EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Roaming Java services for intelligent networks

Derikx, J.H.

Award date:
1999

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 20. Mar. 2025

Eindhoven University of Technology
Department of Electrical Engineering
Information- and Communication Systems Group
P.O. Box 513
5600 MB Eindhoven
The Netherlands

Ericsson Telecommunications B.V.
Business creation & Applied research Company
P.O. Box 8
5120 AA Rijen
The Netherlands

Roaming Java Services for Intelligent Networks

J.H. Derikx

Graduation Report

Rijen, August 1999

Coaches:

Prof.ir. J. de Stigter (TUE)
ir. J. van der Meer (Ericsson)

The Faculty of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of student projects and graduation reports.

Preface

The report before you was written as part of my final project of the Information Technology studies at the Eindhoven University of Technology. The project was carried out at the Business creation & Applied research Company (BAC) which is part of the R&D business line of Ericsson Telecommunication B.V. in Rijen. It was carried out from November 1998 until August 1999.

During the first part of this project I researched the possibilities of mobile code, the security involved and how to integrate the concept in an Intelligent Network. During the second part I developed and implemented a prototype system. The project gave me the opportunity to learn many new things like ERLANG , IN, Java and UML, and how to apply these to create a prototype. Working in a large company like Ericsson gave me a chance to closely observe how fast the telecom market is changing and how rapidly companies will have to adapt to these changes. I enjoyed these nine months at BAC and am convinced that it was a valuable experience.

I would like to thank my coaches, Prof.ir. J. de Stigter and ir. J. van der Meer for giving me the opportunity to do this project, their guidance and useful comments. Furthermore I would like to thank my colleagues at BAC for making me feel at home, for their advice and help. Special thanks to Erik van der Velden and Eltjo Boersma for their input and help with Erlang/RSP and Java. I thank my friends and roommates for the good times we had in the past years. Finally, I would like to thank my parents who always stood by me and gave me their full support. Thank you all!

Bertho Derikx
August 1999

Summary

When a subscriber moves abroad with his cellular phone he is no longer able to access his Intelligent Network (IN) services. There are several solutions to this problem. The CAMEL[1] standard presents one. This report describes another solution: "roaming IN services".

The IN services of a subscriber are stored in his home Service Control Function (SCF). Once he travels beyond the service area of his home network, into the service area of a visited network, his IN services are no longer accessible. The CAMEL standard solves this by establishing a signaling connection from the visited Service Switching Function (SSF) to the home SCF thus accessing the IN services. This report investigates the possibility of moving the IN service from the home SCF to the visited SCF and executing the service locally. The report roughly consists of two parts. The first part describes the research on the mobile code concept and the security involved with it. It presents an architecture for a "roaming IN services"-system. The second part describes the development and implementation of a prototype system using the Java programming language.

Three mobile code paradigms are discussed: the "remote evaluation", the "code on demand" and the "mobile agent"-paradigm. Currently the mobile code concept is not implemented in the IN architecture. The prototype shows that it can be successfully integrated in the IN architecture.

Moving code fragments to another computational environment entails a big security risk for both the host and the code. The host needs to be protected against the code and vice versa. Host security is thoroughly described in literature. This report describes the potential risks and the presented solutions. Examples of the latter are: digital signatures, security policies and an interpreter. Protecting code against a hostile host proves difficult. It remains impossible to *prevent* tampering without the use of secure hardware. All protection mechanisms are based on *detecting* possible attacks. Juristic measures may suffice to protect both host and code.

Java has several mechanisms to protect the host against the code. It uses an interpreter in combination with protection domains and a security policy. Optionally a cryptographic extension can be used. The latter is not used in the prototype. Java does not provide measures to protect the code.

The "roaming IN services"-architecture comprises four types of nodes: the SSF node, the SCF node, the Java Service Environment (JSE) node and the IN server node. The SSF node has the same functionality as in a normal telephone system. The SCF node is reduced to a relay node. The JSE node is the heart of the architecture and responsible for downloading and executing the IN service. The IN server node is a centralized server that holds all available IN services, service data and subscriber data.

The prototype implements the basic features of the architecture. It uses the Rapid Service Prototyping (RSP) software, programmed in Erlang, to simulate a telephone system. The JSE and the IN server node are programmed in Java. The Jive application provides the communication between Erlang and Java. Java has several mechanisms to transport code. They were presented with their advantages and disadvantages. Java Remote Method Invocation (RMI) was selected to

[1]CAMEL: Customized Applications for Mobile network Enhanced Logic

be used for the prototype. Java RMI provides the communication between the JSE and IN server node.

The prototype is operational. When a subscriber registers in a network, detection points are transferred from the IN server to the JSE. These detection points are installed in the SSF when a call is initiated. When the subscriber triggers an IN service, the service is downloaded from the IN server by the JSE and executed locally. The detection points and IN services are cached in the JSE while the subscriber remains registered in the network.

Java does not provide explicit means to unload classes. The use of RMI makes this process even more difficult. It remains unclear whether Java is able to reliably unload classes. RMI itself also has several disadvantages that justify research on other communication protocols. It also remains unclear how Java will perform on a heavily loaded system. Because the prototype uses the local network, it is not possible to determine how much the transport delays would increase in a real network. These subjects are left for further study.

Contents

Preface i

Summary iii

1 Introduction 1

1.1 Scope of the Project 1

1.2 Problem Definition 2

1.3 Report Outline 2

2 Backgrounds 5

2.1 An Introduction to Intelligent Networks 5

2.2 An Introduction to CAMEL 9

2.3 An Introduction to Java 11

2.4 An Introduction to Erlang and Jive 12

2.5 An Introduction to Rapid Service Prototyping 14

3 Mobile Code 15

3.1 Mobile code 15

3.2 Roaming Services in an IN 17

3.3 Java and mobile code 17

3.3.1 Custom Class Loaders 17

3.3.2 Object Serialization 18

3.3.3 Java Remote Method Invocation 18

3.3.4 Java IDL 20

3.3.5 RMI-IIOP 20

3.4 Summary 21

4 Security Aspects 23

4.1 Security and mobile code 23

4.2 Host security 23

4.3 Service security 24
4.4 Java and security 25
4.4.1 The JDK 1.2 Security Model 25
4.4.2 Java security applied to mobile code 28
4.5 Summary 28

5 Roaming IN Services **29**
5.1 Roaming Services in an IN 29
5.1.1 Tracking the subscriber 30
5.1.2 Arming Detection Points 30
5.1.3 Retrieving and Invoking IN Services 30
5.2 Service Management 32
5.3 Services and mobility 33
5.4 The Virtual Private Network Service 33
5.5 Summary 34

6 RJS System Design **35**
6.1 The RJS system 35
6.2 The Server Node 35
6.2.1 Use Case Diagram 35
6.2.2 Class Diagram 38
6.3 The Java Service Environment Node 38
6.3.1 Use Case Diagram 38
6.3.2 Class Diagram 39
6.4 The Service Control Function Node 43
6.4.1 Use Case Diagram 43
6.4.2 Class Diagram 43
6.5 IN Services 43
6.6 A Single-threaded vs. Multi-threaded Implementation 44
6.7 Summary 45

7 The RJS Prototype **47**
7.1 The RJS System 47
7.2 The Service Control Function 48
7.3 The Java Service Environment 49
7.3.1 Singletons in Java 49
7.3.2 The ScfGateway and the SSFList classes 49
7.3.3 The SSF, the SSFInterface and the CallList classes 50

7.3.4 The JobHandler, the Queue and Job classes 51
7.3.5 The Call class 51
7.3.6 The ServerGateway and ServerThread class 51
7.3.7 The Cache class 52
7.4 The IN Server 53
7.4.1 The Server Package 53
7.4.2 The Subscriber package 54
7.4.3 The Service Package 54
7.4.4 The JSE Manager package 55
7.5 Interfaces 55
7.5.1 The SCF - JSE interface 55
7.5.2 The IN Server - JSE interface 56
7.6 IN Services 58
7.7 Common Classes 59
7.8 Limitations of the prototype 60
7.8.1 The IN Server 60
7.8.2 The JSE 60
7.9 Unloading classes 61
7.10 Service Interworking 61
7.11 Summary 62

8 Conclusions **63**

8.1 Problem Definition 63
8.2 Architecture of the RJS system 63
8.3 Security Aspects 64
8.4 The RJS prototype 64
8.5 A monopolistic environment 65
8.6 Recommendations for future work 66

A The Basic Call State Model **67**

A.1 The Originating Basic Call State Model 67
A.2 The Terminating Basic Call State Model 68
A.3 RSP Originating Basic Call Model 70
A.4 RSP Terminating Basic Call Model 71

B Information Flows **73**

B.1 Information Flows in the RSP 73
B.2 Information Flows in the roaming IN services architecture 74

B.2.1 IFs between the SSF and the SCF 74
B.2.2 IFs between the SCF and the SE 74
B.2.3 IFs between the SE and the IN server 74
B.2.4 IFs between the SE and the local cache 75
B.3 Standardized Information Flows 75

C Activity Diagrams **77**
C.1 The Server Node 77
C.2 The Java Service Environment Node 80

D Sequence Diagrams **83**

E User Manual **87**
E.1 Compiling the IN Server and the JSE 87
E.1.1 The se.ericcson.etm.rjs.common package 87
E.1.2 The se.ericsson.etm.rjs.cache package 88
E.1.3 The IN Server 88
E.1.4 The JSE 88
E.2 The IN Server GUI 89
E.2.1 The JSE GUI 91
E.3 The RSP-software 92
E.4 Using prototype services 92

Acronyms **95**

Bibliography **97**

Chapter 1

Introduction

This first chapter introduces the project and its problem definition. It shortly describes its context and gives an outline of the forthcoming chapters.

1.1 Scope of the Project

In the last decade communication facilities and their availability to the general public have grown fast. People have got used to the idea that they can reach almost anybody, almost anywhere. Intelligent Networks, the Internet, cellular phones, they all contributed to this idea.

Intelligent Networks (IN) boomed in the mid-eighties after the US telecommunications monopoly was broken. The idea of the IN architecture concept is to facilitate the introduction of new services in the network. The ability to create new services must be independent of the service and of the network implementation. Ericsson has integrated the IN-concept in their exchanges.

The use of cellular phones is rapidly increasing. The liberalization of the telecommunication market has led to a decrease of entree fees and rates. More can afford a mobile phone. In the Netherlands one third of the population is said to have a cellular phone[1]. Finland is the world's first country in which the number of cellular subscriptions has exceeded the number of fixed subscriptions[2].

Distances become shorter, people travel and boundaries fade. But when a subscriber moves beyond the service area of his home network's SCP, he is no longer able to access his IN services. Three solutions have been presented for this problem:

- Select a limited set of IN services, standardize them and make them available to roaming subscribers. Of course this is not a real solution; the fact that the set in limited and the services standardized is undesirable.
- Establish a signaling connection from the visited network to the SCP in the home network and access the IN service. Refer to figure 1.1. Two variants exist. Establish the connection between the two SCP nodes or between the visited SSP node and the home SCP node. The CAMEL standard defined by ETSI implements the second variant.
- Transport the IN service from the home network to the visited network. Refer to figure 1.2.

This third solutions will be the subject of this project: *roaming IN services*.

Hype or not, Java has become a widely accepted programming language. Java™ is an object-oriented, platform independent programming language developed by Sun Microsystems. Will it be possible to implement these roaming IN services and their execution engine using Java?

[1]Source: De Telegraaf - 07/07/1999
[2]Source: NRC Handelsblad - 23/12/1998

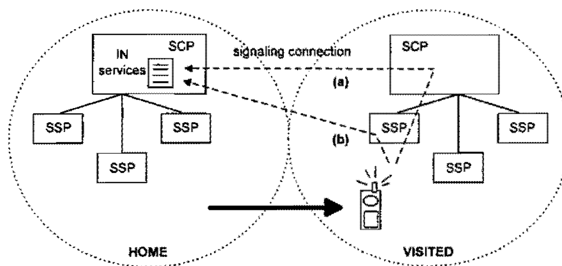

Figure 1.1: Access by means of a signaling connection, two variants

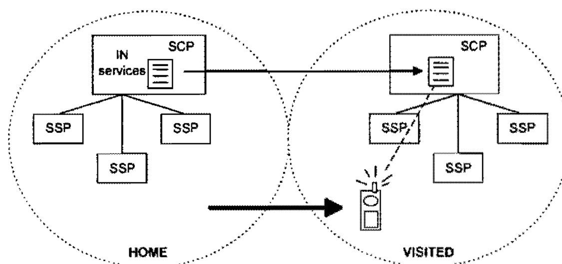

Figure 1.2: The IN service is moved to the visited SCF

1.2 Problem Definition

What is known about the mobile code concept and can it be applied to the IN architecture? Has it already been used and what are its advantages and disadvantages?

Code-fragments and data will have to be transferred to another execution environment. This raises major security issues. How to protect the host, the IN service and the data? And can the privacy of the subscriber be guaranteed?

Java is relatively new and still under development. Is this programming language capable of satisfying the necessary security requirements and how does it implement code-mobility?

This project comprises of two phases. The first phase will try to find answers to the questions raised above. In the second phase a prototype will be designed and implemented. The prototype will implement a to be developed "roaming Java IN services" architecture.

1.3 Report Outline

This section gives a brief survey of the remaining chapters of the report.

Chapter 2 gives an introduction to some basic concepts that will be used in the report. The next topics will be discussed:

- Intelligent Networks
- CAMEL
- Java
- Rapid Service Prototyping
- Erlang and Jive

Readers already familiar with a concept can skip the appropriate section.

Chapter 3 covers the mobile code concept. It presents a taxonomy of mobile code paradigms. What are its advantages and disadvantages? And how does Java implement the mobile code concept?

Chapter 4 describes the security aspects of mobile code. When code is moved from one machine to another, both the visited host and the traveling code and data need to be protected. Does Java implement these security measures?

Chapter 5 presents the requirements and an architecture for roaming IN services. It presents the architecture and its nodes and it describes their actions. It further shows which type of IN services are most likely to be used.

Chapter 6 covers the design of the roaming Java services prototype. It describes the three nodes of the architecture: the SCF node, the JSE node and the Server node. Several types of UML diagrams are used to clarify the description.

Chapter 7 covers the implementation of the prototype. It uses the architecture presented in the previous chapter. It describes the classes that implement the three nodes and the interfaces between them.

Chapter 8 lines up the conclusions from the previous chapters and gives some recommendations for future work.

Chapter 2

Backgrounds

The chapter Backgrounds introduces to the reader the concepts and technologies used during the project. It starts with an introduction to Intelligent Networks followed by a description of Sun's programming language Java™ . The last two sections deal with the programming language Erlang and the Rapid Service Prototyping software used for prototyping.

2.1 An Introduction to Intelligent Networks

The concept of intelligent networks (IN) boomed in the mid-eighties after the US telecommunications monopoly was broken.[1] The goal of the intelligent network concept was and is to reduce the time-to-market of new services. This is accomplished by the introduction of reusable building blocks and a network-implementation-independent provisioning of services. The network-intelligence has been separated from the switching function.

The International Telecommunication Union (ITU) has defined a standardized model for intelligent networks in the Intelligent Networks Recommendations (Q-series): the IN Conceptual Model (INCM) [33]. Capability Set 1 (CS-1) is the first standardized stage of the Intelligent Network architecture [34].

The INCM model, depicted in figure 2.1, represents the IN at four different abstraction levels called planes:

- The service plane (Q.1202)
- The global functional plane (GFP) (Q.1203)
- The distributed functional plane (DFP) (Q.1204)
- The physical plane (Q.1205)

The *service plane* models the IN as a set of services. A service is characterized by its service features (SFs). The service plane gives no information whatsoever about the implementation of services in the network. An example of a service is the "Premium Rate" service. Callers pay an extra fee which will be forwarded by the network operator to the called party. Some of this service's features are "Premium Charging", "One Number", "Call queueing" and "Call logging".

The next step is to model the IN as one big entity. The *global functional plane* shows that service features are put together from service independent building blocks (SIBs). Each block performs a specific function and can be reused for multiple services. Service logic glues the building blocks

[1] In January 1984 Bell System, forced by the U.S. government, split up in one long-distance carrier and seven regional telephone holding companies.

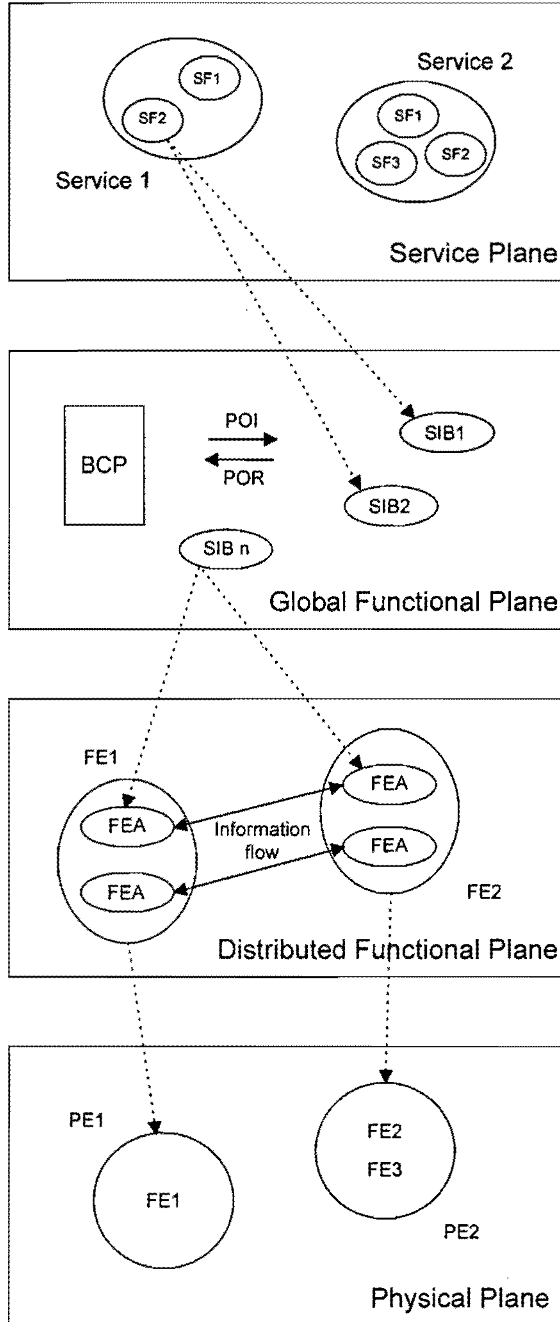

Figure 2.1: The IN conceptual model

together. There is one special building block that handles the Basic Call Process (BCP). This block is connected to the chains of SIBs by points of initiation (POI) and points of return (POR). Refer to figure 2.2. The "Limit"-block is an example of a building block. It makes sure that the number of calls that pass through does not exceed a parameter specified by the operator. The number of calls per minute should, for instance, not exceed x.

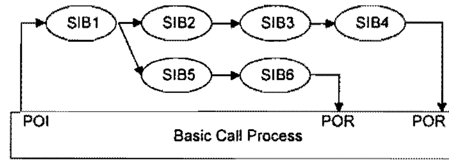

Figure 2.2: A service is created by gluing SIBs together

In the *distributed functional plane*, the IN is divided in unique groups of functions called functional entities (FEs). Each functional entity performs a number of functional entity actions (FEAs). The functional entities communicate through information flows (IFs). Figure 2.3 shows a typical functional model diagram containing the following entities:

- Call control agent function (CCAF) – This function handles the subscriber access to the network. It is found in local exchanges.
- Call control function (CCF) – The CCF handles the call processing and control. It also provides a trigger mechanism to access IN functions.
- Service switching function (SSF) – This function handles the communication between a CCF and the SCF.
- Service control function (SCF) – The SCF function is the heart of the IN network. It contains the centralized intelligence of the network.
- Service data function (SDF) – The SDF contains subscriber and network data which can be accessed by the SCF.
- Specialized resource function (SRF) – The SRF provides specialized resources needed by the IN services. It is used to send and receive data to and from subscribers.
- Service creation environment function (SCEF) – This function handles defining, developing and testing of IN services and their transportation to the SMF.
- Service management agent function (SMAF) – This function handles the operator/provider access to the network.
- Service management function (SMF) – The SMF manages the distribution, the control and the maintenance of IN services.

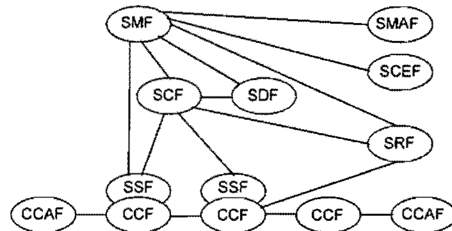

Figure 2.3: The distributed functional plane model

In the *physical plane* the functional entities are mapped to physical entities (PE). The mapping is shown in table 2.1.

Physical entity	Functional entity
Service switching point (SSP)	SSF and CCF
Service control point (SCP)	SCF
Service data point (SDP)	SDF
Intelligent peripheral (IP)	SRF
Adjunct (AD)	SCF and SDF
Service node (SN)	SSF, CCF, SCF, SDF and SRF
Service switching and control point (SSCP)	SSF, CCF, SCF, SDF
Service management point (SMP)	SMF
Service creation environment point (SCEP)	SCEF
Service management access point (SMAP)	SMAF

Table 2.1: Mapping functional entities to physical entities

In the DFP, the Basic Call State Model is introduced. The *Basic Call State Model* (BCSM) is a high level finite state machine description of call processing for basic call control. A *basic call* is a two party non-IN call. The components of the BCSM are shown in figure 2.4. A state of the FSM is called a *Point In Call* (PIC). A *Detection Point* (DP) is a point in the basic call processing at which an event may be reported to an SCF and transfer of processing control may occur. POIs that are introduced in the GFP, are mapped to DPs in the DFP. Refer to Appendix A for the Originating and Terminating BCSM.

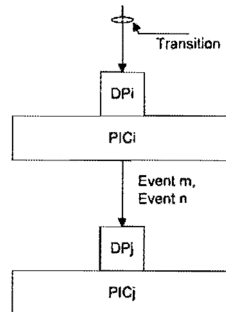

Figure 2.4: BCSM components

Detection Points are points in the BCP at which an IN service *may* be invoked. DPs are characterized by four attributes:

- Arming/disarming mechanism – A DP can be statically armed or dynamically armed. DPs are armed statically through the SMF service feature provisioning. When armed statically, the DP is referred to as a *Trigger Detection Point* (TDP). A TDP can only be disarmed by the SMF. The SCP is able to dynamically arm a DP. A dynamically armed DP is called an *Event Detection Point* (EDP). An EDP is disarmed when triggered.
- Criteria – When an armed DP is encountered, the SCF is notified only when the additional DP Criteria are met.
- Relationship – When an armed DP is encountered and the criteria are met, the SSF/CCF establishes a relationship with the SCF. Two types exist – In a control relationship the SCF is able to influence call processing via the relationship and in a monitor relationship the SCF in *not* able to influence call processing via the relationship.
- Call processing suspension – Call processing may or may not suspended to allow the SCF to influence the subsequent call processing. This only applies for a control relationship.

In the past, the intelligence of the network resided in the individual local exchanges. Introducing a new service was time-consuming and difficult; it took two to five years. The goal of the intelligent network concept is to reduce this to less than six months. therefore the service logic was moved to a centralized location: the service control point.

In today's mobile world, subscribers roam. Unfortunately, they lose their IN services when they move beyond the service area of their home SCP. One possible solution to is to set up a signaling connection to the home SCP. For the GSM network for instance, the ETSI has defined a standard called Customized Applications for Mobile network Enhanced Logic (CAMEL). Refer to section 2.2 for more information on CAMEL. A second solution is to move the subscriber's service profile to the visited network's SCP. The service profile should contain the subscriber's customized services and the related detection points.

2.2 An Introduction to CAMEL

In the past, when subscribers traveled beyond the boundaries of their home GSM network, most of their services were no longer accessible. Only a few standardized GSM services could be used. Nowadays operators try to differentiate themselves by offering their clients value-added customized services. ETSI has developed a standard, CAMEL[2] [2], that gives operators the opportunity to offer *Operator-Specific Services* (OSS) outside their own networks. CAMEL can be seen as the integration of IN into the GSM architecture.

Figure 2.5 depicts the functional entities in the CAMEL (phase 1) supporting GSM Network. When a roaming subscriber accesses one of his IN services, a signaling connection is established between the gsmSSF and the gsmSCF. The IN service is executed in the gsmSCF and the result is returned to the gsmSSF.

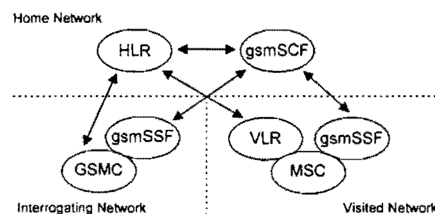

Figure 2.5: The Functional Architecture for support of CAMEL

The *Home Location Register* (HLR) stores all the administrative information of each subscriber registered in GSM network, the current location of the mobile equipment and the Originating and Terminating CAMEL Subscription Information (O-CSI and T-CSI).
The *Visitor Location Register* (VLR) stores selected information from the HLR, among which the O-CSI, for each subscriber currently in its service area.
The *Mobile services Switching Center* (MSC) acts like a switching node but additionally provides all the functionality needed to handle a mobile subscriber.
The *Gateway MSC* (GMSC) is an MSC that also contains a table that links the subscriber's phone number to his corresponding HLR.
The *gsmSCF* contains the CAMEL service logic that implements the OSS.
The *gsmSSF* interfaces the MSC/GMSC to the gsmSCF.

For every subscriber that uses CAMEL services the HLR contains both originating and terminating CAMEL subscription information. The O-CSI applies to those CAMEL services available when

[2]This section describes CAMEL phase 1 only.

the subscriber makes the call: a mobile originated call. The T-CSI is used when the subscriber is being called: a mobile terminated call. The CSI contains four data fields:

- The *gsmSCF address* is used to access the gsmSCF of a particular subscriber.
- The *Service Key* identifies the service logic to the gsmSCF.
- The *Default Call Handling* indicates whether the call should be released or continued after a communications error between the gsmSCF and gsmSSF.
- The *Trigger Detection Point list* indicates on which detection points triggering shall take place, i.e. after which events in the call process the MSC notifies the gsmSCF.

When a CAMEL subscriber switches his mobile phone on in a new location area or moves to a new location area or other network, it registers with the network to update its current location. The visited MSC/VLR notifies the home HLR which responds by sending the previously mentioned subset of subscriber information, including the O-CSI, to the visited MSC/VLR. The old MSC/VLR is notified by the HLR to cancel the old registration. Refer to figure 2.6.

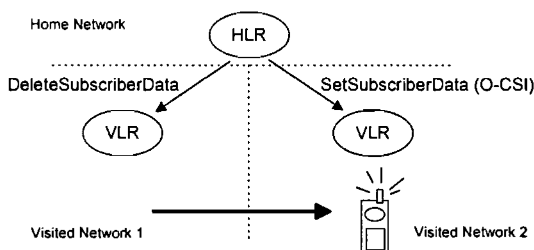

Figure 2.6: Location Update

The Trigger Detection Point list is part of the CSI. When the CSI is sent to the MSC, the listed DPs are armed for that subscriber. When the BCSM reaches an armed DP and the criteria are met, control is transferred to the gsmSCF. For the O-CSI only DP2, and for the T-CSI only DP12 is used. Note that CAMEL also supports the following EDPs: DP7, DP9, DP15, DP17. These can be installed by the SCP. (Refer to Appendix A for an explanation of the DPs.)

CAMEL differentiates between a mobile originated call, a mobile terminating call and call forwarding. Refer to figure 2.7, figure 2.8 and figure 2.9.

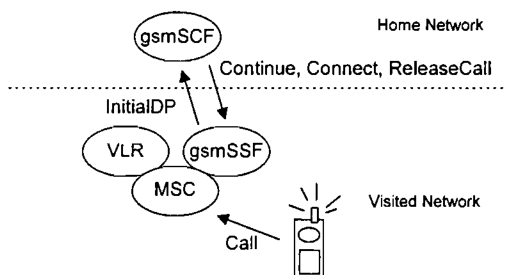

Figure 2.7: Mobile originated call setup

Mobile Originated Call:
When, during call setup, the O-CSI is found in the VLR for the A-subscriber, the visited SSF will send the `InitialDetectionPoint` message to the gsmSCF and suspend the call processing if a detection point is triggered. The message contains the `Service Key` which unambiguously identifies the requested CAMEL service to the gsmSCF. The gsmSCF processes the appropriate service logic and, depending on the outcome, returns the `Connect`, `Continue` or `ReleaseCall` message. The gsmSSF then takes the indicated course of action.

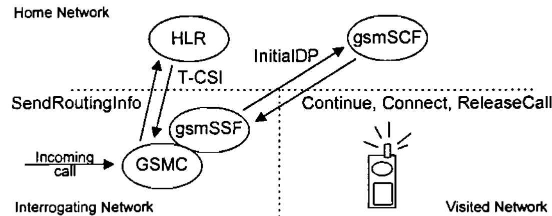

Figure 2.8: Mobile terminating call setup

Mobile Terminating Call:
When an incoming call to the B-subscriber reaches the GMSC, the GMSC sends a `SendRoutingInfo` message to the HLR. The HLR returns the T-CSI. The GMSC acts according to the T-CSI of the B-subscriber and, if a detection point is triggered, processing is suspended and an `InitialDP` message is sent to the gsmSCF.

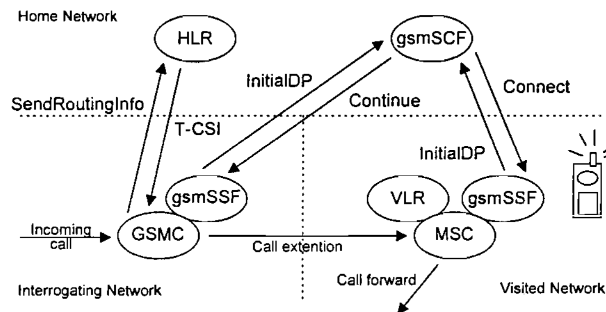

Figure 2.9: Call forwarding at the GMSC

Call Forwarding
Call forwarding is a combination of the two previously mentioned situations and can be triggered at the GMSC or at the visited MSC depending on whether the `Connect` message has the `O-CSI Applicable` flag set. Figure 2.9 shows the situation where the call is forwarded at the visited MSC. After optional processing by the gsmSCF, based on the T-CSI of the B-subscriber, the GMSC extends the call to the MSC servicing the B-subscriber. There the call is forwarded by the gsmSCF, based on the O-CSI of the B-subscriber, to the C-subscriber.

2.3 An Introduction to Java

The Java™ programming language and environment was designed by Sun Microsystems and was first released publicly in November 1995. In December 1998 Sun released version 1.2 of their Java Development Kit (JDK). Most people who have heard of Java associate it with the applets used on the World Wide Web to brighten up web pages. However, Java is much more.

Sun describes Java as a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi threaded and dynamic language [32]. Let us have a closer look at some of these buzzwords.

Java being designed closely to C/C++ will look very familiar to most programmers. However, some of the "more complex" features of C++ can not be found in Java. Java does for instance not implement pointer arithmetic or multiple inheritance. Sun's developers added an automatic

garbage collector which relieves the programmer of all memory management responsibilities. The collector is a low priority background process that gathers unused memory for reuse in idle moments.

Object-oriented languages facilitate code reuse by means of inheritance. An object is a programming model. It has a state and a behavior. The behavior can change the state of the object. A car for instance can be modeled by an object. The model's state could be the speed at which it is traveling. The behavior could be "to accelerate" or "to brake". For modeling a *blue* car the model of the car can be reused; the blue car model inherits from the car model. In Java a model is called a class, the state of the model is represented by the class' instance variables and the behavior is defined by its methods. Java implements the single inheritance model which means that a sub-class may only inherit from *one* super-class.

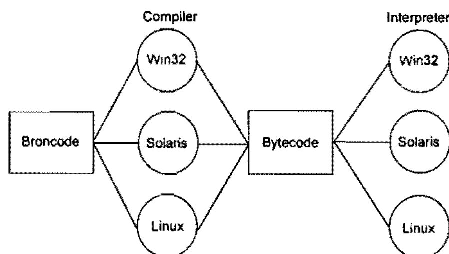

Figure 2.10: Java bytecode

The Java compiler generates architecture independent code called byte code. The byte code can be executed by an architecture-dependent Java interpreter. Given an interpreter for your particular platform, you will be able to run all Java programs. Refer to figure 2.10.

Since Java is intended to be used in a distributed environment, security is one of its key attributes. Java security spans more than one level. Java is said to be secure on the language-level, the Java Virtual Machine-level and on the API-level [20]. For an outline of Java security refer to section 4.4.

The JDK contains the Java Development tools, the Java Runtime environment (JRE) and additional libraries. The tool-set is used to create and build applications and contains tools like the compiler and the debugger. The JRE consists of the Java™ Virtual Machine (JVM) and the Java Application Programming Interface (API) [18]. The Java Virtual Machine is an abstract computing machine to be implemented on top of existing processors. Like all computers it has an instruction set, registers and a stack. The Java API forms a standard interface to applications regardless of the underlying operating system. Refer to figure 2.11 for an overview of the Java platform.

2.4 An Introduction to Erlang and Jive

ERLANG is a declarative language developed at the Ericsson and Ellemtel Computer Science Laboratories. The development of ERLANG started as an investigation of whether modern declarative programming paradigms could be used for programming large industrial telecommunications switching systems [1]. Erlang is now marketed by Erlang Systems.

Erlang implements a process-based model of concurrency; multiple processes can run simultaneously. Processes, on the same system or on distributed nodes, communicate through asynchronous message passing. A process is identified by its process-id (Pid). Refer to figure 2.12. Like Java, Erlang is an interpreted language and has a garbage collector. An important feature of Erlang is that it supports hot code update: Old code can be replaced in a running system and old and

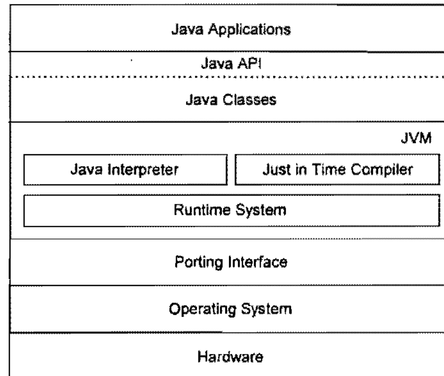

Figure 2.11: The Java platform

new versions of the code may co-exist. Various error detecting mechanisms can be used to build a fault-tolerant system.

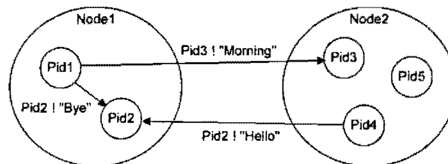

Figure 2.12: Erlang message passing

The Erlang Development Environment consists of the ERLANG runtime system and development tools. The Erlang runtime system consists of

- The Erlang virtual machine which runs on top of the host operating system.
- The kernel which provides low-level services for the applications.
- The standard libraries.

The ERLANG package comes with an application called JIVE . Jive allows Erlang processes to communicate with Java applications and applets. Refer to figure 2.13.

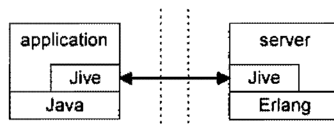

Figure 2.13: The Jive Interface

The communication is socket based and completely hidden from the programmer. The Java API contains a number of wrapper classes for each of the Erlang variable types. The `EVar` is an abstract super class of all other Erlang wrapper classes. The `EInteger`, `EFloat`, `EString`, `EAtom`, `EBinary`, `Elist` and `ETuple` correspond to similarly named Erlang variables. The `EProcess` is the wrapper class of the Erlang process-id (Pid).
A Java application can

- Spawn new Erlang processes.

- Send messages to and from Erlang processes.
- Do an "apply" on Erlang functions.

Erlang processes can send messages to Java objects.

For more information about ERLANG and Jive refer to [1] and [24].

2.5 An Introduction to Rapid Service Prototyping

RSP, which stands for Rapid Service Prototyping, is a software package developed by Ericsson to develop and demonstrate prototype IN services. The RSP software is written in Erlang. It simulates all the main parts in an IN network: the SCF, SSF, SDF, SMF, SRF and various types of phones. The RSP software is also capable of connecting to a real SSF and phones.

Figure 2.14 shows the structure of the software that is relevant to this project. The SCF node shows three modules. The `lim_pad10` module handles all communications with the SSF node. This means both call and non-call related messages. The call-related messages are relayed to the `scf10` module and the non-call related messages to the `scf_man10` module. The `scf10` module also takes care of starting and stopping the other modules. On the SSF side, the `lim_pad10` communicates with the `resourse50` and `ssf50` modules. The resource manager handles all requests from the SCF not related to a call session. The `ssf50` sets up a call session with the SCF and checks for breakpoints on every state change in either halves of the call state model.

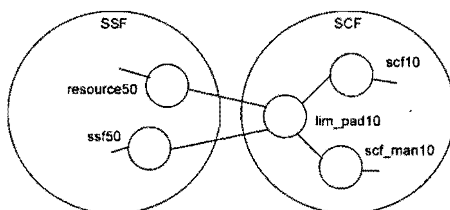

Figure 2.14: The RSP software package

Appendix A shows the call state model used by the RSP software. The messages between the SCF and SSF are described in appendix B.1. What is referred to as a "trigger detection point" in the ETSI standard, is called a "static breakpoint" in the RSP context. An "event detection point" is called a "dynamic breakpoint".

Refer to [10] for more information on the SSF node.

Chapter 3

Mobile Code

This chapter discusses mobile code. What is mobile code and what types of mobility exist? Is there such thing as mobile code in today's IN? In addition, it discusses in what way Java supports code mobility.

3.1 Mobile code

In [38] Vigna gives a taxonomy of mobile code technologies and paradigms. Figure 3.1 shows the model that is used for mobile code systems. The core operating system is located immediately above the hardware. Support for communications between two systems is provided by the network operating system. The computational environment implements the facilities for executing units to move from one host to another.

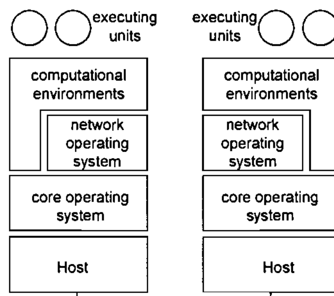

Figure 3.1: The mobile code system

Two forms of mobility are distinguished: strong and weak mobility. In technologies that implement strong mobility both the code and the execution state of an executing unit can be moved to another computational environment. Weak mobility on the other hand only provides for the code to be transferred.

In a distributed computing environment Vigna characterizes four paradigms. One is the traditional client-server paradigm. The other three: the remote evaluation, the code on demand and the mobile agent paradigm, implement some form of mobile code. They differ with respect to the location of the code before and after the service execution.

The paradigm most frequently used today is the *client-server paradigm.* The server has a number of predefined services available for its clients. A client sends a command to the server who executes

the appropriate code and sends the response back to the client. Both the code and the resources reside at the server-side. Figure 3.2 shows this paradigm.

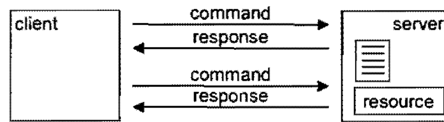

Figure 3.2: The client-server paradigm

The second paradigm is the *remote evaluation paradigm*. The client sends the code to be executed to the server which provides the resources. The server sends the results back to the client. In this setup the client initiates the service.

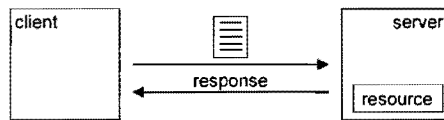

Figure 3.3: The remote evaluation paradigm

In the *code on demand paradigm* it is the server who initiates the service. The resources are available, but not the code. The server sends a request for the code to the client and executes the code on receipt.

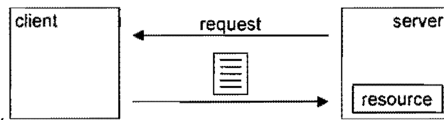

Figure 3.4: The code on demand paradigm

In the *mobile agent paradigm* the client has some of the required resources available but not all. It starts the execution of the code locally and at some point in the program moves the code and the execution state to the server. The server resumes where the client left off. When finished, the response is sent to the client.

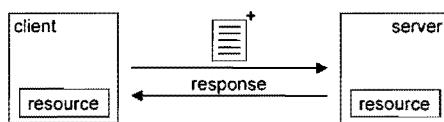

Figure 3.5: The mobile agent paradigm

Literature [38, 5] mentions several benefits of the mobile code approach:

- Service customization – In the client-server paradigm it is only possible to select one of the at the server-side available services. Mobile code gives the client the possibility to customize the services.
- Automation of installation procedures. The mobile code can implement an installation procedure for a software package to be installed.

- Improved fault tolerance – Since the client-side has the source code available it can, when an error is suspected, check the returned result of the remote computation.
- Data management flexibility and protocol encapsulation – When exchanging and processing data, the data access protocol is usually predefined and fixed. In a mobile code setup the protocol could be retrieved along with the data, making the data-structure and protocol more flexible.
- Asynchronous communication – When network transmission latency is high, real-time interaction with the server can be time consuming.

Mobile code systems also entail disadvantages [5]. For instance:

- The need for highly secure execution environments and source code.
- Performance and functional limitations as a result of the security measures.
- Dependence on the willingness of other parties to switch to the mobile approach.

The need for increased security seems to be a major drawback. Authorities will not allow foreign code running on their systems without a guarantee that it is unable to do any harm.

3.2 Roaming Services in an IN

In today's INs, services stay where they are. When a subscriber moves to another network most of his services can no longer be used. In GSM networks a few basic services have been standardized so that they can be accessed across network boundaries. Operator specific services can not be used. The ETSI has defined the CAMEL standard to overcome this. Refer to section 2.2 for a description of CAMEL. However, CAMEL does not move the IN services. It sets up a signaling connection from the visited SSP to the home SCP.

3.3 Java and mobile code

Most people with a connection to the Internet know what Java can do for a web-page; graphics, user interfaces, sounds and so on. These "curiosities" are made possible by so-called applets. An applet is a Java program designed to be included in a HTML page and executed in a Java-enabled browser. A browser examines a retrieved HTML page and discovers that it includes a tag to a specific applet. The applet's class-file, containing the bytecode that implements the applet, is downloaded to the local host. The browser creates an instance of the class and executes it. This is a good example of the "code on demand" paradigm.

Java supports several mechanisms to transport code to and execute it on the local host.

3.3.1 Custom Class Loaders

A Class loader is used to introduce new classes in the running Java environment. New classes are loaded when they are referenced in a class that is already running. So only a running class can introduce a new class. The *primordial class loader* is embedded in the virtual machine and is the default implementation of a class loader. Normally the Java virtual machine loads classes from the local file system. Some classes however may originate from other sources such as a network. therefore Java allows users to implement a custom class loader. An abstract class `java.lang.ClassLoader` is provided as a starting point. The `java.lang` package also provides two extended class loaders: the `SecureClassLoader` and the `URLClassLoader`. The latter is used to load classes and resources from a search path of URL-locations. It is the class loader used by web browsers.

A custom class loader should implement the following procedure [21]:

- Verify the requested class name. A custom class loader should refuse to load any class that claims to be part of the Java API.
- Check if the requested class has already been loaded. If true, do not load it again but return a reference.
- Check if the primordial class loader can resolve the class name and load the class.
- Try to load the requested class from this class loader's repository. The user can retrieve the byte code from any resource he sees fit.
- Define the class for the virtual machine. The byte code's legitimacy is verified.
- Resolve the requested class. The classes referenced by the requested class are loaded and verified.
- Return the requested class to the caller.

The steps for using a custom class loader are [4]:

- Create an instance of the custom class loader.
- Call its `loadClass` method to find and retrieve the requested class file.
- Create a new instance of the retrieved class and cast the returned `Object` reference into the appropriate interface.
- Access the newly loaded class through the interface.

Note that a class loader retrieves the class file and not an object. Objects are created in the local environment after the class has been resolved. Class loaders thus only support *weak* mobility.

3.3.2 Object Serialization

Instead of transporting class files it is also possible to transport an object. therefore, the object has to serialized first. Serialization is the ability to store an object and its state so that is can be reconstructed later. Serialization is an implementation of the *strong* mobility concept.

Two streams in the `java.io` package – `ObjectInputStream` and `ObjectOutputStream` – are special in that they can read and write objects. Java uses serialization for its remote method invocation, the topic of the next subsection, and *lightweight persistence*[1]. In order for a program to correctly use a de-serialized object, the program needs the object's byte code files. The previously mentioned streams are *processing streams* meaning that they must be constructed on other streams. Other streams being for instance a file stream or a socket stream. Thus, object serialization is the basis for transporting objects over a network.

3.3.3 Java Remote Method Invocation

Java Remote Method Invocation (RMI) [30, 29] is Java's implementation of the "client-server paradigm". RMI implements a distributed object system allowing a client to invoke methods on remote objects available at a server. A distributed object system has to:

- Locate remote objects – A remote object is accessed through a reference. References can be obtained in two ways: They can be registered and retrieved from the registry or an application can pass and return them.
- Communicate with other objects – RMI handles all the communication between remote objects.
- Load class bytecodes for objects that are passed as parameters or return values – When a class is required but not locally available, RMI provides the mechanism for retrieving the class file.

[1]Persistence: The archival of an object for later use by the same program. Lightweight referring to the fact that explicit calls have to be made to store and retrieve the object.

A remote object is accessed through a reference. In Java RMI this reference is called a stub. At the client-side the stub is dynamically linked in the client application. A stub is an instance of the stub class available in the stub class file.

As mentioned in the previous section, RMI uses object serialization to transport objects. This applies to objects that are passed as parameters, but also to stubs. (Note that a stub is also an object.) In order for a client to use a de-serialized object, it needs the object's class files. The server therefore makes these class files available on its web server.

Figure 3.6 shows the setup of a typical distributed application. The server first makes its stub and the class files the stub relies on available on its web server (1). It then registers with the `rmiregistry` (2). The registry gets a stub (the instance) and downloads the appropriate class files from the web server (3). The client looks up the server in the registry and receives the stub (4). The client also needs the object's class files and downloads them from the web server (5). The client can now access the remote objects on the server. These five steps are known as dynamic stub downloading.

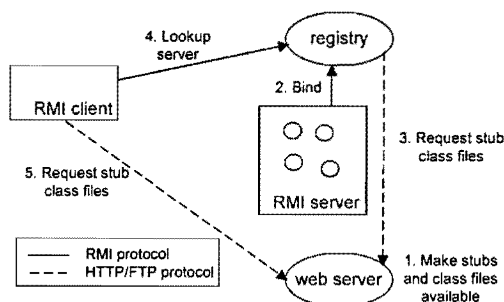

Figure 3.6: Accessing the server

Java RMI allows the client and the server to pass and return full objects as arguments. The objects are passed by value, meaning that the actual object is copied and not just a reference to that object. Note that not just the data in the object is copied but also its behaviors i.e. its methods. When an object is passed, the receiving party needs the object's class files to de-serialize the object. The files will be downloaded from the other party's web server. Refer to figure 3.7.

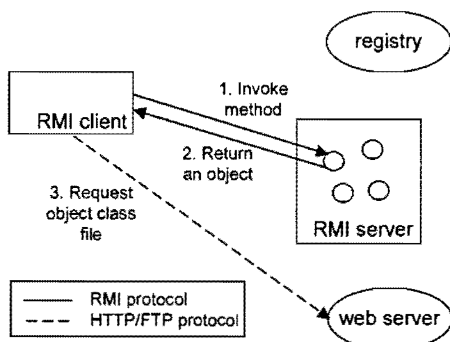

Figure 3.7: Transferring objects as parameters

The fact that Java RMI has to use a web server for its class file distribution is a major disadvantage. The location of the class files is represented by an URL. Java RMI only supports the File-URL, the Http-URL and the anonymous-Ftp-URL. From these three, only the two last ones are usable in a heterogeneous network. Both the Http and the Ftp protocol do not support user authentication,

meaning that everybody who has access to the server can download every class file.

3.3.4 Java IDL

Java IDL implements the *Common Object Request Broker Architecture* (CORBA) as designed by the Object Management Group (OMG). CORBA is an architecture for an open software bus a.k.a. an *Object Request Broker* (ORB). Distributed objects use the bus to communicate. The CORBA object bus defines the form of the objects and how they interoperate. The CORBA architecture does not include any implementations, only interface specifications. Specifications are written in the *Interface Definition Language* (IDL). It does not matter in what programming language an object is written. As long as the language has a mapping to IDL, the object will be able to interoperate with other objects on the bus. ORBs are able to communicate using the Internet Inter-ORB Protocol (IIOP). Refer to figure 3.8.

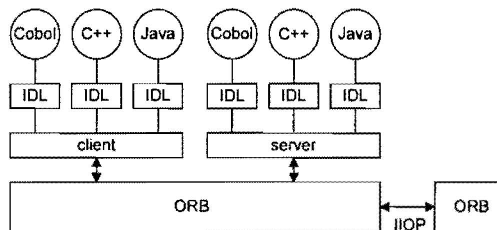

Figure 3.8: The Object Request Broker

Since these distributed objects communicate, the question arises whether they are able to pass objects by value, like RMI, an thus move code. The current version of CORBA does not include protocols for passing objects by value. Version 3.0 of the CORBA specification, which will be released mid-1999 is said to include this feature.

3.3.5 RMI-IIOP

RMI-IIOP was jointly developed by Sun and IBM. The code is currently in a beta-stage. The architecture of RMI-IIOP is similar to that of Java RMI. The main difference is that it uses the Internet Inter-ORB Protocol (IIOP) for its communications between server and client. Like in Java RMI, objects are passed by value. It does however not implement the Common Object Request Broker Architecture (CORBA). Refer to figure 3.9

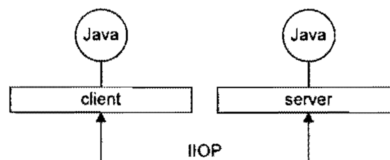

Figure 3.9: RMI-IIOP

The architecture of RMI-IIOP is similar to that of Java RMI. therefore the same security problem as with Java RMI arises. Refer to section 3.3.3.

3.4 Summary

Vigna distinguishes three mobile code paradigms: the remote evaluation paradigm, the code on demand paradigm and the mobile agent paradigm. The first two implement weak mobility and the latter strong mobility. When using one of these mobile code concepts a remote service will gain flexibility. The fact that the client may customize the service is a big advantage. The main disadvantage are the strict security measures necessary.

Java supports the concept of code mobility on several levels. Class loaders give the programmer the ability to retrieve bytecode from a remote location and dynamically link it in the application. An instance of a class can be transported using object serialization. Java RMI combines these two. Java IDL implements IMG's CORBA architecture.

A mobile code system could be implemented using custom class loaders. For IN services however, not only the code, but also a data part has to be downloaded. This is not included in the class loader concept.

CORBA 2.0 does not have the ability to transport objects by value. Implementations of CORBA 3.0 have not yet been released. The ability to transport objects by value is required by a system supporting strong mobility. This makes Java IDL unsuitable for that purpose.

Java RMI makes use of its own class loader and uses object serialization to transport objects by value. Java RMI is suitable for implementing a system supporting strong mobility.

RMI-IIOP resembles Java RMI. RMI-IIOP uses the Internet Inter-ORB Protocol for its client-server communications while Java RMI uses a Sun proprietary protocol. RMI-IIOP is currently in beta stage[2].

Java RMI and RMI-IIOP have the same disadvantage: they use a web server to publish their class files. Everybody who has access to the server can download any class file.

[2] RMI-IIOP was officially released in June 1999

Chapter 4

Security Aspects

When a host has to execute foreign code, security becomes a main issue. Both host and code have to be protected. This chapter is devoted to security aspects of mobile code and examines which of these features are supported by Java.

4.1 Security and mobile code

When executing code obtained from a remote host and belonging to a different authority, certain security risks exist that have to be dealt with. First off all the local host has to be protected. The question is whether the foreign code will do any harm, intentionally or unintentionally? It is not a risk one is prepared to take. But the traveling code also needs protection. Its source may be subject to copyrights or it main contain private information. The local host or a program running on it may try to obtain this information or corrupt the code. The next sections deal with these problems and review how Sun has solved these in their Java programming language.

A malicious program may also attack the mobile code. This problem is however not described separately in this chapter. The solution can be found in the solutions to the two former problems: host security and service security. The host should prevent any program from accessing resources it is not entitled to access, including the memory space of other programs. Roaming code should protect itself against unauthorized use, also by other programs.

4.2 Host security

When a host receives a request from a client to execute the supplied code, it will first want to make sure that the code will not misuse its resources.

Feigenbaum and Lee [11] enumerate the following threads to a host:

- Damage to the host's file system – Files or the file system itself could be damaged or changed by a malicious executing unit.
- Excessive or non-authorized use of the host's resources – By excessive or non-authorized use of for instance the CPU or the host's memory, the correct execution of other processes could be endangered.
- Leakage of private information belonging to the host's users – A malicious executing unit could gain access to private information.
- Access to a private network the host is connected to – Once access to a private network has been gained, the amount of damage that can be done by the executing unit will increase

significantly.

Harmful mobile code can be divided into three categories. The code that was written with the sole intention to damage the host. A virus is the best example in this category. The second category contains the code that was originally written without any bad intentions but has been changed in some way. For instance, some bits in the code may have been flipped during transport. Erroneous code makes up the third group. The code contains one or more undiscovered bugs that may harm the host.

Solutions to the problem of malicious mobile code usually involve *authentication*, *authorization* and *enforcement* [15]. Authentication is the process of establishing the owner of the mobile code. Authorization is the assignment of rights and resource limits to the executing unit based on the owner's identity. Enforcement is handled by a security manager making sure the limits are not exceeded.

Authentication is mostly done using digital signatures. Digital signatures are a way to label objects so that the creator of that object may be positively identified to the recipient or user. Asymmetric cryptography is mostly used.

Authorization involves the setup of a security policy. The policy dictates which authorities and therefore which code can use and access which resources.

The security policy needs to be enforced. Moore [23] describes a set of three techniques to prevent so-called unsafe actions: safe interpreters, fault domain and code verification.

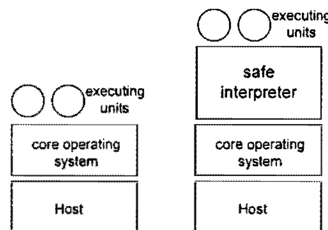

Figure 4.1: An interpreter hides "unsafe" functions

When using an interpreter, the mobile code no longer runs directly on top of the operating system. An extra layer between the code and the operating system, the interpreter, hides the "unsafe" functions from the mobile code. Refer to figure 4.1. Each instruction is examined and depending on the access rights of the code, executed or not. Secondly, the code is loaded in its own part of the address space: a fault domain also known as a sandbox. It is not allowed to store at or jump to any address outside its sandbox. The third technique is proof-carrying code (PCC). Feigenbaum and Lee [11] describe PCC as "a mechanism that supports the construction of easily checkable mathematical proofs of program properties, as well as the formal specification of behavioral safety properties". The proof is sent along with the code so that the host can verify that the code is safe.

In a trusted environment a lower level of security may be possible. Legal restrictions may even suffice [39]. The costs of security always have to be weighed to its benefits.

4.3 Service security

In a mobile code system it is not only the host that can be compromised. The mobile code itself may also be attacked by a malicious host. These attacks can be divided in two groups:

- Extraction of information from the mobile code – Mobile code often carries private information. The code itself may be subject to copyrights (code privacy) or it may contain private

data about an end-user; a secret key or electronic money (user privacy).
- Preventing the code from following its intended thread – The execution state could be altered to influence the program flow, the code could be executed multiple times or not at all.

Chess [6] claims that it is impossible to *prevent* tampering with the mobile code unless trusted hardware i.e. a secure coprocessor is available. All known security measures aim to *detect* violations or try to prevent meaningful alterations: the code can be altered but the resulting behavior is not predictable.

Again costs and benefits have to be weighed. Legal constructions like contracts may be sufficient if both parties share some level of mutual trust [39]. If the code travels only via trusted hosts, strict security measures may be superfluous [15].

Meadows [22] suggests the use of "detection objects". These are dummy data objects which will never be changed by the code itself. Changed values thus indicate that the code has been tampered with.

In order for a malicious host to manipulate a program, it needs to understand the code. Code obfuscation or "code mess up" as Hohl [17] calls it, combined with a limited lifetime of both code and data will make it harder to understand the code. If the lifetime of the code and data is set right, the analysis will be useless. Hohl presents three mechanisms to "mess up" code: variable recomposition, structure dissolving and conversion of compile-time control flow elements into run-time data-dependent jumps. Variable recomposition takes the set of program variables, mixes the contents of these variables up and creates a new set of variables. Structure dissolving tries to eliminate the program structure. Introducing run-time data-dependent jumps makes it harder to analyse the control flow. The lifetime of an object is limited by adding an expiration date to the object and signing the combination using a digital signature.

Vigna [38] proposes to trace the execution of the mobile code. The hash values of the trace and the output are sent back to the client. Digital signatures are used to protect the traces. If the client suspects foul play it can demand to be shown the trace. A disadvantage is that the client is not alerted of foul play. Secondly, the hosts have to store their execution traces in case someone demands them.

Sander and Tschudin [27] present a method to prevent a malicious host from tampering. They claim that it is possible to execute an encrypted function without having to decrypt it. The client has a function f of which the algorithm is to remain secret. The server is willing to compute $f(x)$ for the client. The following protocol is presented:

1. The client encrypts f: $E(f)$.
2. The client creates a program that implements $E(f)$: $P(E(f))$.
3. The program $P(E(f))$ is sent to the server.
4. The server evaluates $P(E(f))$ at x.
5. The server send $P(E(f))(x)$ back to the client.
6. The client decrypts $P(E(f))(x)$ and finds $f(x)$.

4.4 Java and security

4.4.1 The JDK 1.2 Security Model

With the release of the Java Development Kit version 1.2, security measures have again been extended. The MageLang Institute tutorial on Java security [20] describes the Java security architecture on three levels: language-level security, JVM-level security and API-level security.

The security features of the Java language are intended to help prevent errors. therefore pointer arithmetic is not allowed since it may lead to invalid memory access if not used carefully. This does not mean that the Java language does not support pointers. In Java pointers are called references and are supported. Automatic garbage collection relieves the programmer from his memory management duties thus preventing errors in this area. The garbage collector automatically releases memory that is no longer referenced to. The Java compiler implements strong type-checking. Illegal casts are detected to make sure that a block of memory is not misinterpreted.

When a program needs an additional class, that class' class file is loaded by the `ClassLoader`. Before any class can be used, the validity of its bytecode is first verified. Only legitimate Java code can be executed. During execution the JVM provides runtime checking. Array bound checking is an example. Each running JVM optionally has a `SecurityManager` installed. The `SecurityManager` allows the programmer to enforce a security policy restricting the operations of a Java program.

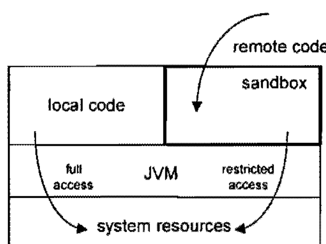

Figure 4.2: The 1.0.x sandbox security model

The JDK 1.0.x introduced a security model known as the sandbox. Remote code is allowed to be executed in a restricted environment. In the JDK 1.0.x all local code is trusted and has access to the system resources. All downloaded remote code is placed in the sandbox and has limited access. Refer to figure 4.2. JDK 1.1.x introduced the concept of signed code. A piece of code signed with a digital signature that is recognized and trusted by the system, is allowed to run outside the sandbox. Refer to figure 4.3. With the introduction of JDK 1.2 a more fine-grained access control is possible [12, 13].

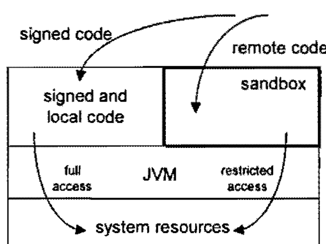

Figure 4.3: The 1.1.x sandbox security model

In JDK 1.2 it is now possible to define a security policy instantiated from the `java.security.Policy` class. The policy is enforced by the `AccessController` with backward compatibility to the earlier `SecurityManager`. A class is fully characterized by its origin and the set of public keys corresponding to the set of private keys used to sign the code. These characteristics are implemented in the `java.security.CodeSource` class. Each class can be given a set of access permissions rooted in the `java.security.Permissions` class. Actually, not the class but a protection domain is granted a set of access permissions as shown in figure 4.4 and figure 4.5. Each class belongs to only one domain. Domains are transparent to programmers and

are created "on demand". If a class tries to access a resource for which it has no permission, the `java.lang.SecurityException` is thrown.

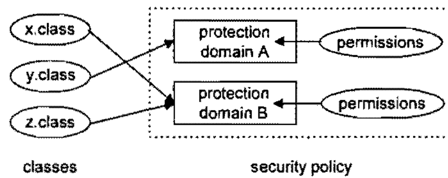

Figure 4.4: The mapping of classes to domains to permissions

A policy entry consists of a set of locations and one or multiple signatures. For the class to match a policy entry both its origin and the signature information must match. The permission of an execution thread is the cross section of the permissions of all protection domains transversed.

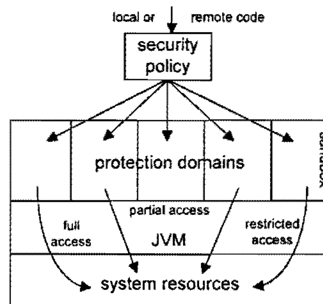

Figure 4.5: The 1.2 security model

Besides the language-level and JVM-level security, Java provides security on the API-level: the Java Cryptography Architecture (JCA). JCA refers to a framework for accessing and developing cryptographic functionality. It includes APIs for digital signatures, message digests and certificate management. It includes no implementations. A cryptographic service provider (CSP) is a set of packages that supplies an implementation of one or more cryptographic services. Sun's JDK comes with a standard provider called SUN. The SUN package provides support for security concepts such as certificates, keys, message digests and signatures. The message digests provided are implementations of SHA-1[1] (NIST[2] FIPS PUB[3] 180-1) and MD5[4](RFC 1321). The package comes with support for generating key pairs using the Digital Signature Algorithm (DSA) and a certificate factory for handling X.509 certificates.

Sun also provides its Java Cryptography Extension (JCE)[5]. JCE is a standard extension package for use with the JDK 1.2. It extends the JCA API to include APIs for encryption, key exchange and Message Authentication Code (MAC) and includes a provider for DES, Diffie-Hellman, RSA and others.

[1]Secure Hash Algorithm
[2]National Institute of Standards and Technology
[3]Federal Information Processing Standard Publication
[4]Message-Digest Algorithm 5
[5]JCE is not exportable outside the U.S. and Canada

4.4.2 Java security applied to mobile code

All code running on the virtual machine is subjective to the installed security manager and thus to the local security policy. The policy designer can grant different authorities/programs the appropriate rights. Each program runs in its own security domain shielding the local host. Depending on the origin of the program and the attached digital signatures, permissions to use resources are granted. These mechanisms shield the host against malicious and altered programs. The third category, erroneous code, still poses a threat to the host. Java verifies the syntax of the bytecode but not its functionality.

Java does not incorporate dedicated mechanisms to prevent a host from tampering with the code it executes. Of course, it is possible to include detection objects, to use code obfuscation techniques or cryptographic traces. Mind that these do not *prevent* tampering.

4.5 Summary

Techniques and mechanisms to protect a local host against harmful (mobile) programs have been studied and implemented by many people around the world, including Sun's. Now researchers are turning to the other side; protecting the code. Without dedicated hardware it remains impossible to protect code and data against analysis or tampering by the local host. Sander and Tschudin have presented a theoretical basis to execute encrypted code. For the time being, authorities will however have to rely on legal constructions to protect their and their customer's rights.

Chapter 5

Roaming IN Services

This chapter describes the introduction of mobile code in an Intelligent Network, i.e. roaming IN services. What kind of architecture is required and which services could benefit from the new approach?

5.1 Roaming Services in an IN

When a subscriber in the traditional IN architecture moves out of the service area of his SCF, most of his IN services can no longer be accessed. One way to solve this problem is to introduce the concept of mobile code in the IN. IN services become mobile and roam wherever the subscriber goes.

Figure 5.1 shows the new architecture. It contains four main entities: the SSF, SCF and a computational environment in the visited network and an IN server in the home network.

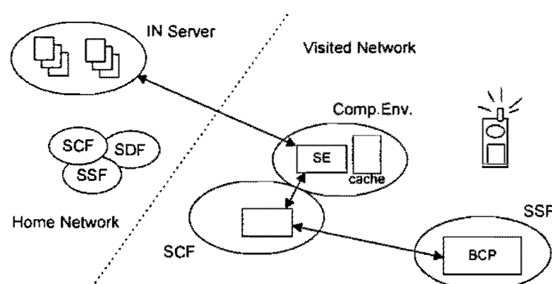

Figure 5.1: Architecture for roaming IN services

The IN server stores the subscribers' service profiles and makes them available to the computational environment. The service profile of a subscriber consists of two parts; the set of IN services he is subscribed to and a list of the corresponding detection points: the DPL. Both parts need to be transferred to and installed in the visited network's SCF/SSF.

The SSF and SCF were introduced in section 2.1. The task of the SSF remains the same. The main task of the SCF, service execution, is now transferred to the computational environment. The SCF is reduced to relaying communications between the SSF and the computational environment.

The computational environment hosts a Service Environment (SE) and a local cache. The SE will download and execute the IN services and DPLs. It also manages the local cache. The cache will

temporarily store the downloaded DPLs and IN services for reuse.

The following subsections will describe the various tasks in more detail. Appendix B.2 lists the information flows between the four entities.

5.1.1 Tracking the subscriber

First of all, the home SCF needs to keep track of its roaming subscribers. It needs to know where to route incoming calls. When a subscriber moves from the service area of one SCF to the service area of another or when he switches on his mobile phone in a new service area, he needs to register with the visited network. The visited network will inform the home network of the subscriber's new location. The home network will store the subscriber location information in the home SDF. As a response, the home network will inform the visited network whether that subscriber is subscribed to roaming IN services. This will be done implicitly by sending or not sending the subscriber's DPL to the SE. Because the DPL is downloaded when the subscriber registers there will be no transport delays when a call is initiated. The home network will also inform the abandoned visited network that the subscriber's personal service profile can be removed. Refer to figure 5.2.

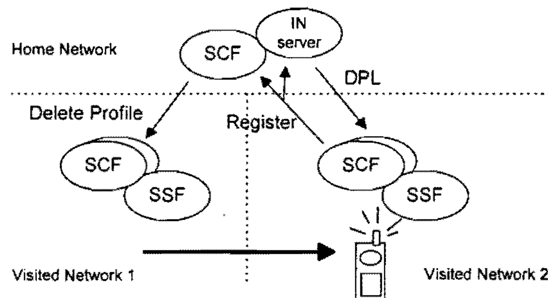

Figure 5.2: Removing and installing a profile

5.1.2 Arming Detection Points

In order to invoke an IN service, the detection points corresponding to the service's POI need to be armed. According to CS-1 and CS-2 only the SMF can arm or disarm a TDP. therefore it is not feasible for the visited network to statically arm and disarm detection points whenever a roaming subscriber enters and leaves the service area. To solve this, DP1 (for originating calls) and DP12 (for terminating calls) will be armed statically for *all* subscribers. These are "the first" detection points in the basic call process. The service logic triggered by these two detection points will request the DPL from the SE, which will retrieve it from its cache. The DPL is passed to the SSF which will arm the listed EDPs. If the subscriber's DPL is not available in the cache, the subscriber is not permitted to use any roaming IN services and the call will be resumed. The procedure is the same for originating and terminating calls. Refer to figure 5.3

The server may command the SE to remove one or more DPLs from the local cache. This will occur when the subscriber registers in another network or when a cache flush is forced by the home network.

5.1.3 Retrieving and Invoking IN Services

There are several possibilities as to how and when to install and remove IN services. As explained, a service is built by combining individual SIBs. The service logic glues the SIBs together forming

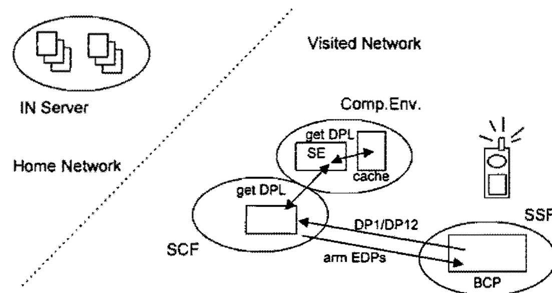

Figure 5.3: The DPL is retrieved and the EDPs are armed

a service. This basic version of the service can be customized to serve the needs of individual customers. These three parts form the IN service that has to be transferred. To the first question as to *what* to copy, there are at least three answers:

- Copy the service logic and the customization data – The implementation of the SIB should be distributed to all the participating SEs beforehand.
- Copy the entire service – Copy the SIBs, the logic and the customization data.
- Copy the complete set of the subscriber's IN services – Thus making all services immediately available when called upon.

The first answer has the disadvantage that every operator's proprietary SIBs have to be stored at every SE of every participating operator. This solution, seemingly elegant at first glance, requires a relatively large storage capacity as well as a management effort with respect to the SIBs. The second answer requires the entire service to be transported. This can be done by copying the service's code files: the SIBs and logic, and the subscriber's data file. A second solution is to make use of object serialization[1] and dynamic binding[2]. An instance of the code implementing the IN service is moved to the SE, bound and executed. The third answer proposes to copy all the subscriber's IN services in one action to the SE. Only one copy action is required.

For now it is assumed that the transport delay of one IN service is negligible or at least acceptable. An IN service will be downloaded when its detection point is triggered.

There are two possibilities as to *when* the services should be copied to the visited SCF:

- Copy the services as soon as the subscriber enters the new service area and registers.
- Copy the service (or all services) only when needed.

The first possibility implies that all services are copied in one go. The advantage is that no transport delay will occur when the subscriber accesses the service. It remains to be seen whether this delay is significant in a real-life situation. The second solution requires less storage capacity. The service is downloaded when the subscriber tries to access it. A possible compromise would be to copy all services as soon as the subscriber accesses one of them. The question is whether the larger transport delay on first access is acceptable.

Once a service is no longer needed, it should be removed from the SE. But *when to remove* the service (or the set of services)?

- Delete a service immediately after use.
- Store the service (set of services) in a local cache until the subscriber leaves the service area.

[1] *Object serialization* is the encoding of an object into a stream of bytes and the complementary reconstruction of the object from the stream.

[2] *Dynamic binding*, a.k.a. *late binding* refers to the linking of the software interface between two objects at runtime, i.e. no recompiling is required.

The first option would be feasible if transport delay and cost were negligible. But since a service is not likely to change often, it may anyhow be cached in the visited network. When the command to delete the profile is received from the home network, the set of services is removed from the cache.

Figure 5.4 shows a possible setup. The EDPs have been armed and when triggered, a message is sent to the SCF. The SCF passes the information to the SE. After analyzing the information, the SE will retrieve the IN service from the local cache or the IN server in the home network and invoke the service.

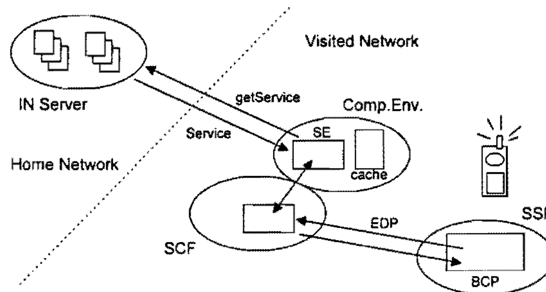

Figure 5.4: The IN service is retrieved and invoked when an EDP is triggered.

5.2 Service Management

Managing roaming IN services involves the following activities:

- Introduction of new IN services
- Cancellation of IN services
- Change of IN services by the operator
- Change of IN services by the subscriber

All these activities must be carried out while the system is online. Taking a telecom system offline is too costly. The management procedures should be designed to support online management.

Introducing a new IN service involves two things. First the operator will have to place the IN service's code files on the IN server. Secondly that service's detection points will have to be added to the DPLs of those subscribers requesting the service. Possibly, some local caches will have to be flushed so that the SE will download the updated versions of the DPLs.

When removing an IN service the operator will first have to remove the corresponding detection points from the DPLs. Then the SE is instructed to remove the outdated DPLs from its local cache and download new ones from the IN server. The next step is to remove the services' code files from the IN server. Finally, the server will instruct the SE to remove the appropriate service from the cache. When an instance of a DPL is however already in use, i.e. the EDPs have already been installed in the SSF, the old service will be requested when its detection point is triggered. Since the service's code files are no longer available the server will instead return an empty service. The empty service will instruct the SSF to resume the BCP.

The procedure for updating an IN service is similar to that when removing a service. The first step is to make the new code files available on the IN server. Secondly the DPLs on the server are changed to reflect the new situation. The procedure ends with removing the old service as described in the previous paragraph.

Some IN services allow the subscriber to change his personal settings. The program code to accomplish this is embedded in the roaming IN service. The IN server will however also need an interface to allow this kind of interaction.

5.3 Services and mobility

It is beyond the scope of this report to determine which of the two solutions – roaming IN services vs. a signaling connection – is "the better" one. The answer to this question involves many issues. One of the most important factors is the level of mutual trust between the two network operators. The chapter on security pointed out that it is hard, if not impossible, to protect the service's code and data against violations. Two competing operators, serving the same market, will think twice before sending their data to the competitor. Secondly, the costs should be considered. What are the costs of invoking the IN service using a signaling connection? How do these relate to the costs of transportation and local execution of the IN service code? Moreover – what will the neighbors do? Since both architectures involve multiple networks, an operator will always depend on what his competitors decide. Without any doubt there will be more issues to consider.

But which categories of IN services are suitable for the roaming IN services architecture? The gain of executing the IN service locally – in the visited network – should justify the transport. If a service requires little or no processing it may be smarter to transport the result instead of the code. This is of course not as simple a statement as it looks. Since the roaming services are cached locally they can be reused. Hence frequent usage may also justify local execution. And how about the transport delay? When looking at a high capacity network, a relatively large service may be transported without any significant delay. If the load is high the main goal may be to minimize transports to reduce it.

IN services that are used frequently are a good candidate. The local cache eliminates the need to setup a connection with the home network every time the service is triggered. The Virtual Private Network (VPN) service is a good example. The next section will describe the VPN service.

5.4 The Virtual Private Network Service

For a more detailed description of the Virtual Private Network (VPN) service please refer to the VPN Service Overview [8].

The VPN service aims to give the subscriber the illusions that his communications take place over a private network while in fact using the public telecom infrastructure. The private numbering plan is the basis of the VPN service. It allows the service users to reach each other using a short private number instead of the long public number.

The service has the following additional features:

- Announcements
  The ability to play an announcement to the caller.
- Call Diversion
  The ability to redirect a call when the call can not be completed.
- Charging
  Successful VPN calls are charged.
- Customer Control
  A service user has the ability to turn the call diversion feature on or off.
- Equal Access
  Removes the Carrier Access Code from a dialed number.

- Forced On-Net
  The ability to translate public numbers to on-net numbers when possible.
- Inter User-Group Dialing
  The ability to call a service user in another user group.
- Number Presentation
  The ability to display the number of the calling party at the called party.
- Off-Net Calls
  The ability to dial destinations outside the VPN by first dialing a special prefix.
- Office Zone
  The ability to indicate whether a mobile call has been made from within or outside a predefined Office Zone.
- Outgoing-Call Screening
  The ability to block specific private and/or public numbers.
- Statistics
  Statistical information on attempted, successful and unsuccessful calls.

5.5 Summary

This chapter presented an architecture for a network implementing roaming IN services. The architecture consists of four main entities: the Service Switching Function, the Service Control Function, a computational environment and the IN server. The IN server stores and makes the roaming IN services and DPLs available to the computational environment. The computational environment is responsible for executing the roaming IN services and processing the DPLs and registration requests. The SCF is reduced to an intermediate, passing information between the computational environment and the SSF. The SSF is a conventional SSF.

IN Services that are used frequently, like the VPN-service, can be used effectively in the roaming IN services architecture. The local caching mechanism reduces communications with the home network.

Chapter 6

RJS System Design

In this chapter, the architecture described in the previous chapter will be given a concrete form using the Unified Modeling Language. A top-down approach is used. The first section will globally describe the total RJS system.The following sections will examine the individual components more closely.

6.1 The RJS system

Figure 6.1 shows the component diagram of the Roaming Java Services system. It contains three types of nodes: the *Server node*, the *Java Service Environment node* (JSE) and the *Service Control Function node.* The Server node, which is located in the home network, is geographically separated from the other two. The SCF node is an already existing node, originally intended for service execution. This functionality will now be performed by the JSE. The SCF is used solely to transfer information between the SSF and the JSE.

The server's main task is to make the roaming IN services and the related DPLs available to the JSEs. In order to accomplish this, it has a *Subscriber Database* and a *Service Database.* The service database holds the roaming IN services available on this server and the service's general configuration ("service specific service data"). The subscriber database contains subscriber specific information. This includes the DPL, personal information and subscriber-specific service configuration data ("subscriber specific service data"). The *Publisher* will make the services and DPLs available and initializes them before they are downloaded by a JSE.

The roaming IN services will be executed on the Java Service Environment node. The *Service Engine* is responsible for downloading and executing IN services as well as DPLs. A *Cache* will store the most recent DPLs and IN Services to relieve the network.

The Service Control Function node has the task to forward requests, coming from the SSF, to the JSE and return the results.

The three nodes will be described in the following sections.

6.2 The Server Node

6.2.1 Use Case Diagram

The Server node contains a number of use cases. The use case diagram in figure 6.2 shows these, their mutual relationships and the information flows.

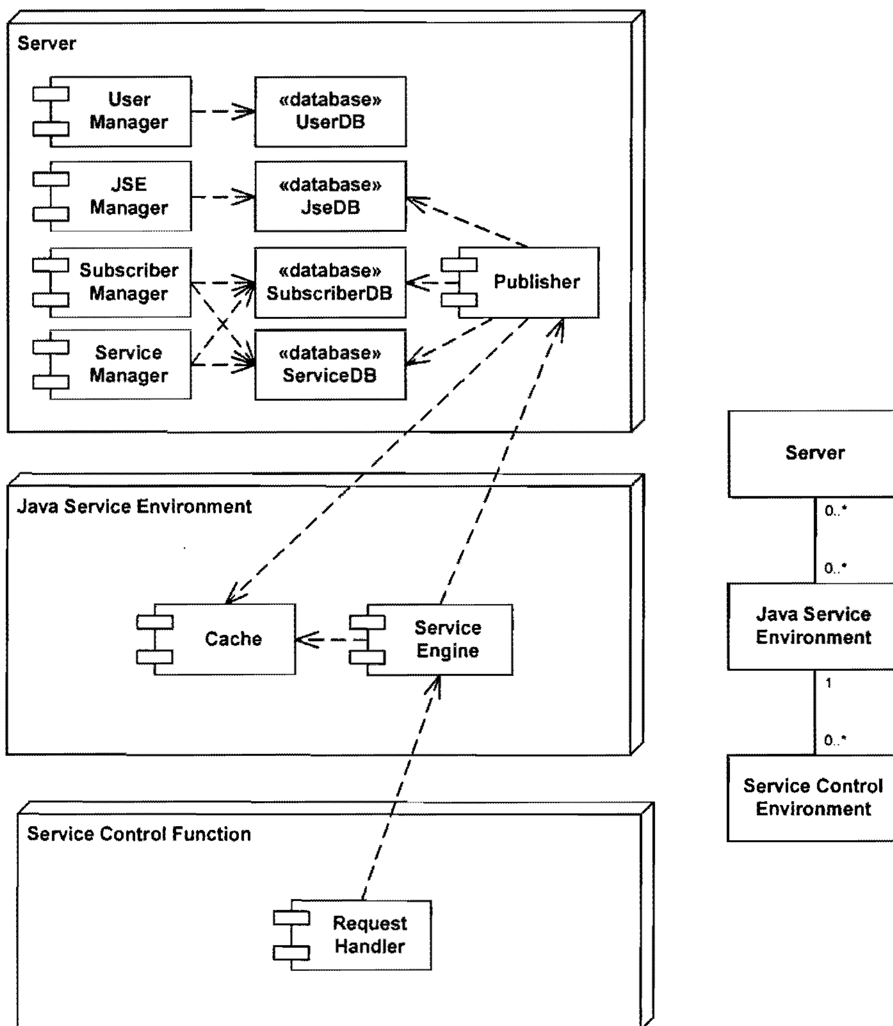

Figure 6.1: Component diagram for the Roaming Java Services system

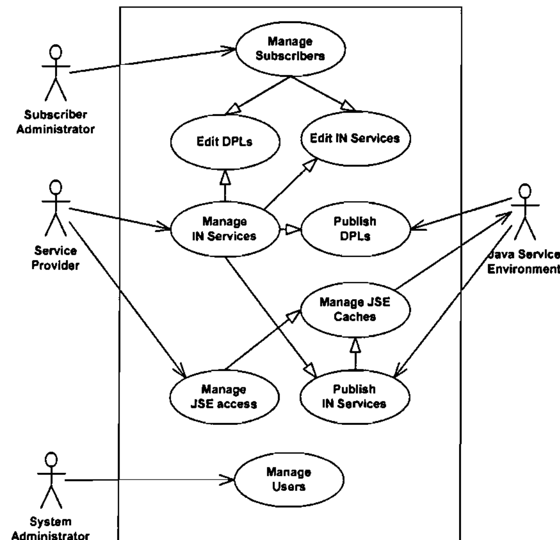

Figure 6.2: Use case diagram of the Server

Description of the use cases:

"*Manage Subscribers*":
Manage the subscribers making use of this server. Add subscribers, remove subscribers and change subscriber data. This involves personal data but also which roaming IN services the subscriber is allowed to use and his DPL.
Refer to figure C.1 in Appendix C for the activity diagram.

"*Manage IN Services*":
Manage the set of roaming IN services that are available from this server. Add to and remove services from the server and change the service specific service data.
Refer to figure C.2 in Appendix C for the activity diagram.

"*Customize IN Services*":
Manage which roaming IN services a subscriber can use and customize the service for that subscriber. The subscriber specific service data is stored in the subscriber database.

"*Customize DPLs*":
Depending on which roaming IN services a subscriber uses, one or more detection points will be added to the subscriber's DPL. The information is stored in the subscriber database.

"*Publish DPLs*":
Make the DPLs available to the JSEs.
Refer to figure C.3 in Appendix C for the activity diagram.

"*Publish IN Services*":
Make the roaming IN services available to the JSEs.
Refer to figure C.4 in Appendix C for the activity diagram.

"*Manage JSE Caches*":
Request a JSE to delete items from its local cache.

"*Manage JSE Access*":
Manage which JSEs are allowed to download roaming IN services from the server.
Refer to figure C.5 in Appendix C for the activity diagram.

"*Manage Users*":

Manage the Service Providers and Subscriber Administrator accounts. Add, remove and change accounts.

6.2.2 Class Diagram

Figure 6.3 shows the class diagram of the Server. A dotted arrow from A to B indicates that A depends on B. In following diagrams, a "normal" arrow from A to B indicates that A inherits from B.

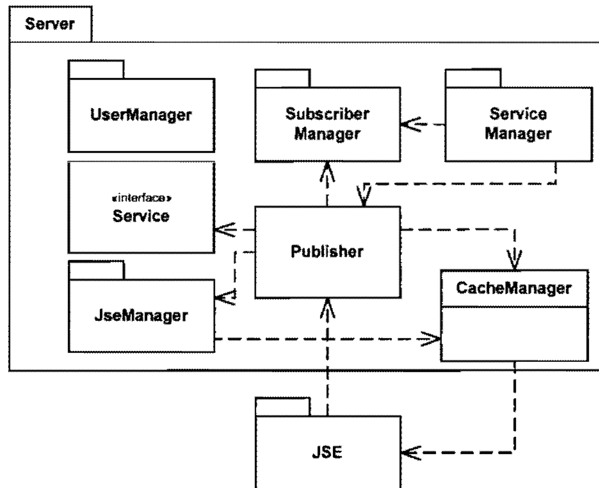

Figure 6.3: Class diagram of the Server

The prototype will not implement the entire Server node as described in the previous subsection. The focus of the project is on the JSE node. Hence databases and user interfaces will be kept to a minimum. The packages `UserManager`, `SubscriberManager`, `Service Manager`, `CacheManager` and `JseManager` will not be fully specified at this point.

The `JseManager` keeps track of which JSEs have access to the server. For this purpose, it keeps a list of identification/password pairs. Whenever a JSE contacts the server, it also sends its identification and password. However, before access is granted the JSE has to send the address of its local cache. The `CacheManager` needs this address to remove outdated entries from the local cache.

The `Publisher` receives requests from the Java Service Environment for DPLs and IN services. When the JSE requests a specific DPL, the `Publisher` will use the `SubscriberManager` package to retrieve the DPL from the subscriber database and return it. When a service is requested, it will first be initialized before it is returned. The `Publisher` uses the `ServiceManager` and `SubscriberManager` packages to initialize the service. The `JseManager` package is used to verify whether a JSE has access to the server.

6.3 The Java Service Environment Node

6.3.1 Use Case Diagram

The Java Service Environment node contains four use cases. Figure 6.4 shows their structure.

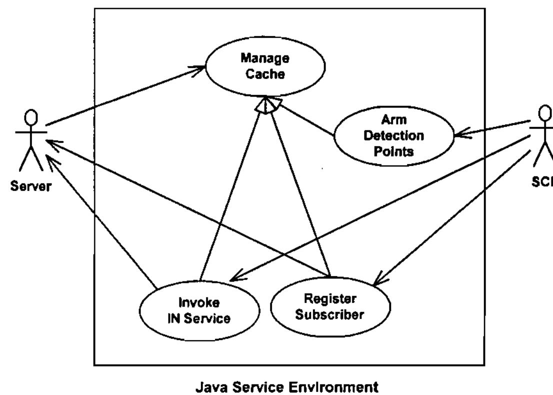

Figure 6.4: Use case diagram of the Java Service Environment

Description of the use cases:

"*Manage Cache*":
Add and remove items, either a DPL or a roaming IN service, to/from the cache.
Refer to figure C.6 in Appendix C for the activity diagram.

"*Register Subscriber*":
When a subscriber registers with the network, retrieve his DPL from his service provider's IN server and store it in the local cache.
Refer to figure C.7 in Appendix C for the activity diagram.

"*Arm Detection Points*":
The SCF requests a subscriber's DPL. Retrieve it from the local cache and send it to the SCF.
Refer to figure C.8 in Appendix C for the activity diagram.

"*Invoke IN Service*":
The SCF requests the JSE to execute a roaming IN service for a specific subscriber. If the service is not yet available in the local cache, retrieve it from the IN server and store it in the cache. Execute the service and return the result to the SCF.
Refer to figure C.9 in Appendix C for the activity diagram.

6.3.2 Class Diagram

For clarity the class diagram of the Java Service Environment has been split into multiple figures. Figure 6.5 shows the first part.

The `ScfGateway` is the first class started on the JSE node. It is responsible for maintaining a list of all connected SSF nodes: the `SSFList`. Whenever an SSF node connects to or disconnects from the SCF node, the SCF node informs the `ScfGateway`. The `ScfGateway` is a singleton, meaning that there can only be one instance of this class.

The `SSFList` maintained by the `ScfGateway` contains representations of all SSF nodes currently connected to the SCF node. Every connected SSF node is represented by an `SSF` object in the list.

An `SSF` object is created by the `ScfGateway` when an SSF node connects to the SCF node. The purpose of the `SSF` class is to hide the communications with the SSF node from the other Java classes. When created, the `SSF` will setup its own connection to the SCF node. Every `SSF` keeps a list, the `CallList`, of ongoing calls on its peer SSF node. Every ongoing call is represented by

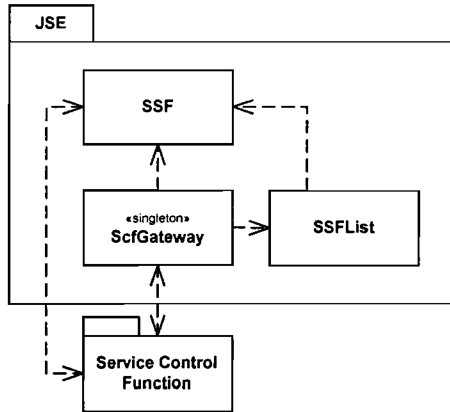

Figure 6.5: Class diagram of the Java Service Environment (part 1): the `ScfGateway`

a `Call` object. If the `SSF` receives a message from the SCF node, it will create a new `CallJob`, put it in the FIFO `Queue` and notify the `JobHandler`. Messages from other classes addressed to its peer SSF node will be forwarded to the SCF node. Refer to figure 6.6.

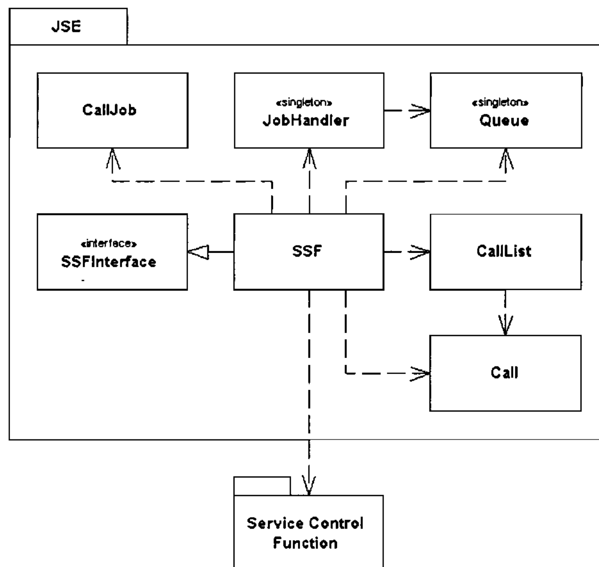

Figure 6.6: Class diagram of the Java Service Environment (part 2): the `SSF`

The `JobHandler` runs in its own thread; it inherits from the `Thread` class. It takes the first job from the `Queue` and executes it. This is repeated until the queue is empty after which the JobHandler suspends itself. The `JobHandler` is a singleton.

A `Job` instance is created when an event has occurred. The `SSF` and the `ServerThread` are the only two object types that create jobs and put them in the queue. A `Job` instance contains the data related with the event and the party for whom the event is intended. Three kinds of jobs exist. All three implement the `Job` interface. Refer to figure 6.7.

A `RegJob` instance is created when an `SSF` receives a registration event from the SCF node. When the job is executed it will request the `ServerGateway` to download the subscriber's DPL from the

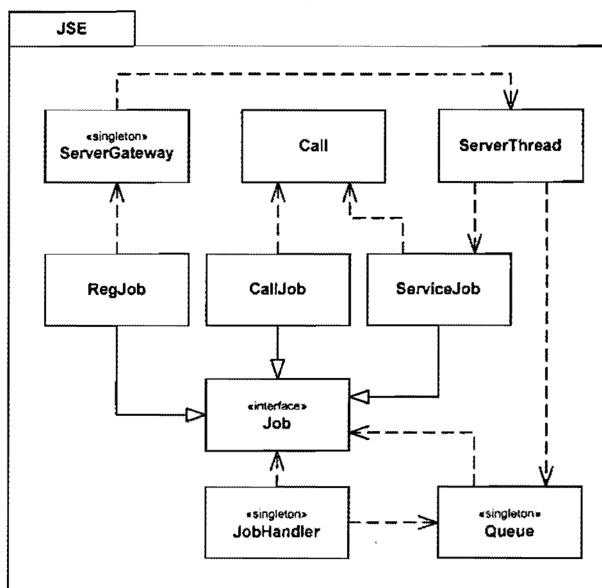

Figure 6.7: Class diagram of the Java Service Environment (part 3): the `JobHandler` and the `Job`

server node and store it in the `Cache`.
A `CallJob` instance is created by the `SSF` when it receives any other event from the SCF node. When executed it will do one of two things. In case a new call is being setup it will create a new `Call` object. In the other case it will pass the received information to the appropriate, already existing, `Call` object.
The third job-type is the `ServiceJob`. This job is created by a `ServerThread` instance when it has downloaded a service from the Server node. When executed, the job will forward the information to the appropriate `Call` object.

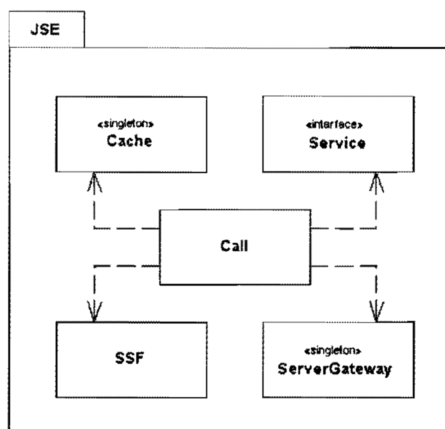

Figure 6.8: Class diagram of the Java Service Environment (part 4): the `Call`

For every call a `Call` instance is created. When created, the class' constructor will initialize the call parameters using the data provided by the incoming event. A statically armed detection point generates more call related information than a dynamically armed detection point. Hence the `Call` instance has to store the "extra" information generated by DP1 (or DP12) so it can be

used in the future. The second task of the constructor is to retrieve the DPL from the cache and arm the EDPs. It uses the SSF object to send messages to its SSF node
When a Call instance receives an event, this means that the subscriber wants to use an IN Service. If the appropriate IN service can be found in the Cache, the Call object will immediately execute the Service. However, if the service is not present in the cache, it will request the ServerGateway to retrieve it from the server.

The ServerGateway keeps a pool of ServerThread objects. When requested to retrieve a DPL or service from the server, it will get an idle thread from the pool and instruct it to download the item. If there are no idle threads in the pool, the gateway may decide to create a new ServerThread object or to ignore the request. The ServerGateway is a singleton.

As the name indicates, the ServerThread is a thread. When created, a ServerThread will immediately suspend itself. When awaken by the ServerGateway, it will download a Service or a DPL from the server node and store it in the Cache. In case of a service, it will create a ServiceJob, put it in the Queue and notify the JobHandler. Finally it will switch back to the idle state. Refer to figure 6.9

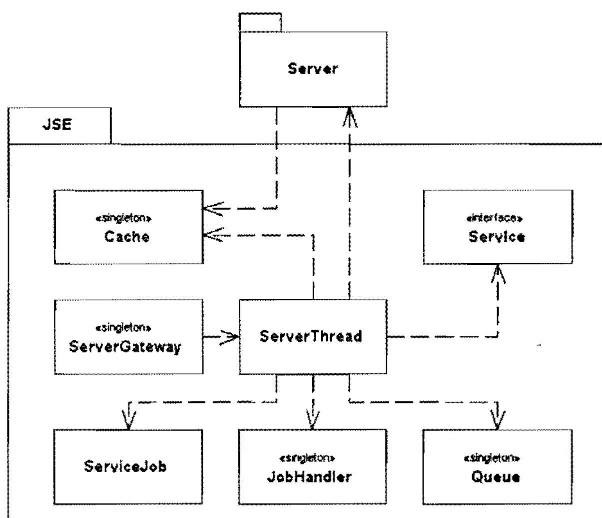

Figure 6.9: Class diagram of the Java Service Environment (part 5): the ServerGateway and serverThread

Interaction with the server node takes place in separate ServerThread threads. Large response times from the server would slow down the system if everything was handled by one single thread. By using separate and multiple ServerThread threads, the overall system can continue while downloading.
When the JSE is started, a pool of threads is created. Creating a new ServerThread object every time interaction with the server node is required, would slow down the system. A pool solves this problem.

The Cache stores the DPL's of the subscribers registered with the network and the most recently used IN services. The Server also has a limited access to the cache. It will notify the cache when a DPL or service has to be updated and when a subscriber has registered in a different network. The Cache is a singleton.

In Appendix D three sequence diagrams are shown. Figure D.1 shows what happens in the JSE when a user registers with the network. Figure D.2 shows the actions occurring after DP1 (or DP12) has been triggered. Finally, figure D.3 depicts the order of actions when a roaming IN service is activated.

6.4 The Service Control Function Node

6.4.1 Use Case Diagram

Figure 6.10 depicts the use case diagram of the Service Control Function node.

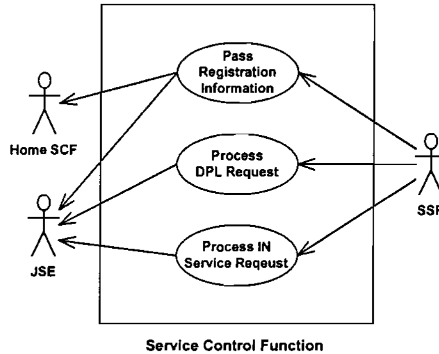

Figure 6.10: Use case diagram of the Service Control Function

Description of the use case diagram:

"*Pass Registration Information*":
Pass the registration information coming from the SSF to the JSE and to the SCF in the subscriber's home network.

"*Process DPL Request*":
Pass the request for the subscriber's DPL coming from the SSF to the JSE. When the JSE returns the list, forward it to the SSF.

"*Process IN Service Request*":
Pass the request to execute a roaming IN service coming from the SSF to the JSE. When the JSE returns the result, forward it to the SSF.

6.4.2 Class Diagram

Figure 6.11 shows part of the module diagram of the Service Control Function node. The `jsegateway` is the interface between the JSE and the SCF and SSF. It hides the communication details from the other SCF- and SSF-modules.

6.5 IN Services

The `Service` interface is the base of all roaming IN service classes. It defines which methods a service has to implement. Figure 6.12 shows the class diagram. An abstract class `ServiceClass` provides a default implementation for some of the methods.

The following methods are enforced by the interface:

- `execute` – Starts the execution of the IN service.
- `getServiceKey` – Returns the service's service key. The service key uniquely identifies the service.
- `getSubscriber`, `setSubscriber` – Returns and sets the subscriber for whom the service is configured. A subscriber is identified by his telephone number.

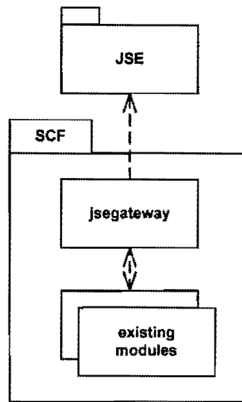

Figure 6.11: Module diagram of the Service Control Function

- `getSubscriberData`, `setSubscriberData` – Returns and sets the subscriber specific service data. The data will in most cases include private information. The service is therefore allowed to encode the data before it is returned. The JSE will not use the subscriber data. Refer to section 7.3.7.
- `inputRepsonse` – When the IN service requests the subscriber for input it is asynchronously sent back to the service by means of this method.
- `clone` – Creates a duplicate of the IN service object.

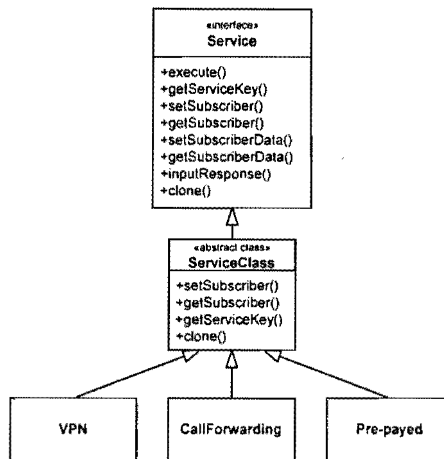

Figure 6.12: Class diagram of the Service class hierarchy

6.6 A Single-threaded vs. Multi-threaded Implementation

The presented architecture is meant to serve a large number of subscribers. This implies that in a small interval of time, multiple events will arrive at the Java SSF objects and that multiple requests will be sent to the Server by the server gateway.

The events arriving at the SSFs can be handled sequentially or concurrently by the JSE. Sequential processing means that the incoming events will be placed in a queue and handled one after the

other. When the events are handled concurrently, a new thread will be started, executing the event handling mechanism, every time an event arrives. The former is referred to as a single-threaded and the latter as a multi-threaded implementation.

Figure 6.13 shows the two situations. The top picture shows the single-threaded and the bottom picture the multi-threaded implementation. A thread is allowed to run for a time interval of t units before the system switches to the next thread. Four events arrive at the same point in time. It is assumed that they each require roughly the same processing time; in this case 4t units.

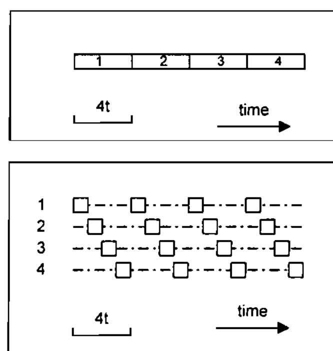

Figure 6.13: Event-driven vs. multi-threading

In both cases the system finishes after 16t units. In practice this is not true. Switching between threads consumes processing power and time. therefore the multi-threading implementation takes slightly more processing time. A closer look shows that events 1, 2 and 3 in the single-threaded implementation are finished before events 1, 2 and 3 in the multi-threaded implementation. For event 4 it makes no difference which implementation is used. The average time before an event has been handled is thus less for the single-threaded implementation.

The event handling in the JSE as described in section 6.3 is single-threaded.

Retrieving an object (service or DPL) from a heavily loaded remote server will take time. Part of his time the thread doing the download will be waiting for the server to respond. In a single-threaded system this means that the entire system would stagnate. If the download process was handled by a separate thread, the overall system would continue.
A third alternative is to start *every* download request in a separate thread. Multiple connections to the server could be open at one time, increasing the overall throughput. A server can however serve only a limited number of clients. therefore, clients should be careful not to overload the server. Creating a *new* thread object for every download also generates overhead. The best solution therefore is to create a pool of suspended threads. Whenever a download is requested, an idle thread is taken from the pool and activated. When the download is complete, the thread suspends itself again and returns to the idle state.

The JSE implements a thread pool from which a thread is taken for every download.

6.7 Summary

This chapter described the design of the RJS prototype. It consists of three nodes: The server node, the JSE node and the SCF node. The server node was not specified in detail because the project's focus is on the JSE node. Several UML diagram types were used to specify how the JSE works. The use case diagram, activity diagrams, class diagrams and sequence diagrams show how the JSE is supposed to react to stimuli. Messages coming from the SCF are placed in a queue and handled successively by a single thread. The gateway to the server node however uses

a thread pool to speed up communications. For the IN services a set of mandatory methods was put together in an interface.

Chapter 7

The RJS Prototype

This chapter describes the implementation phase of the RJS system; how have the three nodes been implemented in the RJS prototype? It starts off with a detailed description of the involved classes. Then the interfaces between the nodes are specified. The limitations of the prototype are revealed in the last sections.

7.1 The RJS System

The roaming IN services architecture consists of four communicating nodes. Figure 7.1 shows these and the communication lines.

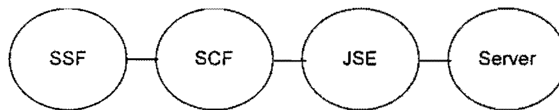

Figure 7.1: The four nodes

The prototype will use the RSP described in section 2.5 to simulate a normal telephone system including all normal IN functionality. The RSP software will be running on a UNIX Solaris system. The Java environment, implementing the JSE node, will be running on a PC-clone with a Windows NT operating system. Note that because Java byte code is able to run on more than one operating system, it can just as easily be run on a UNIX platform. The server, including the web-server, will also be running on a UNIX Solaris system. In this case one UNIX system will be used to host both the RSP and the server node. Figure 7.2 shows this.

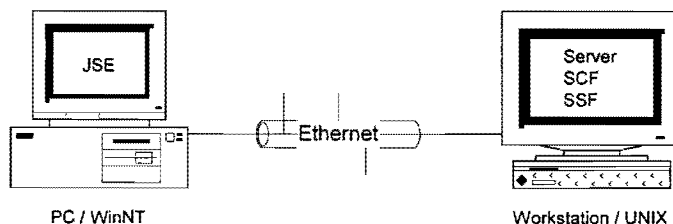

Figure 7.2: A Windows NT PC-clone and a Solaris UNIX system will be used.

The next sections show how the individual nodes are implemented. For more details about the classes and methods refer to the API document generated by javadoc[1]..

7.2 The Service Control Function

The SCF node and the SSF node are part of the RSP-tool. Figure 7.3 shows the relevant modules of the SCF software. The `lim_pad10` and `scf_man10` modules are started (and stopped) by the `scf10` module. The `lim_pad10` module handles the communication between the SSF and SCF node. The `scf_man10`, among other things, keeps track of which SSF nodes are being connected to and disconnected from the SCF node.

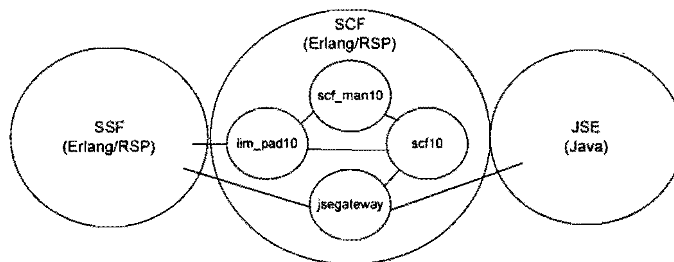

Figure 7.3: The relevant modules of the SCF software

The `jsegateway`, although also implemented in Erlang, is not part of the RSP software. It has been added to handle all communications between the JSE and the SCF/SSF. Refer to section 7.5.1 on interfaces. Its only task is to forward messages to and from the JSE thus hiding the communication mechanism from the other RSP modules.

Changes to the RSP software have been kept to a minimum. The `scf10` has been changed to also start and stop the `jsegateway`. The `scf_man10` module has been changed to also notify the JSE (via the `jsegateway`) when an SSF node connects to or disconnects from the SCF node. It also reports when a subscriber registers or unregisters at an SSF. The `lim_pad10` was used as a model for the `jsegateway`.

When the `jsegateway` is started, it starts the Jive server. In order to relay messages, the gateway has to keep track of the paired Erlang and Java addresses. Addresses come in pairs: one for the Erlang domain and one for the Java domain. These pairs are stored in the gateway's process dictionary. Every new connected SSF node and every new call comes with a new address pair.

In the RSP software a registration IN service is implemented. This service enables a subscriber to register at an arbitrary phone set. Once registered at a phone set, all calls for that subscriber will be routed to that phone. During the registration procedure the SCF node will send a message to the phone with the registration information. This message is now also forwarded to the `jsegateway` and thus to the JSE. The JSE uses this conventional IN service to simulate a roaming subscriber. In a mobile telephone system a roaming subscriber "automatically" registers when entering a new mobile network. In this prototype the subscriber will have to invoke the registration service. The JSE will receive a copy of the registration message and act accordingly.

[1]javadoc is the Java API documentation generator. It parses the declaration and documentation comments in a set of Java source files and produces a set of HTML pages.

7.3 The Java Service Environment

The Java Service Environment consists of a number of classes described shortly in the previous chapter. The next subsections will describe these classes in more detail and deal with the implementation specifics.

7.3.1 Singletons in Java

A singleton is a class of which only one instance can exist. The key to this idea is to make sure that nobody, except the class itself, can create instances of the class. The first thing that has to be done is to make all constructors `private`. At least one private constructor has to be implemented to prevent the compiler from creating a `friendly` default constructor. The next thing is to make sure that the class does indeed create that one instance. There are two ways to do this. The first is to create it statically and the second is to create it on demand in the public access method. The class should also be made final to prevent cloning.

Below is an example from the `Queue` class. The one instance is created statically and the public access method, `getHandle`, returns a handle to that one instance.

```
public final class Queue //singleton
{
private static Queue q = new Queue();
private Queue() {}
public static Queue getHandle() {return q;}
}
```

7.3.2 The ScfGateway and the SSFList classes

The `ScfGateway` is one of the two classes that communicates with the Erlang environment. Section 7.5.1 goes into detail on how that communication takes place. The `ScfGateway` only *receives* messages from the `jsegateway` but does not send any back. However, after connecting to the Jive server, the `ScfGateway` executes the `set_jseaddress` Erlang function in its constructor. This function stores the `ScfGateway`'s address in the Erlang environment. The `ScfGateway` is implemented as a singleton.

The one public method of this class is the `receive` method that gets called by the Erlang environment. It receives and processes five messages:

- `jsegateway` – It contains the address of the `jsegateway` needed by the JSE to send messages to it.
- `setup_ssfs` – Indicates that one or more SSF nodes have been connected to the SCF node. It contains the names of the newly connected SSFs. A new `SSF` object is created for every new SSF node. The objects are added to the `SSFList` and instructed to arm their two static breakpoints. Note that the SCF, according to the ETSI standards, is not allowed to arm static breakpoints in the SSF. However, the RSP software does allow this.
- `unset_ssfs` – Indicates that one ore more SSF nodes have been disconnected from the SCF node. It contains the names of the disconnected SSFs. The corresponding `SSF` objects are instructed to `cleanup` and are removed from the list.
- `stop` – Indicates that the RSP software is being shut down. All `SSF` objects are instructed to `cleanup` and the JSE is shut down too.
- `exit` – Indicates that (part of) the RSP software stopped. All `SSF` objects are instructed to `cleanup` and the `jsegateway` address is reset.

Before the `ScfGateway` accepts any of the other four messages it first needs to receive the `jsegateway`'s address. Until that happens, all other messages are ignored.

The `SSFList` class holds a synchronized `ArrayList`. Since the `ScfGateway`'s `receive` method is not synchronized, access to the list needs to be. The methods of this class are however not synchronized. This is left to the user. The reason for this is that these methods are frequently called from within loops. It is more efficient to synchronize only once before the loop is entered. The implemented methods are: `add`, `get`, `remove`, `indexOf` and `size`.

7.3.3 The SSF, the SSFInterface and the CallList classes

The `ScfGateway` and the `SSF` are the only two classes capable of communicating with the Erlang environment. Therefore these two are also the only classes depending on and making use of Jive. Data forwarded to other Java classes is first converted to "normal" Java datatypes. Data destined for the Erlang environment is converted to one of the Erlang wrapper datatypes.

The `SSFInterface` defines the `SSF` class for the outside world, i.e. for the IN services. IN services are able to send commands to the SSF node. Refer to the introduction of the RSP software in section 2.5. The `SSFInterface` defines which commands Java IN services are allowed to send to the SSF node. Currently the following commands are allowed: "load_data", "set_monitor", "transfer", "continue", "input" and "message". Refer to appendix B.1 for their meaning.

A `SSF` object represents an SSF node. As far as the other Java objects are concerned, it *is* the SSF node. When created, the constructor connects to the Jive server, sends its address to the `jsegateway` and sets an unconditional static breakpoint on both the SSF's originating and terminating "null" state.
The `jsegateway` distinguishes seven different incoming messages from the SCF and SSF nodes: "provide", "event", "register", "unregister", "results", "simulation_end" and "exit". The "initiate" message is not forwarded to the `SSF` since it is immediately followed by "provide". When one of the first three messages is received it is stored in a job and put in the queue. The others are handled immediately by the `SSF`. Refer to appendix B.1 for their meaning.

The `CallList` stores address information about every call. It enables the `SSF` to route messages to the correct parties. An entry in the `CallList` contains four elements. The first is the `EProcess` of the call. This is the address of the call in the Erlang environment. The second element is an integer value that identifies the call in the Java environment. This `callId` is directly derived from the `EProcess`. The third element is the `Call` object. Figure 7.4 shows the mapping between these three elements. The fourth element is a reference to the last executed IN service in relation to this

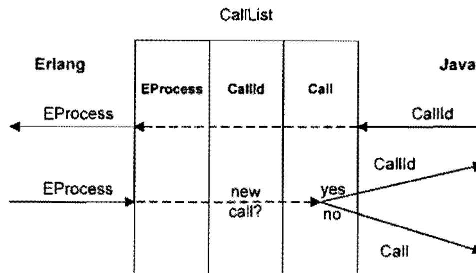

Figure 7.4: The CallList class

call. It is stored when an IN service requests a subscriber for input. The response to this request is passed back asynchronously. therefore the `SSF` temporarily stores the service's address in the `CallList`.

7.3.4 The JobHandler, the Queue and Job classes

The `JobHandler` class extends the `Thread` class. Once the thread it implements is started, it starts checking the FIFO queue in an endless loop. If it finds a job in the queue it will execute it. If there is no job in the queue the `JobHandler` will suspend itself until another job is put in the queue. The `JobHandler` is implemented as a singleton.

A thread can be suspended by calling the `wait` method. The thread can be re-activated by calling the `notify` or `notifyAll` method. These three methods are implemented by the `Object` class. (The `Object` class is the root of the Java class hierarchy.) These methods can only be called if the thread owns the object's monitor. The `notify` method awakens one *arbitrary* thread that was waiting for the object's monitor. The `notifyAll` method awakens *all* threads waiting for the object's monitor.

The `queue` class implements a FIFO queuing mechanism. It contains a synchronized `ArrayList` that stores the job objects, and implements two methods. The `add` method adds an element to the end of the list and the `getFirst` method returns and removes the first element of the list. Both methods are synchronized. The `queue` is implemented as a singleton.

The JSE knows three kind of jobs and all three adhere to the `Job` interface. The `Job` interface dictates that all jobs must implement the `execute` method.

7.3.5 The Call class

The `Call` class has two relevant methods: its constructor and the `handleEvent` method. The constructor will first send the `SSF` a pointer to itself, which the `SSF` will store in the `CallList`. Secondly, it will request the subscriber's DPL from the cache. If this request comes up empty, the "transfer" command is sent to the `SSF`. If a DPL is returned, the `SSF` is instructed to set the dynamic breakpoints and a "continue" message is sent.

7.3.6 The ServerGateway and ServerThread class

The `ServerGateway`'s private constructor will create a predefined number of `ServerThread` objects and stores them in a synchronized `ArrayList`; the thread pool. The JSE will not be granted access to the server before it has sent the address of its local cache. therefore the constructor will obtain a free thread from the freshly created pool and instruct it to send the cache's address to the server. Besides its constructor, the gateway has two `friendly` an one `private` method. The latter is the `getServerThread` method which looks for an idle thread in the pool. If it can not find one, it will check the size of the pool against a predefined maximum size. If the maximum size has not yet been reached, it will create a new `serverThread`, add it to the pool and return it. The maximum size of the pool determines the maximum number of open connections to the server. The two other methods are the `getDPL` and `getService` method. They will use the `getServerThread` method to obtain a free thread, initiate it and call `notifyAll` to start the download. Notice that the `notifyAll` method is used instead of the `notify` method. The `notify` method awakens one *arbitrary* `ServerThread`. There is no way to be sure that the right thread is awakened. therefore all threads are notified. The ones already busy will ignore this message. The once that were idle will start running, check whether they were initialized and if so, download the requested object. When not initialized, it will immediately return to the idle state. The `ServerGateway` is implemented as a singleton.

The `ServerThread` class extends the `Thread` class. In its `run` method it will lookup the `Publisher` in the RMI registry and then go into an endless loop. The loop consists of three parts. In the first part the thread suspends itself using the `wait` method. In the second part, after the thread has been awakened, it will check whether it has been initialized and if so, download the requested

object and store it in the Cache. If the object is a Service, a new ServiceJob is created, it is added to the Queue, and the Queue is notified. In the third part the initialization variables are reset.

The ServerThread responds to three types of exceptions thrown by the IN server: the NoAccessException, the InvalidServiceException and the RemoteException. When it receives one of these exceptions, the ServerThread will do another lookup and then try again. If the lookup fails or the request is denied a second time, the call will continue unchanged. The subscriber will not be able to use his IN services.

7.3.7 The Cache class

The local cache stores DPLs and IN services. It provides means to add, retrieve and remove elements. Not only the local environment can remove elements but also the remote IN server. For this purpose the cache implements an RMI server. Figure 7.5 shows the class diagram of the cache.

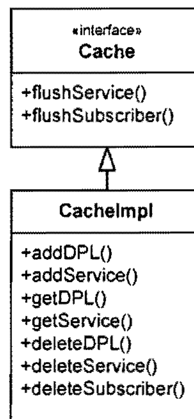

Figure 7.5: The local cache

The Cache interface defines which methods the IN server is allowed to execute. The CacheImpl implements these methods and adds some others that are only accessible from the local environment.

The cache actually stores three types of elements: the DPL, the IN service and the subscriber specific service data. The first two are clear but the third one might need an explanation. As far as the cache is concerned, an IN service consists of two parts: code and configuration data. The code will be the same for every subscriber. The configuration data may however differ. The cache makes use of this. When it receives an IN service, it will extract the configuration data using the service's getSubscriberData method and store it separately. This way the service code will only be stored once. When a service is requested from the cache, the cache will duplicate the service code using the clone method from the Service interface, set the subscriber specific service data using the setSubscriberData method and return the cloned object. Note that cloning is necessary because two or more subscribers may be running the same service at the same time.
The deleteService method will remove the service's code and all data elements related to the service. The deleteSubscriber will remove the subscriber's DPL and all his data elements. The flushService and flushSubscriber use the two previous methods.

7.4 The IN Server

The previous chapter globally described the server package. The classes and packages it described will not be fully implemented in the prototype. The prototype will contain only the basic functionality of the Server node.

7.4.1 The Server Package

The `server` package contains a number of classes and packages. The `subscriber`, `service` and `jse` packages are described in the next subsections.

The first class is the `PublisherImpl` class. It implements an RMI server from which the JSE can request DPLs and services. Figure 7.6 shows the class diagram of the publisher. The `Publisher`

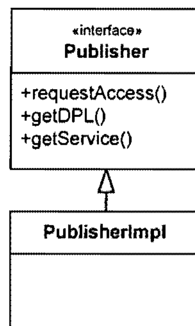

Figure 7.6: The publisher

interface defines which methods a JSE is allowed to execute. The `PublisherImpl` class implements these methods. The interface contains three remote methods: the `requestAccess`, `getDPL` and `getService` method. The first two arguments of all three methods are the same: the identification and password of the JSE. If this pair does not correspond to that in the JSE database, a `NoAccessException` is thrown. The `requestAccess` methods requires a reference to the local JSE cache as a third argument. Every JSE *has to* call this method before it can call one of the other two. If it does not adhere to this restraint, the publisher will also throw a `NoAccessException`.
The `getDPL` additionally requires the subscriber's telephone number. If it can find the subscriber in the subscriber database, it will return his DPL. If not, it will return `null`. This method will also update the subscriber's present location in the subscriber database and instruct the `CacheManager` to flush the local cache at the subscriber's previous location.
The `getService` method requires both a service key and the subscriber's telephone number as additional arguments. The service key identifies the IN service requested. If this service is not found in the service database, the publisher will throw an `InvalidServiceException`. Secondly, the subscriber is retrieved from the subscriber database. If the subscriber is not found or he is not registered as being at the JSE making the request, an `InvalidServiceException` is thrown. If, however, all information is correct, the IN service is initialized with the subscriber's data and returned to the JSE.

The second class in the package is the `CacheManager`. The cache manager implements four methods to access remote JSE caches:

- `flushService` – Instruct the specified JSE to remove the specified service from its cache.
- `flushService` – Instruct *all* JSEs to remove the specified service from their cache.

- flushSubscriber – Instruct the JSE at the specified subscriber's present location to remove all data (DPL and services) regarding the subscriber from its cache.

If the cache manager is unable to contact a JSE, that JSE is denied access to the IN server until it sends its cache's address using the publisher's requestAccess method.

7.4.2 The Subscriber package

The server.subscriber package contains the Subscriber and SubscriberDatabase class. The Subscriber class contains information about a subscriber and methods to retrieve and change this information. The following data is stored:

- The subscriber's telephone number,
- The subscriber's present location,
- The subscriber's DPL,
- A list of names of the services the subscriber is subscribed to.
- A list of subscriber-specific service data elements. For every subscribed service, the list contains one element that contains that service data that is specific to the subscriber.

The methods getNumber, getLocation, getDPL and getServices return the corresponding data elements. The following additional methods are defined:

- getDataList – Return the subscriber-specific service data for the specified IN service.
- setLocation – Set the subscriber's current location to the specified location.
- resetLocation – Set the subscriber's current location to the default location.
- addService – Subscribe the subscriber to the specified service.
- removeService – Unsubscribe the subscriber from the specified service.

The SubscriberDatabase stores Subscribers. The database is implemented using two synchronized ArrayLists. The first list stores *all* elements in the database. The second stores a subset of the first list, namely those subscribers that are presently allowed to use their IN services on this IN server. The ServiceDatabase is implemented as a singleton. The database implements the following methods:

- getHandle – Return a handle to this class' only instance.
- add – Add the specified subscriber to the database.
- remove – Remove the specified subscriber from the database.
- publish – Allow the specified subscriber to access his IN services.
- unpublish – Disallow the specified subscriber to access his IN services.
- get – Retrieve the specified subscriber from the published list.
- getTotal – Retrieve the specified subscriber from the list containing all subscribers.
- removeService – Unsubscribe all subscribers in the database from the specified service.
- whoHas – Return the list of subscribers that are subscribed to the specified service.

7.4.3 The Service Package

The server.service package contains the Java IN services and the ServiceDatabase class. The ServiceDatabase class implements a simple database using two synchronized ArrayLists. The first list stores *all* elements in the database. The second stores a subset of the first list, namely those Java IN services that may presently be downloaded from the IN server. The ServiceDatabase is implemented as a singleton. The ServiceDatabase class has seven methods:

- getHandle – Return a handle to this class' only instance.
- add – Add the specified IN service to the database.

- `remove` – Remove the specified IN service from the database.
- `publish` – Allow the specified IN service to be downloaded by a JSE.
- `unpublish` – Disallow the specified IN service to be downloaded by a JSE. The `CacheManager` is instructed to remove the service from all JSE caches.
- `getService` – Retrieve the specified IN service from the service database.
- `getDPL` – Return the list of detection points related to the specified IN service that need to be armed.

7.4.4 The JSE Manager package

The `server.jse` package contains only one class. This is the `JSEDatabase` class. It implements a simple database using two synchronized `ArrayLists`. The one list stores *all* elements in the database. The other list stores a subset of the first list, namely those JSEs that are allowed to connect to the IN server. For every JSE the database stores three elements:

- The JSE's identification: a string that uniquely identifies the JSE,
- The JSE's password: a password string,
- The JSE's cache address: to allow the IN server to flush the JSE's cache.

The class has nine public methods:

- `getHandle` – Return a handle to this class' only instance.
- `add` – Add the specified JSE to the database.
- `remove` – Remove the specified JSE from the database.
- `publish` – Allow the specified JSE to gain access to the IN server.
- `unpublish` – Deny the specified JSE to gain access to the IN server.
- `setCache` – Update the cache address for the specified JSE.
- `getCache` – Return the cache address for the specified JSE.
- `resetCache` – Reset the cache address for the specified JSE. The address is set to `null`.
- `accessGranted` – Determine whether the specified JSE has access to the IN server.

The method `accessGranted` is called by the IN server when it needs to know whether a JSE is allowed to retrieve DPLs or services from the IN server. Four conditions must be met: First, the JSE must be in the database. Secondly, the JSE must have been published. The JSE must have sent the address of its cache using the publisher's `requestAccess` method. The address is stored in the database using the `setCache` method. Finally, the supplied password must be correct.

When a JSE is unpublished, the `CacheManager` is instructed to flush the JSE's local cache. Furthermore, the JSE's cache address is reset and the subscriber database is instructed to reset the present location of all subscribers currently in the JSE's service area. The JSE is no longer allowed to access the IN server. When `remove` is called, the JSE is first unpublished and then its entry is entirely removed from the database.

7.5 Interfaces

7.5.1 The SCF - JSE interface

Section 2.4 provides an introduction to Erlang and Jive. The Jive server provides a communication mechanism between an Erlang application and a Java application.

Figure 7.7 shows how a Java-Erlang interface is implemented using Jive. The procedure is the same for the `ScfGateway` and the `SSF` class.

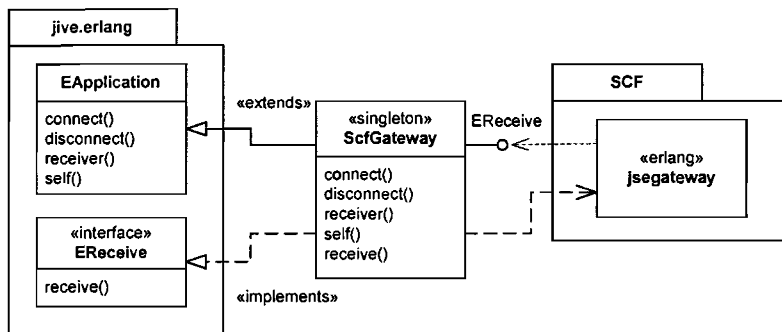

Figure 7.7: The interface between the SCF and the JSE

The first thing a Java application has to do to communicate with an Erlang application is connect to the Jive server. To be able to do this, it has to extend the `EApplication` class. This class' methods enable the Java application to connect to and disconnect from the Jive server. When connected to the Jive server, a Java application can send messages to an Erlang processes, spawn Erlang processes and execute Erlang functions.

Every time an `EApplication` connects to the Jive server, the Jive server starts an Erlang process. The Java application knows this Erlang process as the self-process. The `self` method returns the wrapper-id (self-PidId) of this self-process. An Erlang application has to know this PidId to be able to send messages to the Java application.

A Java application has to implement the `EReceive` interface and register this implementation with the Jive server's `EReceiver` object in order to receive messages from an Erlang application. The `EApplication`'s `receiver` method returns this `EReceiver` object. After registering, the Java application receives an identification (Id), which it has to send to the Erlang application along with its self-PidId.

Before an Erlang process can receive messages from a Java application, it has to register its process-id (Pid) with the Jive Server. It receives a wrapper-id (PidId), its address, which it should send to the Java application. The Java application receives this PidId as an `EProcess`. Figure 7.8 shows the structure.

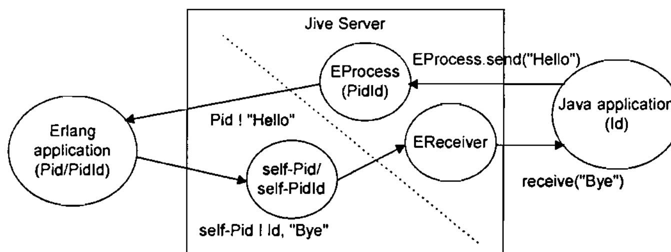

Figure 7.8: The Jive server structure

7.5.2 The IN Server - JSE interface

Section 3.3 shows an overview of how mobile code could be implemented using Java. Currently Java RMI is the best candidate for implementing the server - JSE interface.

Section 3.3.3 describes the general operation of Java RMI. A Java RMI server consists of an interface and a class. The interface defines which remote methods the RMI client may invoke on the RMI server. It has to extend the `Remote` interface which signifies that its methods can be called from a remote Java machine. Every remote method in the interface must be defined as being able to throw the `RemoteException` exception. The `Server` class provides the implementation of the interface and extends the `UnicastRemoteObject` class. Figure 7.9 shows the class diagram.

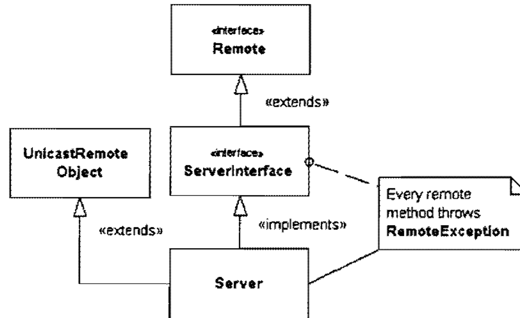

Figure 7.9: The class diagram of an RMI server

Between the JSE and the IN server, four types of requests and responses are sent.

- The JSE requests access to the IN server.
- The JSE requests a DPL from the IN server.
- The JSE requests an IN service from the IN server.
- The IN server requests the JSE to delete a number of entries from its cache.

To implement these messages, the IN server and the JSE both need to implement an RMI server as well as an RMI client as shown in figure 7.10.

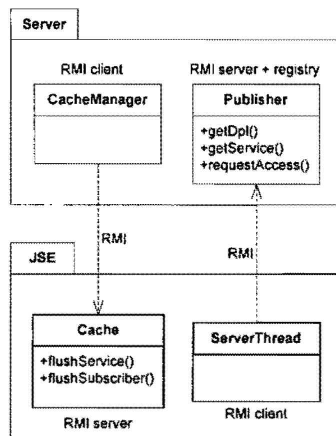

Figure 7.10: The interface between the Server and the JSE

The prototype will take advantage of the fact that the interfaces of both RMI servers are not likely to be changed. This eliminates the need for dynamic stub downloading: The class files of both the RMI servers' stubs will be *locally available* to the RMI clients.

Figure 7.11 shows the situation in which the IN server (`Publisher`) implements the RMI server and the JSE (`ServerThread`) implements the RMI client. When comparing this to figures 3.6 and

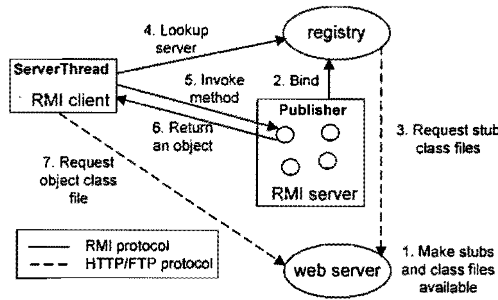

Figure 7.11: RMI interaction between the `Publisher` and the `ServerThread`

3.7, notice that after the RMI client has retrieved a reference to the RMI server from the registry, it does not have to download the stub's class files.

In section 7.3.6, it was explained that the JSE will not be granted access to the IN server before it has sent the address of its local cache. What this "address" exactly is was not explained. The "address" is in fact a reference to the cache's RMI server. This procedure is called "server callback". Refer to figure 7.12.

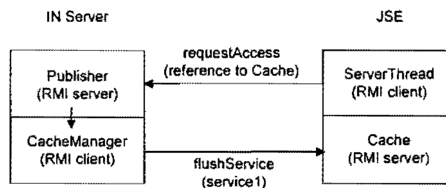

Figure 7.12: RMI server callback

7.6 IN Services

An IN service consists of four parts:

- the service code,
- the service DPL,
- the service specific service data and
- the subscriber specific service data.

A Java IN service by definition extends the abstract `ServiceClass` class and thus implements the `Service` interface. The `Service` interface defines a number of methods the IN service has to implement. Some of these have a default implementation in the `ServiceClass`. However, the IN service is free to override these. The `execute` method, that has to be implemented by every service, must send the continue message to its SSF object once it finishes.
Every Java IN service has one or more dynamic breakpoints that trigger the service. These breakpoints are stored in an `ArrayList`. A dynamic breakpoint is implemented by the `DBP` class. The `DBP` class stores the five characteristics of a dynamic breakpoint:

- The call state at which the breakpoint is to be installed. This can be an originating or terminating state.

- The call data conditions that have to be fulfilled to trigger the breakpoint.
- The service key of the service that has to be executed when the breakpoint is triggered.
- The data elements that the SSF node has to provide to the JSE when the breakpoint is triggered.
- The telephone numbers that have to be added to the analysis set.

The call state is an integer as defined in `se.ericsson.etm.rjs.common.INString`. Service key and telephone numbers are stored in a `String`. The data to be provided and the telephone numbers are stored in `ArrayLists`. The conditions are stored in an `EventDataList`. It uses an `ArrayList` to store its elements. Every element consists of a name-string and a value-string. The name-string stores the name of the call data element and the value-string stores its value.

The service specific service data is that part of the service data that is the same for every subscriber. In a VPN service, the numbering plan is part of the service specific service data. In the prototype, this data is considered part of the service code. This means that it can not be changed at runtime. Of course, this is not acceptable in real life.

The subscriber specific service data is that part of the service data that is different for every subscriber. A subscriber's barring list is an example. The `Service` interface defines the `getSubscriberData` method to extract this data from the service object. The service implementation is free to encode or process the data before it is returned to the calling object. In this way, the privacy of the subscriber can be guaranteed. This may not be entirely true as stated in section 4.3 on service security.

The prototype does not impose any restrictions on the structure of an IN service as long as all the classes the service depends on are available on the web server for download by the JSE.

The prototype includes two example Java IN services: the Java VPN service and the Java police service. Refer to appendix E on their usage.

7.7 Common Classes

Several classes exist that are required at both the server node and the JSE node. These classes can be divided in two groups. The first group contains the RMI interfaces and the RMI stub classes. These class files are distributed so that the RMI clients do not have to download them from the web server. These classes are:

- the `Publisher` interface in the `server` package,
- the `PublisherImpl_Stub` class in the `server` package,
- the `Cache` interface in the `cache` package and
- the `CacheImpl_Stub` class in the `cache` package.

The second group contains all other classes and interfaces that both the server and the JSE use. They are bundled in the `common` package and are also distributed to all server and JSE nodes. The `common` package contains:

- The `Service` interface – Defines which methods an IN service *must* implement.
- The `ServiceClass` class – Contains a default implementation for some of the methods defined in the `Service` interface.
- The `DBP` class – Implements a dynamic breakpoint.
- The `EventDataList` class – Implements a list that can store name-value pairs. Every element in the list consists of a name-field and a value-field.
- The `INString` interface – Defines a number of frequently used string constants.

- The `SSFInterface` interface – Defines which `SSF` methods an IN service is allowed to call. It defines the possibilities and restrictions of an IN service regarding call control.
- The `NoAccessException` class – Implements the exception that is thrown by the `Publisher` when access is denied to a JSE.
- The `InvalidServiceException` class – Implements the exception that is thrown by the `Publisher` when a requested service is not available at the IN server.

7.8 Limitations of the prototype

The RJS prototype was built to show how an architecture with roaming IN services can be implemented using Java. The prototype was not intended to be a full grown system. The previous sections showed what the prototype is capable of. This section will point out its limitations.

7.8.1 The IN Server

The IN server lacks a proper database with interface which would allow services, subscribers, JSEs and their parameters to be edited. Currently, all this data is hard-coded. Changing the data would mean having to take the server down. There is one exception to this. The IN server does allow new services to be introduced. This was done to demonstrate that Java allows new code to be introduced in a running program. The user is prompted for the name of the class of the service to be added. It is not allowed to specify a name of a class that has already been loaded. It must be a new class. The IN server identifies services not by their class name but by their service key. Upgrading a service `x`, implemented in class `X1`, therefore means introducing a new class `X2` which also implements the service `x`. Note that the old class `X1` is *not* unloaded leading inevitably to shortage of memory. Unloading a class is theoretically possible but will, in combination with RMI, prove tricky as section 7.9 will show.

The RMI connection between the IN server and a JSE is not secure. A simple TCP/IP connection without encryption is used. However, with RMI it is possible to install a custom RMI socket factory which creates sockets that, for instance, encrypt or compress the data sent across. The reason that the prototype does not have a socket factory implementing an SSL (Secure Socket Layer) is that a non-commercial implementation of an SSL is not available.
At this moment, Sun does not provide an SSL implementation in its JDK. Development of a Java RMI Security Extension is on its way [31]. The extension adds features like communication integrity, communication confidentiality, server and client authentication and delegation to RMI. The EspreSSL[2] team is another party developing a free SSL implementation. The product is however not finished and therefore not available.

The server is unable to immediately detect when a JSE goes down. This could easily be solved by periodically polling every JSE. Java does not provide a mechanism that informs an RMI server when one of its RMI clients goes down.

The IN server does not allow a subscriber to change his subscriber specific service data. To allow this, the IN server and its RMI interface would have to be adapted.

7.8.2 The JSE

The prototype JSE can connect to one IN server only. In order to be able to connect to multiple IN servers, the `ServerGateway` and `ServerThread` classes will have to be altered so that they initialize and connect to the appropriate server. The JSE will have to keep a list of the IP address - port pair of every IN server.

[2]For more information on EspreSSL and the current project status please refer to http://www.vonnieda.org/jSSL

Currently, the JSE is not protected against erroneous code. When a service ends up in an endless loop or forgets to send the continue message, the JSE is lost. A watchdog thread would solve this problem. A watchdog thread is a thread with the highest priority monitoring the other threads. It will sleep most of the time but wake up periodically to check whether all is well. In this case it would have to check two things. The first is whether the `JobHandler` thread is still running properly. If a service has been executing too long, it should stop the thread and restart the `JobHandler`. Secondly, it has to check whether calls are terminated properly. If a service has not sent the continue message, this has to be detected.

7.9 Unloading classes

Unloading a class from a JVM is tricky. Section 12.8 of [14] states that "a class may not be unloaded while any instance of it is still reachable" and that "a class or interface may not be unloaded while the `Class` object that represents it is still reachable". In [28] a clarification was published saying that "a class or interface may be unloaded if and only if its class loader is unreachable".

In practice, unloading a class involves the following steps:

1. Place the class file that should be unloaded in a separate location. Do not add this location to the classpath.
2. Create a custom classloader (subclass the `java.lang.ClassLoader`) and program it to load classes from the location mentioned above.
3. Load the class with the custom classloader.
4. When it is time to unload the class, discard the custom classloader making it unreachable.
5. Discard all instances of the class making them not reachable.
6. Eventually, the garbage collector will unload all the classes loaded by the custom classloader.

If desirable, the class can now be reloaded. Create a new instance of the classloader and instruct it to load the class.

How this mechanism works in the distributed environment of the RJS prototype, or in any distributed environment for that matter, is left for further study. Several question arise:

- If the IN server should be able to unload every IN service class separately, is it necessary to create a classloader instance for every IN service class? Or is it possible and feasible to use one classloader instance to load all IN service classes and reload them with every update?
- RMI transports instances of the class to be unloaded from the IN server, to distributed JSE nodes using the `RMIClassLoader`. Is it necessary, and if so, possible, to discard and recreate the RMI classloader in order to unload the class at both IN server and JSE node?

7.10 Service Interworking

Service interworking refers to two or more services used together during a call, either simultaneously or in sequence [35]. One speaks of service interaction when two services are used together but the result is not exactly the sum of the two single services. In most cases service interaction yields a negative result.

The RJS prototype also shows a form of service interworking. When one of the armed TDPs (DP1 or DP12) is triggered, a control relationship is established between the SSF and the JSE. An SSF

can have only one active control relationship. Therefore, when DP1 or DP12 is triggered all other TDPs are temporarily deactivated until the control relationship is broken. The JSE breaks the relationship when the subscriber is not subscribed to Java IN services, when a Java IN service sends the "transfer" message or when the call is released. Until that moment all TDP-activated services can not be used.

Java IN services do not depend on TDPs for their activation. They use EDPs as was explained earlier. However, the "conventional" IN services do depend on TDPs. This means that a subscriber that uses Java IN services can not use "conventional" IN services and vice versa. The RJS prototype partly solves this problem by introducing an escape service. The escape service breaks the control relationship. This deactivates the Java IN services and reactivates the "conventional" IN services. A subscriber can activate the escape service by dialing the escape number defined in the configuration file.

Service interaction happens with the Java VPN service. The Java VPN service translates a two-digit telephone number to a six-digit number. This six-digit number is used by the registration IN service to forward the call to the appropriate telephone set. Refer to section 7.2. A "conventional" IN service is used immediately after a Java IN service. This is only possible when the control relationship between the JSE and SSF is broken. This means that the Java VPN service has to send the "transfer" message. This also means that from that moment, the subscriber is no longer able to use any other Java IN service for the duration of the call.

7.11 Summary

The RJS prototype is up and running. The IN server, the JSE and the logic in the SCF have been implemented. The prototype has some limitations. A proper database is missing in the IN server and the interface between the server and the JSEs is not secure. The JSE is at the moment not capable of connecting to multiple IN servers. Unloading a class in Java is tricky and may be a problem.

Chapter 8

Conclusions

This final chapter presents the results of the graduation project. It describes the conclusions of the research into mobile code and the security involved. It summarizes the features and limitations of the prototype. Finally it gives some recommendations for future work.

8.1 Problem Definition

A subscriber can no longer activate his Intelligent Network (IN) services when he leaves the service area of his home network's Service Control Function (SCF). The CAMEL[1] standard solves this problem by establishing a signaling connection between the visited Service Switching Function (SSF) and the home SCF. This report presented an alternative solution: "roaming IN services". The IN service and subscriber data are transported from the home SCF to the visited SCF. The IN service is executed locally.

The following sections summarize the answers found to the questions raised in the Introduction.

8.2 Architecture of the RJS system

The Roaming Java Services (RJS) architecture consists of four nodes. Two of these nodes, the SSF and SCF nodes, already exist in today's telephone system. The functionality of the SSF has not changed in the new architecture. The SCF, however, was reduced to a relay node. The SCF's former task, which is service execution, is to be performed by a new node; the Java Service Environment (JSE). The fact that the SCF is implemented in Erlang and the JSE was to be implemented in Java, led to the creation of a separate JSE node. The SCF and JSE do however have a one to one relation. One JSE connects to one SCF and vice versa. Therefore it is also possible to integrate the two nodes. The Server node is a centralized storage facility of IN services and subscriber data. JSE nodes may access it when a subscriber requires an IN service. Refer to sections 5.1 and 5.2 for more details.

The mobile code concept is presently not implemented in the IN architecture. However, the RJS architecture does implement the concept. In [38] Vigna gives a mobile code taxonomy. It is summarized in section 3.1 of this report. When compared to the mobile code taxonomy, the RJS architecture comes closest to the "code on demand" paradigm. The JSE requests a code fragment that it does not have available locally. The code is executed on receipt. The architecture also has

[1]CAMEL: Customized Applications for Mobile network Enhanced Logic

elements of the "mobile agent" paradigm. The code sent from IN Server to JSE is after all already initialized and thus has formally started execution. This is known a "strong mobility".

The DPL is downloaded from the IN server to the JSE as soon as the subscriber registers at the network. This eliminates transport delays when a call is initiated and the Event Detection Points (EDP) have to be armed. During the project it was assumed that the transport delay of one IN service was acceptable, hence the IN service is downloaded after its detection point has been triggered. Because IN services are not subject to frequent change, they are cached locally by the JSE instead of being immediately deleted after use. The IN server is allowed to delete Detection Point Lists (DPL) and IN services from the local caches.

IN services that are frequently used, are suitable for use in the RJS architecture. The local cache eliminates repeated transport. The Virtual Private Network (VPN) service is used as an example. Refer to sections 5.3 and 5.4.

8.3 Security Aspects

While flexibility is the major advantage of mobile code, the strict security measures necessary are its major disadvantage. Refer to section 3.1. When executing code in a foreign environment, both the environment and the code need to be protected. Three types of harmful code exist:

- Code written to be intentionally damaging - for instance a virus.
- Code that has been (un)intentionally altered - for instance because of errors during transport.
- Code that contains programming errors.

The first two risks can be eliminated by using digital signatures in combination with a security policy. The risk of the third category can be reduced by placing an interpreter between the application and the operating system. For more details on host security, refer to section 4.2.

Attacks on code can be divided in two categories:

- Attempts to extract information from the code.
- Attempts to influence the program flow.

Without trusted hardware it remains impossible to *prevent* tampering. All security measures attempt to detect foul play or prevent meaningful alterations. Examples of mentioned security measures are obfuscation, traces and detection objects. Refer to section 4.3.

Costs and benefits have to be weighted. In some cases legal constructions may provide sufficient protection.

Java makes use of an interpreter to shield the operating system from the applications. An access controller puts the running applications in protection domains depending on their origin and the signatures attached. The security policy grants each domain certain permissions. The Java Cryptography Architecture (JCA) allows use of cryptographic services implemented by a cryptographic service provider (CSP). This shows that Java is well equipped to protect the host against attacks. Java itself does not include measures to protect the code. Refer to section 4.4 on Java and security.

When mapped to the RJS architecture it seems that the Java environment is capable of protecting the JSE node. The IN services and personal data of the subscribers may be at risk. However, with the close cooperation of telecom operators nowadays, legal measures may suffice.

8.4 The RJS prototype

The prototype implements the RJS architecture. It uses the Rapid Service Prototyping (RSP) software to simulate the telephone network. Some modules of the SCF were changed but changes

were kept to a minimum. A gateway to the JSE was added to the SCF. The gateway is for shielding the communication specifics from the other modules. The Jive application implements the Erlang - Java interface. Refer to sections 7.2 and 7.5.1.

The JSE is implemented in Java and therefore its code, but also downloaded IN services, adhere to the security policy of the local environment. Not all security measures available in Java have been implemented in the prototype.

The JSE successively handles the call events coming from the SSF and SCF. This is faster compared to a multi-threaded (parallel processing) solution. Interaction with the IN server however is handled by multiple threads. This way, multiple connections to the IN server exist simultaneously. These threads are ready-made and need only be initialized before activation. Refer to section 6.6.

The communication with the IN server is provided by Java Remote Method Invocation (RMI). Several other options exist to transport code from one environment to another, but RMI was chosen. CORBA does not allow objects to be passed by value and RMI-IIOP[2] does not have any significant advantages compared to RMI. The disadvantage of RMI is that it uses a web-server to publish the class files. Refer to sections 3.3 and 7.5.2.

Although a secure connection between the JSE and IN server is very important and although RMI supports the use of a Secure Socket Layer (SSL), it has not been implemented in the prototype. The reason for this is that a non-commercial implementation of an SSL is not available. Sun is currently developing a security extension to RMI. It will implement features like communication integrity, communication confidentiality and server/client authentication. Refer to section 7.8.

The question whether Java is suitable to implement the RJS architecture remains partly unanswered. How does Java behave if the system is heavily loaded? In a telephone system reliability and short response times are important. It is also not clear whether Java is able to reliably unload classes in a distributed system. Refer to section 7.9.

8.5 A monopolistic environment

The report showed that a centralized IN server and the mobility concept are useful in a multi-operator environment. The IN services are transported to the network of a fellow operator. They are executed and are temporarily cached in that network. Suppose there was only one operator, would the architecture lose its added value?

Thörner [35] describes three phases in the development of an IN. In the first phase the intelligence is separated from the switching function. The IN services are moved from the local switching nodes to an SCF. In the second phase intelligence is flexibly allocated in the network. Some services reside in the terminal, some in the dedicated SCF and some in the network nodes. The third phase implements the "intelligence on demand"-concept. A service may be retrieved from a server whenever the subscriber needs it. It is downloaded into either the local exchange, the SCF or into the terminal.

Thörner said that in the future subscribers of a Personal Communication Network (PCN) will have full mobility. They have a personal number combined with a personal service profile. The profile contains the IN services and the personal data. It is stored on a server. When the subscriber goes to a new place, the profile accompanies him. The architecture he describes is very similar to the RJS architecture. However, Thörner places no restrictions on where the service is executed.

The project focused on a multi-operator environment. The architecture also has advantages in a single-operator environment. The JSE can be located in a dedicated node, but also in a local exchange or even a terminal. When the subscriber moves to a new terminal, his profile would accompany him to that terminal and/or the local exchange serving the terminal and/or a dedicated

[2]RMI-IIOP was officially released in June 1999

node serving the local exchange.

8.6 Recommendations for future work

Since the SCF node's functionality is drastically reduced, the question arises whether it can be left out completely. The SSF node can be modified to communicate directly with the JSE. In the prototype the RSP software simulates the telephone system. A gateway to the JSE node was added to relay communications. This gateway is part of the SCF node but can easily be transferred to the SSF node. Further study is needed to determine whether the SCF can completely be replaced by the JSE in a telephone system.

Java RMI was best suited to implement a prototype. During the development and implementation additional disadvantages of RMI surfaced. Disadvantages of RMI are:

- It needs a http or ftp-server to distribute the class files, only anonymous logins are allowed.
- Neither the security extension nor a non-commercial SSL implementation is available.
- Java does not provide explicit means to unload classes. To unload a class, class loaders will have to be altered. This also applies to the RMI class loader.
- It is not possible to get a notification when a remote server goes down.

Research into other communication standards, like the new CORBA 3, is desired.

To come to an operational implementation some other aspects will have to be examined:

- On the local network the transport delays are minimal. How big or small will transport delays be in a telephone network?
- Data security can not be guaranteed by means of a software solution. Operators will have to come to a mutual understanding and juridically solve this issue.
- Java has the means to implement the RJS architecture. But it is unclear how Java behaves when the system is heavily loaded. Tests will have to be conducted to show whether Java is suitable to implement a time-critical system.
- It is not possible to explicitly unload classes in Java. Does this make Java unsuitable? Further study is needed.
- A proper database implementation will have to be developed for the IN server.

Appendix A

The Basic Call State Model

The Basic Call State Model (BCSM) is described in chapter 4 of the ITU Recommendation Q.1214 [34]. The model is implemented in a somewhat changed form by the RSP software. It is described in [10].

A.1 The Originating Basic Call State Model

Figure A.1 shows the Originating BCSM according to the ITU standards. The numbered squares represent DPs and the blocks represent PICs.

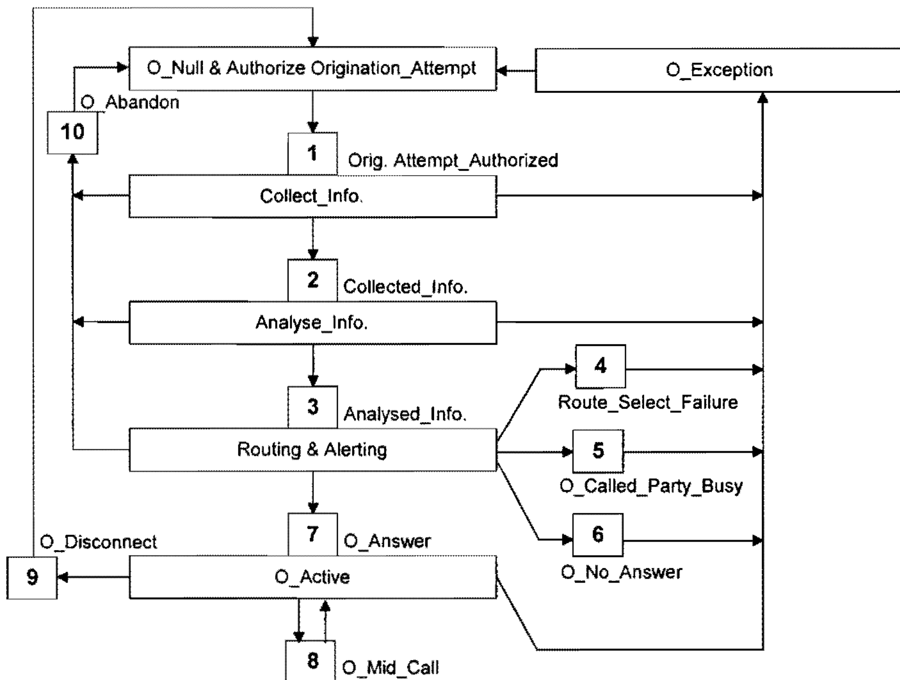

Figure A.1: The Originating BCSM for CS-1

O_Null & Authorize_Origination_Attempt:
The line/trunk is idle. When the originating party tries to place an outgoing call (e.g. phone off-hook) his authority/ability is verified.

DP 1 Origination Attempt_Authorized:
The originating party is trying to place a call and the authority/ability to do so has been verified.

Collect_Information:
The initial information string is being collected from the originating party.

DP 2 Collected_Information:
The complete initial information string is available from the originating party.

Analyse_Information:
The information is analysed and/or translated according to the dialling plan in order to determine the routing address and call type.

DP 3 Analysed_Information:
The routing address and its nature is available.

Routing & Alerting:
The routing address and call type are interpreted, the authority of the originating party to place this call is verified and the call is processed by the T-BCSM.

DP 4 Route_Select_Failure:
All routes are busy.

DP 5 O_Called_Party_Busy:
The terminating side is busy.

DP 6 O_No_Answer:
The terminating side does not answer.

DP 7 O_Answer:
The call is accepted and answered by the terminating party.

O_Active:
The connection between the originating and terminating party has been established.

DP 8 O_Mid_Call:
A service feature request is received from the originating party.

O_Exception:
An exception has occured and is now being handled.

DP 9 O_Disconnect:
One of the parties has disconnected the call.

DP 10 O_Abandon:
The call is abandoned before a connection could be established.

A.2 The Terminating Basic Call State Model

Figure A.2 shows the Terminating BCSM according to the ITU standards. The numbered squares represent DPs and the blocks represent PICs.

T_Null & Authorize_Termination_Attempt:
The line/trunk is idle. When an incoming call arrives, the authority/ability of the terminating party is verified.

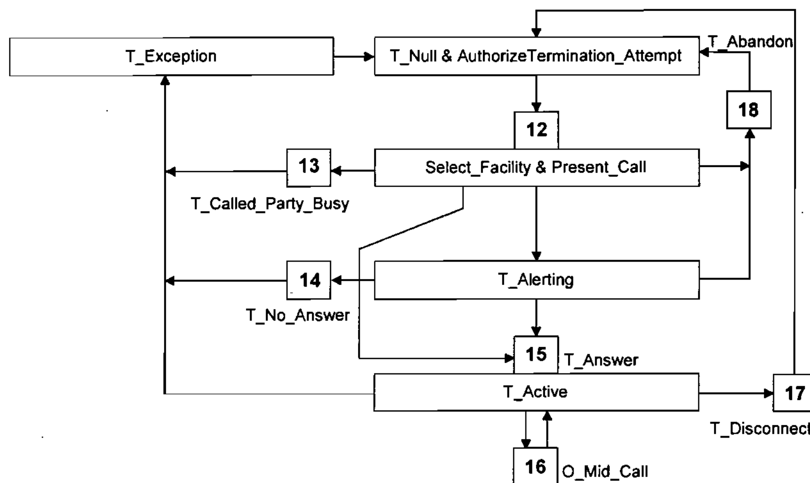

Figure A.2: The Terminating BCSM for CS-1

DP 12 Termination_Attempt_Verified:
An indication for an incoming call is received and the terminating party's authority/ability is verfied.

Select_Facility & Present_Call:
Select an available resource from the resource group and inform the terminating party of an incoming call.

DP 13 T_Called_Party_Busy:
The terminating party is busy.

T_Alerting:
Waiting for the call to be answerd by the terminating party. The originating party is informed that the terminating party is being alerted.

DP 14 T_No_Answer:
The ringing timer has expired before the terminating party answered.

DP 15 T_Answer:
The call is accepted and answered by the terminating party.

T_Active:
The connection between the originating and terminating party has been established.

DP 16 T_Mid_Call:
A service feature request is received from the terminating party.

T_Exception:
An exception has occured and is now being handled.

DP 17 T_Disconnect:
One of the parties has has disconnected the call.

DP 10 T_Abandon:
The call is abandoned before a connection could be established.

A.3 RSP Originating Basic Call Model

Figure A.3 shows the Originating BCM as implemented by the RSP software. It was taken from [10].

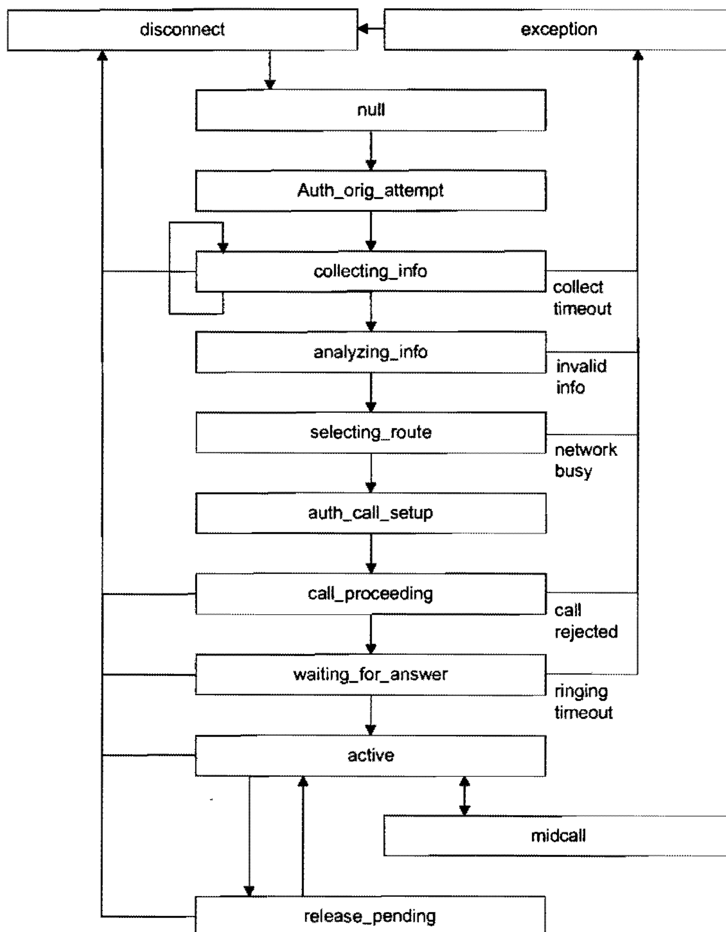

Figure A.3: The Originating BCM for the RSP software

null:
Start of call.

auth_orig_attempt:
Check if the originating call is allowed for this subscriber.

collecting_info:
Get information from the originating side.

analyzing_info:
Analyze the collected info.

selecting_route:
Select an outgoing route.

auth_call_setup:
Check if call setup is allowed.

call_proceeding:
Set up the call to the terminating side.

waiting_for_answer:
Wait for the Address Complete message from the other side.

active:
A speech connection between the originating and the terminating party is established.

midcall:
A hookflash is received.

release_pending:
The terminating subscriber has terminated. Wait for re-answer or disconnection of the originating side.

exception:
A fault occured.

disconnect:
The originating side disconnected.

A.4 RSP Terminating Basic Call Model

Figure A.4 shows the Terminating BCM as implemented by the RSP software. It was taken from [10].

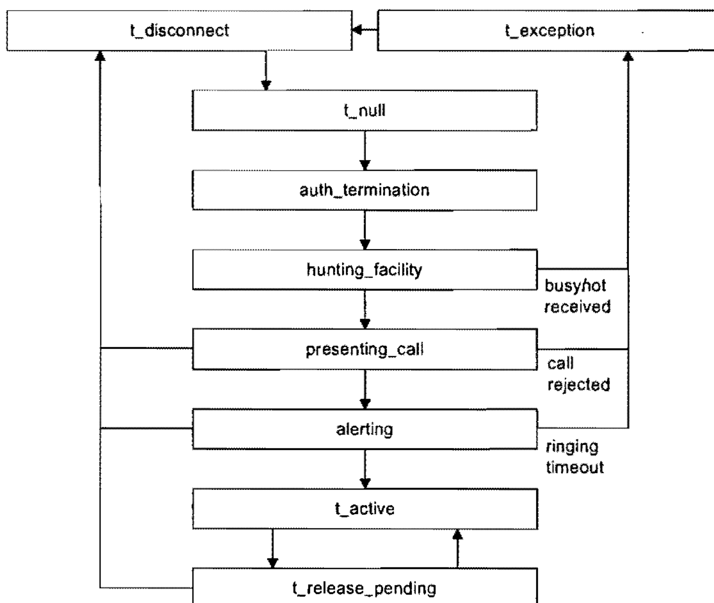

Figure A.4: The Terminating BCM for the RSP software

t_null:
Start of call.

auth_termination:
Check if the call is allowed to terminate.

hunting_facility:
Select an outgoing device.

presenting_call:
Offer the call to the outgoing side.

alerting:
The outgoing side is being alerted, wait for answer.

t_active:
A speech connection between the originating and the terminating party is established.

t_release_pending:
One of the parties has terminated. Wait for the other party to terminate as well.

t_exception:
A fault occurred.

t_disconnect:
The call is disconnected.

Appendix B

Information Flows

B.1 Information Flows in the RSP

This section describes the information flows between the SCF node and the SSF node in the RSP (refer to section 2.5).

Messages from the SSF to the SCF:

- "initiate" – Initiates the communication session between the SSF and SCF.
- "simulation end" – Terminates the communication session between the SSF and SCF. The SCF should respond with "simulation ended".
- "provide" – Indicates that a static breakpoint has been triggered.
- "event" – Indicates that a dynamic breakpoint has been triggered.

Messages from the SCF to the SSF:

- "update breakpoint set" – Set a static breakpoint in the SSF.
- "update breakpoint unset" – Unset a static breakpoint in the SSF.
- "update analysis set" – Add a telephone number (one or more) to the number analysis set.
- "update analysis unset" – Remove a telephone number (one or more) from the number analysis set.
- "create" – Create a call (a virtual call) with the given service data.
- "register" – Register a subscriber on the given telephone set.
- "un register" – Unregisters a subscriber.
- "lineMonitor" – Perform a linemonitor on the given telephone set.
- "load data" – Substitute the call data by the provided new call data.
- "set monitor" – Set a dynamic breakpoint in the SSF.
- "unset monitor" – Unset a dynamic breakpoint in the SSF.
- "transfer" – Request the SSF to terminate communications. The SSF will respond with "simulation end".
- "continue" – Tells the SSF to continue with the next state in the call state model.
- "message" – Display specified message on the telephone's display.
- "input" – Prompt the user for information.
- "scf interrupt" – Force the call to the state "scf interrupt". The SCF takes immediate control of the call.

Refer to [10] for the detailed message syntax.

B.2 Information Flows in the roaming IN services architecture

This section describes the information flows (IF) in the roaming IN services architecture as presented in chapter 5.

B.2.1 IFs between the SSF and the SCF

IFs originating from the SSF are:

- Registration message – The mobile terminal tries to register with the network.
- DP1/DP12 triggered – Either DP1 or DP12 has been triggered. The EDPs need to be armed.
- EDP triggered – An EDP has been triggered indicating that a roaming IN service is requested.
- Standardized IFs as described in Recommendation Q.1214 section 6.4 [34]. See also section B.3.

IFs originating from the SCF are:

- Standardized IFs as described in Recommendation Q.1214 section 6.4 [34]. See also section B.3.

B.2.2 IFs between the SCF and the SE

IFs originating from the SCF are:

- Registration message – The mobile phone tries to register with the network.
- DP1/DP12 triggered – The SCF requests the subscriber's DPL.
- EDP triggered – An EDP has been triggered indicating that a roaming IN service is requested.
- Standardized IFs as described in Recommendation Q.1214 section 6.4 [34]. See also section B.3.

IFs originating from the SE are:

- DPL available – The SE passes the subscriber's DPL to the SCF.
- Standardized IFs as described in Recommendation Q.1214 section 6.4 [34]. See also section B.3.

B.2.3 IFs between the SE and the IN server

IFs originating from the SE are:

- Registration message – The SE requests the subscriber's DPL.
- Request IN service – The SE requests a roaming IN service.

IFs originating from the IN server are:

- Registration response – The server returns either the subscriber's DPL or a Negative Acknowledgement (NACK).
- IN service – The server returns the requested IN service or a NACK.
- Remove DPL – The server instructs the SE to remove one or more DPLs from the local cache.

Information Flow	Camel	JSE	VPN
Activity Test	X	X	
Activity Test Response	X	X	
Apply Charging	X		
Apply Charging Report	X		
Call Information Report	X	X	
Call Information Request	X	X	
Connect	X	X	X
Connect to Resource	X		
Continue	X	X	X
Disconnect Forward Connection	X		
Establish Temporary Connection	X		
Event Report BCSM	X	X	X
Furnish Charging Information	X	X	
Initial DP	X	X	X
Release Call	X	X	X
Request Report BCSM Event	X	X	X
Reset Timer	X	X	X
Send Charging Information	X		

Table B.1: Information flows in CAMEL, the JSE and the VPN service

- Remove IN service – The server instructs the SE to remove one or more IN services from the local cache.

B.2.4 IFs between the SE and the local cache

IFs originating from the SE are:

- DP1/DP12 triggered – The SE requests the subscriber's DPL.
- Request IN service – The SE requests a roaming IN service.
- Remove DPL – The SE instructs the cache to remove one or more DPLs from the local cache.
- Remove IN service – The SE instructs the cache to remove one or more IN services from the local cache.

No IFs originate from the local cache.

B.3 Standardized Information Flows

Not all IFs between the SSF and SCF defined in CS-1 should be implemented in a roaming IN services system. Some IFs originating from the SCF could cause unwanted behavior in the visited network. The "Initiate Call Attempt" is an example. The table below shows which IFs (between SSF and SCF) are implemented by the CAMEL standard [3], the ones that will be implemented in the roaming IN services prototype and the ones necessary to implement a VPN-service [9].

Appendix C

Activity Diagrams

C.1 The Server Node

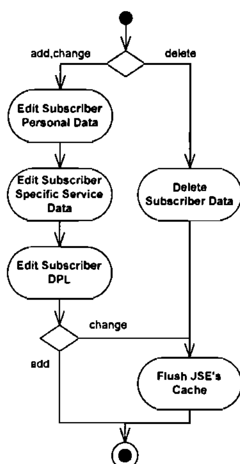

Figure C.1: Activity diagram of the "Manage Subscribers" use case

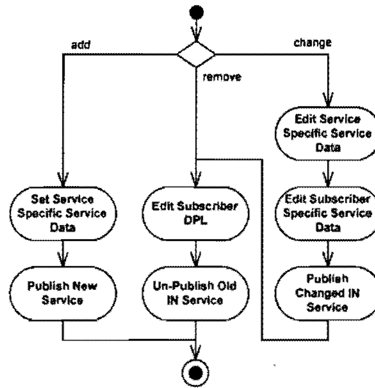

Figure C.2: Activity diagram of the "Manage IN Services" use case

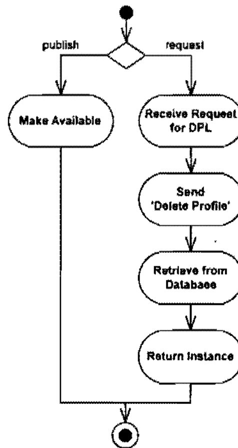

Figure C.3: Activity diagram of the "Publish DPLs" use case

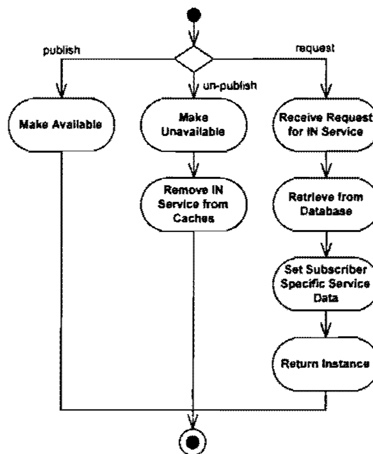

Figure C.4: Activity diagram of the "Publish IN Services" use case

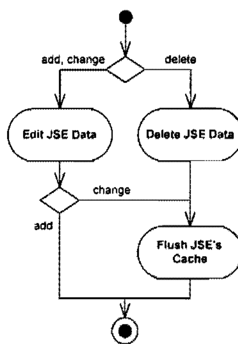

Figure C.5: Activity diagram of the "Manage JSE Access" use case

C.2 The Java Service Environment Node

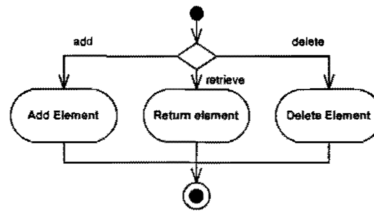

Figure C.6: Activity diagram of the "Manage Cache" use case

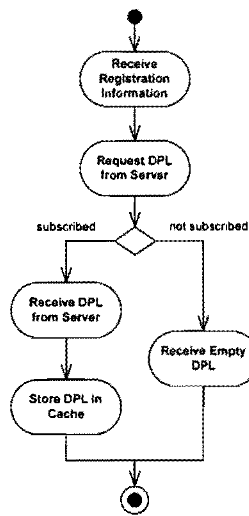

Figure C.7: Activity diagram of the "Register Subscriber" use case

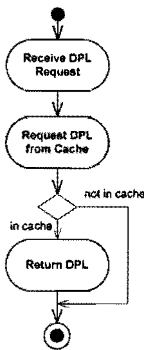

Figure C.8: activity Diagram of the "Arm Dectection Points" use case

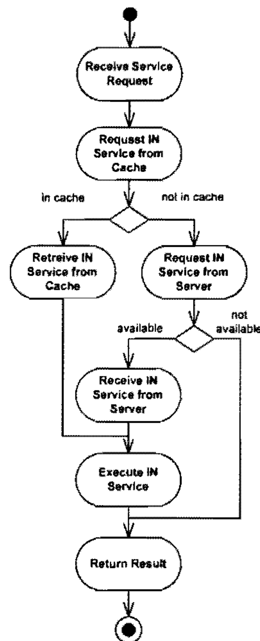

Figure C.9: Activity diagram of the "Invoke IN Service" use case

Appendix D

Sequence Diagrams

The following pages show the sequence diagrams for the Java Service Environment. The first diagram shows the actions taken in the JSE when a subscriber registers with the network. The second shows the actions as the subscriber goes off hook (DP1 is triggered) and as a call for a subscriber registered in this network comes in (DP12 is triggered). The third sequence diagram shows what happens when a subscriber invokes a roaming IN service.

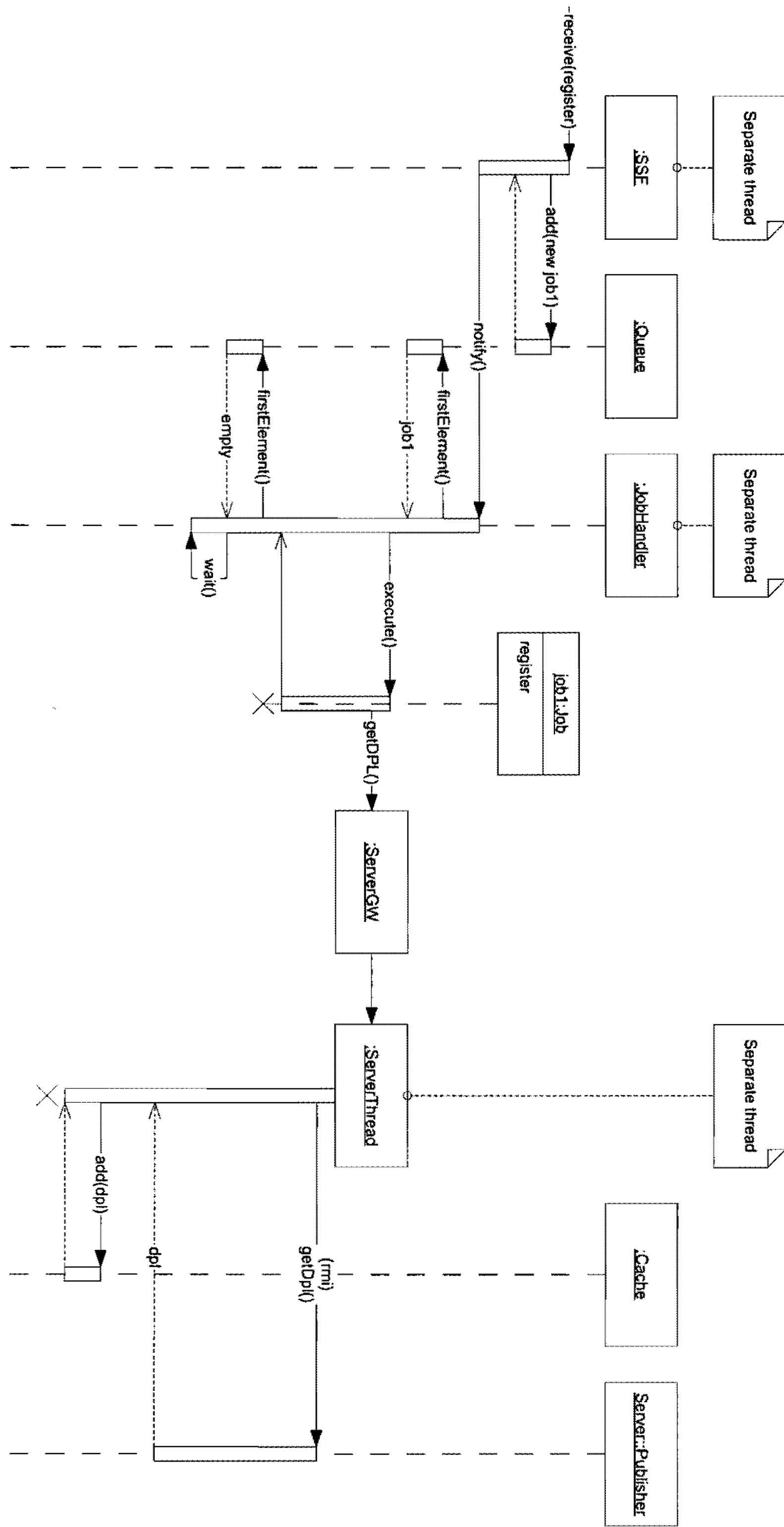

Figure D.1: Sequence diagram for registering a subscriber

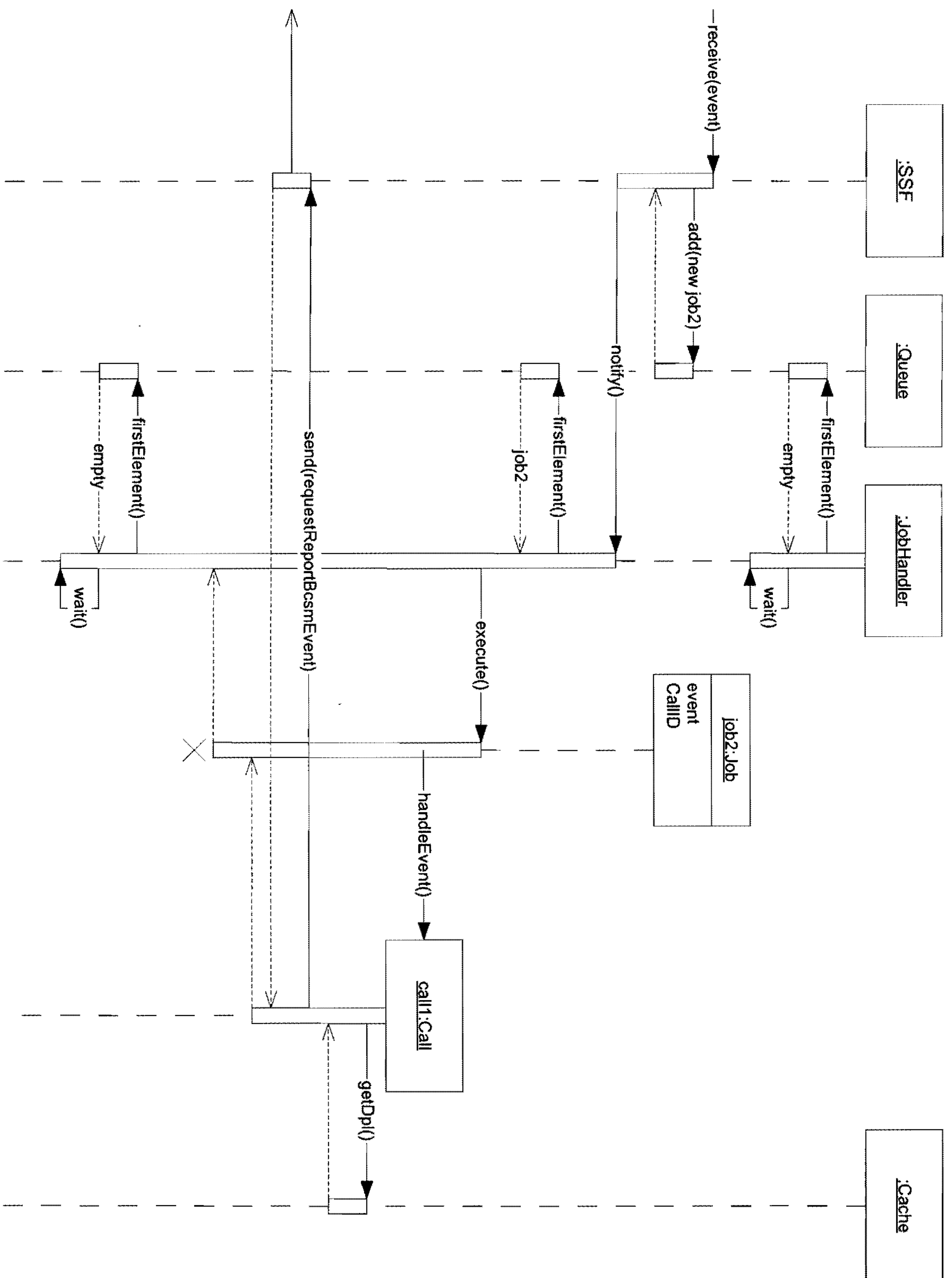

Figure D.2: Sequence diagram for arming the detection points

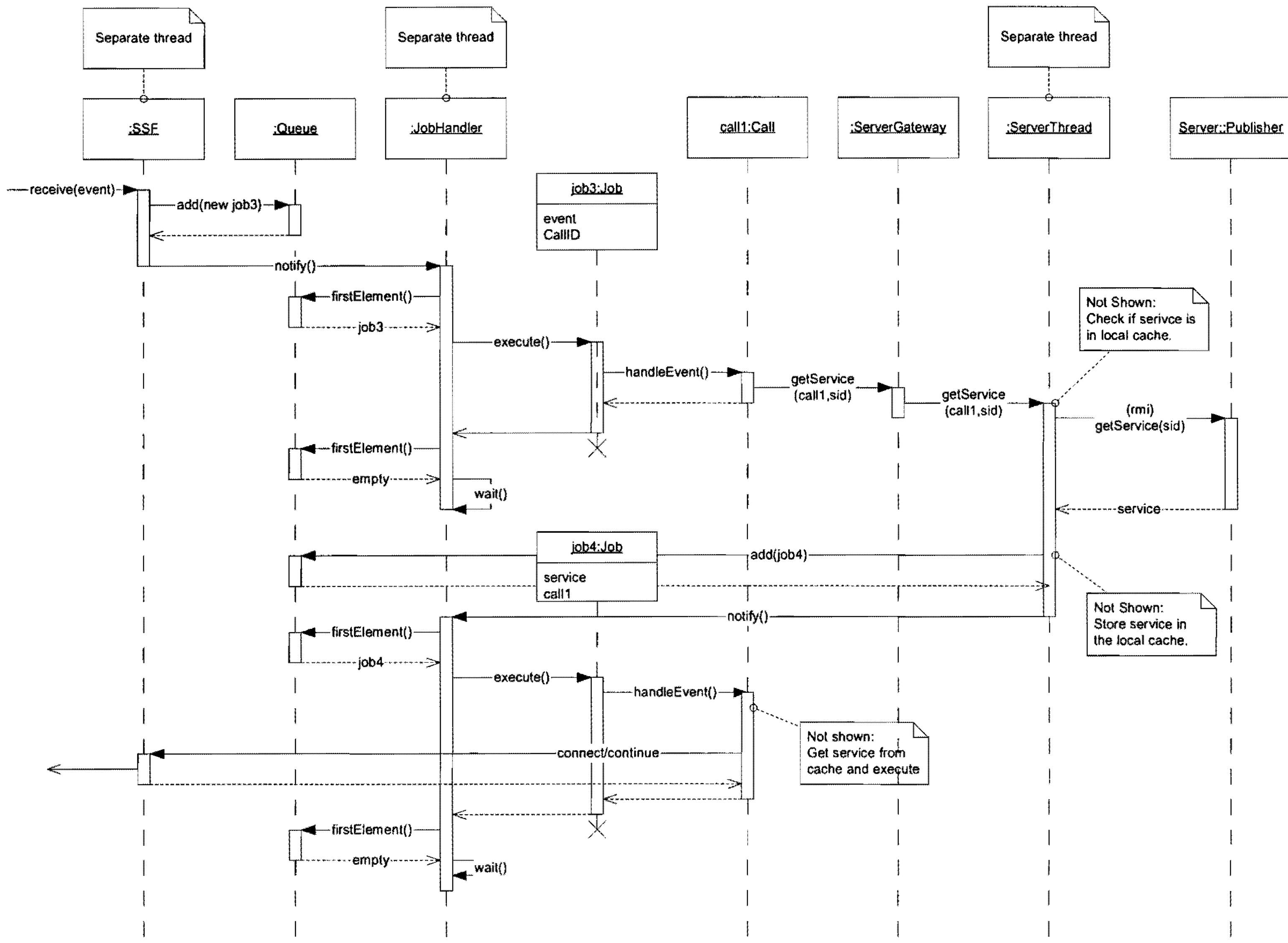

Figure D.3: Sequence diagram for invoking an IN Service

Appendix E

User Manual

The RJS package consists of three individual programs:

- the IN Server,
- the Java Service Environment (JSE) and
- the RSP software.

To work correctly, the JSE has to be started *after* the IN Server and the RSP software. How these programs are compiled, started and used, is described in the next sections.

E.1 Compiling the IN Server and the JSE

To compile the IN Server and the JSE, one requires JDK 1.2. The directory where the Java 1.2 binaries are located will be referred to as `<javabin>`.

The JSE depends on the Jive package which is part of the Erlang 47.4.1 libraries. The directory where the Jive class files can be found will be referred to as `<jiveclass>`. Usually this is the `lib/jive-1.2/priv/jive/erlang` directory found in the Erlang root directory.

The directory where the IN Server source files can be found will be referred to as `<serverroot>`. The directory where the JSE source files can be found will be referred to as `<jseroot>`.

The host the IN Server is running on also needs to run a web server. On this web server the IN Server can make its service class files available for the JSEs. The directory where the web server stores the class files is referred to as `<webdir>`. The URL from which the JSEs may download these class files is referred to as `<weburl>`.

Since the IN Server and JSE are mutually dependent, compiling both packages is not trivial. To compile both packages follow the next steps:

- Compile and distribute the se.ericsson.etm.rjs.common package.
- Compile the se.ericsson.etm.rjs.cache package and distribute the cache stub files.
- Compile the IN Server and distribute the server stub files.
- Compile the JSE.

E.1.1 The se.ericcson.etm.rjs.common package

The common package is part of the IN Server. Apart from the stub files, it contains those files that will be used by both the IN Server and the JSE. To compile the common package, first set the `CLASSPATH` environment variable to the `<serverroot>`. Now go to the `<serverroot>` and run:

<javabin>/javac se/ericsson/etm/rjs/common/*.java

To distribute the common package copy the files in the directory
<serverroot>/se/ericsson/etm/rjs/common to
<jseroot>/se/ericsson/etm/rjs/common.

E.1.2 The se.ericsson.etm.rjs.cache package

The cache package is part of the JSE. To compile the cache package, first set the CLASSPATH variable to the <jseroot>. Go to the <jseroot> directory and run:
<javabin>/javac se/ericsson/etm/rjs/cache/*.java
To create the cache stub files run:
<javabin>/rmic -v1.2 -d . se.ericsson.etm.rjs.cache.CacheImpl

To distribute the cache stub files copy the
<jseroot>/se/ericsson/etm/rjs/cache/Cache.class and
<jseroot>/se/ericsson/etm/rjs/cache/CacheImpl_Stub.class files to
<serverroot>/se/ericsson/etm/rjs/cache.

E.1.3 The IN Server

In the <serverroot>/se/ericsson/etm/rjs/server/Configuration file several configuration settings can be customized:

- The GUI constant defines whether the GUI should be displayed or not.
- The DEFAULTLOCATION constant sets the name of the home network.
- The RMI_REGISTRY_PORT sets the port on which the RMI registry will be started.

It is not necessary to set the host IP address at which the RMI registry will start. This is always the IP address of the host the IN Server is running on.

Before the IN Server can be compiled the JAVAROOT and WWWROOT variables in the
<serverroot>/compileserver file have to be edited, so that they reflect the location of respectively the Java compiler (<javabin>) and the web server root (<webdir>). Running compileserver will compile the IN server source files and make the service class files available at the web server.

To distribute the server stub files copy the
<serverroot>/se/ericsson/etm/rjs/server/Publisher.class and
<serverroot>/se/ericsson/etm/rjs/server/PublisherImpl_Stub.class files to
<jseroot>/se/ericsson/etm/rjs/server.

E.1.4 The JSE

the <jseroot>/se/ericsson/etm/rjs/jse/Configuration files contains several parameters that can be customized:

- The JSE_IDENTIFICATION constant stores the name of the JSE and is used to gain access to the IN Server.
- The JSE_PASSWORD constant stores the password of the JSE and is used to gain access to the IN Server.
- The GUI constant defines whether the GUI should be displayed or not.
- The DEBUG constant defines whether additional debug windows should be displayed or not.

- The RMI_REGISTRY_IPADDRESS holds the IP address of the RMI registry. This is the IP address of the IN server.
- The RMI_REGISTRY_PORT holds the port number of the RMI registry. It must match the setting in the IN Server configuration file.
- The JIVE_SERVER_IPADDRESS sets the IP address of the host the Jive server is running on.
- The ESCAPE_NUMBER defines the telephone number that activates the escape service. This is the IP address of the host the RSP software is running on.
- The JIVE_SERVER_PORT sets the port number of the Jive server. If this setting is changed to a non-default value, one will also have to change the port in jive:start(<port>) in the <rsproot>/jsegateway file. The default port number is 4711.
- The INITIAL_NUMBER_OF_SERVERTHREADS is the number of ServerThreads in the thread pool at startup.
- The MAX_NUMBER_OF_SERVERTHREADS is the maximum number of ServerThreads in the thread pool.

Before the JSE can be compiled the JSEROOT, JIVEROOT and JAVAROOT variables in the <jseroot>/compilejse file have to be edited, so that they reflect the location of the JSE source files (<jseroot>), the Jive library (<jiveclass>) and the Java compiler (<javabin>). Running compilejse will compile the JSE source files.

E.2 The IN Server GUI

When the IN Server is started with GUI, the display shown in figure E.1 will appear. It shows a

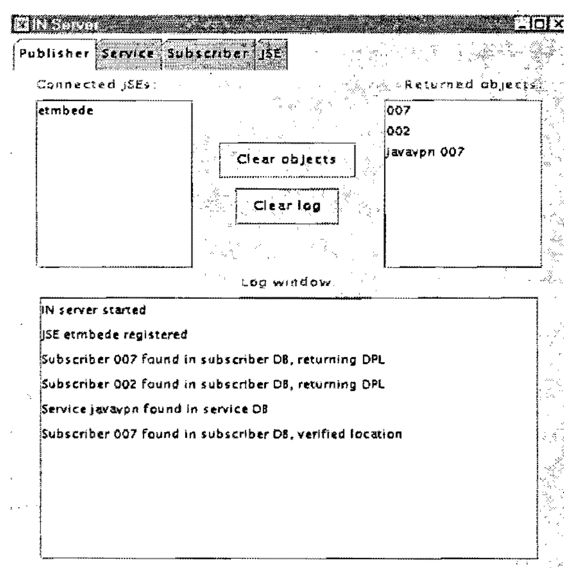

Figure E.1: The publisher tab of the IN Server GUI

tabbed pane with four tabs. Currently, the publisher tab is active. In the top left corner a list of currently connected JSEs is displayed. In the top-right corner a list of recently returned objects (DPLs and IN services) is displayed. Below these lists is a log window. In between the two lists are two buttons; one to clear the list with returned objects and one to clear the log window.

The second tab is the service tab. It is shown in figure E.2. The top-left list shows all known services on the IN server. In this case two services, police and javavpn, are known. The top-right

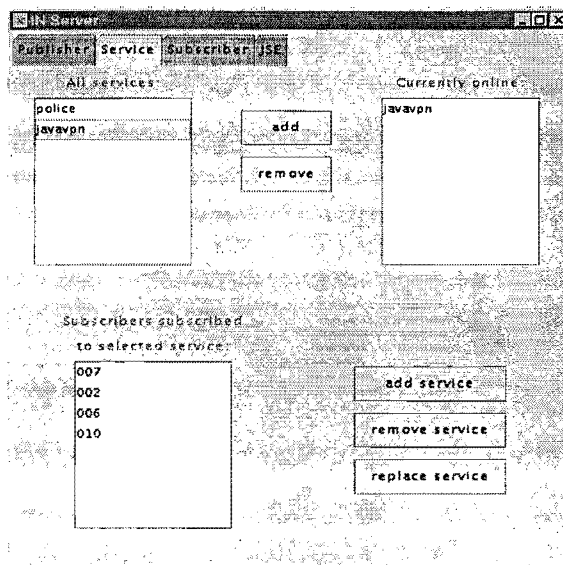

Figure E.2: The service tab of the IN Server GUI

list shows which of these services are currently online. When a service is online it means that a JSE is able to download it from the server. The `add` and `remove` buttons allow services to be added to and removed from the online-list. When a service is selected in the top-left window, as is currently the `javavpn` service, the subscribers subscribed to this service are displayed in the bottom list. The three buttons in the lower-left corner allow manipulation of the all-list. When `add service` button is pressed a window will pop up prompting for the class name of the service to be added. The `remove service` button removes the in the all-list selected services. The `replace button` prompts for the class name of the to be added service and replaces the selected service.

Figure E.3 shows the subscriber tab. The top-left list shows all known subscribers. In this example

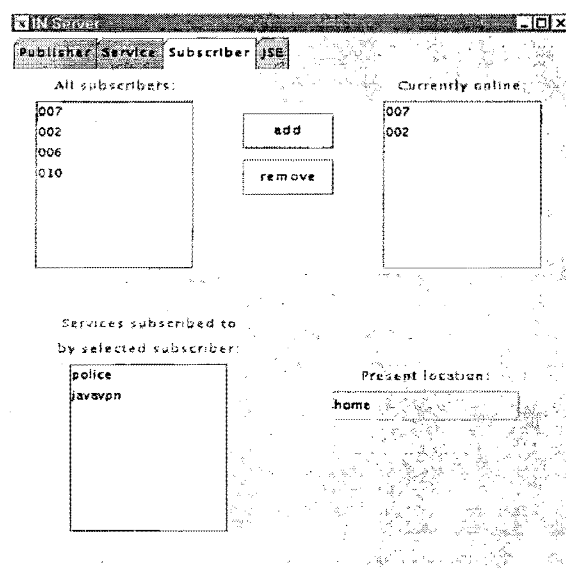

Figure E.3: The subscriber tab of the IN Server GUI

four subscribers are known. The top-right list shows which of these subscribers are allowed to access their subscribed services. Currently, only subscriber 007 and 002 can use their services. The two middle buttons allow subscribers to be added to and removed from the online-list. When a subscriber is selected in the all-list, the services he is subscribed to are shown in the bottom list and his present location is displayed. Subscriber 007 is subscribed to both the `police` and `javavpn` service and is currently `home`. By default the `DEFAULTLOCATION` variable mentioned in section E.1.3 is set to `home`. It it not possible to add subscribers to the all-list or for a subscriber to subscribe or unsubscribe to an IN service. Currently, this is hard-coded.

The fourth and last tab is the JSE tab shown in figure E.4. All known JSEs are shown in the

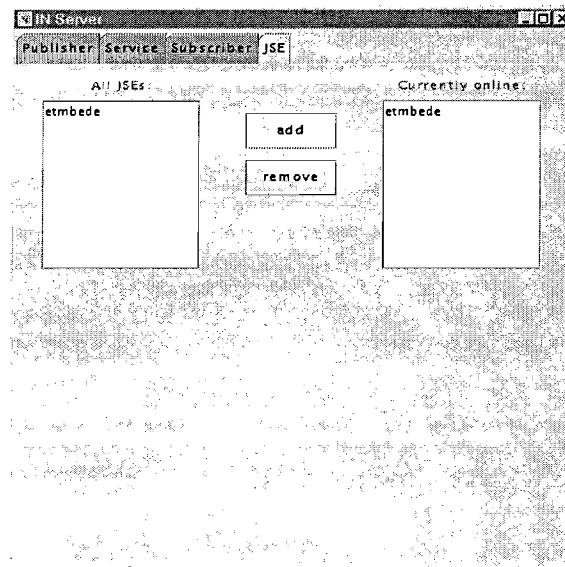

Figure E.4: The JSE tab of the IN Server GUI

top-left list. The ones in the top-right list are allowed to gain access to the IN Server. The buttons allow JSEs to be added to and removed from the online-list. It is not possible to add JSEs to the all-list.

E.2.1 The JSE GUI

The GUI in figure E.5 will appear when the JSE is started. The title bar of the window shows the name of the JSE, in this case `etmbede`. The window itself consists of four areas: "SCF Gateway", "SSF", "JobHandler" and "Cache". In the SCF Gateway area, a list of currently to the JSE connected SSFs is displayed. In the example, `Oosterhout@dennett` and `Rijen@dennett` are connected to the JSE. The list in the SSF area remains empty until an SSF is selected in the SCF Gateway area. It then displays the calls for the selected SSF that the JSE is monitoring. In the example, `Rijen@dennett` is selected. The JSE monitors two calls; call number 29 for subscriber 007 and call number 31 for subscriber 010. The list in the JobHandler area shows the eight most recent executed jobs. The three types of jobs are displayed in different colors. The Cache area contains a list of elements that are currently stored in the cache. DPLs, services and data elements are displayed in different colors. When one or more of these elements are selected and the Flush button is pressed, the elements are removed from the local cache.

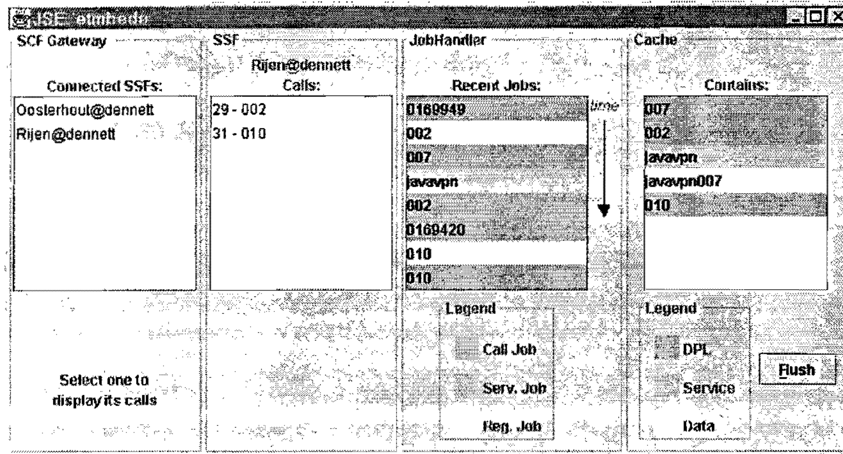

Figure E.5: The JSE GUI

E.3 The RSP-software

The directory in which the RSP software is installed will be referred to as `<rsproot>`.

Some modules of the RSP software have been changed to make communication with the JSE possible. Copy the following files:
Copy `jsegateway` to the `<rsproot>` directory.
Copy `lim_pad10`, `scf_man10` and `scf10` to the `<rsproot>/scf` directory.
Compile the RSP software by running `make` in the `<rsproot>`.

Start the RSP software from the `<rsproot>` with: `erl -sname etmbede -s in`.

E.4 Using prototype services

The RJS package comes with three prototype services:

- the Java VPN service,
- the Police service and
- the Test service.

The subscriber database contains four subscribers:

- James Bond (007) is subscribed to the Java VPN, Police and Test service.
- Bill Fairbanks (002) is subscribed to the Java VPN service.
- Alexander Trevelyan (006) is subscribed to the Java VPN service.
- Austin Powers is subscribed to the Java VPN and Police service.

The Java VPN service creates a virtual private network for its subscribers. Every subscriber has a two digit telephone number and a barring list.

The Police service is activated by dialing 112. It prompts the subscriber for a personal code number. Only if the number is entered correctly the call is forwarded to a predefined number, in this case 0169110. The personal code number for James Bond and Austin Powers are respectively 007112 and 010112.

The Test service prints its version number on the subscriber's telephone display. It was developed to demonstrate how a Java IN service can be updated during runtime. The `TestService10` class

Name:	Tel.nr.	VPN Tel.nr.	Barred:
James Bond	111007	33	35
Bill Fairbanks	111002	34	35 & 36
Alexander Trevelyan	111006	35	33
Austin Powers	111010	36	34

Table E.1: The VPN service

file contains version 1.0 and `TestService20` version 2.0 of the service. There is no real difference between the version, just the number.

Acronyms

AD	Adjunct
API	Application Programming Interface
BCP	Basic Call Process
BCSM	Basic Call State Model
CAMEL	Customized Application for Mobile network Enhanced Logic
CCAF	Call Control Agent Function
CCF	Call Control Function
CORBA	Common Object Request Broker Architecture
CS-n	Capability Set n
CSI	CAMEL Subscription Information
CSP	Cryptographic Service Provider
DFP	Distributed Functional Plane
DP	Detection Point
DPL	Detection Point List
DSA	Digital Signature Algorithm
EDP	Event Detection Point
ETSI	European Telecommunication Standards Institute
FE	Functional Entity
FEA	Functional Entity Action
GFP	Global Functional Plane
GMSC	Gateway Mobile services Switching Center
GSM	Global System for Mobile communications
HLR	Home Location Register
HTML	Hypertext Markup Language
IDL	Interface Definition Language
IF	Information Flow
IIOP	Internet Inter-ORB Protocol
IN	Intelligent Network
INCM	Intelligent Network Conceptual Model
IP	Intelligent Peripheral
ITU	International Telecommunication Union
JCA	Java Cryptography Architecture
JCE	Java Cryptography Extension
JDK	Java Development Kit
JRE	Java Runtime Environment
JSE	Java Service Environment node
JVM	Java Virtual Machine
MSC	Mobile services Switching Center
O-CSI	Originating CAMEL Subscription Information
OMG	Object Management Group
ORB	Object Request Broker

OSS	Operator-Specific Services
PCC	Proof-Carrying Code
PCN	Personal Communication Network
PE	Physical Entity
PIC	Point in Call
PID	Process Identification
POI	Point of Initiation
POR	Point of Return
RJS	Roaming Java IN Service(s)
RMI	Remote Method Invocation
RSP	Rapid Service Prototyping
SCEF	Service Creation Environment Function
SCEP	Service Creation Environment Point
SCF	Service Control Function
SCP	Service Control Point
SDF	Service Data Function
SDP	Service Data Point
SE	Service Environment
SF	Service Feature
SIB	Service Independent Building Block
SMAF	Service Management Access Function
SMAP	Service Management Access Point
SMF	Service Management Function
SMP	Service Management Point
SN	Service Node
SRF	Specialized Resource Function
SSCP	Service Switching and Control Point
SSF	Service Switching Function
SSP	Service Switching Point
T-CSI	Terminating CAMEL subscription Information
TDP	Trigger Detection Point
UML	Unified Modeling Language
URL	Uniform Resource Locator
VLR	Visitor Location Register
VPN	Virtual Private Network

Bibliography

[1] J. Armstrong, R. Virding and M. Williams, "Concurrent Programming in ERLANG ", ISBN 0-13-285792-8, Prentice Hall International, 1993, available at http://www.erlang.se/erlang/sure/main/news/erlang-book-part1.pdf

[2] Special Mobile Group (SMG) of the ETSI, "Digital cellular telecommunications system (Phase 2+); Customized Applications for Mobile network Enhanced Logic (CAMEL) - Stage 2 (GSM 03.78 version 5.1.0)", version 5.1.0, August 1997, inside Ericsson available at: http://www.gsm.ericsson.se/standard/gsmph2+/camel/0378-510.pdf

[3] Special Mobile Group (SMG) Technical Committee (TC) of the ETSI, "Digital cellular telecommunications system (Phase 2+); Customised Application for Mobile network Enhanced Logic (CAMEL) - Phase 2; Stage 2 (GSM 03.78)", October 1997, inside Ericsson available at: http://www.gsm.ericsson.se/standard/gsmph2+/camel/378a08r5.pdf

[4] K. Campbell, "Custom Class Loaders Using Java", Developer's Journal, Interface Technologies, Inc, 1998, available at ftp://ftp.iftech.com/DevJournal/ccl/ccl.pdf

[5] D.M. Chess, C.G. Harison and A. Kershenbaum, "Mobile agents: Are they a good idea?", IBM Research Report, T. J. Watson Research Center, October 1994, available at http://www.research.ibm.com/massive/mobag.ps

[6] D. Chess, B. Grosof, C. Harrison, D. Levine and C. Parris, "Itinerant Agents for Mobile Computing", IBM Research Report, T. J. Watson Research Center, March 1995, available at http://www.research.ibm.com/massive/rc20010.ps

[7] B. Eckel, "Thinking in Java", ISBN 0-13-659723-8, Prentice-Hall Inc., 1998, available at ftp://www.mindview.net/pub/eckel/printabl.pdf

[8] , No author given, "Virtual Private Network VPN 2.2.1, Service Overview, Structure and Feature Description", EN/LZT1120012.R2B, Ericsson Telecom AB, September 1997, inside Ericsson available at: http://sad.etm.ericsson.se/products/documents/vpn221R4so.pdf

[9] , No author given, "Network Overview and Traffic Cases Description for VPN 2.2.1", No.: 1/190 59-FAM 511 02 UEN, Ericsson, May 1998, inside Ericsson available at: http://sad.etm.ericsson.se/products/documents/vpn221R4no.pdf

[10] , No author given, "The FSP Pototype, The SSF Part", ETM/C/XU 93:046, Ericsson, November 1993

[11] J. Feigenbaum and P. Lee, "Trust Management and Proof-Carrying Code in Secure Mobile-Code Applications", DARPA workshop on Foundations for Secure Mobile Code, March 1997, available at http://www.cs.nps.navy.mil/research/languages/statements/leefei.ps

[12] L. Gong, M. Mueller, H. Prafullchandra and R. Schemers, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java™ Development Kit 1.2", In Proceedings of the USENIX Symp. on Internet Technologies and Systems, Monterey, California, December 1997, available at http://java.sun.com/people/gong/papers/jdk12arch.ps.gz

[13] L. Gong and R. Schemers, "Implementing Protection Domains in the Java™ Developement Kit 1.2", In Proceedings of the Internet Society Symp. on Network and Distributed System Security, San Diego, CA, March 1988, available at http://java.sun.com/people/gong/papers/jdk12impl.ps.gz

[14] J. Gosling, B. Joy and G. Steele, "The Java™ Language Specification", ISBN 0-201-63451-1, Addison Wesley, 1996, available at ftp://ftp.javasoft.com/docs/specs/langspec-1.0.pdf

[15] R.S. Gray, D. Kotz, G. Cybenko and D. Rus, "D'Agents: Security in a Multiple-language mobile-agent system", Dartmouth College, 1998, available at http://actcomm.dartmouth.edu/papers/gray:security-book.ps.Z

[16] C.G. Harrison, "Smart Networks and Intelligent Agents", IBM T. J. Watson Research Center, available at http://www.research.ibm.com/massive/smartnw.ps

[17] F. Hohl, "An Approach to Solve the Problem of Malicious Hosts in Mobile Agent Systems", University of Stuttgart, IPVR, 1997, available at http://www.informatik.uni-stuttgart.de/ipvr/vs/mitarbeiter/hohlfz/sosp97.ps

[18] Douglas Kramer, "The Java™ Platform, A white paper", May 1996, available at http://java.sun.com/docs/white/platform/CreditsPage.doc.html

[19] Laura Lemay and Rogers Cadenhead, "Teach Yourself Java™ 1.2 in 21 days", ISBN 1-57521-390-7, Sams Publishing, May 1998, refer to http://prefect.com/java21/

[20] No author given, "Fundamentals of Java Security", MageLang Insitute, 1998, available at http://developer.java.sun.com/developer/onlineTraining/Security/abstract.html

[21] C. McManis, "The basics of Java class loaders", Java World, October 1996, available at http://www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth.html

[22] C. Meadows, "Detecting Attacks on Mobile Agents", DARPA workshop on Foundations for Secure Mobile Code, March 1995, available at http://www.cs.nps.navy.mil/research/languages/statements/meadows.ps

[23] J.T. Moore, "Mobile Code Security Techniques", Department of Computer and Information Science, University of Pennsylvania, May 1998, available at http://www.cis.upenn.edu/ jonm/papers/cis700.ps

[24] K. Nygren, J. Grebenö, "Jive Application (JIVE)", part of the Open Source Erlang documentation, may 1997, available at http://www.erlang.org/doc/doc/lib/jive-1.2/doc/book.ps.gz

[25] J.J. Ordille, "When agents roam, who can you trust?", Computing Science Research Center, Bell labs, 1996, available at http://cm.bell-labs.com/cm/cs/doc/96/5-09.ps.gz

[26] T. Sander and C.F. Tschudin, "Protecting Mobile Agents Against Malicious Hosts", in Lecture Notes in Computer Science (LNCS), Vol.1419, Springer-Verlag, June 1998, available at http://www.icsi.berkeley.edu/ sander/publications/MA-protect.ps

[27] T. Sander and C.F. Tschudin, "Towards Mobile Cryptography", Proceedings of the 1998 IEEE Symp. on Security and Privacy, May 1998 available at http://www.icsi.berkeley.edu/ sander/publications/satschu.ps

[28] No author given, "Clarifications and Amendments to The Java Language Specification, Technical Rationale on Class Unloading", Sun Microsystems, available at http://java.sun.com/docs/books/jls/unloading-rationale.html

[29] No author given, "Java Remote Method Invocation - Distributed Computing for Java", white paper, Sun Microsystems, available at http://java.sun.com/marketing/collateral/javarmi.html

[30] No author given, "Java™ Remote Method Invocation Specification", Revision 1.50, JDK 1.2, Sun Microsystems, October 1998, available at ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf

[31] No author given, "Java™ RMI Security Extension, Early Look Draft", Sun Microsystems, May 1999, available at http://www.javasoft.com/products/jdk/rmi/rmi-security.pdf

[32] No author given, "The Java™ Language: An overview", Sun Microsystems, 1994, available at ftp://ftp.javasoft.com/docs/papers/java-overview.ps

[33] Telecommunication Standardization Sector of ITU, Recommendation Q.120x, inside Ericsson available at:
http://mirad.etm.ericsson.se/COMPAREA/STANDARD/STANDARD/ITU/INAP/

[34] Telecommunication Standardization Sector of ITU, Recommendation Q.121x, inside Ericsson available at:
http://mirad.etm.ericsson.se/COMPAREA/STANDARD/STANDARD/ITU/INAP/

[35] Jan Thörner, "Intelligent Networks", ISBN 0-89006-706-6, Artech House, 1994

[36] No author given, "UML Notation Guide", version 1.1, Rational Software Corporation, September 1997, available at http://www.rational.com/uml

[37] Giovanni Vigna (Ed.), "Mobile Agents and Security", ISBN 3-540-64792-9, Lecture Notes in Computer Science, Springer, 1998

[38] Giovanni Vigna, "Mobile Code Technologies, Paradigms, and Applications", PhD Thesis, Politecnico di Milano, February 1998, available at http://www.cs.ucsb.edu/ vigna/pub/phdthesis.ps.gz

[39] B. Yee, "A Sanctuary for Mobile Agents", March 1995, available at http://www.cs.nps.navy.mil/research/languages/statements/bsy.ps