key="protocol" op="equals" value="plaintextUpload"/>
            </ATTRIBUTES>
        </ADAPT_COMPONENT>
    </ADAPT>
</PROACTIVE>
```

Figure 6.13: Application Profile - Example of Proactive Encoding.

component may not be available locally; in this case, discovery, download and deployment of a component implementation is required; as will be illustrated for reactive discovery, this process is almost identical to the one that has been discussed above, as components

are treated as yet another type of resource.

Reactive Discovery. The reactive part of the profile describes how the application reacts to context changes. The reactive encoding shown in Figure 6.14 states that, when the remaining battery power is greater than 30% (`<LOCAL_CONTEXT>`) and there is a video sensor in reach that broadcasts images at a resolution of 800x600 in the JPEG format (`<REMOTE_CONTEXT>`), a binding to that sensor should be established (tag `<BIND>`). As the example shows, context can be composed of both local resources (e.g., battery) and remote resources (e.g., video sensor); the tag `<CONDITION>` is used for the former, and `<ATTRIBUTES>` for the latter. Continuing with the example, after binding to a video sensor, the data received should be displayed on the local device using a component (tag `<ADAPT_COMPONENT>`) that can display images at the specified resolution, and that can cache at least 1024KB of the data received; once again, if a local implementation of that component is not available, one has to be discovered that satisfies the listed conditions. If, after the screening performed using context conditions, there are still multiple video sensors to bind to, or multiple implementations of the desired component, the utility function selects the one to be used (see Section 6.4.2).

The information encoded in an application profile will now be summarised and generalised.

- In the **first part**, the initial local and remote contexts act as pre-conditions to perform discovery and adaptation. Context can be composed of local resources (e.g., memory, battery, CPU, etc.), and remote resources (e.g., a video sensor), which are represented by advertisable SATIN components. The conditions associated to a remote resource (tags `<ATTRIBUTE>`) are used during the discovery of the remote resource itself, to cut down the number of suitable answers. When at least one local context is enabled *and* at least one remote context is enabled, then the precondition to discovery and adaptation holds and a binding to a remote resource may be required. Note that this 'context change-triggers-binding' type of behaviour represents the very nature of reactive adaptation, that demands monitoring of context and prompt reaction to changes; for proactive adaptation, instead, these general local and remote context specifications will typically be left blank, and context conditions will rather be associated to specific bindings (second part), so to be evaluated only on-demand, when resource discovery is explicitly triggered. In other words, defining context conditions independently of a specific binding, or associated to it, will lead to the same result; what changes is the semantics of the context monitoring: continuous for independent conditions (thus typical of reactive adaptation), on-demand for bind-related conditions (thus typical of proactive adaptation).
- The **second part** specifies what bindings are necessary (either as a consequence of context change, or as a result of an application service request), and what con-

```
<REACTIVE id="1">
    <LOCAL_CONTEXT id="1">
        <CONDITION name="battery" op="greaterThan" value="30%"/>
    </LOCAL_CONTEXT>

    <REMOTE_CONTEXT id="2">
            <ATTRIBUTES>
                <ATTRIBUTE key="sensor" op="equals" value="videoSensor"/>
                <ATTRIBUTE key="resolution" op="equal" value="800x600"/>
                <ATTRIBUTE key="format" op="equals" value="jpeg"/>
            </ATTRIBUTES>
    </REMOTE_CONTEXT>

    <BIND>
        <BIND_RESOURCE name="videoSensor"/>
    </BIND>

    <ADAPT>
        <ADAPT_COMPONENT id="3">
            <LOCAL_CONTEXT/>
            <REMOTE_CONTEXT/>

            <ATTRIBUTES>
                <ATTRIBUTE key="type" op="equals" value="displayVideo"/>
                <ATTRIBUTE key="cache" op="greaterThan" value="1024KB"/>
                <ATTRIBUTE key="resolution" op="greaterThan" value="800x600"/>
            </ATTRIBUTES>
        </ADAPT_COMPONENT>
    </ADAPT>
</REACTIVE>
```

Figure 6.14: Application Profile - Example of Reactive Encoding.

text information should be used to reduce the number of plausible resources to bind to. Reactive encoding will typically require a binding to the very same resource discovered during the pre-condition, thus leaving the context associated to the binding empty; proactive encoding will specify here the context conditions necessary to prune the binding instead.

- The **third part** specifies what adaptation is required on the device itself, in order to bind to the selected remote resource. Each adaptation alternative may have a context associated to it, to state what alternative is most suited in different contexts. Adaptation refers to i) binding a SATIN component to a remote resource; ii) requesting, downloading and deploying a remote component to the local container;

```
<UTILITY_FUNCTION id="uf1">
    <RETURN>
        <EVALUATE>
            <ATTRIBUTE key="cost" op="greaterThan" value="10$"/>
        </EVALUATE>
        <FILTER>
            <ATTRIBUTE key="cost"/>
        </FILTER>
    </RETURN>
    <MAXIMISE>
        <ATTRIBUTE key="battery" weight="10"/>
        <ATTRIBUTE key="memory" weight="5"/>
    </MAXIMISE>
</UTILITY_FUNCTION>
```

Figure 6.15: Example of a Utility Function.

or iii) both.

Application profiles enable context-aware resource discovery; however, more than one resource (be it a service provider, a component implementation or a sensor) may fit in the current context. The next section illustrates how to use utility functions to select exactly one resource out of the shortlisted ones.

Utility Functions

Utility functions are used to select the best resource out of the context-fit ones, according to the current non-functional requirements of the user. Similar to how profiles are handled, there exists a utility function for each application, so that user preferences may vary depending on the application under consideration.

Assume that the system is looking for a component implementation to be downloaded and executed on the local device, in order to bind to a remote sensor. Figure 6.15 illustrates an example of a utility function encoding. As shown, the encoding is divided into two parts: a `<RETURN>` part, and a `<MAXIMISE>` part. The latter is discussed first.

Under the tag `<MAXIMISE>`, the application lists the non-functional parameters it is interested in, together with a weight that expresses their relative importance. Let us assume that these weights vary in a range $[0, 10]$; the example indicates that the application is willing to select a component that maximises battery and memory savings; also, saving energy is twice as important as saving memory. The maximise part of the utility function is executed on a resource descriptor, as a summation of products (i.e., normalised

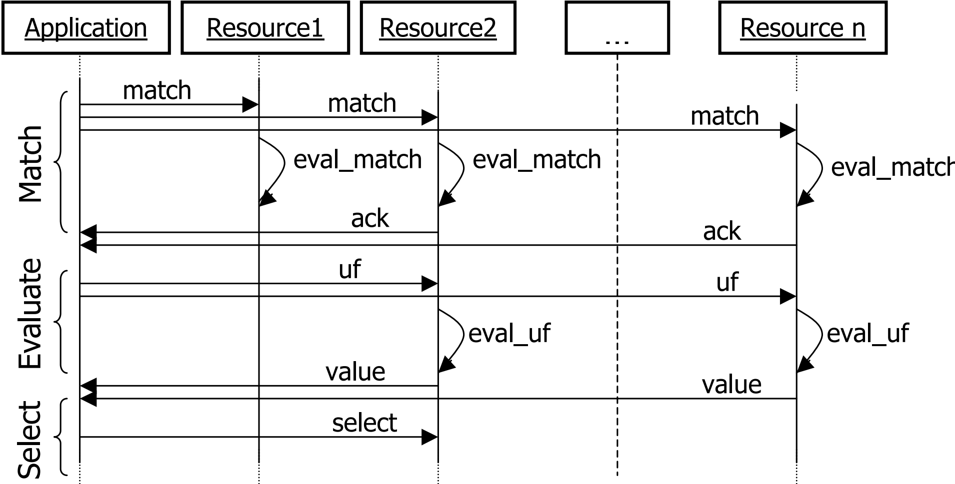


Figure 6.16: 3-Step Resource Discovery Protocol.

estimates multiplied by weights, as found in the resource descriptor and utility function respectively); it returns a single value that can be used to compare the quality of different resources. Note that high weights associated to parameters such as battery and memory mean that the user aims at sparing them; however, resource descriptors estimate their consumption, rather than their saving. In order to give higher scores to the remote resources that reduce consumption, the value: `saving = maximum consumption - estimated consumption` is used. The resource discovery concludes with the selection of the resource that scored highest (i.e., the one that maximises the user utility). However, there are cases in which it is not desirable to have a fully automated selection process. For example, it may not be desirable to download a component that maximises the non-functional requirements, in case it is too expensive. The first part of the utility function specification (tag `<RETURN>`) is used, when intervention on behalf of the application or user is required. For example, Figure 6.15 dictates that discovery and selection can be automated if the cost of the component is less than $10; otherwise, information has to be prompted to the user to make the final decision. This information includes, besides the result of the maximisation part, all the attributes listed in the `<FILTER>` part of the function (these attributes are a subset of those that appear in the resource descriptor, and usually coincide with the ones used in the `<EVALUATE>` part). Applications that require notification need to implement the `NeedsNotification` facet.

Discovery Protocol

This section presents a conceptual description of the discovery protocol that Q-CAD adopts in order to achieve QoS and context awareness.

As shown in Figure 6.16, Q-CAD discovery protocol consists of three main steps: *matching*,

evaluation and *selection*. These steps are exactly the same, regardless of the type of remote resource that is being looked for, and of whether reactive or proactive search is being performed.

Matching. The first step of the protocol uses the information encoded in the application profile to perform context-aware resource discovery. On behalf of the application, Q-CAD queries any discovery services that are registered with the SATIN container about remote components that satisfy the attributes given in the application profile.

Evaluation. Once the replies of the matching phase are received, the second step of the protocol uses the information encoded in the utility function to perform QoS evaluation. The nodes hosting the advertisable components that match the attributes required are sent a message containing the application's utility function, using the Deployer. Each node then evaluates the function over the relevant resource descriptors and returns an answer to the querying application. Note that a resource may refuse to perform this computation, either because it does not have the capabilities to do so (as could be the case for a sensor), or because it does not want to consume local resources. Vice versa, the application may not be willing, for privacy reasons, to disclose its utility function. In these cases, the resource descriptor may be returned instead, and the application itself will compute the utility function over the descriptor locally.

Selection. If no user intervention is required, the application selects the resource that maximises its utility based on the answers received and/or the local computation performed; if the intervention of the user is required, the returned values are passed to the application to obtain a final choice. After the selection has been made (either automatically or after user intervention), the protocol concludes.

The full potential of the language used to encode application profiles allows for a cascading execution of the discovery protocol: for example, to bind to an arbitrary number of sensors that are relevant to the application, then to find a service provider that can process the data coming from the sensors, and finally to locate and download the components needed to talk to the sensors and service provider. This work considers the profiles illustrated in Section 6.4.2 to be already expressive and representatives of most realistic situations. In these cases, the discovery protocol is repeated at most twice: first to discover a service provider or sensor, and then to discover a component to talk to it.

Figure 6.17 shows two example instances of the Q-CAD discovery protocol, the first related to the proactive encoding of Figure 6.13, and the second related to the reactive encoding of Figure 6.14. The proactive discovery simulates the case were PhotoLab1 does not pass the pruning performed by the first phase (for example, because PhotoLab1 does not provide users with the 100MB required by the application); after execution of the utility

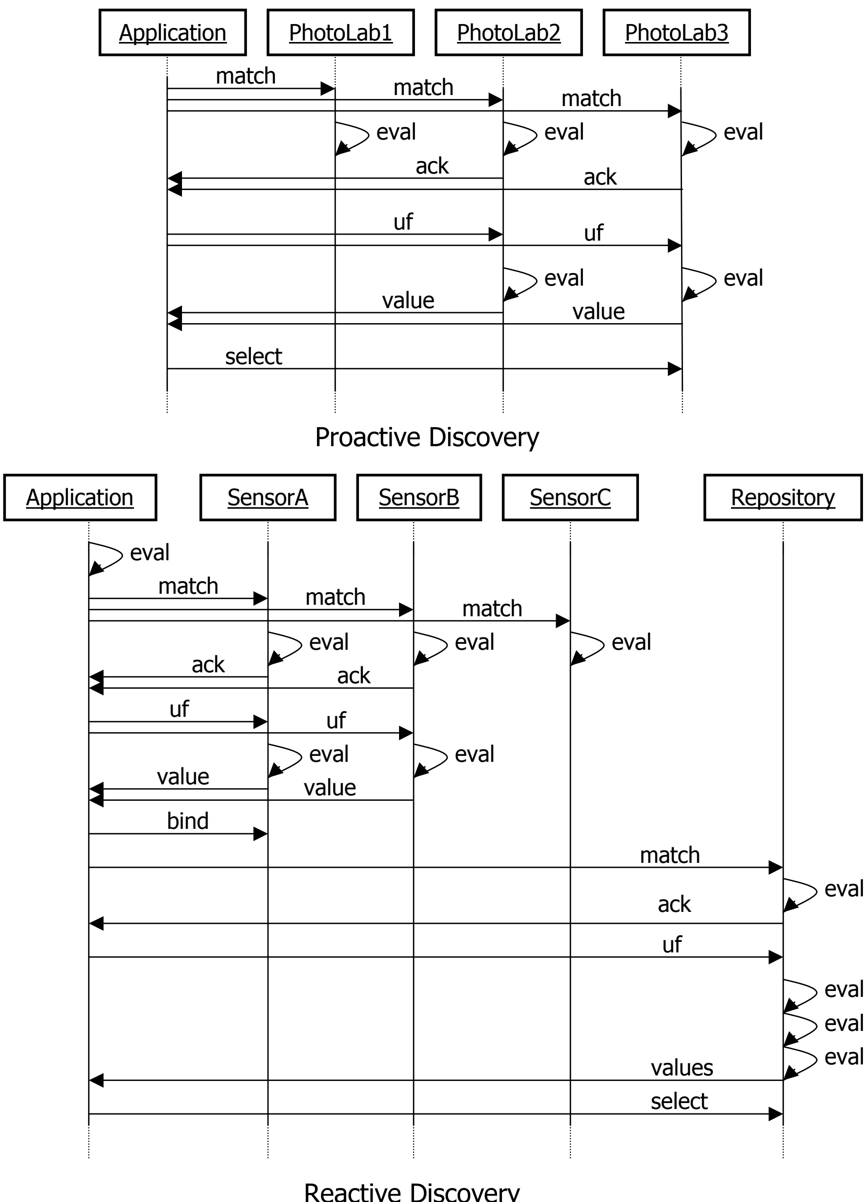


Figure 6.17: Proactive and Reactive Discovery Protocol Examples.

function, PhotoLab3 is chosen. The reactive discovery performs two executions of the protocol: sensor discovery first, followed by component discovery. During the first step of sensor discovery, SensorC is pruned (for example, because it does not send images at the resolution required by the application), and then SensorA is selected as the one maximising the utility function; a component discovery then starts to find an implementation of the component needed to display the data sent by the sensor. As the example shows, only one repository is contacted, but this may contain several implementations of the same component; it thus runs the utility function over the descriptors of all the resources it

possesses and that passed the pruning. Also note that, unlike proactive discovery, reactive discovery starts after a context evaluation (i.e., it is triggered as a consequence of a context change, not after an explicit request).

This concludes the presentation of the Q-CAD model. The following section analyses how this model has been mapped onto an efficient architecture, taking advantage of the flexibility and offered functionality of the SATIN component model and middleware system.

6.4.3 Q-CAD Architecture

The Q-CAD architecture is organised into four conceptual layers: the Application Meta-Interface layer, the Information layer, the Decision layer and the Action layer. This is shown in Figure 6.18. These are built as components of the SATIN middleware system, which is instanced on top of a networking operating system layer. Q-CAD reuses a number of abstractions defined in the SATIN component model and middleware system in Chapters 4 and 5. The architecture is further refined in Figure 6.19 as a collection of interacting SATIN components. Figure 6.19 uses the notation defined for the SATIN meta component model, which is presented in Section 4.5. The next paragraphs discuss each individual layer, the interaction between the components and issues related to their instantiation.

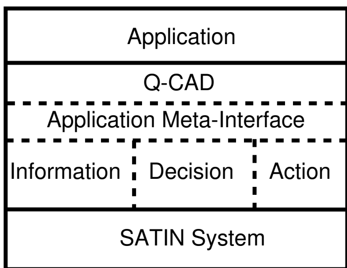


Figure 6.18: A High-Level Overview of the Q-CAD Architecture.

Q-CAD Applications. Q-CAD applications are represented as a collection of SATIN components. An application component must implement a specialisation of the `Application` facet, the `NeedsNotification` facet, which allows the Q-CAD framework to notify the application when its input is needed.

The Application Meta-Interface Layer. This layer encapsulates the interaction of the applications with the Q-CAD architecture. It is composed of the *PolicyRepo* and *NotificationService* components, an instance of which is given to each application upon starting up. PolicyRepo allows for the dynamic inspection and modification of the application profile and utility function. The latter are represented as SATIN classes.

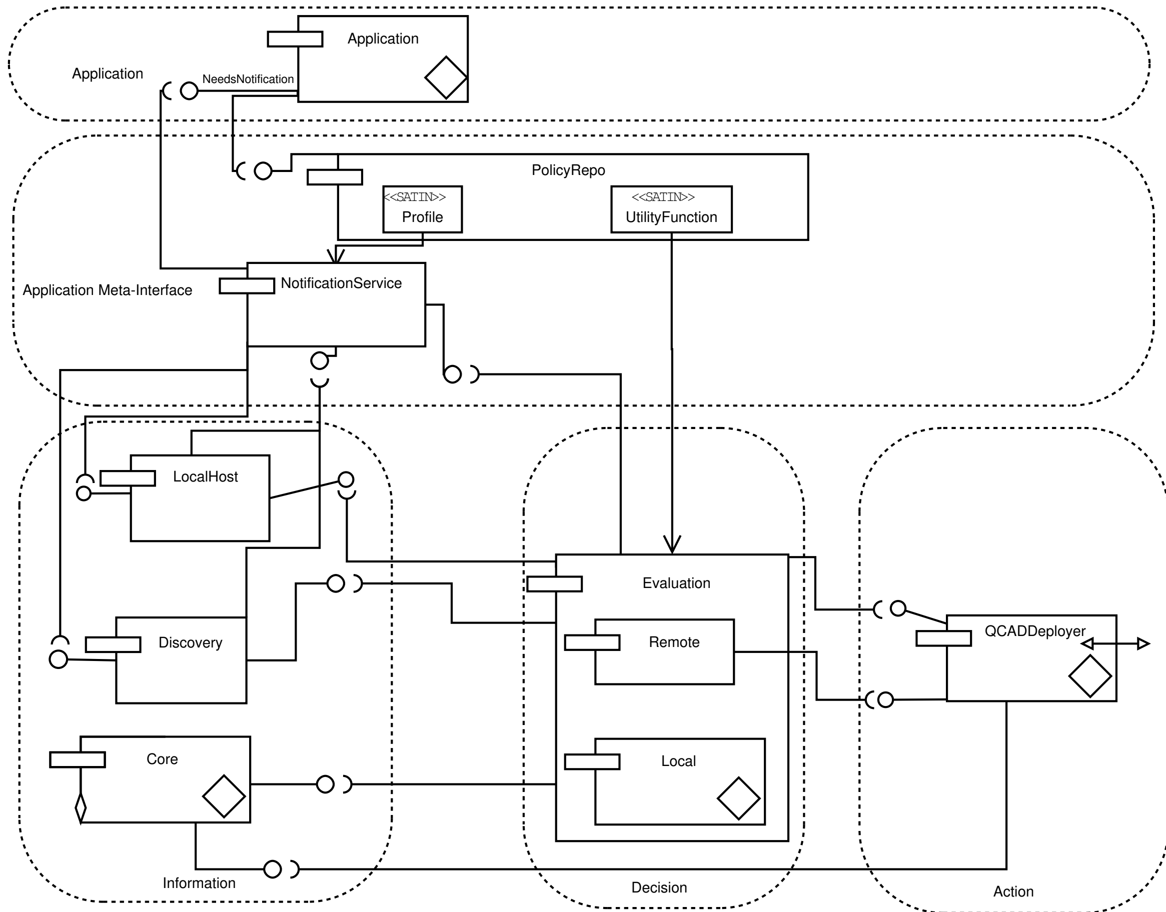


Figure 6.19: The Q-CAD Architecture.

The NotificationService is responsible for the following: extracting the information from the application profile and passing it on to the components in the Information layer; notifying the Information layer of changes that happened in the profile via the PolicyRepo component; and returning the result of a resource discovery to the application. The attributes in the application profile are directly mapped into SATIN *MatchAttributes*, with the operator (`op`) being a `MatchFilter`.

The Information Layer. This layer is responsible for the management of all the local and remote context-related information. Context information is mainly used for three purposes: during reactive adaptation, to continuously monitor the status of the system and trigger the discovery and adaptation process when a configuration of interest is entered; during proactive adaptation, to evaluate the status of the system on-demand; and during the matching part of the discovery and selection protocol (both for reactive and proactive encodings), to distribute the discovery message and prune the number of potential matches for resource binding. The Information layer takes care of all these tasks.

In particular, the *LocalHost* component, provided by the SATIN middleware (see Chapter 5), is responsible for monitoring the state of the local system (e.g., available memory, remaining battery power, etc.). The *Discovery* component represents any SATIN `Discovery` components that are available in the local node. It is responsible for detecting the remote resources (services, sensors and components), encapsulated as SATIN advertisable components, currently available to the local host, that the application is interested in (i.e., those listed in the application profile and that satisfy the specified constraints). Together, these components are thus responsible for the conditions set in the various `<LOCAL_CONTEXT>` and `<REMOTE_CONTEXT>` elements of the application profile. Note that the realisation of these components may require the instantiation of multiple (sub)components, each monitoring different parts of the context, using different techniques such as polling and interrupts. Only conditions expressed in the profiles of running applications are monitored at each time so not to waste local resources; moreover, computationally inexpensive conditions (e.g., remaining battery power) are checked first, and only when these are satisfied will more expensive conditions (e.g., existence of a remote sensor) be monitored (*and* semantics of context conditions). Note also that it is the responsibility of the Information layer to monitor the validity of the bindings established to remote resources (e.g., sensors and services) and to re-establish them when they are invalidated.

The *Core*, which is the instance of the SATIN *Container* in the system, is responsible for storing references to all component instances available in the local host (including all of the Q-CAD components). As such, the *Discovery* and *Container* components are responsible for evaluating the conditions set in the `<ADAPT>` elements of the application profile.

The Decision Layer. This layer encapsulates the evaluation and selection aspects of the Q-CAD protocol. After the Information layer has performed its pruning, the Decision layer evaluates the utility function against the shortlisted resource descriptors, and selects the one that maximises the application's utility. As previously mentioned, the utility function can be either evaluated remotely, on the host that is offering the resource, or locally, provided that the remote resource sends its descriptor. The *Evaluation* component of the Decision layer thus comprises both a *Local* and a *Remote* component, for local and remote evaluation of the utility function respectively. The Local component is a SATIN *Reflective* component. As such, it can receive remote utility functions and execute them against their associated resource descriptors.

The Local component interacts with the Information layer to get the resource descriptors against which to evaluate the utility function locally. As will be discussed below, the Remote component uses the functionality provided by the Action layer to distribute the utility function to the hosts where it is going to be evaluated. The utility function is encapsulated in an LMU and sent to the appropriate *Local* component of the host that hosts the resource where it is going to be executed. The

execution of the Evaluation component may generate events that need application input, as defined when describing utility functions (see Section 6.4.2). If that is the case, the NotificationService component in the Application Meta-Interface layer is used to pass the events to the application and get the required input. This is performed by calling a method exported by the `NeedsNotification` facet that the application component needs to implement.

The Action Layer. This layer encapsulates the logical mobility aspects of SATIN. As such, it consists of *QCADDeployer* which is an instance of the SATIN *Deployer* component. Of particular interests are the `Remote Evaluation` and `Code On Demand` paradigm realisations of the Deployer. Remote evaluation is used to distribute the utility function to remote nodes; it is encapsulated in an LMU and its *logical target* (`LTARG`) is the *Local* component instance of the Decision layer of the node that hosts the resource. Code on demand is used to download components that are needed. The downloaded components are registered with the SATIN `Container` component (the *Core* component in the diagram), so that the Information layer maintains an up to date status of the system.

6.4.4 Summary

This section has described Q-CAD, a QoS and context aware resource discovery framework for pervasive environments that is built using SATIN. Q-CAD combines expressive specifications of the preferences of the user with efficient processing. In particular, users specify the context conditions that should influence the discovery and selection of resources in application profiles, while the non-functional requirements are encoded in utility functions. The Q-CAD discovery and selection protocol efficiently uses the information contained in application profiles to prune the number of matches; it then uses utility functions to select the best resource out of the pruned ones.

Q-CAD shows how the SATIN middleware can be extended to provide the functionality needed to build an *application-aware* adaptable system. It also shows how the SATIN metamodel can be used to build a mobile system, showing how the SATIN abstractions can be used for adaptation. The logical mobility aspects of SATIN are used to offer a customisable mechanism to adapt a mobile system and bind to remote resources, based on changes in either its local or remote context.

To conclude, the design of Q-CAD found that the SATIN metamodel and middleware system provide useful and meaningful abstractions for creating an adaptable resource discovery framework. Large aspects of the middleware system were reused without any change, whereas for concepts that the middleware system provided no equivalent (such as the Utility Function and the Notification Service), the concepts defined in the metamodel proved adequate for describing them. The component properties were found to be very

useful in mapping to resource descriptors and proved a good guide for reasoning about context. Match Attributes also mapped well to context evaluation mechanisms. Also, the Deployer and the LMU concept was found adequate for sending and receiving utility functions. Hence, it can be asserted that despite its minimalism, the functionality provided by SATIN is suitable for a resource discovery framework. Finally, the abstractions proved adequate to build functionality that is missing from SATIN, namely a decision logic for adaptation. A project for implementing Q-CAD is currently being organised.

6.5 Qualitative Evaluation: ZION

The ZION [Chatterjee et al., 2004] project was a Masters thesis that produced a lightweight security framework implementation for pervasive computing using SATIN. The code and documentation was given to a group of students undertaking a post-graduate degree in Computer Science (in particular, a degree in Distributed Systems and Networking), to use it in their Masters thesis. From the point of view of this thesis, the aim of this experiment was to find out if SATIN has utility in the security domain and if the design and implementation of SATIN is usable by third parties.

ZION assumes an ambient computing environment, where an application needs some form of security at all times; however, the environment can be very dynamic. Hence, the security mechanism needed depends on the current status of the environment. As such, ZION uses Ponder [Damianou et al., 2001], a policy specification language, to allow the applications to specify security policies, SATIN components to implement those policies and the SATIN container, advertising, discovery and deployer components to load and use the components that implement the security policies when the context changes. Thus, the ZION project integrated SATIN, Ponder and the Java Cryptography classes, to offer a component based, adaptable security framework.

Figure 6.20 shows a high level overview of the ZION architecture. At the top level, applications use the *Core Engine* to define security policies using Ponder. A security policy defines the type of *encryption* and *authentication* to use, based on the current context. The *Context Domain* is responsible for reasoning and notifying about changes in context. The *Security Manager* is responsible gathering the necessary information about context from the context domain, deciding if a policy needs to be enforced, and translating between a policy and the required SATIN components needed to implement it. The SATIN middleware system was used to locate, locally or remotely and instantiate those components.

Facets were used to define a common interface that all security components must implement and the SATIN attribute system to describe each component. For example, an encryption component implementing 128-bit encryption, advertised this information using the following attributes:

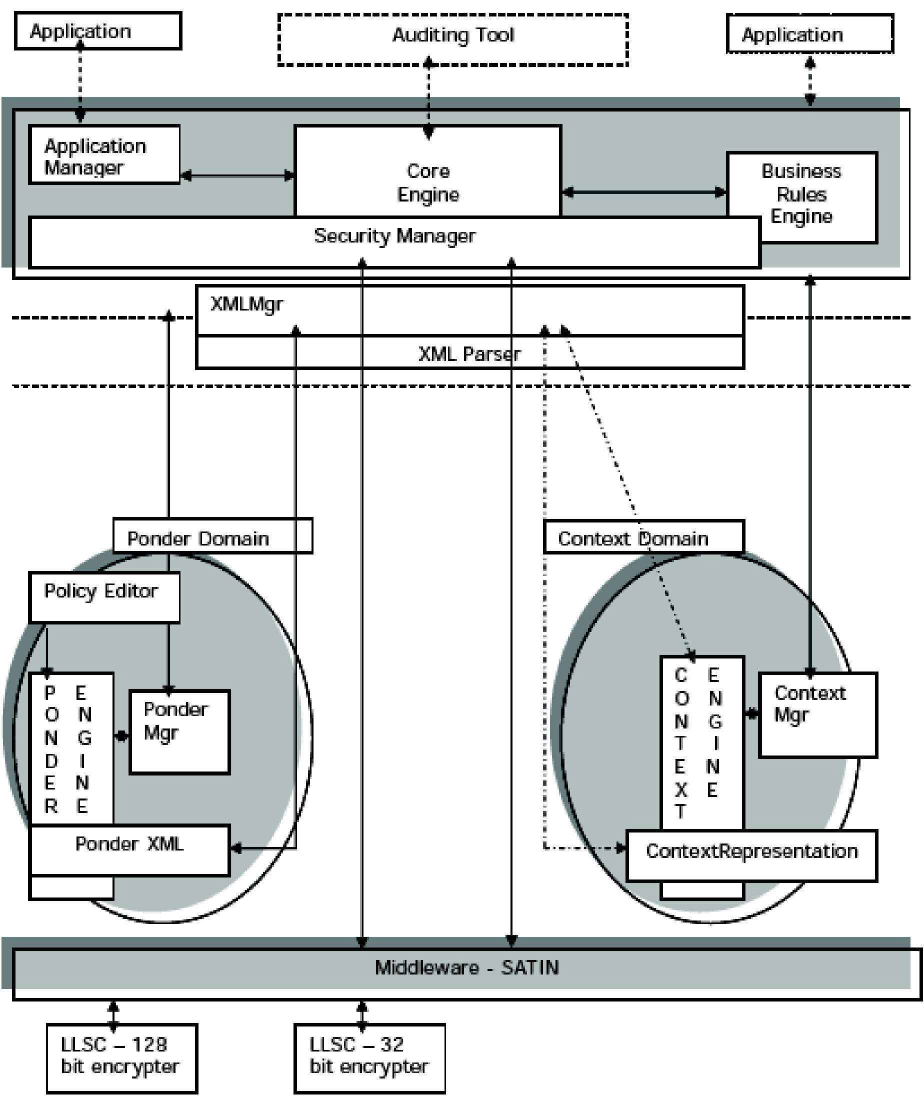


Figure 6.20: The ZION Architecture. Taken from [Chatterjee et al., 2004].

Key	**Value**
`ID`	STN:ENCRYPTER128
`TYPE`	SECURITY
`ENCRYPTLEVEL`	128
`ALGORITHM`	AES

A report was requested from the ZION project on the usability of SATIN. The report and Masters thesis stated that the

> *"features of SATIN make it ideal for this security framework. All components in the system will be represented as SATIN components with a set of properties.*

> *These components would register themselves with the SATIN container and advertise their presence. On context change, the system would re-organise i.e. appropriate components would be dynamically loaded using the discovery service based on attributes defined in the security policy.*"

From this, it can be deduced that the ideas behind SATIN (i.e. a local component meta-model offering the flexible use of logical mobility primitives), appear to be highly applicable in creating a modular and dynamic security framework.

Moreover it was claimed that

> "*The successful, seamless usage of Ponder, SATIN, XML, JAVA (particularly Java Crypto classes), has illustratively demonstrated that the component based middleware SATIN can be successfully deployed for a security environment and a high-level security policy definition language can be used effectively to map on to low-level security components whenever defined context change takes place.*"

This quote shows that SATIN was successfully used to build a security framework.

The report further claims that the "*component-based design is logical*" (i.e. the component model is usable) and that it "*allows plug-in of low-level components*" (i.e., it allows system-level, rather than only application-level adaptation). The students claimed they utilised most of the abstractions in the SATIN design.

Finally, it was claimed that SATIN is easy to develop with and to integrate with the rest of the project. "*to integrate SATIN with ZION (our system), we* [the students] *have added several classes and interfaces. We performed the implementation and integration with ease*". Understanding the SATIN middleware system Application Programmers Interface and the SATIN component model abstractions and confidently using them took 9 days. Developing the appropriate SATIN components and integrating with the rest of the system took 11 days and ended one day ahead of schedule. SATIN "*did 95% of what we* [the students] *expected it to do*".

It can thus be concluded that the students found utility in SATIN, while also claiming that it was easy to understand and use.

6.6 Discussion

The *validation* and *evaluation* of SATIN show the following:

Feasibility: The implementation of SATIN shows that a small footprint mobile computing middleware system, built using the SATIN component model and thus offering the

flexible and systematic use of logical mobility techniques, is *feasible*. The testing of the applications, shows that the implementation, which has not been optimised yet, allows mobile applications to use logical mobility primitives to *adapt* to changes of context, with limited performance trade offs.

Applicability: The applications developed using SATIN demonstrate how it can be used to offer adaptation to applications that address different domains. The ease of adding functionality to the system was also demonstrated, by adapting BeanShell to run as a SATIN component.

Ease of Use: The report obtained from the ZION project, as well as the adaptation of BeanShell and JOrbis to SATIN components demonstrate that the programming interface that SATIN offers is easy to use and develop against. Moreover, converting existing projects for use under SATIN is straightforward. The componentised applications use the SATIN primitives to gain *mobility*.

Suitability for Adaptable Mobile Systems Development: The experience in designing Q-CAD, as well as the use of SATIN in the ZION project, demonstrate that the abstractions, functionality and extensibility offered by the SATIN component model and middleware system are powerful enough to build adaptive mobile systems. Q-CAD further demonstrates how SATIN can be extended and used to build an *application-aware* adaptable system.

It was not the purpose of this chapter to extensively benchmark the various performance aspects of SATIN. It is expected that the deployer could implement various different client/server protocols, the registrar could use many different conflict detection and reconciliation algorithms, etc. The modularity and extensibility of the SATIN component model and middleware system makes it possible to use different components that implement different algorithms, while still exporting the same facets.

The SATIN middleware system and its implementation are offered as a proof of concept middleware and its use in applications and projects was detailed in order to evaluate whether the abstractions that the SATIN component model offers are powerful enough to support adaptive mobile systems.

6.7 Summary

This chapter has presented the implementation of the instantiation of the SATIN component model as a middleware system and discussed its evaluation. In particular, it was demonstrated that a lightweight middleware system using the SATIN component model and thus offering the flexible use of logical mobility techniques is feasible. The applications built take advantage of the abstractions and functionality offered by SATIN, to demonstrate

various aspects of mobile adaptation. The modularity and extensibility of the system was demonstrated, by showing how to add core functionality to the middleware. Moreover, the ZION report and the adaptation of existing applications showed that the system is easy to use. The design of the Q-CAD system also showed how SATIN can be used as a core framework to build *application-aware* adaptable mobile systems.

It is thus concluded that the use of the SATIN component model and middleware system facilitates the development of adaptive mobile systems, by allowing for the exploitation of powerful abstractions by imposing little overhead on the developer or the system.

Chapter 7

Conclusion and Future Work

This thesis identified a limitation in current mobile systems; namely, mobile computing systems are often monolithic and static and fail to adapt to accommodate to changes to their requirements. Given the physical mobility and constrained resources of mobile devices, the requirements of a mobile system may change - thus, the need to adapt to accommodate these changes is a likely eventuality.

The main goal of the work presented is to provide a systematic framework for mobile adaptation. To that end, the advantages of the flexible use of logical mobility primitives for mobile computing were identified; in particular, the thesis presented a framework that offers the flexible use of logical mobility primitives and which was used to dynamically transfer functionality between mobile nodes. The SATIN component meta model was defined and used to engineer adaptable systems, representing them as a collection of interoperable collocated components. The component metamodel offers the flexible use of logical mobility primitives to applications. Its utility was shown by instantiating it to design and implement a component-based middleware system, which was then used to evaluate this approach by developing and adapting a number of applications with it. Q-CAD, a QoS and context aware resource discovery framework was also built as an extension of the middleware system and provided a decision logic on how a system should adapt.

This last chapter summarises the main contributions of the thesis, critically evaluates it and outlines some ideas for future research.

7.1 Contributions

The following paragraphs summarise the contributions of this thesis to mobile computing research.

Use of Logical Mobility for Mobile Adaptation

This work identified the advantages of the systematic and flexible use of logical mobility primitives in various mobile application domains. Thus, it concluded that a flexible solution is needed, that is able to implement any logical mobility paradigm.

To that end, the Logical Mobility Entity abstraction was defined as a generalisation of a class, an instance or application data. Thus, a logical mobility entity instance represents a single aspect of the logical layer of a mobile system. The Logical Mobility Unit was defined as a container that is formed as a composition of many logical mobility entities. The logical mobility unit was described using a set of attributes.

A framework was also defined, that is able to send and receive logical mobility units and export this functionality flexibly to applications. It was illustrated that it is capable of implementing any logical mobility paradigm. Logical mobility was used to transfer functionality between mobile nodes, thus permitting mobile system adaptation.

Component Metamodel for Mobile Adaptation

Defining a platform that can be used to export the flexible use of logical mobility is not enough to enable mobile system adaptation. The system itself also needs to be *engineered* for adaptation.

As such, the logical mobility framework was encapsulated and offered as a first class citizen of an in-process component metamodel, targeting mobile systems. The model offers a structured way to build a laissez-faire adaptable mobile system and defines the notion of a Reflective component, as a component that can adapt via the use of the logical mobility primitives offered by the component model.

The SATIN component model was realised as the SATIN mobile computing middleware system, which was implemented and used for a number of applications. Finally, Q-CAD, a decision logic on how a mobile system should adapt based on application profiles and utility functions, was built using the SATIN middleware, transforming it from a laissez-faire adaptation system to an application-aware one.

7.2 Critical Evaluation

As pointed out in Chapter 1, the goal of this work is to offer adaptation in a constrained environment. This section evaluates SATIN with respect to this criterion.

Mobile Adaptation

This thesis has identified a number of abstractions and primitives for mobile adaptation. In particular, the Logical Mobility Entity, Logical Mobility Unit and Reflective component abstractions, along with the corresponding attributes to describe LMUs and components, were defined and used to offer adaptation.

The evaluation of the work, as illustrated in Chapter 6, illustrates that these abstractions are sufficient to allow reasoning about and adapting to changes to context. In particular, the applications and systems that were built demonstrate the applicability of these abstractions to offer different aspects of adaptation.

Lightweight System

SATIN is specifically designed to target mobile devices, which are, by their very nature, resource constrained, as argued in Chapter 2. The goal of a mobile computing middleware system is to offer higher-level primitives and functionality, while imposing as little overhead as possible.

The realisation of SATIN as a component - based middleware system and its implementation using Java 2 Micro Edition illustrates that a lightweight and dynamically adaptable system can be built using the SATIN component metamodel and that the added functionality can be offered without imposing a significant overhead on the device.

7.3 Future Work

Broader Definition of Context

SATIN defines reasoning about the context as discovering what components are available locally and remotely. In particular, it is assumed that remote resources are advertised as SATIN advertisable components and detected by a discovery service. Moreover, little emphasis is placed on evaluating the status of the local node. It is assumed that this information would be available through the use of the `LocalHost` component. A flexible

attribute system was defined to describe components - an ontology for that system was not been specified.

This information is adequate to allow a system to adapt based on the functionality that is available locally and remotely. However, realistically, the local context (for example, remaining battery power, memory, etc.) will need to be taken into account, as discussed in Chapter 2. This work can be extended by having a broader definition of context and by redefining the `LocalHost` component to implement it. Moreover, a rich ontology for the SATIN attributes needs to be defined. In reality, it would be difficult to get all parties interested to agree on a single ontology. It is thus expected, that a translation mechanism would be needed. If a common ontology is not defined and used and a translation mechanism is not in place, then interoperability in a pervasive computing environment is impossible.

Decision Logic for Adaptation

SATIN provides a model and a platform to allow systems to adapt. As such, systems and applications using SATIN can detect changes to context and may use the exported functionality to adapt. What is not defined, however, is the *decision logic* on *how* to adapt, in an environment where there are many potential ways to adapt.

There are a number of ways in which this can be addressed. Q-CAD, outlined in Section 6.4, uses a richer definition of local and remote context, along with application specified profiles and quality of service requirements, to decide how a system should adapt. As such, the decision logic is automated, but is influenced by the application and, consequently, the end-user. Q-CAD was built using the SATIN component model and middleware and thus applications can express Q-CAD as a requirement in their dependencies attribute. Other approaches, such as genetic algorithms or expert systems [Power, 1990, Parunak and Brueckner, 2001] can be used to decide on how a system should adapt. These can also be built and offered using the component model.

Constraining Adaptation

SATIN does not provide mechanisms to constrain how a reflective component or the system in general can adapt. As outlined in Chapter 5, SATIN allows a system to adapt and mutate freely, potentially leaving it to an undesirable or even non-functional state. A general solution would be to extend the reflective component concept with a rule based mechanism which would constrain how it can adapt, or to extend the model to support component frameworks [Szyperski, 1999, Parlavantzas et al., 2000].

Multiple Containers

Recent versions of OpenCOM define the notion of a *capsule*, as a container that can encapsulate various components. An OpenCOM instance, can have multiple capsules. By using multiple capsules and by assigning different address spaces to different capsules, an instance of OpenCOM can operate on heterogeneous but tightly controlled hardware. Using this concept, a SATIN instance could have multiple containers, each with different components. The containers could then implement access policies on inter-container communications. A *secure* or *immutable* container could then be introduced, with components that must be present for an instance of SATIN to operate.

A Distributed Component Model

In Section 4.2, this thesis argued why the SATIN component model is a collocated one. In targeting SATIN to very dynamic environments, it was claimed that a local component model requires less resources than a distributed one and that it allows for autonomous operation in case of network failure, which was considered to be a frequent event. In a less dynamic environment, where it can be assumed that mobile devices are connected for large periods of time to services provided by an infrastructure over a high speed network connection, distributed component models can provide powerful abstractions with which to interact with the remote services.

A simple extension to the metamodel would be to have two types of containers on each node: One for components available locally and one for components available remotely. Together with a networked component reference that handles marshaling and unmarshaling of requests and replies, this extension could be used to make SATIN able to handle both local and distributed components. The instantiation of this extended metamodel into a middleware system could use the Q-CAD framework to reestablish a binding to a remote component if the current binding is lost. SATIN attributes could be used to express dependencies on remote components.

Security

As stated in Chapter 1, security and trust are outside the scope of this thesis. The issue was partially discussed in Chapter 3, when detailing the Trust & Security layer of the framework for logical mobility. There are a number of issues here: Making sure that the code received behaves as advertised, avoiding to send code to nodes that are not trusted etc. Section 3.3.3 outlined a number of potential solutions to this problem; heuristic antivirus checks, proof carrying code, digital signatures, trust models and encryption. The SATIN framework could be expanded to implement those ideas.

Aspect-Oriented Adaptation

Aspect-oriented programming defines constructs called Aspects, which treat the concerns of a separate set of objects or classes. In particular, aspects can be weaved into points of intersection to modify the behaviour of software. An aspect-oriented development methodology can be used to formally specify a SATIN system; the logical mobility primitives can be used to send and receive aspects to be weaved into points of intersection. As such, this can provide for a more formal method of deploying classes and objects into reflective components. Related approaches have been proposed in [Blair et al., 2000, Popovici et al., 2003].

Model Driven Architecture

The Model Driven Architecture (MDA) [Miller and J.Mukerji, 2001] is a specification that allows specifying a system in an abstract level in terms of systems functionality, separating the architecture from the implementation. The Platform Independent Model (PIM) is expressed in UML, and formally specifies how the system should be engineered and behave. It is then mapped into a Platform Specific Model (PSM), and alternative PSM implementations can be integrated by using various bridging solutions. With respect to SATIN, MDA could potentially be used to engineer platform-specific SATIN-based systems, by using the metamodel as the platform independent model.

7.4 Further Dissemination

SATIN is going to be released as an open source project and is currently being used internally as part of the Security Expert Initiative (SEINIT) [SEINIT Project, 2003] project, a European Union / Information Society project that aims to create a trusted and dependable security framework for ubiquitous computing. SATIN is being used to dynamically locate components that implement various security policies. SEINIT partially addresses the security concerns of SATIN, by using trust management techniques and IPSec tunnels to send information securely. Moreover, there are initial discussions with the European Space Agency to use SATIN for Programmable Active Networking in satellite networks. The general role of SATIN would be to inject components in various nodes in the network topology, which would transcode multimedia streams based on currently available bandwidth and data packet loss. This will require building signalling and network status sensing components as SATIN components. Finally, Q-CAD is currently being implemented.

7.5 Conclusions

This thesis identified and set out to investigate a problem with mobile systems: By its very nature, a mobile system is highly dynamic. Fluctuations in battery, network connectivity and resource availability (which are already limited) strongly influence the system execution context, while the physical mobility of the devices makes the remote host and service availability strongly variable. This variation implies that the system requirements may change and hence the application may have to adapt to accommodate to the change. As such, this thesis showed that monolithic architectures and static approaches to mobile systems engineering do not offer the flexibility that this dynamic environment requires, as mobile systems will have to *adapt*.

Hence, this thesis proposed a synergy of two approaches, logical mobility and component systems, to offer adaptation to mobile systems. Logical mobility, was used as a computational primitive to allow systems to adapt, by sending and receiving functionality at runtime, whereas, SATIN, a component metamodel that offers the use of logical mobility primitives as a first class citizen, was used to engineer a system for adaptation. The metamodel was instantiated as a middleware system, and used to develop new systems and applications, as well as adapt existing ones.

Thus, the contributions of this work are the following: It managed to prove that the combination of logical mobility and components, which is novel in the literature, can be used to adapt a mobile system. In particular, the examination of a general logical mobility framework for mobile systems contributes a study of general logical mobility primitives, modelled using a process algebra to verify correctness. The SATIN component metamodel contributes a system that can be used to build lightweight adaptable systems, taking advantage of the logical mobility framework. The middleware system instantiation of that metamodel contributes a general adaptable middleware system, that offers modular advertising and discovery mechanisms and allows the general use of logical mobility as a computational primitive. The evaluation was used to prove the feasibility, utility, usability and applicability of this work.

Reflecting on the work presented, it can be concluded that mobile systems can benefit from the flexibility offered by component systems offering logical mobility primitives, *without suffering significant performance penalties*. In particular, SATIN allows for building novel adaptable systems and also permits the porting of existing systems and applications, with the ports benefiting from the reconfigurational primitives that SATIN provides. The performance of an unoptimised version of SATIN running both new applications and ports of existing systems on mobile computing hardware that is several years old was found to be adequate and the time needed to adapt was measured to be minimal.

An alternative approach to using components and logical mobility, would be to create a programming language that allows the specification of modular systems but that also

offers built in logical mobility primitives, such as XMILE [Mascolo et al., 2001]. The major drawback of such an approach would be, however, that the complete software development chain, from the linker to the compiler to the development environment to the adaptable software itself, would have to be written from scratch. In contrast, SATIN allows the use of existing software and tools.

Looking into the future, the incorporation of a trust management system, instantiated as a collection of SATIN components and offered by the middleware seems like the most immediate step forward, with significant research questions to be answered in the mobile trust arena. Moreover, the porting of this architecture to even smaller devices such as sensors is under consideration - the severely limited resource availability of a sensor platform, offers a new set of research challenges.

Appendix A

Q-CAD Metadata Encoding

A.1 Application Profile

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

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


   <xs:element name="PROACTIVE">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="LOCAL_CONTEXT"/>
            <xs:element ref="REMOTE_CONTEXT"/>
            <xs:element ref="BIND"/>
            <xs:element ref="ADAPT"/>
         </xs:sequence>
         <xs:attribute name="id" type="xs:string" use="required"/>
      </xs:complexType>
```

```
</xs:element>

<xs:element name="REACTIVE">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="LOCAL_CONTEXT"/>
         <xs:element ref="REMOTE_CONTEXT"/>
         <xs:element ref="BIND"/>
         <xs:element ref="ADAPT"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="required"/>
   </xs:complexType>
</xs:element>


<xs:element name="LOCAL_CONTEXT">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="CONDITION" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="required"/>
   </xs:complexType>
</xs:element>
<xs:element name="CONDITION">
   <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="op" type="xs:string" use="required"/>
      <xs:attribute name="value" type="xs:string" use="required"/>
   </xs:complexType>
</xs:element>

<xs:element name="REMOTE_CONTEXT">
   <xs:complexType>
      <xs:choice>
         <xs:element ref="CONDITION" minOccurs="0" maxOccurs="unbounded"/>
         <xs:element ref="ATTRIBUTES" minOccurs="0" maxOccurs="unbounded"/>
      </xs:choice>
      <xs:attribute name="id" type="xs:string" use="required"/>
   </xs:complexType>
</xs:element>
<xs:element name="ATTRIBUTES">
   <xs:complexType>
```

```
        <xs:sequence>
          <xs:element ref="ATTRIBUTE" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="ATTRIBUTE">
      <xs:complexType>
        <xs:attribute name="key" type="xs:string" use="required"/>
        <xs:attribute name="op" type="xs:string" use="required"/>
        <xs:attribute name="value" type="xs:string" use="required"/>
      </xs:complexType>
    </xs:element>



    <xs:element name="BIND">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="BIND_RESOURCE" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="BIND_RESOURCE">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="REMOTE_CONTEXT"/>
          <xs:element ref="LOCAL_CONTEXT"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
    </xs:element>

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

```
            <xs:element ref="LOCAL_CONTEXT"/>
            <xs:element ref="REMOTE_CONTEXT"/>
            <xs:element ref="ATTRIBUTES"/>
         </xs:sequence>
         <xs:attribute name="id" type="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

A.2 Utility Function

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

   <xs:element name="UTILITY_FUNCTION">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="RETURN"/>
            <xs:element ref="MAXIMISE"/>
         </xs:sequence>
         <xs:attribute name="id" type="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>


   <xs:element name="RETURN">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="EVALUATE"/>
            <xs:element ref="FILTER"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="EVALUATE">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="ATTRIBUTE" minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
```

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

   <xs:element name="ATTRIBUTE">
      <xs:complexType>
         <xs:attribute name="key" type="xs:string" use="required"/>
         <xs:attribute name="op" type="xs:string"/>
         <xs:attribute name="value" type="xs:string"/>
         <xs:attribute name="weight" type="xs:int"/>
      </xs:complexType>
   </xs:element>

   <xs:element name="MAXIMISE">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="ATTRIBUTE" minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Appendix B

Programming With SATIN

This appendix provides some actual examples that show how to program using the SATIN metamodel and middleware system.

B.1 Transferring Components

In this section, component `HelloWorld` is created, with the following attributes:

Key	Value
ID	STN:HELLOWORLD
BITRATE	64
VER	1
DEP	
FACETS	ComponentFacet

It only implements the `ComponentFacet`. The component is located on Node A, and is represented by class `HelloWorld`. The `main()` method initialises the middleware and uses the Deployer to send the component to Node B, to be registered on its container.

The listing for class `HelloWorld` follows.

```
package edu.UCL.satin.apps.test;

import edu.UCL.satin.arch.attribute.GenericAttribute;
import edu.UCL.satin.arch.components.Component;
import edu.UCL.satin.arch.components.container.Container;
import edu.UCL.satin.arch.facets.ComponentFacet;
import edu.UCL.satin.arch.facets.DeployerFacet;
import edu.UCL.satin.arch.facets.RegistrarFacet;
```

```
import edu.UCL.satin.arch.lmu.LMU;
import edu.UCL.satin.impl.MiToolkitDeployer.HashLMU;
import edu.UCL.satin.impl.MiToolkitDeployer.MiToolkitDeployer;
import edu.UCL.satin.impl.container.Core;

/**
 * @author <a href="mailto:s.zachariadis@cs.ucl.ac.uk">Stefanos Zachariadis</a>
 *
 */
public class HelloWorld extends Component implements ComponentFacet {

        public HelloWorld() {
                super("STN:HELLOWORLD");
        }


        //this is the constructor of the component
        public boolean construct() {
                System.out.println("Hello SATIN world");
                //add a few attributes
                addAttribute(new GenericAttribute("BITRATE",new Integer(64),true));
                addAttribute(new GenericAttribute("VER",new Integer(1),true));
                addAttribute(new GenericAttribute("DEP","",true));
                addAttribute(new GenericAttribute("FACETS","ComponentFacet",true));
                return(true);
        }


        //this is the destructor of the component
        public void destroy() {
                System.out.println("Bye-bye cruel SATIN world");
                System.gc();
        }


        public boolean isEnabled() {
                //doesn't really do much in this component
                return true;
        }


        public void setEnabled(boolean enabled) {
                // doesn't really do much in this component
        }

        public static void main(String[] args) {
                new Core(); //initialise the container
```

```
                //get a reference to it
                Container container = Container.getContainer();

                //get a reference to the registrar
                RegistrarFacet registrar = container.getDefaultRegistrar();

                Component c=new HelloWorld(); //initialise the component

                //register the component. If registration failed, then die
                if (!registrar.registerComponent(c)) {
                        System.err.println("Registration failed. Exiting...");
                        System.exit(0); //this shouldn't really happen ;-)
                }

                //enables debug information
                //debuging is handled by a component
                container.getComponent("STN:DEBUG").setEnabled(true);

                DeployerFacet d = new MiToolkitDeployer(); //Initialise the deployer
                registrar.registerComponent((Component)d); //register it with the middleware
                d.setEnabled(true); //enable it

                /* The following statement creates a new LMU,
                 * with target "hamsalad.cs.ucl.ac.uk" and
                 * defines its destination as "STN:CONTAINER",
                 * which is the container of the recipient host.
                 *
                 * This effectively defines that the LMU should be sent to the
                 * container of hamsalad.cs.ucl.ac.uk.
                 *
                 * The container will try to register it.
                 */
                LMU lmu=new HashLMU("hamsalad.cs.ucl.ac.uk","STN:CONTAINER");
                lmu.addComponent(c); //adds the component to the LMU.

                d.send(lmu);//sends the LMU
        }

}
```

Node B runs code, represented by class `GetHelloWorld`. `GetHelloWorld` implements the `ComponentListener` facet. Using various attributes, `GetHelloWorld` requests to be notified when a component that has a `BITRATE` attribute with a value greater or equal than 32 and a `VER` attribute of exactly 1 is found. When it is found, it removes it from the system.

The listing for class `GetHelloWorld` follows.

```
package edu.UCL.satin.apps.test;

import java.util.Hashtable;

import edu.UCL.satin.arch.attribute.GenericAttribute;
import edu.UCL.satin.arch.components.Component;
import edu.UCL.satin.arch.components.container.Container;
import edu.UCL.satin.arch.facets.ComponentListener;
import edu.UCL.satin.impl.MiToolkitDeployer.MiToolkitDeployer;
import edu.UCL.satin.impl.arch.attribute.GreaterEqualThanFilter;
import edu.UCL.satin.impl.arch.attribute.MatchAttribute;
import edu.UCL.satin.impl.container.Core;

/**
 * @author <a href="mailto:s.zachariadis@cs.ucl.ac.uk">Stefanos Zachariadis</a>
 *
 */
public class GetHelloWorld implements ComponentListener {

        //this method is called when a component that was requested for was found
        public void componentFound(Component c) {
                System.out.println("The component "+c+" was registered");
                Container.getContainer().getDefaultRegistrar().removeComponent(c);
                System.out.println("The component was removed");
        }

        public static void main(String[] args) {

                //instantiates the container and the deployer. also enables debug
                Container c=new Core();
                c=Container.getContainer();
                c.getComponent("STN:DEBUG").setEnabled(true);
                c.getDefaultRegistrar().registerComponent(new MiToolkitDeployer());
                c.getComponent("STN:MITKDEP").setEnabled(true);

                GetHelloWorld obj=new GetHelloWorld();

                /*
                 * Creates a new query template. This template asks for the following:
                 * A "BITRATE" attribute with value greater or equal to 32
                 * A "VER" attribute with value of 1 (exactly)
                 */
                Hashtable template=new Hashtable();
                template.put("BITRATE",new MatchAttribute("BITRATE",new Integer(32),
                                   new GreaterEqualThanFilter()));
```

```
                template.put("VER",new GenericAttribute("VER",new Integer(1),true));

                //adds a listener for a component based on that template
                c.addListener((ComponentListener)obj,template);

        }
}
```

Starting Node B first and Node A second, the output that is obtained on Node B is the following (debugging output has been omitted for clarity):

```
[szachari@hamsalad devel]$ java  -classpath satin:MiToolkitDeployer:
                         /home/szachari/tmp/satin-tests
                         edu.UCL.satin.apps.test.GetHelloWorld
Hello SATIN world
The component ID=STN:HELLOWORLD,Bitrate=64,FACETS=ComponentFacet,DEP=,VER=1
was registered
Bye-bye cruel SATIN world
The component was removed
```

It shows the component received, constructed, registered and removed, with a notification of its registration being sent in between.

B.2 Advertising, Discovering and Requesting Components

Instead of passively waiting for a component to be sent, this section modifies the example given above to use the advertising and discovery framework of SATIN middleware system. In particular, the `HelloWorld` component implements the `Advertisable` facet and the example uses the centralised publish/subscribe instantiation of the advertising and discovery framework. As such, Node A runs the publish/subscribe server and an advertising client, which advertises `HelloWorld`. Note that it does not actively send the component to B.

The listing for the modified class `HelloWorld` follows.

```
package edu.UCL.satin.apps.test;

import edu.UCL.satin.arch.attribute.GenericAttribute;
import edu.UCL.satin.arch.components.Component;
import edu.UCL.satin.arch.components.container.Container;
import edu.UCL.satin.arch.facets.ComponentFacet;
import edu.UCL.satin.arch.facets.DeployerFacet;
```

```
import edu.UCL.satin.arch.facets.RegistrarFacet;
import edu.UCL.satin.facets.Advertisable;
import edu.UCL.satin.impl.MiToolkitDeployer.MiToolkitDeployer;
import edu.UCL.satin.impl.advertising.central.clients.advertising.CentralAdvertising;
import edu.UCL.satin.impl.advertising.central.server.AdvertisingServer;
import edu.UCL.satin.impl.container.Core;

/**
 * @author <a href="mailto:s.zachariadis@cs.ucl.ac.uk">Stefanos Zachariadis</a>
 *
 */
public class HelloWorld extends Component implements ComponentFacet, Advertisable {

        public HelloWorld() {
                super("STN:HELLOWORLD");
        }

        //this is the advertising message of the component
        public Object getMessage() {
                return("Say hello to the world of advertising.");
        }

        //this is the constructor of the component
        public boolean construct() {
                System.out.println("Hello SATIN world");
                //add a few attributes
                addAttribute(new GenericAttribute("BITRATE",new Integer(64),true));
                addAttribute(new GenericAttribute("VER",new Integer(1),true));
                addAttribute(new GenericAttribute("FACETS","ComponentFacet,
                                                      Advertisable",true));
                return(true);
        }


        //this is the destructor of the component
        public void destroy() {
                System.out.println("Bye-bye cruel SATIN world");
        }


        public boolean isEnabled() {
                //doesn't really do much in this component
                return true;
        }


        public void setEnabled(boolean enabled) {
```

```
                // doesn't really do much in this component
        }

        public static void main(String[] args) {
                new Core(); //initialise the container
                Container container = Container.getContainer(); //get a reference to it
                //get a reference to the registrar
                RegistrarFacet registrar = container.getDefaultRegistrar();
                Component c=new HelloWorld(); //initialise the component

                //register the component. If registration failed, then die
                if (!registrar.registerComponent(c)) {
                        System.err.println("Registration failed. Exiting...");
                        System.exit(0); //this shouldn't really happen ;-)
                }

                //enables debug information
                //debug is handled by a component
                container.getComponent("STN:DEBUG").setEnabled(true);

                //creates a new advertising server, registers it and starts it
                AdvertisingServer srv=new AdvertisingServer();
                registrar.registerComponent(srv);
                srv.setEnabled(true);

                DeployerFacet d = new MiToolkitDeployer(); //Initialise the deployer
                registrar.registerComponent((Component)d); //register it with the middleware
                d.setEnabled(true); //enable it


                //creates a new advertising client, registers it,
                //gives it the location of the advertising server
                //registers HelloWorld to be advertised and starts it.
                CentralAdvertising adv=new CentralAdvertising();
                registrar.registerComponent(adv);
                adv.setIP("localhost");
                adv.addAdvertisable((Advertisable)c);
                adv.setEnabled(true);

        }

}
```

Similarly, class `GetHelloWorld` is modified to start a discovery service, bound to the server running on A. Additionally to listening for the local availability of the `HelloWorld` component (as explained in the previous section), `GetHelloWorld` registers a listener with

the discovery service to be notified when `HelloWorld` is located remotely. When this occurs, it is requested, downloaded and installed locally, using the Deployer.

The listing for the modified class `GetHelloWorld` follows.

```
package edu.UCL.satin.apps.test;

import java.util.Hashtable;

import edu.UCL.satin.arch.attribute.GenericAttribute;
import edu.UCL.satin.arch.components.Component;
import edu.UCL.satin.arch.components.Deployer;
import edu.UCL.satin.arch.components.container.Container;
import edu.UCL.satin.arch.facets.ComponentListener;
import edu.UCL.satin.impl.MiToolkitDeployer.MiToolkitDeployer;
import edu.UCL.satin.impl.advertising.central.clients.discovery.CentralDiscovery;
import edu.UCL.satin.impl.arch.attribute.GreaterEqualThanFilter;
import edu.UCL.satin.impl.arch.attribute.MatchAttribute;
import edu.UCL.satin.impl.arch.components.comms.RemoteComponent;
import edu.UCL.satin.impl.container.Core;

/**
 * @author <a href="mailto:s.zachariadis@cs.ucl.ac.uk">Stefanos Zachariadis</a>
 *
 */
public class GetHelloWorld implements ComponentListener {

        //this method is called when a component that was requested for was found
        //note that in this case, the component can be both local and remote
        public void componentFound(Component c) {

                if(c instanceof RemoteComponent) { //the component was found remotely
                        System.out.println("The component " + c + " that says " +
              ((RemoteComponent)(c)).getMessage() +
              " was found remotely, at location "
              +((RemoteComponent)(c)).getLocation().asString());
                        //gets an instance of the deployer
                 =(MiToolkitDeployer)(Container.getContainer().getComponent("STN:MITKDEP"));
                        d.asyncReceive((RemoteComponent)(c),Container.getContainer());
                }
                else { //the component was found locally
                        System.out.println("The component "+c+" was registered");
                        Container.getContainer().getDefaultRegistrar().removeComponent(c);
                        System.out.println("The component was removed");
                }
        }
```

```
        public static void main(String[] args) {

                //instantiates the container and the deployer. also enables debug
                Container c=new Core();
                c=Container.getContainer();
                c.getComponent("STN:DEBUG").setEnabled(true);
                c.getDefaultRegistrar().registerComponent(new MiToolkitDeployer());
                c.getComponent("STN:MITKDEP").setEnabled(true);

                GetHelloWorld obj=new GetHelloWorld();


                //creates a new discovery component, tells it where to discover from
                //and registers it
                CentralDiscovery disc=new CentralDiscovery();
                c.getDefaultRegistrar().registerComponent(disc);
                disc.setIP("sandwich.cs.ucl.ac.uk");


                /*
                 * Creates a new query template. This template asks for the following:
                 * A "BITRATE" attribute with value greater or equal to 32
                 * A "VER" attribute with value of 1 (exactly)
                 */
                Hashtable template=new Hashtable();
                template.put("BITRATE",new MatchAttribute("BITRATE",new Integer(32),
                      new GreaterEqualThanFilter()));

                //adds a listener for a component based on that template.
                //the listener is added both to the container and the discovery service
                c.addListener((ComponentListener)obj,template);
                disc.addListener((ComponentListener)obj,template);


                //starts the discovery service
                disc.setEnabled(true);
        }
}
```

Starting Node A first and Node B second, the output that is obtained on Node B is the following (debugging output has been omitted for clarity):

```
[szachari@hamsalad devel]$ java  -classpath satin:MiToolkitDeployer:
             /home/szachari/tmp/satin-tests edu.UCL.satin.apps.test.GetHelloWorld
The component ID=STN:HELLOWORLD,BITRATE=64,FACETS=ComponentFacet,VER=1
that says Say hello
```

```
to the world of advertising. was found remotely, at location 128.16.66.118
Hello SATIN world
The component ID=STN:HELLOWORLD,BITRATE=64,FACETS=ComponentFacet,VER=1 was registered
Bye-bye cruel SATIN world
The component was removed
```

It shows a component being located remotely, received, constructed, deployed and removed.

Bibliography

[Abowd et al., 1997] Abowd, G., Atkeson, C., Hong, J., Long, S., Kooper, R., and Pinkerton, M. (1997). Cyberguide: A Mobile Context-Aware Tour Guide. *Baltzer/ACM Wireless Networks*, 3(5):421–433.

[Anderson et al., 2002] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. (2002). SETI@home: An Experiment in Public-resource Computing. *Comm. ACM*, 45(11):56–61.

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

[Bailey, 2000] Bailey, E. C. (2000). *Maximum RPM.* Red Hat, Inc.

[Baldi et al., 1997] Baldi, M., Gai, S., and Picco, G. P. (1997). Exploiting Code Mobility in Decentralized and Flexible Network Management. In *Proceedings of the First International Workshop on Mobile Agents*, pages 13–26, Berlin, Germany.

[Baldi and Picco, 1998] Baldi, M. and Picco, G. (1998). Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In *Proc. of the* 20th *Int. Conf. on Software Engineering.*

[Becker et al., 2004] Becker, C., Handte, M., Schiele, G., and Rothermel, K. (2004). PCOM - A Component System for Pervasive Computing. In *Proceedings of the 2nd International Conference on Pervasive Computing and Communications*, Orlando, Florida.

[Becker and Schiele, 2003] Becker, C. and Schiele, G. (2003). Middleware and Application Adaptation Requirements and Their Support in Pervasive Computing. In *Proceedings of the 3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (DARES), collocated with ICDCS'03*, pages 98–103. IEEE Computer Society.

[Bieszczad et al., 1998] Bieszczad, A., Pagurek, B., and White, T. (1998). Mobile Agents for Network Management. *IEEE Communications Surveys*, 1(1):2–9.

[Blair et al., 1998] Blair, G., Coulson, G., Robin, P., and Papathomas, M. (1998). An Architecture for Next Generation Middleware. In *Proc. of Middleware '98*, LNCS, pages 191–206. Springer Verlag.

[Blair et al., 2001] Blair, G., G. Coulson, G., Andersen, A., Blair, L., Costa, M. C. F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., and Saikoski, K. (2001). The Design and Implementation of OpenORB v2. *IEEE Distributed Systems Online, Special Issue on Reflective Middleware*, 2(6).

[Blair et al., 2000] Blair, G. S., Blair, L., Andersen, A., and Jones, T. (2000). A Formal View of Aspects in the Development of Component-Based Distributed Systems. In *SCI'2000 invited session on Generative and Component-Based Software Engineering*, Orlando, Florida, USA.

[Blair et al., 2002] Blair, G. S., Coulson, G., Blair, L., Duran-Limon, H., Grace, P., Moreira, R., and Parlavantzas, N. (2002). Reflection, Self-Awareness and Self-Healing in OpenORB. In *Proceedings of the first workshop on Self-healing systems*, pages 9–14. ACM Press.

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

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

[Bromberg and Issarny, 2004] Bromberg, D. and Issarny, V. (2004). Service Discovery Protocols Interoperability in the Mobile Environment. In *Proceedings of the International Workshop Software Engineering and Middleware (SEM).*

[Capra, 2003] Capra, L. (2003). *Reflective Mobile Middleware for Context-Aware Applications.* PhD thesis, University College London, United Kingdom.

[Capra, 2004] Capra, L. (2004). Engineering Human Trust in Mobile System Collaborations. In *Proc. of the 12th International Symposium on the Foundations of Software Engineering (SIGSOFT 2004/FSE-12)*, Newport Beach, CA, USA. ACM Press.

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

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

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

[Capra et al., 2005] Capra, L., Zachariadis, S., and Mascolo, C. (2005). Q-CAD: QoS and Context Aware Discovery Framework for Adaptive Mobile Systems. In *International Conference on Pervasive Services (ICPS05)*, Santorini, Greece. IEEE.

[Carzaniga et al., 1997] Carzaniga, A., Picco, G. P., and Vigna, G. (1997). Designing Distributed Applications with Mobile Code Paradigms. In *Proceedings of the 19th International Conference on Software Engineering*, pages 22–32, Boston, MA. ACM Press.

[Cervantes and Hall, 2002] Cervantes, H. and Hall, R. (2002). BEANOME: A Component Model for the OSGi Framework. In *Software Infrastructures for Component-Based Applications on Consumer Devices*, Lausanne.

[Cervantes and Hall, 2004] Cervantes, H. and Hall, R. (2004). Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *Proceedings of the 26th International Conference of Software Engineering (ICSE 2004)*, pages 614–623, Edinburgh, Scotland. ACM Press.

[Chatterjee et al., 2004] Chatterjee, K., Ho, P., Mehdi, W., Shariff, F., and Solangi, M. (2004). ZION: A Lightweight SEINIT based Security Framework Implementation for Pervasive Computing. Master's thesis, University College London, United Kingdom.

[Cheverst et al., 2000] Cheverst, K., Davies, N., Mitchell, K., Friday, A., and Efstratiou, C. (2000). Developing a Context-aware Electronic Tourist Guide: Some Issues and Experiences. In *Proceedings of CHI 2000*, pages 17–24, Netherlands.

[Chinnici et al., 2003] Chinnici, R., Gudgin, M., Moreau, J.-J., and Weerawarana, W. (2003). *Web Services Description Language (WSDL) 1.2*. World Wide Web Consortium.

[Clarke et al., 2001] Clarke, M., Blair, G. S., Coulson, G., and Parlavantzas, N. (2001). An Efficient Component Model for the Construction of Adaptive Middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 160–178. Springer-Verlag.

[Conti et al., 2004] Conti, M., Maselli, G., Turi, G., and Giordano, S. (2004). Cross-layering in Mobile ad Hoc Network Design. *IEEE Computer*, 37(2):48–51.

[Coulson et al., 2004] Coulson, G., Blair, G. S., Grace, P., Joolia, A., Lee, K., and Ueyama, J. (2004). OpenCOM v2: A Component Model for Building Systems Software. In *Proceedings of IASTED Software Engineering and Applications (SEA'04)*.

[Cugola and Picco, 2002a] Cugola, G. and Picco, G. (2002a). Peer-to-Peer for Collaborative Applications. In *Proceedings of the IEEE International Workshop on Mobile Teamwork Support, Collocated with ICDCS'02*, pages 359–364.

[Cugola and Picco, 2002b] Cugola, G. and Picco, G. P. (2002b). PeerWare: Core Middleware Support for Peer-to-Peer and Mobile Systems. Submitted for publication.

[Damianou et al., 2001] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. (2001). The Ponder Policy Specification Language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38. Springer-Verlag.

[Eddon and Eddon, 1998] Eddon, G. and Eddon, H. (1998). *Inside Distributed COM*. Microsoft Programming Series. Microsoft Press, Redmond, WA.

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

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

[Fallside, 2000] Fallside, D. (2000). XML Schema. Technical Report http://www.w3.org/TR/xmlschema-0/, World Wide Web Consortium.

[Ferscha et al., 2004] Ferscha, A., Hechinger, M., Mayrhofer, R., and R.Oberhauser (2004). A Light-Weight Component Model for Peer-to-Peer Applications. In *2nd International Workshop on Mobile Distributed Computing*. IEEE Computer Society Press.

[Fuggetta et al., 1998] Fuggetta, A., Picco, G., and Vigna, G. (1998). Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361.

[Gavalas et al., 1999] Gavalas, D., Greenwood, D., Ghanbari, M., and O'Mahony, M. (1999). Using Mobile Agents for Distributed Network Performance Management. In Albayrak, S., editor, *Proceedings of the 3rd International Workshop on Intelligent Agents for Telecommunication Applications (IATA-99)*, volume 1699 of *LNAI*, pages 96–112, Berlin. Springer.

[Georgiadis et al., 2002] Georgiadis, I., J, M., and Kramer, J. (2002). Self-Organising Software Architectures for Distributed Systems. In *Proceedings of the first workshop on Self-healing systems*, pages 33–38. ACM Press.

[Grace et al., 2003a] Grace, P., Blair, G., and Samuel, S. (2003a). ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability. In *Proceedings of International Symposium on Distributed Objects and Applications (DOA 2003)*, pages 1170–1187, Sicily, Italy. Springer.

[Grace et al., 2003b] Grace, P., Blair, G. S., and Samue, S. (2003b). Middleware Awareness in Mobile Computing. In *Proceedings of First IEEE International Workshop on Mobile Computing Middleware (MCM03) (co-located with ICDCS03)*, pages 382–387.

[Granroth, 2000] Granroth, K. (2000). Using KDE Components (KParts). Unpublished Invited Talk at Annual Linux Showcase 2000.

[Grassi and Mirandola, 2002] Grassi, V. and Mirandola, R. (2002). PRIMAmob-UML: a Methodology for Performance Analysis of Mobile Software Architectures. In *Proceedings of the 3rd International Workshop on Software and Performance (WOSP-02)*, pages 262–274, New York. ACM Press.

[Grassi and Mirandola, 2003] Grassi, V. and Mirandola, R. (2003). Derivation of Markov Models for Effectiveness Analysis of Adaptable Software Architectures for Mobile Computing. *IEEE Transactions on Mobile Computing*, 2(02):114–131.

[Hall and Cervantes, 2003] Hall, R. S. and Cervantes, H. (2003). Gravity: Supporting Dynamically Available Services in Client-side Applications. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 379–382. ACM Press.

[Hall et al., 1999] Hall, R. S., Heimbigner, D., and Wolf, A. L. (1999). A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 174–183. IEEE Computer Society Press / ACM Press.

[Hamilton, 1997] Hamilton, G. (1997). JavaBeans. Version 1.01.

[Hashman and Knudsen, 2001] Hashman, S. and Knudsen, S. (2001). The Application of Jini Technology to Enchance the Delivery of Mobile Services. http://www.sun.com/jini/whitepapers/PsiNapticMIDs.pdf.

[Holder et al., 1999] Holder, O., Ben-Shaul, I., and Gazit, H. (1999). Dynamic Layout of Distributed Applications in FarGo. In *Proceedings of International Conference on Software Engineering*, pages 163–173.

[Ijaha, 2004] Ijaha, M. (2004). Mitoolkit. Master's thesis, University College London, United Kingdom.

[JCraft, 2001] JCraft (2001). Jorbis – pure java ogg vorbis decoder. http://www.jcraft.com/jorbis/.

[Kangas and Oening, 1999] Kangas, K. and Oening, J. (1999). Using Code Mobility to Create Ubiquitous and Active Augmented Reality in Mobile Computing. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*.

[Katz, 1994] Katz, R. H. (1994). Adaptation and Mobility in Wireless Information Systems. *IEEE Personal Communications*, 1:6–17.

[Keegan and O'Hare, 2003] Keegan, S. and O'Hare, G. (2003). EasiShop: Enabling uCommerce through Intelligent Mobile Agent Technologies. In *5th Int. Workshop on Mobile Agents for Telecommunication Applications (MATA03)*, pages 200–209. Springer.

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

[Kon et al., 2002] Kon, F., Costa, F., Blair, G. S., and Campbell, R. H. (2002). The Case for Reflective Middleware. *Commun. ACM*, 45(6):33–38.

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

[Kort and Dulaney, 2004] Kort, T. and Dulaney, K. (2004). Sony's Exit from the PDA Market Hurts PalmSource. http://www4.gartner.com/resources/121100/121196/sonys_exit_from.pdf.

[Lange and Oshima, 1998] Lange, D. B. and Oshima, M. (1998). *Programming and Deploying Java^{TM} Mobile Agents with Aglets^{TM}*. Addison-Wesley.

[Lindholm and Yellin, 1999] Lindholm, T. and Yellin, F. (1999). *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc.

[Magee and Kramer, 1999] Magee, J. and Kramer, J. (1999). *Concurrency: Models and Programs - From Finite State Models to Java Programs*. John Wiley.

[Mascolo et al., 2002a] Mascolo, C., Capra, L., and Emmerich, W. (2002a). Middleware for Mobile Computing (A Survey). In E. Gregori and G. Anastasi and S. Basagni, editor, *Advanced Lectures on Networking*, number 2497, pages 20–58. Springer.

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

[Mascolo et al., 2001] Mascolo, C., Zanolin, L., and Emmerich, W. (2001). XMILE: an XML based Approach for Incremental Code Mobility and Update. Submitted for Publication.

[Mettala, 1999] Mettala, R. (1999). Bluetooth Protocol Architecture. http://www.bluetooth.com/developer/whitepaper/.

[Mikic-Rakic and Medvidovic, 2002] Mikic-Rakic, M. and Medvidovic, N. (2002). Architecture-Level Support for Software Component Deployment in Resource Constrained Environments. In Bishop, J. M., editor, *Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*, pages 31–50. Springer.

[Miller and J.Mukerji, 2001] Miller, J. and J.Mukerji (2001). Model Driven Architecture. OMG Document number ormsc/2001-07-01.

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

[Murdock, 1994] Murdock, I. (1994). Overview of the Debian GNU/Linux system. *Linux Journal*, 6.

[Murphy et al., 2001] Murphy, A., Picco, G., and Roman, G.-C. (2001). Lime: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-01)*, pages 524–536, Los Alamitos, CA. IEEE Computer Society.

[Necula, 1997] Necula, G. C. (1997). Proof-carrying code. In *The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM SIGACT and SIGPLAN, ACM Press.

[Niemeyer, 1997] Niemeyer, P. (1997). BeanShell - Lightweight Scripting for Java.

[OMG, 1995] OMG (1995). *The Common Object Request Broker: Architecture and Specification Revision 2.0.* 492 Old Connecticut Path, Framingham, MA 01701, USA.

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

[OMG, 2000] OMG (2000). Meta Object Facility (MOF) Specification. Technical report, Object Management Group.

[OMG, 2003] OMG (2003). *Unified Modeling Language*. Object Management Group. B((Version 1.5 B)), http://www.omg.org/docs/formal/03-03-01.pdf.

[PalmInfocenter, 2004] PalmInfocenter (2004). Sony to Suspend Clie Handheld Line. http://www.palminfocenter.com/view_story.asp?ID=6866.

[PalmSource, 2004a] PalmSource (2004a). Conduits and the Palm OS Platform. http://www.palmos.com/dev/support/docs/conduits/win/Intro_CondPPPlatform.html.

[PalmSource, 2004b] PalmSource (2004b). Palmsource Developers Program. http://www.palmsource.com/developers/.

[Parlavantzas et al., 2000] Parlavantzas, N., Coulson, G., and Blair, G. S. (2000). Applying Component Frameworks to Develop Flexible Middleware. In *Proceedings of the Workshop on Reflective Middleware (RM'2000)*, New York.

[Parunak and Brueckner, 2001] Parunak, H. V. D. and Brueckner, S. (2001). Entropy and Self-Organization in Multi-agent Systems. In *Proceedings of the fifth international conference on Autonomous agents*, pages 124–130. ACM Press.

[Picco, 1998a] Picco, G. (1998a). μCODE: A Lightweight and Flexible Mobile Code Toolkit. In Rothermel, K. and Hohl, F., editors, *Proc. 2nd Int. Workshop on Mobile Agents*, LNCS 1477. Springer.

[Picco et al., 1999] Picco, G., Murphy, A., and Roman, G.-C. (1999). LIME: Linda meets Mobility. In *Proc. 21st Int. Conf. on Software Engineering (ICSE-99)*, pages 368–377. ACM Press.

[Picco, 1998b] Picco, G. P. (1998b). μCODE: A Lightweight and Flexible Mobile Code Toolkit. In Rothermel, K. and Hohl, F., editors, *Proceedings of the 2nd International Workshop on Mobile Agents*, Lecture Notes in Computer Science, pages 160–171, Berlin, Germany. Springer-Verlag.

[Picco, 1998c] Picco, G. P. (1998c). *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility.* PhD thesis, Politecnico di Torino, Italy.

[Picco et al., 2001] Picco, G. P., Roman, G.-C., and McCann, P. J. (2001). Reasoning About Code Mobility With Mobile UNITY. *ACM Trans. Softw. Eng. Methodol.*, 10(3):338–395.

[Popa et al., 2004] Popa, L., Athanasiu, I., Raiciu, C., Pandey, R., and Teodorescu, R. (2004). Using Code Collection to Support Large Applications on Mobile Devices. In *Proceedings of the 10th annual international conference on Mobile computing and networking*, pages 16–29. ACM Press.

[Popovici et al., 2003] Popovici, A., Frei, A., and Alonso, G. (2003). A Proactive Middleware Platform for Mobile Computing. In *Middleware 2003*, pages 455–473, Rio De Janeiro, Brazil.

[Power, 1990] Power, J. (1990). Distributed Systems and Self-Organization. In *Proceedings of the 1990 ACM annual conference on Cooperation*, pages 379–384. ACM Press.

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

[Roman et al., 1997] Roman, G.-C., McCann, P. J., and Plun, J. Y. (1997). Mobile UNITY: Reasoning and Specification in Mobile Computing. *ACM Trans. Softw. Eng. Methodol.*, 6(3):250–282.

[Roman et al., 2000] Roman, G.-C., Murphy, A. L., and Picco, G. P. (2000). Software Engineering for Mobility: A Roadmap. In *The Future of Software Engineering - 22nd Int. Conf. on Software Engineering (ICSE2000)*, pages 243–258. ACM Press.

[Roman and Islam, 2004] Roman, M. and Islam, N. (2004). Dynamically Programmable and Reconfigurable Middleware Services. In *Proceedings of Middleware '04*, Toronto.

[Roman et al., 2001] Roman, M., Kon, F., and Campbell, R. H. (2001). Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online Journal, Special Issue on Reflective Middleware*.

[Satyanarayanan, 1996] Satyanarayanan, M. (1996). Accessing Information on Demand at Any Location. Mobile Information Access. *IEEE Personal Communications*, 3(1):26–33.

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

[SEINIT Project, 2003] SEINIT Project, T. (2003). Security Expert Initiative. http://www.seinit.org.

[Shaw and Garlan, 1996] Shaw, M. and Garlan, D. (1996). *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall.

[Solheim, 2004] Solheim, S. (2004). Analysts: Sony's Clie Pullback Jibes with Forecasts. http://www.eweek.com/article2/0,1759,1605466,00.asp.

[Sun Microsystems, 1998a] Sun Microsystems (1998a). Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/index.html.

[Sun Microsystems, 1998b] Sun Microsystems (1998b). *Java Object Serialization Specification*.

[Sun Microsystems, 1998c] Sun Microsystems (1998c). *Java Remote Method Invocation Specification*, Revision 1.50, JDK 1.2 edition.

[Sun Microsystems, 1998d] Sun Microsystems (1998d). *The Java Reflection API*.

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

[Sun Microsystems, Inc., 2001a] Sun Microsystems, Inc. (2001a). Java 2 Enterprise Edition. http://java.sun.com/products/j2ee/.

[Sun Microsystems, Inc., 2001b] Sun Microsystems, Inc. (2001b). Jxta Initiative. http://www.jxta.org/.

[Sun Microsystems, Inc., 2003] Sun Microsystems, Inc. (2003). Mobile Media API 1.1.

[SyncML, 2000] SyncML (2000). Building an Industry-Wide Mobile Data Synchronization Protocol. http://www.syncml.org/technical.htm.

[Szyperski, 1999] Szyperski, C. (1999). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.

[The Distributed.net Project, 1995] The Distributed.net Project (1995). Distributed.NET. http://www.distributed.net.

[The GNOME Project, 2001] The GNOME Project (2001). The Bonobo Component Model. http://developer.gnome.org/arch/component/bonobo.html.

[The Mozilla Foundation, 2003] The Mozilla Foundation (2003). *Cross Platform Component Object Model (XPCOM)*.

[The OSGi Alliance, 1999] The OSGi Alliance (1999). The OSGi framework. http://www.osgi.org.

[The Seti At Home Project, 1997] The Seti At Home Project (1997). SETI@home, The Search for Extraterrestrial Intelligence. http://setiathome.ssl.berkeley.edu.

[The Urban Tapestry Project, 2002] The Urban Tapestry Project (2002). Urban Tapestries. http://urbantapestries.net/.

[The Xiph.org Foundation, 1998] The Xiph.org Foundation (1998). The OGG Vorbis Project. http://xiph.org/ogg/vorbis/.

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

[V. Hayes, 1996] V. Hayes, S. (1996). IEEE P802.11. Specification, Institute of Electrical and Electronics Engineers, Inc. (IEEE).

[Waldo, 1998] Waldo, J. (1998). JavaSpaces Specification 1.0. Technical report, Sun Microsystems.

[Waldo, 1999] Waldo, J. (1999). The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82.

[Weinsberg and Ben-Shaul, 2002] Weinsberg, Y. and Ben-Shaul, I. (2002). A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices. In *Proceedings of the 24th International Conference on Software Engineering*, pages 374–384.

[Wheeler, 2004] Wheeler, D. A. (2004). SLOCCount.

[Wilson and Kesselman, 2000] Wilson, S. and Kesselman, J. (2000). *"Java Platform Performance: Strategies and Tactics"*. Sun Microsystems.

[Wong et al., 1999] Wong, D., Paciorek, N., and Moore, D. (1999). Java-based Mobile Agents. *Communications of the ACM*, 42(3):92–102.

[Zachariadis and Mascolo, 2003] Zachariadis, S. and Mascolo, C. (2003). Adaptable Mobile Applications Through SATIN: Exploiting Logical Mobility in Mobile Computing Middleware. In *1st UK-UbiNet Workshop*, London, United Kingdom.

[Zachariadis et al., 2002] Zachariadis, S., Mascolo, C., and Emmerich, W. (2002). Exploiting Logical Mobility in Mobile Computing Middleware. In *Proceedings of the IEEE International Workshop on Mobile Teamwork Support, Collocated with ICDCS'02*, pages 385–386.

[Zachariadis et al., 2003] Zachariadis, S., Mascolo, C., and Emmerich, W. (2003). Adaptable Mobile Applications: Exploiting Logical Mobility in Mobile Computing. In *5th Int. Workshop on Mobile Agents for Telecommunication Applications (MATA03)*, pages 170–179, Marrakech, Morocco. Springer.

[Zachariadis et al., 2004] Zachariadis, S., Mascolo, C., and Emmerich, W. (2004). SATIN: A Component Model for Mobile Self-Organisation. In *International Symposium on Distributed Objects and Applications (DOA)*, Agia Napa, Cyprus. Springer.

[Zelkowitz and Walace, 1997] Zelkowitz, M. V. and Walace, D. (1997). Experimental Validation in Software Engineering. In *Empirical Assessment & Evaluation in Software Engineering*, Staffordshire, United Kingdom.