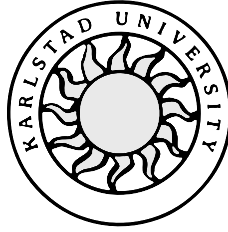

Computer Science

Harald Grytberg

Kristofer Wiklund

Implementing

JAIN MAP API RI

Bachelor’s Project

2000:14

Java API Integrated Network

Mobile Application Part

Application Programmers Interface

Reference Implementation

JAIN MAP API RI

Harald Grytberg

Kristofer Wiklund

© 2000 The authors and Karlstad University

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Harald Grytberg

Kristofer Wiklund

Approved, 2000-05-30

Advisor: Nils Dåverhög

Examiner: Stefan Lindskog

Abstract

This report is written as a Bachelor's Project in Computer Science. It is supposed to give an insight in our work of producing a reference implementation for the JAIN MAP application programmers interface. The document includes a description of how we worked during the project, our ideas, our problems, and how the reference implementation works with all of its components.

The reference implementation is supposed to be a program that should be used by people developing applications who uses the application programmers interface. By running their application against the reference implementation they will see if they are using the application programmers interface in a correct way. Within the limits of this Bachelor's Project not the entire API functionality should be tested, only the parts concerning sending and receiving SMS messages.

To be able to test the application programmers interface, there is another Bachelor's Project called Implementation of JAIN MAP API CTS [4] developed by Henrik Bergkvist and Nicklas Jansson. The conformance test suite is an application that communicates with the reference implementation using the interface in JAIN MAP application programmers interface.

Sammanfattning

Den här rapporten är skriven i ett examensarbete i datavetenskap. Det är menat att ge en inblick i vårt arbete med att tillverka en referensimplementation (RI) för JAIN MAP applikationsprogrammerarinterfacet (API). Dokumentet innehåller en beskrivning av hur vi arbetade under projektets gång, våra idéer, våra problem, och hur RI fungerar med sina alla komponenter.

RI är menat att vara ett program som ska användas av personer som utvecklar applikationer mha API:et. Genom att köra sin applikation mot RI kan de se om de använder API:et på ett korrekt sätt. Inom ramen för detta examensarbetet kommer inte hela API funktionalitet att testas, bara de delarna som har att göra med att skicka och ta emot SMSmeddelanden.

För att kunna testa API:et finns det ett annat examensarbete vid namn Implementation of JAIN MAP API CTS [4], utvecklat av Henrik Bergkvist och Nicklas Jansson. CTS är en applikation som kommunicerar med RI genom att använda API.

Contents

1 Introduction 1

2 Background, problems and purpose 2

2.1 Background 2

2.2 Problems...3

2.2.1 History
2.2.2 Construction
2.2.3 Implementation
2.2.4 Compiling
2.2.5 Execution

2.3 Purpose 4

3 Conditions and requirements 6

3.1 Conditions 6

3.2 Equipment 6

3.3 Requirements 6

3.3.1 The stack implementation
3.3.2 The provider implementation
3.3.3 RI design
3.3.4 SMS Primitives

4 Description of the construction method 12

4.1 Preparation 12

4.2 Designing 13

5 Implementation and testing 14

5.1 Implementation 14

5.2 Compilation...14

5.2.1 Javac
5.2.2 Rmic
5.2.3 Make for Java

5.3 Execution 15

5.3.1 Java
5.3.2 Rmiregistry

5.4 Testing...16

5.4.1 White Box Test Model
5.4.2 Black Box Test Model
5.4.3 Test Execution

6 Experiences and recommendations 20
6.1 Methods 20
6.1.1 RMI
6.1.2 Package
6.1.3 Factory model
6.1.4 Listeners
6.2 Programs 22
6.2.1 Visual SlickEdit 5.0
6.2.2 ClearCase NT3.2
6.2.3 Make for Java 1.3
7 Conclusions 24
References 25
Appendix 26
A Abbreviations 26
B Expressions 26
C Requirements specification - A simple JAIN MAP API RI 27
C.1 Background 27
C.2 Task 27
C.3 Purpose 28

List of Figures

Figure 2.1: SS7 stack layers 3
Figure 2.2: Application programming using the API 5
Figure 2.3: CTS RI interaction 5
Figure 3.1: Interaction between RI classes 9
Figure 4.1: API interface 13
Figure 5.1: RMI communication structure 15
Figure 5.2: White box test case 17
Figure 5.3: Black box test case one 18
Figure 5.4: Black box test case two 18
Figure 5.5: CTS application 19
Figure 6.1: Factory model 21
Figure 6.2: Listener model 21

1 Introduction

As students at University of Karlstad we have to do what is called a Bachelor's Project in order to receive our Bachelor's Degree. The course is a 10 points course, which means that the effort we put in the assignment should comprise ten weeks of work, but since this is a course studied at half rate, the work was spread out over 20 weeks. This document is the final result of our work together with the application we developed.

Our project involves developing a reference implementation for a Java Application Programmers Interface. This interface was to be developed at Ericsson Infotech AB at the same time as our reference implementation.

In this document we have tried to write down the most important steps of our work. In the first section called Background, problems and purpose we try to explain how the JAIN MAP API project started, and why. We also explain problems of the developement project.

The following section describes the conditions we had to adapt to, and also what result we were expected to produce. We will have a look at the desired design. The section includes some thoughts about these topics as well.

After that we want to tell a little more about how we constructed our program. How did we prepare? What happened then?

Another important topic is how we worked during the implementation phase. We will see how we tried to implement our design. This topic also includes information about compilation as well as testing. This means testing the whole API, and not only our reference implementation.

In the end of the documents there are two close-related topics in which we explain recommendations we have to people working with similar things, and our own experiences; what we learned, our mistakes.

Finally we try to summarize the conclusions we came to after completing our project, including methods we prefer, and dead-ends.

2 Background, problems and purpose

In this section we want to describe how, and by whom the JAIN project was started, and hopefully explain why the standardized API will be widely used in telecom application programming. We also want to explain what TCAP and MAP are, and their role in the project.

Here you can also find information about the different types of problems that occurred throughout the project.

Finally this section reveals the main purpose of the reference implementation, what it will be used for, and the general functionality.

2.1 Background

All data over the Public Land Mobile Network, PLMN, is transported using a protocol by the name of Signaling System 7, SS7. Somewhere in this network we want servers that are able to communicate with the mobile stations, MS, on the PLMN as well as the Internet.

[2] Back in June 9, 1998 Sun Microsystems announced to the world that a few telecommunication companies had taken the initiative of a project which purpose was to bring the benefits of Java technology to the Intelligent Network (IN) infrastructure of the public switched telephone network. This project was called Java Advanced Intelligent Network (JAIN), and it was to be the first Java platform-based solution for building advanced telecom services that blend IN and Internet technologies. It was supposed to take the position of a standard for programming in Java over the SS7 stack.

Three leading SS7 protocol stack vendors wanted to participate in the project from the very beginning: ADC NewNet, DGM&S Telecom[1] and Ericsson Infotech AB. In addition, Apion Ltd. joined in at a very early stage too.

Transaction Capabilities Application Part (TCAP) is an SS7 protocol which provides the means for transferring information between nodes and provides generic services to applications, while being independent of them. It is used in nodes of the existing telephone network.

---

[1] DGM&S Telecom changed name to Ulticom Inc. in 1999

Mobile Application Part (MAP) is a SS7 protocol layer too, but it is placed in a higher level in the SS7 stack.

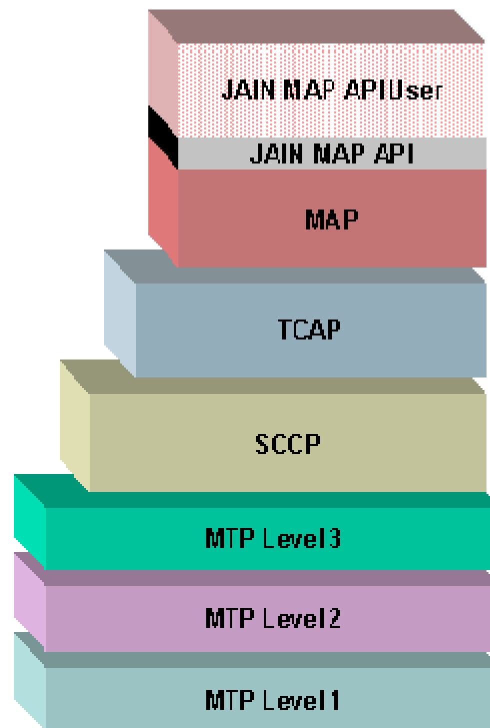

Figure 2.1: SS7 stack layers

By August 1999, the time of the JAIN MAP specification[1] completion, the following companies had joined in for the standardization project:

- Telcordia
- Trillium Digital Systems
- NTT
- Nokia

Previously a similar project was developed for the existing telephone network for stationary telephones called JAIN TCAP API.

2.2 Problems

2.2.1 History

The stack-built SS7 protocol is what we can call a closed protocol. Programming on this protocol is difficult to master, partly because of the layered structure, but also because of the

number of variables that the application programmer has got direct access to. It therefore takes too much time to develop applications that can work with the protocol.

2.2.2 Construction

Under the construction phase the biggest problem was that we didn’t really know what the problem was because of the complexity of the project and therefore we didn’t know what to do. Another problem was that the API wasn’t totally developed when we constructed our system. A solution to that problem was that we froze the API at one point so that we could complete our Bachelor’s Project.

2.2.3 Implementation

At first we tried to build the communication on Java sockets, but that turned out not to be the best solution for our problem. Instead we used RMI as described in section 6.1.1.

Even if RMI was the best solution for us it wasn’t the easiest thing to understand. It was very difficult to understand even though we had a lot of information to study.

2.2.4 Compiling

When the whole project was to be inserted into ClearCase as described in section 6.2.2 a big problem occurred to us. In JMK as described in section 6.2.3 there were many classpaths that should be merged together with the package structure, and this was a problem because of the path lengths. For example one of the paths looked like this:
m:\qinxkwi_dev_exwork\lovberj\anf10101\cra11937_stack\cna21413_map\caa20117_3\src\ericsson\ein\ss7\map\capability\message\transaction\server\MessageTransactionProviderImpl_Server.java

2.2.5 Execution

In the execution we had similar problems as in the compiling part. It is difficult to execute files inside a ClearCase database, because of the complex structure and the long paths. The problems with the long paths don’t depend on ClearCase but on Ericsson product hierarchy.

2.3 Purpose

The purpose of the JAIN MAP API is to make programming over the SS7 stack much easier by only supplying the programmer with what he needs, having the API doing as much as possible automatically and out of sight for the application programmer.

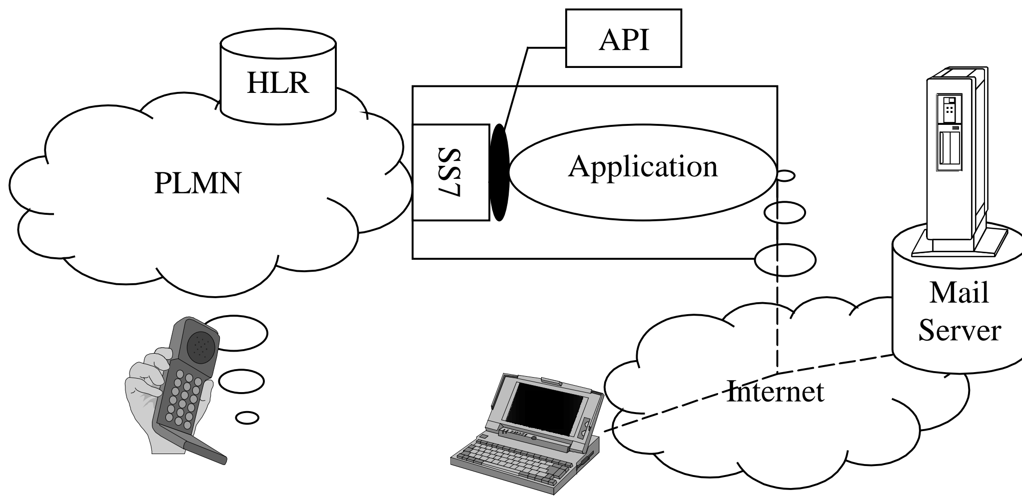

Figure 2.2: Application programming using the API

The reference implementation, RI, is supposed to be an implementation of the JAIN MAP API specification. Being able to take care of all the method calls that the Java application will use in order to perform whatever the application using the API wants the layers below to perform.

The RI does not include an SS7 protocol stack including a MAP protocol, but it will simulate the functions of the stack. The purpose of the RI is to provide a test bed to verify that Java applications are compatible with the JAIN MAP API Specification.

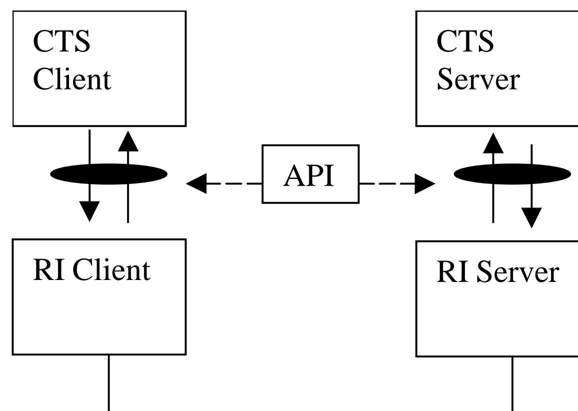

Figure 2.3: CTS RI interaction

To see whether or not the primitives works correctly downward, we will use a connection to another instance of RI (with the above layer CTS server) stationed at a different computer.

3 Conditions and requirements

In this section we want to describe the conditions we had and the requirements we had to fulfil with our reference implementation.

3.1 Conditions

Since the JAIN MAP API project is being proceeded on Sun Microsystems initiative they have been much involved in the supervision of the project. Of course we had to implement everything in Java.

3.2 Equipment

To fulfil this assignment, each of us had a workstation with the following specifications:

Name: Dell OptiPlex GX1

CPU: Intel Pentium III 500 MHz

RAM: 128 MB SDRAM

Installed software: Windows NT 4.00.1381
Internet Explorer 4.72.3110.8
Outlook 8.5.6614.0
Visual SlickEdit 5.0
NetBeans DeveloperX2 2.1
Rational Rose 98i
Word 97 SR-2
ClearCase NT3.2
JDK 1.2.2
…and more

3.3 Requirements

In this section we describe the interfaces of the stack and the provider which we were supposed to implement.

3.3.1 The stack implementation

The RI must provide a stack implementation. It should contain four different providers, but due to lack of time only one is implemented within the Bachelor's Project, the SMS provider. To complete the RI for the three other providers, more methods have to be implemented.

The interface that the stack implementation should contain is jain.protocol.SS7.map.JainMapStack, which contains as follows:

3.3.1.1 Public JainMapMessageProvider createMessageProvider()

Whenever an application user calls this method, a new message provider will be created, and a reference to that object is returned. The provider object is inserted in a provider list within the stack.

3.3.1.2 Public void deleteProvider(JainMapProvider providerToBeDeleted)

If the application uses this method, the provider specified is to be deleted from the provider list.

3.3.1.3 Public void detach(JainMapProvider jainMapProvider)

Detaches the specified vendor specific Peer JAIN Map Provider from this JainMapStackImpl

3.3.1.4 Public Vector getProviderList()

Returns the vector of Peer JAIN MAP Providers that has been created by this JainMapStackImpl.

3.3.1.5 Public String getStackName()

This method returns the name of the stack.

3.3.1.6 Public void setStackName(String stackProtocol)

Sets the Name of this Stack.

3.3.1.7 Public int getProtocolStandard()

This method returns the protocol standard version number.

3.3.1.8 Public void setProtocolStandard (int protocolVersion)

This method sets the protocol standard version number.

3.3.2 The provider implementation

The provider must provide implementation of the interface in jain.protocol.SS7.-map.capability.message.transaction:

3.3.2.1 Public void addMessageTransactionListener(MessageTransactionListener listener, MapUserAddress userAddress)

Adds a MessageTransactionListener to the list of registered Event Listeners of this JainMapProviderImpl.

3.3.2.2 Public void mtMessageReq(MtMessageReqEvent event)

This method is used to send a request message from a computer to an MS. The mt prefix means mobile terminated.

3.3.2.3 Public void mtMessageReq(MtMessageReqEvent event, MessageTransactionListener listener)

This method is used to send a message from a computer to an MS. The confirm message to this request message will only be sent to the specified listener.

3.3.2.4 Public void moMessageResp(MoMessageRespEvent event)

This method is used to send a response message from a computer to an MS. The mo prefix means mobile originated.

3.3.2.5 Public long getNewTransactionId()

This method returns a new transactionId in long format.

3.3.2.6 Public void releaseTransactionId(long transId)

This method removes the specified transactionId from the transactionId list.

3.3.2.7 Public void removeMessageTransactionListener(MessageTransactionListener listener)

This method removes specified listener from the MessageTransactionListener list.

3.3.2.8 Public JainMapStack getAttachedStack()

This method returns the stack to which the provider is attached.

3.3.2.9 Public boolean isAttached()

Returns true if provider is attached to a stack. In other case the returned value is false.

3.3.3 RI design

This section contains information about how we designed the reference implementation

3.3.3.1 Class diagram

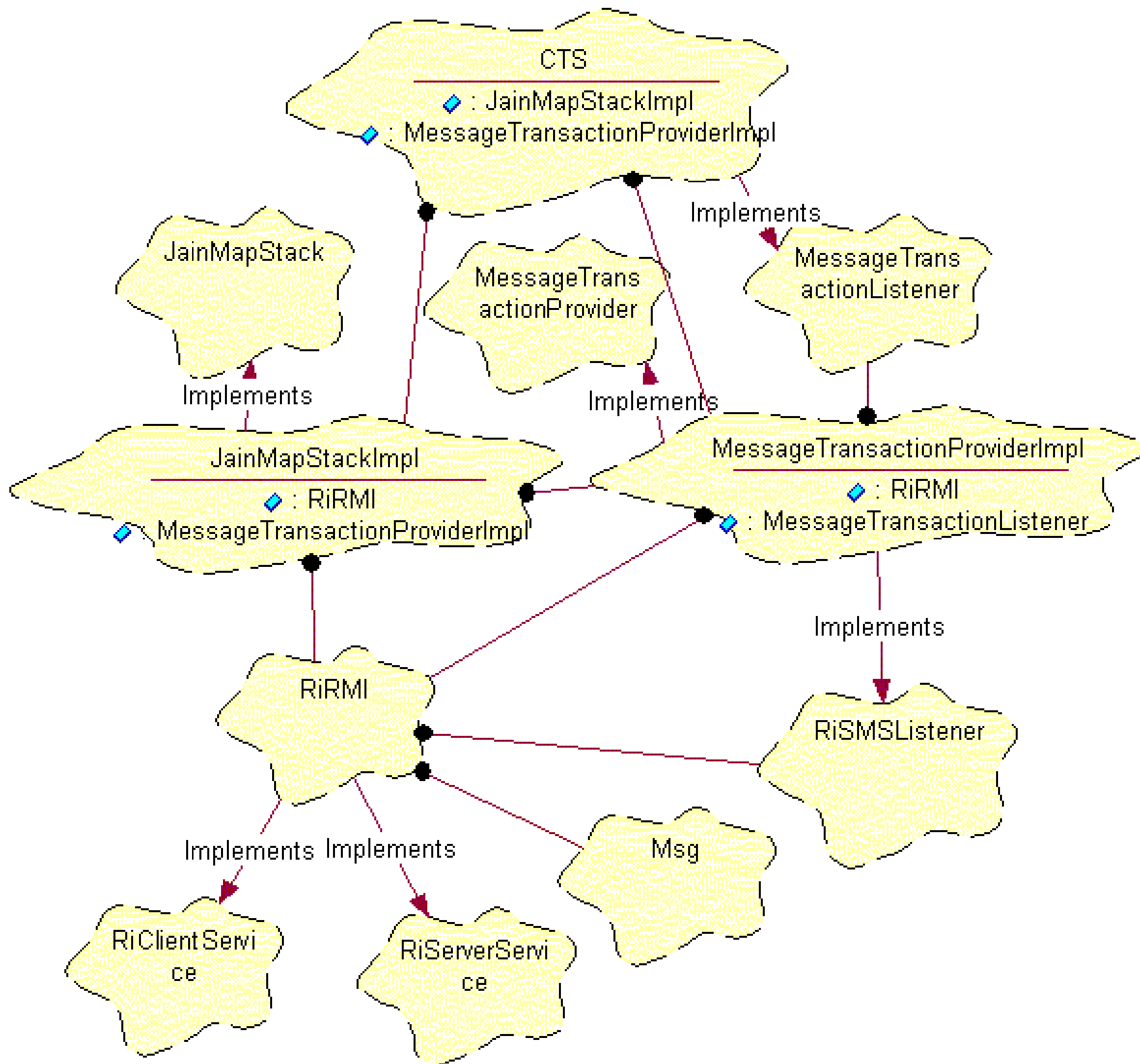

Figure 3.1: Interaction between RI classes

3.3.3.2 Description of class diagram

The whole test application is started by the CTS. CTS are an application that initiates a stack, JainMapStack, from which several different providers can be created. JainMapStackImpl implements JainMapStack, the API standard that should be tested.

JainMapStackImpl can create different providers. One of these is the MessageTransactionProviderImpl. This class implements the MessageTransactionProvider, which is also included in the JAIN MAP API standard, and should be tested as well.

When CTS creates a new MessageTransactionProviderImpl from the JainMapStackImpl, the JainMapStackImpl returns a MessageTransactionProvider object, because we want a direct communication between CTS and the provider.

JainMapStackImpl initiates RiRMI[2] that handles the communication between client and server over the Internet. When a provider is created, a reference to the RiRMI object is supplied, so that we can have a direct communication between the provider and the RiRMI.

When RiRMI in client has sent a message to RiRMI in server, the server should send it to the CTS. This is achieved by having a RiSMSListener implemented in the provider, and the provider adds himself to the list of listeners in the RiRMI. When a message arrives, every listener on the list receives this message. In the same way the provider sends the message up to the CTS by MessageTransactionListener.

3.3.3.3 RiRMI

The RiRMI handles the communication over the Internet between client and server. It consists of RiClientService, RiServerService and Msg. RiRMI uses RMI (Remote Method Invocation) [3] for the communication between different computers. RiClientService and RiServerService are two interfaces that RiRMI implement. RiClientService is used by server to send messages to client, and RiServerService is used by client to send messages to server. This means that both sides of the communication acts as both client and server. A message is an instantiated object of the Msg class, which is a wrapper class. Msg contains the primitive that the CTS want to send.

[2] RiRMI is separately described in chapter 3.3.3.3

3.3.4 SMS Primitives

Under this headline you will find the primitives that are included in the API, and therefore also must be taken care of in our RI. The sendable primitives are those that the application may send to the RI, and the receivable ones are those that will be sent up to the user as answers.

3.3.4.1 Sendable

MtMessageReqEvent

MtMessageRespEvent

3.3.4.2 Receivable

MtMessageConfEvent

MtMessageIndEvent

3.3.4.3 Capabilities of the Jain MAP API

- Send and receive SMS
- Communicate with a service application, USSD
- Find out the status (on/off/occupied) and the location (which cell) of a MS
- Find out the position (x,y) of a MS, LCS

4 Description of the construction method

In this section we describe how we designed our program.

4.1 Preparation

The first week we tried to understand what the assignment really consisted of. We thought that it was a little bit hard to get an insight to that, because none of us were familiar with signaling at this level. However, things became clearer after awhile. We had an important meeting 2000-02-04 where we got much of the information that we needed from the other persons involved in the project to be able to start designing. Here are some of the most interesting parts that concerned the RI.

In Figure 4.1 it is illustrated how an application can use the services provided by the API. The main idea is the three parts API, CTS and RI. The API is the standard that is being developed by Ericsson. Inside the API there are *interfaces*, *primitives*, *parameters*, *exceptions* and a *factory*. The interface is there because it is desirable to have a definite interface between the application and the communication part. A primitive is the object to be sent between client and server. It contains different parameters. A parameter contains the values of different basic types, such as *int*, *char* and *string*. It is necessary to standardize exceptions too, so that the communication part is throwing the same exception the application can catch. To have a standard so that different applications can be put together with different communications the factory is used. Factory is started by the application, and then told by the application to create a communication it can use. When communication is demanded the factory create a stack. A reference to the stack is returned to the application, so it can communicate directly with the layers below.

When we knew what different parts we should have in our program, we tried to visualize these by drawing class diagrams over the system.

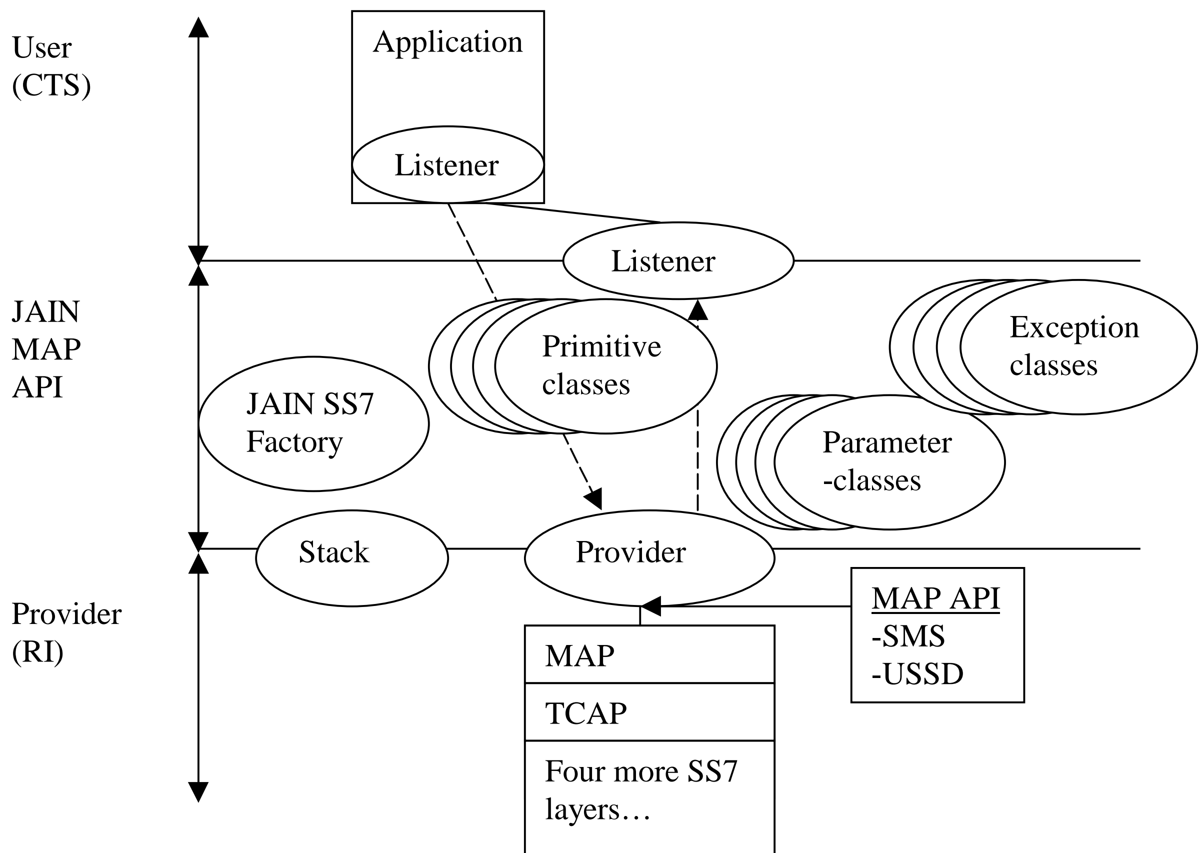

Figure 4.1: API interface

4.2 Designing

In the beginning it was difficult to understand the complexity of the program we should implement, so we started by constructing small parts. The first thing we did was to try to get a connection between two computers with Java sockets. We soon realized that this wasn't the best solution for our purposes, because if anything would be changed in the API, we would have to change a lot of code. The alternative to Java sockets is remote method invocation, RMI, which we found out would suit us better. One of the advantages with RMI is that we could put every primitive in a wrapper class, so that we wouldn't have to change anything in the communication details. The usage of stack, provider and listener was given from the start, and we should only take care of the underlying communication and logics. When we designed the RI we got a little help from other people involved in the project.

5 Implementation and testing

In this section we are going to describe how we implemented the reference implementation. We also wanted describe how we compiled and executed the implementation, and what tools we used for those purposes. Finally this section contains a few words about testing process for the RI.

5.1 Implementation

We used the evolutionary method for implementing the code for the reference implementation. We started to write a little program that used RMI to communicate. When this was finished we changed the program so that both side could act as client and server, because we wanted a full-duplex communication.

When the communication worked successfully, we built a shell of the reference implementation. In this shell could we fill in the methods we needed in order to fulfil the project. When we had developed a working program to transfer a message object, we had to start implementing the program logic. For example when a *request* is sent, it will appear as an *indication* on the other side and a *confirm* message will appear as a *response* according to the OSI model.

5.2 Compilation

At first we had all Java-files in the same directory, but that isn't very beautiful, so it had to be changed. The obvious advantage of having it that way was that we didn't have to import our own code, because everything knew about everything else. At EIN, however, they have standard ways of structuring different parts of programs, so of course we had to adapt to them. We built it into a structure according to this standard and created a package for the RI. To facilitate the compilation we used a tool called Make for Java, JMK. Because of the modular structure it is necessary to compile three times with JMK, one time for each package (CTS, API and RI).

5.2.1 Javac

This is the command used to compile a Java source file into a Java class file.

5.2.2 Rmic

Rmic is an abbreviation for RMI compiler. It creates two new class files from another one, which use RMI. These files are the *stub* and the *skeleton*, and they are used to apply transparency for the communication between the client and the server. This way remote method calls looks like local method calls. This simplifies programming for communication over the Internet. An example is shown in Figure 5.1.

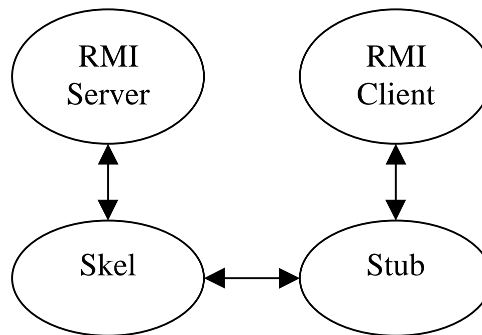

Figure 5.1: RMI communication structure

5.2.3 Make for Java

Make for Java, JMK, is an application in which you specify paths, arguments and different compiling operations that you want to use[3]. When we ran the makefile, it compiled and copied all necessary files to the pre-specified directory structure. One advantage of JMK is that we didn't have to remember all the classpaths, or write them in a command line for every time we wanted to compile. Because of the project structure contained three parts, CTS, API and RI we had to use one makefile for each part.

5.3 Execution

To make it possible for multiple users to execute the CTS, API and RI together, we put all the code into ClearCase[4].

5.3.1 Java

Java is the command we used to execute the Java class files

---

[3] For further information about Make for Java see section 6.2.3.
[4] For further information about ClearCase see section 6.2.2.

5.3.2 Rmiregistry

For a program that uses RMI it is necessary to start a background process called rmiregistry. This is a database containing all remote services available to an RMI client. When a client wants to know where a specific service is situated, it sends a query to the rmiregistry. The registry checks in the remote service database, and if the correct service is found it tells the client how to find it. If the service is not found in the database, an exception is thrown back to the client.

5.4 Testing

Since this project of developing the RI, CTS and the API has proceeded simultaneously, we have been able to test our code against each other, but in order to test the performance of the reference implementation we would have to develop a test application that would only test the performance. We couldn't do this within the range of the Bachelor's Project due to lack of time. However we could test the logic and the communication between the different parts using the CTS.

The intention of RI is to be able to test the applications usage of the API against something. This something (RI) is going to receive messages sent by the CTS above.

The CTS uses the primitives in the API with different parameters and order. To see whether or not the primitives works correctly downwards, we will use a connection to another instance of RI (with above layers such as API and CTS Server) stationed at a different computer.

5.4.1 White Box Test Model

Figure 5.2 describes the different steps performed in our test.

How to check the steps in the test model, we print out a line in the terminal for each step. There could we see the parameters in the events and check that it's correct.

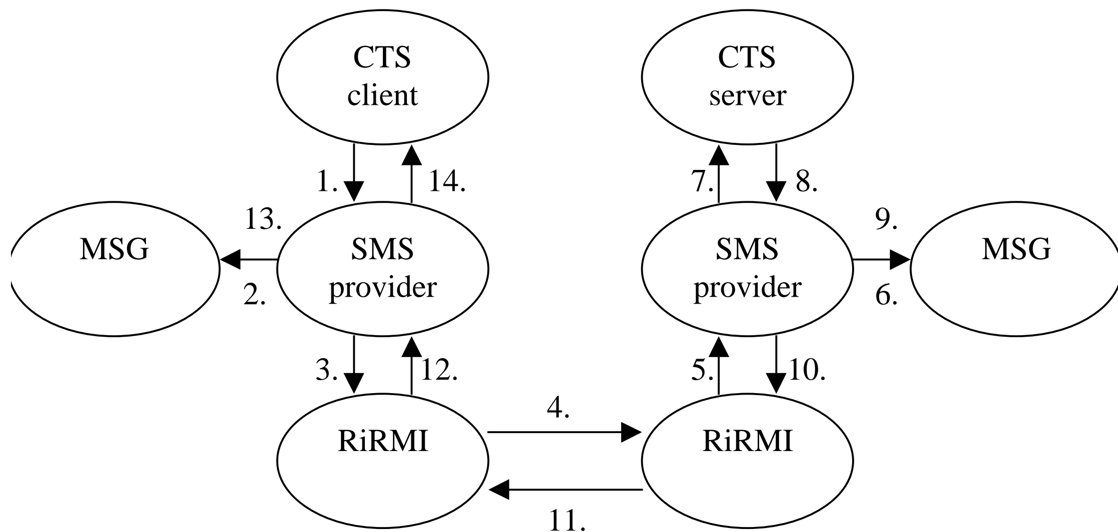

Figure 5.2: White box test case

1. CTS application send a *MtMessageReqEvent* to SMSProvideer.
2. Provider creates a Msg and put the event in to it.
3. SMSProvider sends the Msg object to RiRMI.
4. RiRMI send the object over to RiRMI server side.
5. RiRMI checks out where this package should be sent, and then sends it there.
6. SMSProvider unpacks the event from Msg object. If the event is a *MtMessageReqEvent* it converts it to *MtMessageIndEvent*.
7. SMSProvider sends the event up to the CTS application.
8. CTS sends a *MtMessageRespEvent* to SMSProvider.
9. Provider creates a Msg and put the event in to it.
10. SMSProvider sends the Msg object to RiRMI.
11. RiRMI send the object over to RiRMI client side.
12. RiRMI checks out where this package should be sent, and then sends it there.
13. SMSProvider unpacks the event from Msg object. If the event is a *MtMessageRespEvent* it converts it to *MtMessageConfEvent*.
14. SMSProvider sends the event up to the CTS application.

5.4.2 Black Box Test Model

In general one could abstract the test case as in Figure 5.3, where you can see if the logic within the program works as is should.

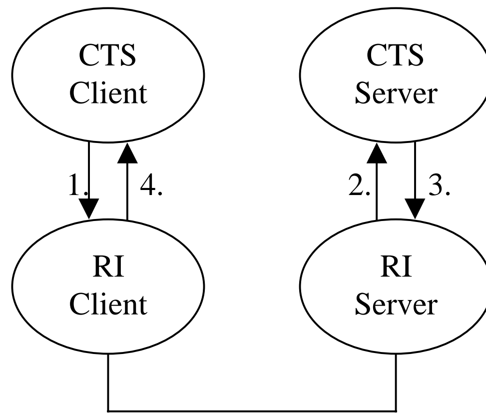

Figure 5.3: Black box test case one

1. CTS Client sends a MtMessageReqEvent to RI Client.
2. RI Server sends a MtMessageIndEvent to CTS Server.
3. CTS Server sends a MtMessageRespEvent to RI Server.
4. RI Client sends a MtMessageConfEvent to CTS Client.

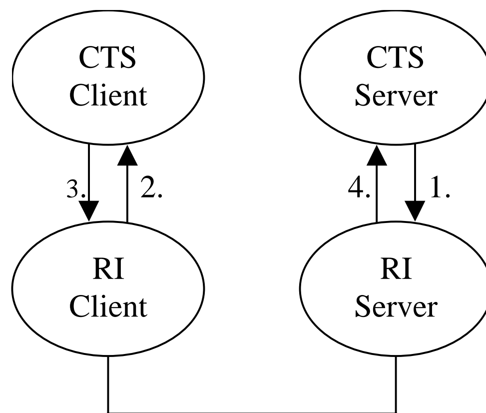

Figure 5.4: Black box test case two

1. CTS Server sends a MoMessageReqEvent to RI Server.
2. RI Client sends a MoMessageIndEvent to CTS Client.
3. CTS Client sends a MoMessageRespEvent to RI Client.
4. RI Server sends a MoMessageConfEvent to CTS Server.

5.4.3 Test Execution

To perform this test case we used the CTS application in Figure 5.5. In this program we can specify different primitives and their parameters that we want to send.

First we added a MtMessageReqEvent with default parameters, then we sent it to the server by pressing *Run*. When comparing what we sent and received, we could make the Black box test in 5.4.2.

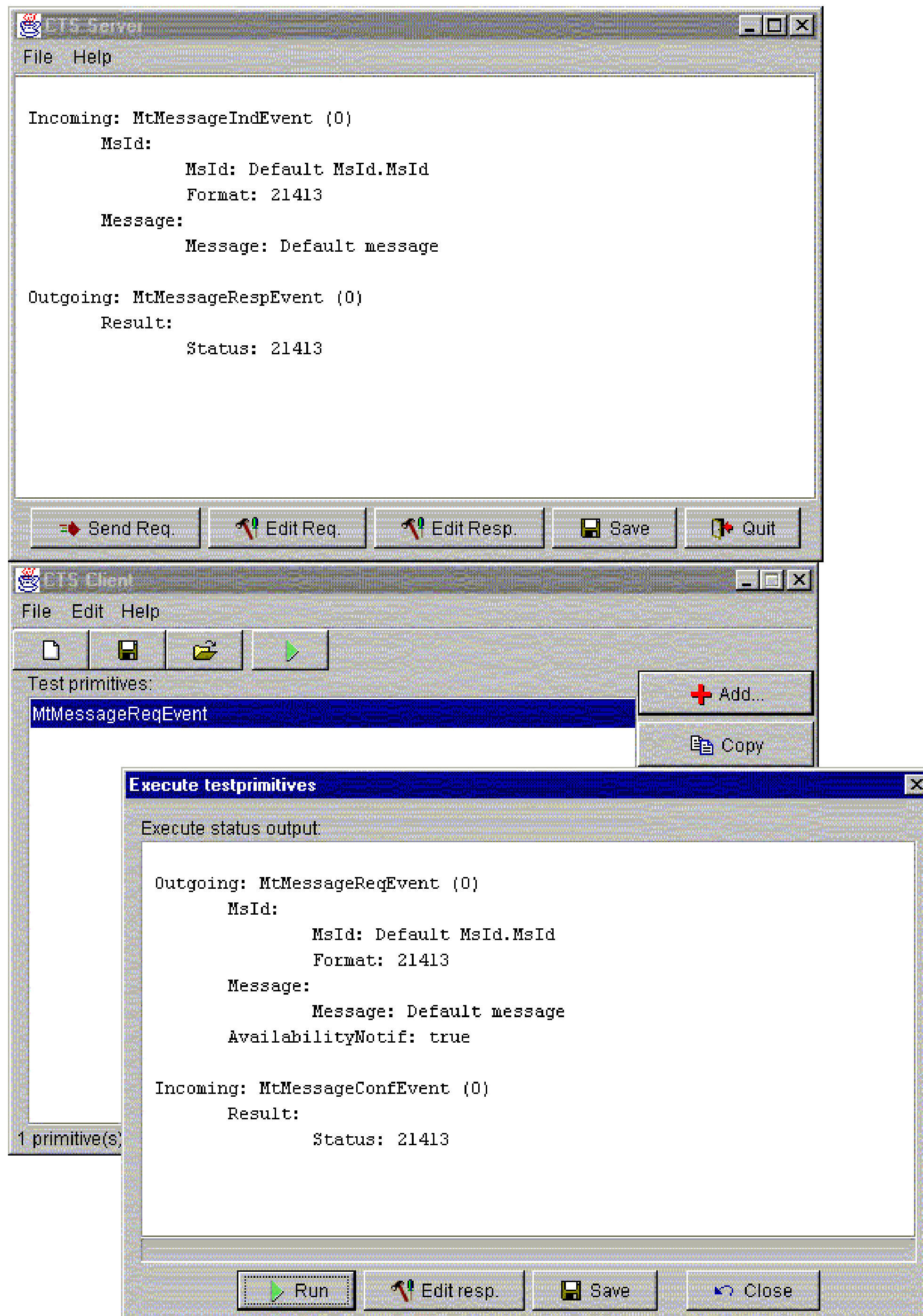

Figure 5.5: CTS application

6 Experiences and recommendations

In this section we want to explain what methods and programs we used. We also want to describe the advantages and the disadvantages of these methods and programs.

6.1 Methods

In this sub-section we want to explain the methods we used to develop our reference implementation. We only explain the ones we thought were worth to mention.

6.1.1 RMI

Remote method invocation, RMI, is a very useful method for calling methods on a non-local process. In the beginning it can be difficult to understand how the different parts of the RMI works together and why they are necessary. We started by making a small working program, and we thought that was a good way to start. We could afterwards rather easy expand the functionality, for example to a full-duplex communication.

To sum up our opinion about RMI we have to say that it is difficult to get started, but once you have understood, it is definitely worth the effort.

6.1.2 Package

Package is used to bind together different parts in a project. It is useful because you get a good structure over all files and you don't have to set so many different classpaths when you compile or execute your project. In our case there are three packages, one for each sub-project: CTS, API and RI.

To sum up our opinion about usage of package we want to say that we recommend it because it is easy to use and it looks professional.

6.1.3 Factory model

The meaning of factory model is to let someone create another object for you, and leave the control of the object to you. Why doing it this way? It is good to have one object that can create several objects for you, so that you don't have to know how to create them. In this way you can change underlying modules without having to change anything in the program that uses the factory. Another advantage with this model is that several processes can use the same object.

An example of factory model is shown in Figure 6.1, in which we describe in four steps how it works.

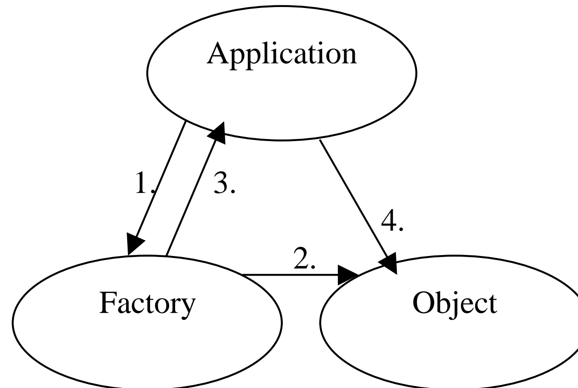

Figure 6.1: Factory model

1. An application asks the factory to create an object.
2. The factory creates an object.
3. Factory returns a reference to the object.
4. The application communicates directly with the object.

6.1.4 Listeners

Listeners are used for asynchronous communication, when two objects want to communicate with each other without having to wait for the other object to be prepared. We felt that listeners method is quite easy to learn and use. It works very well so we strongly recommend it.

An example of listeners is shown in Figure 6.2, in which we describe in six steps how it works.

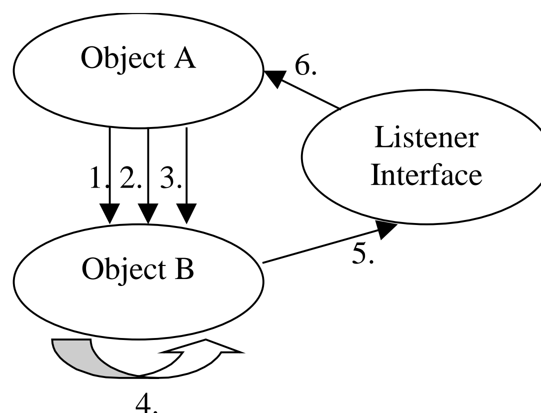

Figure 6.2: Listener model

1. Object A creates Object B

2. Object A adds it self to the listener in Object B
3. Object A calls Object B and then continues to work
4. Object B works on that Object A sent
5. Object B goes through the listener's list and send an event to every one on the listener's list.
6. Object A gets the event

6.2 Programs

In this sub-section we want to inform the reader about the programs we used to develop our reference implementation.

6.2.1 Visual SlickEdit 5.0

This is a code editor that can be used for several various programming languages. For example it is possible to edit code in C, C++, Pascal, Java, but also batchfiles, HTML code and JavaScript. We haven't explored the program's capabilities in the other languages, since we were only coding in Java, but we found that it was a very powerful tool for our purposes. The program features highlighted reserved words and a little cute window containing methods of opened Java-files amongst other goodies.

We learnt by mistake that one better install the Java Development Kit, JDK before installing Visual SlickEdit, because the program searches the computer for installed compilers during the installation process. If one install for example JDK afterwards, one would have to specify paths manually, which easily can be, and therefore should be, avoided.

Visual SlickEdit was the program we used the most when implementing the code for the reference implementation.

6.2.2 ClearCase NT3.2

This program is a version handler for files distributed on a server. When a user want to change the content of a file, he has to check out the file, make the change, and finally check in. The files are placed in a database on the server. If two programmers are working with the same file, it is possible to merge the result together so that nothing is lost.

The program has its advantages, but some of the operations featured could be a little less complicated to perform.

6.2.3 Make for Java 1.3

This is a program that makes the compiling process a little bit easier than to compile in a terminal window. It is easy to understand and to use, but you have to learn a new programming language to be able to use it. The basic idea is that you specify paths, arguments and different compiling operations that you want to use in a *makefile*. These operations are for example Javac and Rmic, which are called with different arguments depending on how you want the source code to be compiled. When running the makefile, it compiles and copies all necessary files to a pre-specified directory structure.

7 Conclusions

When working with this project we felt that we learned very much, and it was interesting and stimulating. At first, when we started, it was very difficult to estimate how long the work was going to take, and as anyone else would do we underestimated the time.

We have learned to use a few design patterns and methods we found very useful. If anyone would do a similar project which includes full duplex communication over the Internet we would strongly recommend Java RMI because of the high level of abstraction for the programmer. To save some time in large projects we strongly recommend to invest a few hours to learn Make for Java. It can simplify compilation very much, and the risk of forgetting any moment of the compilation is heavily reduced.

To complete the reference implementation, it is necessary to implement the rest of the providers within the API, including:

- Communicate with a service application, USSD
- Find out the status (on/off/occupied) and the location (which cell) of a MS
- Find out the position (x,y) of a MS, LCS

We would recommend that the implementation of these providers follow the same structure as we created in the SMS provider. In that way a lot of the code from the SMS provider can be reused.

In the final phase of the project we found that the execution of class files can be slower then necessary if the programmer includes entire libraries, for example the initial line "import java.io.*;" is inefficient, because it includes more than the program uses in most cases. A solution to make it faster is to include only the names of the used methods without wildcards.

If we had more time in this project, and if we were more involved in the development work of the API, we would try to enforce a few features not included now. For example a clone method for message events that are being sent.

References

[1] Matti Drisin, JAIN MAP Specification, http://java.sun.com/aboutJava/communityprocess/jsr/jsr_029_map.html, August 11, 1999

[2] Sun microsystems launches drive to bring benefits of Java™ software to telecom intelligent network -- blending intelligent networks and Internet technologies http://www.sun.com/smi/Press/sunflash/9806/sunflash.980609.9.html , 1998

[3] Sun Microsystems Inc, Java Remote Method Invocation (RMI), http://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html, 1999

[4] Henrik Bergkvist and Nicklas Jansson, Implementation of JAIN MAP API CTS, May 30, 2000

Appendix

A Abbreviations

API: Application Protocol Interface
CTS: Conformance Test Suite
EIN: Ericsson Infotech AB
HLR: Home Location Register
IN: Intelligent Network
JAIN: Java API Integrated Network / Java Advanced Intelligent Network
LCS: Location Service
MAP: Mobile Application Part
MS: Mobile Station (Mobile Telephone)
MTP: Message Transfer Part
OSI: Open Systems Interconnection
PLMN: Public Land Mobile Network
RI: Reference Implementation
RMI: Remote Method Invocation
SCCP: Signaling Connection Control Part
SMS: Short Message System
SS7: Signaling System No 7
TCAP: Transactions Capabilities Application Part
USSD: Unstructured Supplementary Service Data

B Expressions

Cell: This term is used to describe the position of a MS. A cell is limited by its horizontal and vertical borders. These borders could be compared to latitudes and longitudes of the common map system. The size of a cell may be X km^2. Where X is still not specified.

C Requirements specification - A simple JAIN MAP API RI

C.1 Background

Sun runs a standardization work called JAIN (Java API's for the Integrated Network) where they develop Java API's for different "telecom-protocols". An example of this telecom-protocol is MAP (Mobile Application Part). Within the limits of JAIN so are Ericsson leading a working team that develops a Java API for MAP. The functions of JAIN MAP API contain transmit/receive message like SMS and to get information about the mobile phone (ex. It's state and position). It's therefore a rather simple API with about five operations.

C.2 Task

As the development of the API progress, so will a reference implementation (RI) of the API be produced. The RI will not be developed on a real MAP-stack but under API it will be a test stub that's takes procedure calls to the API and returns back answers. The RI shall also be able to initiate test traffic on the Internet. Possibly make the RI a Java socket so it will be able to communicate with a RI on another machine, and possible a MMI (Man Machine Interface) will be needed.

The examination job contains to develop RI and if the time allows a performance test will be done. It should also develop a demonstrator for JAIN MAP API that can be shown on different exposition.

To sum up the examination job parts:

- In consultation with the staff at EIN do a system design for RI
- Write a specification in English for RI
- Implement RI
- Test RI including the API sources.
- Possible a performance test.
- Possible develop a demonstrator.

C.3 Purpose

The purpose of this examination job is to develop a good API-standard and then test it. To do that we test RI and it represent also that the API source will be tested and therefore it might change the API-standard.