A Transit Telephony Exchange Simulator Implemented in Erlang

Malin Pihl

March 2, 2006
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Ola Ågren
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Simulation tools are essential in building large and complex systems which both include lots of different hardware components, but also to a large extent, a complex and vast amount of software. A common practice is for companies to use simulation tools for the tedious testing efforts that are needed. This is to facilitate faster tests as well as to minimize the hardware costs. Hardware might not even exist and then simulation tools are the only way forward.

This report describes the implementation of a simulation tool, for analysis of ISUP signaling, as an extension to an already existing tool used to test telecommunication products.

The final product was a simulation tool implemented in Erlang and C. The report describes the development of the tool and also gives an insight in telecommunication systems and the tools used for the implementation. Moreover, the report contains a study of how the language Erlang is suited for development of telecommunication applications.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 The Task . . . 2
1.3 Layout of the Thesis . . . 3

2 Basic Concepts and Prerequisites **5**
2.1 The ISUP Protocol . . . 5
2.1.1 The Protocol Messages . . . 6
2.1.2 ISUP Procedures . . . 8
2.2 The Signaling System number 7 (SS7) Protocol Stack . . . 10
2.2.1 Overview of the Protocol Stack . . . 10
2.2.2 Definitions . . . 12
2.2.3 Configuration . . . 12
2.2.4 The API Interface . . . 13
2.3 The Engine Integral Solution . . . 15
2.3.1 The AXE-10 . . . 16
2.3.2 The Mediation Logic (ML) . . . 16
2.3.3 The Media Gateway (MG) . . . 16
2.4 Other Relevant Protocols . . . 17
2.4.1 The Transport Handler Protocol (TRH) . . . 17
2.4.2 The Switch Device Management Protocol (SDM) . . . 17
2.4.3 The Connection Management Protocol (CM) . . . 17

3 Problem Description **19**
3.1 Motivation . . . 19
3.2 The Assignment . . . 19
3.3 Timeplan and Method . . . 20

4 Erlang in Telecommunication Applications **23**
4.1 Introduction to Erlang . . . 23
4.1.1 History . . . 23

4.1.2 Overview of the Erlang Language . . . 24
4.1.3 Sequential Programming . . . 25
4.1.4 Concurrent Programming . . . 28
4.1.5 Error Handling . . . 29
4.1.6 Distributed Programming . . . 31
4.1.7 Ports . . . 31
4.1.8 Dynamic Code Change . . . 31
4.1.9 Generic Behaviors . . . 31
4.1.10 Supervision Hierarchies . . . 32
4.1.11 Open Telecom Platform (OTP) . . . 32
4.2 Erlang and Telecommunication Applications . . . 33
4.2.1 Requirements on Telecommunication Applications . . . 33
4.2.2 Overview of Erlang Design . . . 35
4.2.3 Erlang used in Telecommunication Applications . . . 35
4.2.4 Discussion . . . 41

5 Accomplishment 43
5.1 Development . . . 43
5.1.1 MS1-MS3 . . . 43
5.1.2 MS4 . . . 44
5.1.3 MS5 . . . 44
5.1.4 MS6-MS7 . . . 44

6 Results 47
6.1 The Application xSIM . . . 47
6.1.1 The Structure of the Erlang Program . . . 47
6.1.2 The Structure of the C Program . . . 49
6.1.3 The Communication Protocol . . . 49
6.1.4 The GUI . . . 50
6.2 The Configuration of the SS7 Protocol Stack . . . 51
6.3 The Traffic Generator . . . 53
6.4 Tests . . . 53
6.4.1 Layout of the Test . . . 53

7 Summary and Conclusions 55
7.1 Problems during the Development . . . 55
7.2 Limitations and Restrictions . . . 55
7.3 Future Work . . . 56
7.3.1 Dynamic and General Solution . . . 56
7.3.2 Controlling the Traffic Generator . . . 56
7.3.3 Error Handling . . . 57

8 Acknowledgments 59

References 61

A Abbreviations and Acronyms 63
A.1 ISUP Protocol 63
A.2 Telephony Systems 63

List of Figures

1.1 User A calling User B 2

2.1 Overview of the PSTN network 6
2.2 Overview of a successful call setup using ISUP 8
2.3 Overview of a normal call release 9
2.4 The protocol layers in the TietoEnator SS7 Protocol Stack 11
2.5 Links, link sets, routes and route sets 12
2.6 Configuration files 13
2.7 The callbacks and the request/response functions 14
2.8 The system architecture of the EIN solution 15

3.1 xSIM in the system 20

4.1 C compared to Erlang 40
4.2 Java compared to Erlang 40

6.1 Overview of xSIM 48
6.2 The GUI for the Application xSIM 50
6.3 The Configuration of the stack and the Spectra 52

List of Tables

4.1 Exceptions in Erlang 30
4.2 Erlang matched against requirements on telecommunication switching systems 42

Chapter 1

Introduction

This thesis was conducted at a test department at Ericsson AB in Älvsjö from September 2005 to the beginning of February 2006.

1.1 Background

The Public Switched Telephony Network (PSTN) is the world's collection of interconnected voice-oriented public telephone networks. It was originally a circuit-switched network, i.e. a network in where a connection via signaling has to be established prior to letting a telephone call through, but parts of it is now beginning to be replaced by an IP-network.

In the field of telecommunications, a telephone exchange (or telephony switch) is a piece of equipment that connects phone calls. It is what makes phone calls work in the sense of making connections and relaying the speech information. When a subscriber makes a telephone call the analog signals on the two copper wires from the telephone are digitized into a channel of the telephone exchange. The digital format is used both internally in the exchange and between exchanges.

In order to establish a call between two telephone exchanges information is exchanged using different and standardized signaling protocols. The most common signaling protocol used in existing telephony systems is Integrated Services Digital Network User Part (ISUP). Both ISUP and telephony systems will be described into more detail later in the report.

When telephony operators expressed interest in replacing the circuit switched connections between the exchanges with a packet backbone network, Ericsson AB provided the Engine Integral Network (EIN) as a solution. In comparison with a circuit switched network, a packet backbone network do not need to establish a path through the network for each call but instead divides data into smaller units called packets and send these individually between nodes in the network to their destination. This enables the links used for routing to be shared by many other nodes.

1.2 The Task

All subscribers (users) in a telephone network are connected to a telephone exchange. The purpose of the telephone exchange is to set up connections to other exchanges in order to let through telephone calls conducted by the subscribers.

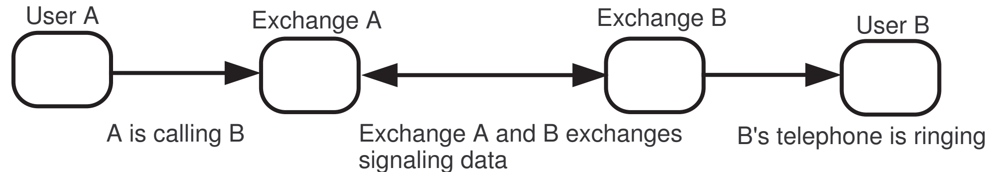

Figure 1.1: User A calling User B

Prior to the establishment of a telephone call the calling subscriber's exchange, A, and the called subscriber's exchange, B, must exchange information, so called signaling data, about the telephone call (as depicted in Figure 1.1). Other information exchanged, is e.g. address information necessary for routing of the call through the telephone network or certain parameters that has to be agreed upon by the two exchanges. Exchange B must also confirm that the called subscriber is available, i.e. that the telephone is not busy, and that no extra features such as "Call Forwarding", are active.

After all necessary information has been exchanged and all valid checks have been performed the called subscriber's telephone will start ringing. When the subscriber answers the telephone call more information will be sent between the two exchanges in order to through-connect the call. When the through-connection has been made the two subscribers can talk with each other.

When one of the subscribers terminates the call, necessary signaling data are once again exchanged between exchanges A and B to terminate and remove the earlier connection.

This signaling data must also be handled in IP-backbone networks in order to set up telephone calls between subscribers, and it is handled in Ericsson's EIN solution. In order for Ericsson to test their products they need to set up a network with all the involved components and test several situations and all possible fault scenarios which might happen. This is a tedious task which often includes advanced and very complex configurations. Sometimes, hardware are too expensive or not available to use for testing, and in such situations software simulating hardware is a preferable choice. Moreover, in order to test a telephone exchange, subscriber's connected to it are needed, and as it is not possible to actually have a setup with thousands of subscribers a simulation tool is preferable also in this case. Therefore Ericsson developed a simulation tool, which generates data signaling, and by that simulates several subscribers calling each other. A limitation with this software was that it was not able to receive and handle real signaling data. The task was to extend the already existing simulation tool with support for analyzing signaling data. This is described in more detail in Section 3.2.

1.3 Layout of the Thesis

This section will give an overview of the different chapters in the thesis.

Chapter 2 explains and describes basic concepts related to the thesis. The information here is needed to understand the thesis.

Chapter 3 contains the detailed problem description of the work. Here concepts in chapter 2 are used to explain what the purpose of the work was.

Chapter 4 examines whether Erlang is a suitable programming language for programming telecommunication systems. An overview of the language is given, followed by what requirements telecommunication applications need to fulfill and thus what requirements Erlang needs to fulfill in order to be a suitable language for implementing such applications.

In chapter 5 the development phase of the application is described.

Chapter 6 presents the resulting simulation tool that was developed. A description of the system together with pictures of the GUI and the system architecture are presented.

In chapter 7 conclusions are derived and discussed regarding the work. Limitations and further restrictions are also discussed. Furthermore, some thoughts about future work is presented.

Chapter 8 acknowledges people who have helped me during the work.

Chapter 2

Basic Concepts and Prerequisites

In order to describe the work using the appropriate abbreviations and terms, knowledge in some specific areas is necessary. The purpose of this chapter is to describe some basic concepts relevant to the thesis.

2.1 The ISUP Protocol

ISUP, or ISDN User Part, defines the protocol and procedures for setting up, managing and releasing trunk circuits in a telephony network. This section will describe the most essential parts of the protocol.

The network used in the description is an old circuit-switched network and not an IP-backbone network. In the circuit switched network telephones are connected to so called exchanges, also referred to as a Service Switching Point (SSP), by a voice trunk. SSPs are in turn connected to each other and also to Signal Transfer Points (STPs) by SS7 signaling links. There are also Service Control Points (SCPs) in the network as can be seen in Figure 2.1. All these components are referred to as Signaling Points (SPs) [23].

SSPs are switches which converts a dialed number from a subscriber line into SS7 signaling messages and send these using the ISUP or the Transaction Capabilities Application Part (TCAP) protocol to other SSPs. The messages are sent in order to setup, manage and release calls made by users. The SSPs may also send requests to the SCPs to acquire information on how to route a call. Basically the SSPs are those that originates or terminates a call [23].

An STP is a router or gateway in the SS7 network and its purpose is to switch SS7 messages between signaling points. Therefore are all SSPs and STPs set up in pairs. Moreover, if an originating SSP does not know the address of a destination SSP the STP must provide it [23].

An SCP is an interface to applications, for example databases, and must provide access to it. The protocol that is used to access and interface a database application is TCAP [23].

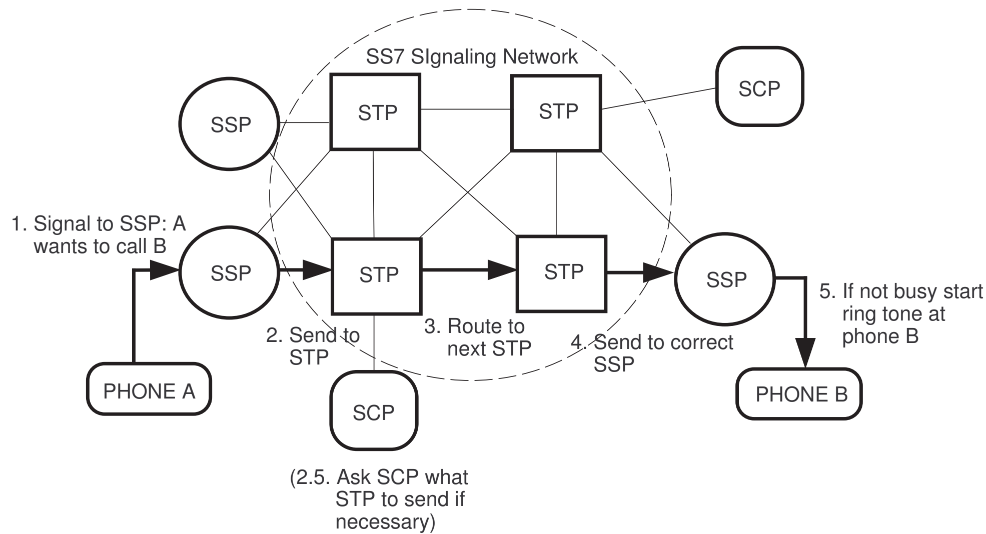

Figure 2.1: Overview of the PSTN network

2.1.1 The Protocol Messages

In order to set up calls and also tear them down, a number of different messages, or primitives, has to be used. Only the most important ones will be described in this section.

2.1.1.1 Initial Address Message (IAM)

The IAM is sent in the forward direction, i.e. from the calling party to the called party, to initiate seizure of an outgoing circuit and to transmit relevant information about the routing and handling of the call [13].

2.1.1.2 Address Complete Message (ACM)

This message is sent in the backward direction and indicates that all the address signals required for routing the call to the called party has been received [13].

2.1.1.3 Answer Message (ANM)

This message is also sent in the backward direction and indicates that the called party has answered the call [13].

2.1.1.4 Release Message (REL)

The message can be sent in either direction to indicate that a specific circuit is being released due to a reason given in the message [13].

2.1.1.5 Release Complete Message (RLC)

Answer to reception of an REL-message or if appropriate to a reset circuit message when the circuit concerned has been brought into the idle condition [13].

2.1.1.6 Suspend Message (SUS)

The SUS-message is sent in either direction and indicates that the calling or called party has been temporarily disconnected [13].

2.1.1.7 Resume Message (RES)

A message which is sent in either direction indicating that the calling or called party has been reconnected after being suspended [13].

2.1.1.8 Continuity Check Request (CCR)

This message is sent by an exchange to an adjacent exchange to request that continuity checking equipment should be attached for a continuity check [13]. Continuity checks need to be performed because signaling always uses the same channel and it is thus not possible to know if the other channels used for the telephone calls are free or blocked at an exchange.

2.1.1.9 Connect Message (CON)

The CON-message is sent in the backward direction to inform the originating access that all address signals required for routing the call to the called party has been received and also, that the call has been answered. This can be sent instead of an ACM and an ANM [13].

2.1.1.10 Continuity Message (COT)

A message sent in the forward direction to inform if there is continuity or not on the preceeding circuits and on the selected circuit to the following exchange [13].

2.1.1.11 Call Progress (CPG)

The CPG-message can be sent in either direction during the setup or active phase of the call to indicate that an event has occurred [13].

2.1.1.12 Segmentation Message (SGM)

This is used when an ISUP message sent has exceeded the octet limit of ISUP messages. The message will then be segmented and a segmentation message is sent to notify the receiving end that an additional segment of the message is coming [13].

2.1.2 ISUP Procedures

This section describes how the protocol works during call setup and release phase together with suspension of calls.

2.1.2.1 Successful Call Setup

To accomplish a successful call setup three messages are used, the IAM, ACM and ANM.

The IAM Message
When a person A would like to call a person B (as in Figure 2.2), data is going from A's telephone to the exchange it is connected to. This exchange determines certain information about the calling party and the called party. When it has analyzed the called number (the number of user B) and knows how to route the call, a selection of a free, suitable, inter-exchange circuit takes place and an IAM-message is sent to the succeeding exchange. Immediately after the IAM is sent the through-connection of the transmission path will be completed in the backward direction. The through-connection in the forward direction will not be completed until an ANM or CON is received [14]. If the IAM being sent exceeds the limit of 272 octets it will be segmented and an SGM-message will be sent to indicate that additional segments will follow as explained in Section 2.1.1.12 [13].

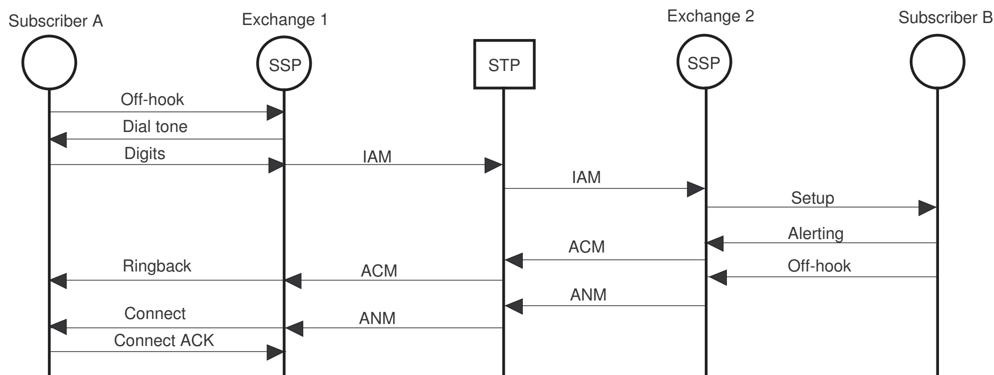

Figure 2.2: Overview of a successful call setup using ISUP

If the exchange receiving the IAM is an intermediate exchange in the route path it will continue to route the message to the next succeeding exchange. Before continuing to send the IAM it will choose a free intermediate circuit and seize this. After sending the IAM the through-connection will be completed in both directions in the intermediate exchange [14].

When a destination or terminating exchange (Exchange 2 in Figure 2.2) receives the IAM-message it will analyze the data within it to determine which locally connected subscriber it should connect to. The exchange will also make sure the called subscriber's circuit is free and available and that nothing prevents a successful call, before setting up a connection to the called subscriber. Moreover, the exchange will also start the ring tone (of the telephone) at the called subscriber (B) [14].

The ACM/CON Message
As soon as the destination exchange determines that the complete called party number has been received it will send an ACM back to the calling party. However, if a connect indication is received from the called subscriber (B) and no ACM has yet been sent a CON will be sent instead of the ACM [14].

Upon reception of an ACM or CON-message in an intermediate exchange the exchange will continue forwarding the message to the preceeding exchange.

When the originating exchange (Exchange 1 in Figure 2.2)receives the ACM-message it will start a timer in order to wait for the following ANM-message. If a timeout occurs, before receiving any ANM, the connection will be released and an indication will be sent to the calling party. If the exchange instead receives a CON it will complete the through-connection in the forward direction and the whole connection is then complete [14].

The ANM Message
When a called party answers the call the destination exchange will through-connect the transmission path and the ringing tone is removed if applicable. An ANM will also be sent to the preceeding exchange [14].

An intermediate exchange will as usual forward the message and when the originating exchange (Exchange 1 in Figure 2.2) receives the ANM, which indicates that the required connections has been completed, the transmission path is through-connected in the forward direction [14].

2.1.2.2 Normal Call Release

The call release procedure is based upon two messages, the REL and the RLC. The REL initiates the release of the circuit switched connection and the RLC is a confirmation of the release.

Release initiated by the Calling Party
When the calling party hangs up the originating exchange will receive a request to release the call from the telephone. It will then immediately start the release of the switched path by sending an REL to the succeeding exchange as can be seen in Figure 2.3 [14].

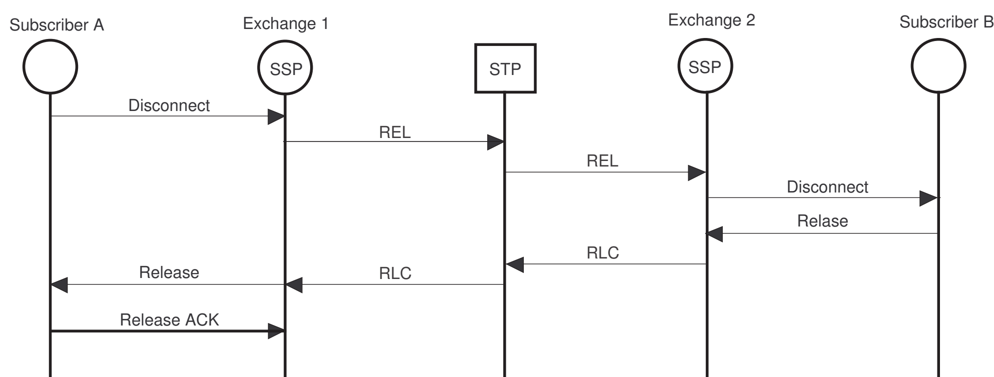

Figure 2.3: Overview of a normal call release

When an intermediate exchange receives a REL it starts the release of the switched path and sends a REL to either the preceeding or succeeding exchange depending on from where the message came. When the circuit is re-selectable (idle) an RLC is sent to the exchange which sent the REL [14].

When the destination exchange receives the REL it will start to release the switched path and when it is ready for circuit re-selection it will send an RLC to the preceeding exchange [14].

2.1.2.3 Suspend and Resume Procedures

The SUS-message indicates that the connection will be temporarily suspended without releasing the circuit. This can only be accepted during the conversation or data phase. When a destination exchange receives an on-hook condition it may send a suspend message to the preceeding exchange and an intermediate exchange will forward the SUS to the preceeding exchange [14].

At the originating exchange, when receiving a SUS-message or on-hook condition, it begins to wait for either an off-hook condition, re-answer message (RES), a REL or an RLC. If applicable it will also send a SUS-message to its preceeding exchange [14].

The RES message indicates a request to restart communication after a preceeding SUS message. Any request to release the call from the calling party will override any SUS or RES message and the connection will be released [14].

On receipt of a re-answer indication at the interworking exchange or off-hook condition at the destination exchange the exchange may send a RES-message to the preceeding exchange in the case when an earlier SUS-message has been sent [14].

When an originating exchange receives an off-hook condition, re-answer indication or a RES it will if applicable send a RES to the preceeding exchange and then will communication continue as before in the call. If the exchange receive a REL it will release the call according to the call release procedure [14].

2.2 The Signaling System number 7 (SS7) Protocol Stack

In this assignment an SS7 protocol stack implemented by TietoEnator was used. An SS7 protocol stack is a collection of protocols used for signaling in telecommunication networks. With signaling one is referring to all data that need to be transferred in a telephony system in order to provide services to its users, i.e all data except for the speech.

There are several different standards of SS7, for example ANSI and ITU. In this case ITU is being used and described.

2.2.1 Overview of the Protocol Stack

The SS7 protocol stack used consists of several layers; the Message Transfer Part Layer 1 (MTP-L1), Layer 2 (MTP-L2) and Layer 3 (MTP-L3) as can be seen in the Figure 2.4.

On top of these layers are user-defined layers which handles different signaling protocols. Some examples of these layers are ISUP, Telephone User Part (TUP) and TCAP. In this case the most essential and used layer, except for the MTP-layers, is ISUP [17].

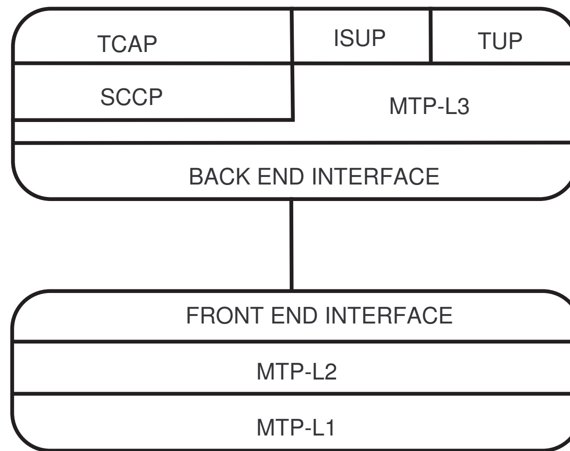

Figure 2.4: The protocol layers in the TietoEnator SS7 Protocol Stack

The MTP-L1 is equal to the physical layer in a TCP/IP stack and has responsibilities such as transmitting binary digits from one Signaling Point (SP) (see Section 2.1.2) to another. Furthermore, the layer is also concerned with physical, electrical and functional properties of the links connecting the SPs [17].

The second layer (MTP-L2) is equivalent to the link layer in an ordinary TCP/IP stack and is responsible for end-to-end transmission of a message across a signaling link. That includes responsibility of flow control, message sequence validation and error checking. The layer is aware of links but not of any nodes (SPs) in the network [17].

The third layer (MTP-L3) of the stack is equivalent to the network layer in a TCP/IP stack. It has awareness of all the SPs in the network each with a unique Signaling Point Code (SPC). The layer is responsible for the routing between an originating signaling point and a destination signaling point and also the link management. Moreover it handles discrimination, distribution and routing of signaling messages [17].

The stack is property of TietoEnator and is implemented as a Front End (FE) and a Back End (BE) (see Figure 2.4). This is to ease the usage of the stack in a distributed environment. The two parts can either be executing on the same machine or on different machines as a distributed system. The BE handles the third layer and the user layers while the FE handles the second and the first layer. Both the BE and the FE has a common interface to use for communication between each other.

A graphical tool was also available from TietoEnator in order to manage the SS7 protocol stack, which is referred to as the ss7-manager.

2.2.2 Definitions

To understand everything about the configuration and the stack some definitions are needed to be explained. Figure 2.5 explains the difference between links, link sets, routes and route sets.

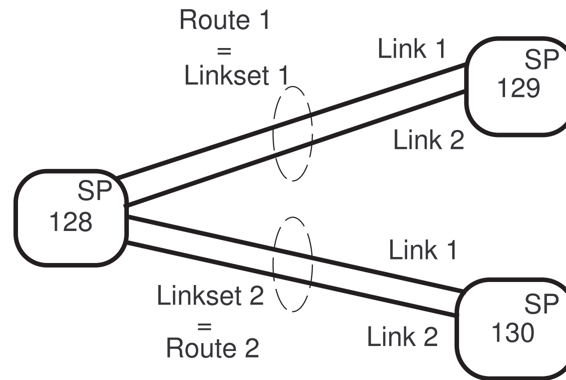

Figure 2.5: Links, link sets, routes and route sets

Every link set consists of one or more links and a route consists of one link set. Furthermore, a route set consists of one or more routes [17].

2.2.3 Configuration

The configuration is done by defining certain parameters in configuration files that has the suffix `.cnf`. The files used are `cp.cnf`, `ss7.cnf`, `l2.cnf`, `mux_v2.cnf`, `ife_v2.cnf` and `ss7mgr.cnf`. The different files configure different parts of the stack layers as can be seen in Figure 2.6.

The configuration of the Common Parts (CP), which handles all the TCP/IP traffic between the layers, is done in `cp.cnf`. In `ss7.cnf` configuration of layers two, three and the user layers takes place. The `ss7mgr.cnf` handles the configuration of the graphical ss7-manager program. The rest of the configuration files handles the configuration of both the FE interface and more hardware related parameters.

2.2.3.1 Configuration of mux_v2.cnf

On the E1-board used there exist two external trunks towards the network. These trunks are referred to as PCMA and PCMB respectively. Between the FE and the BE in the stack there is an internal and logical trunk called PRA.

Both the external trunks and the internal trunk has 32 timeslots or channels available for use. In order to let messages go from the external trunks over the internal trunk and up to the BE, a mapping must be made between the timeslots in the external trunks and in the internal trunk. This is accomplished in the file `mux_v2.cnf`.

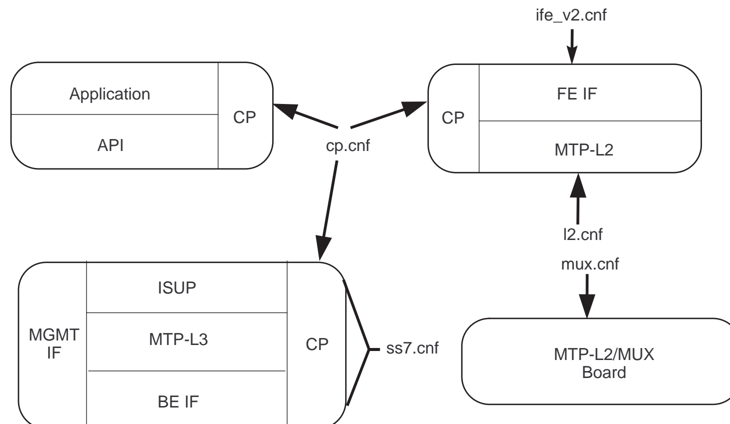

Figure 2.6: Configuration files

2.2.3.2 Configuration of cp.cnf

The Common Parts (CP) takes care of all TCP/IP traffic between the layers and the processes. Moreover, it also handles the logs, memory allocation and timer handling. In the file `cp.cnf` is common parts configured regarding message size, log file settings, TCP/IP ports and redirection of layers to common ports.

2.2.3.3 Configuration of ss7.cnf

In this file the configuration of MTP-L3, the BE interface and all the user modules is performed. Each layer will have its own section with configuration parameters.

The configuration of the BE interface mostly sets some timeout values and performs a mapping between the logical channels in the BE and the physical channels in the FE. The purpose is to give the channels in the internal trunk logical names to be accessed by the BE.

The layer above the BE interface is the MTP-L3 which defines links, routes, link sets and route sets. Different timers and priorities of messages are also defined there.

Other user layers such as ISUP or TCAP are configured above this section. The main goal of this configuration is to define the network architecture which the stack is connected to.

2.2.4 The API Interface

The ISUP API is divided into two parts, the request- and response functions and the indication- and confirmation functions. Both parts are implemented as C-functions where each request and response or indication and confirmation function is mapped to a specific ISUP message.

The API package consists of C libfiles (.a) and libraries, which the application using the API must link into its source code. The libraries are threadsafe and can thus be be used by threaded applications without any problems [17].

2.2.4.1 Using the API

Before the API can be used it must be initiated by calling some setup functions. In the same way before terminating the stack some functions must be called, e.g., to release allocated memory.

In the API each request and response function represents an ISUP message that can be sent through the stack to some other node in the network. Every request and response has its own function which checks all data sent to it before transmitting it.

The interface has two processor flags where one, if defined, means that all pre-defined callback functions are unavailable. A callback function is a function that is called when an indication or a confirmation is received from the stack. If the pre-defined functions are not available the users must write their own callback functions. Before use, the user must also register the callback functions in the stack via yet another API call [17].

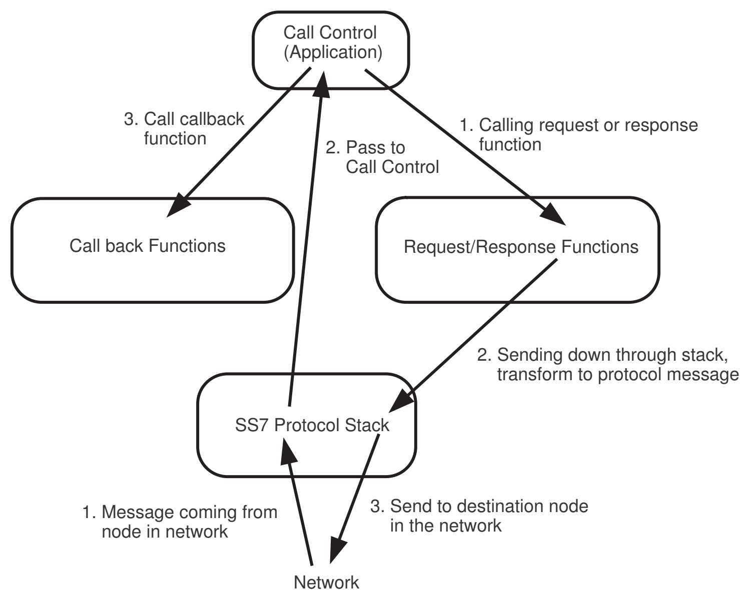

Figure 2.7: The callbacks and the request/response functions

2.2.4.2 Requests and Responses

Basically the request and response functions will tell the stack to send a specific ISUP message to some node in the network as can be seen in Figure 2.7 (1-3 on the right hand side). The parameters to the function will provide the message with data.

2.2.4.3 Indications and Confirmations

When an ISUP message is received by the stack it will interpret the message and then pass it to call control (application or user). Call control calls the correct defined callback function by using an API function as can be seen in Figure 2.7 (1-3 on the left hand side).

2.3 The Engine Integral Solution

The EIN solution, provided by Ericsson AB, consists of three components. The AXE-10 (telephony exchange), the AXD 301 and a number of Media Gateways (MGs). The AXD 301 is also referred to as the Mediation Logic (ML). The AXE-10 and the ML will together form a so called Media Gateway Controller (MGC) or telephony server. The server is the controlling part which analyzes the signaling and sends orders to the MGs. The actual connections through the network are made in the MGs [19]. The architecture of the system is shown in Figure 2.8.

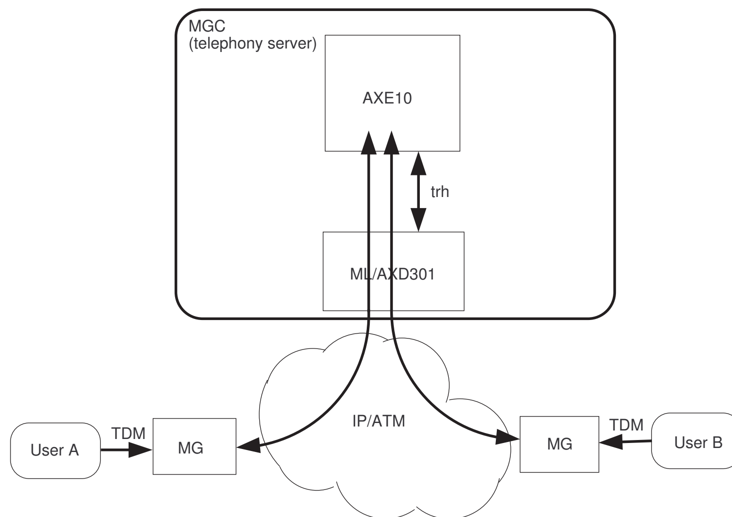

Figure 2.8: The system architecture of the EIN solution

If a user A (as in Figure 2.8) is to make a phone call to user B, signaling data (ISUP) will be transmitted from A to the MG A is connected to. The MG will tunnel the signaling data through the backbone network and the ML up to the AXE-10. The AXE-10 will analyze the signaling and send directives to the ML using a control link between them. The directives are sent using a protocol called Transport Handler (TRH). The ML sends instructions to the MGs involved instructing them to set up a connection. The protocol used for this is the standardized H.248 protocol[1] [19].

[1] H.248 is a common protocol used to control one or many MGs in a standardized way [21]

2.3.1 The AXE-10

This is the part of the EIN solution where the ISUP signaling being tunneled from the MGs through the ML is analyzed and handled. The AXE-10 will instruct the ML what to do in response to the received ISUP traffic [19].

2.3.2 The Mediation Logic (ML)

The ML is controlled by the AXE-10 and is connected to it by a control link, which is used by the AXE-10 to send control directives to the ML.

The main purpose of the ML is to translate commands and messages going between the AXE-10 and the MGs. It also maintains the data consistency between the MGC and the MGs by using separate so called audits between itself, the AXE-10 and all the MGs [19].

To transfer data over the trh-link between the AXE-10 and the ML the TRH protocol is used. The trh-link carries seven protocols where the TRH protocol is one of them. The seven protocols are:

1. Transport Handler (TRH)
2. Switch Device Management (SDM)
3. Connection Management (CM)
4. Bearer Request Management (BRM)
5. Device Audit Management (DAM)
6. Media Gateway Management (MGM)
7. SemiPermanent Connection Management (SCM)

The most used protocols in this thesis is the TRH protocol, the Switch Device Management (SDM) protocol and the Connection Management (CM) protocol. Therefore, these will be described into more detail than the others in Section 2.4.

2.3.3 The Media Gateway (MG)

The media gateways are in fact an AXD 301 platform, but configured in a slight different manner in comparison with the ML. It is possible to configure the AXD 301 platform to host more than one MG [19].

A port is an interface to the physical link on an Exchange Terminal (ET), where each port can either have 31 or 24 terminations depending on what hardware is being used. A physical termination belongs to a single MG and can be used to set up connections to other physical terminations on other MGs [19].

The main purpose of MGs are to translate the TDM-data coming in on one side into Asynchronous Transfer Mode (ATM) data or IP-packets on the other side depending on the type of network used [19]. The MGs will also notify the MGC of any changes in the service state of a termination or a group of terminations.

2.4 Other Relevant Protocols

The AXE-10 uses seven different protocols to control the ML. The purpose of this Section is to describe the protocols essential to this thesis.

In all protocols the AXE-10 will be referred to as the (switch) master and the ML will be referred to as the (switch) slave.

2.4.1 The Transport Handler Protocol (TRH)

This protocol is a transmission protocol for messages between an application in the AXE-10 and an application in the AXD 301. The information provided in this protocol is used for routing of messages between e.g. two applications in AXD 301 and AXE-10. Moreover, the protocol also implements a heartbeat function in order to check the availability of the links residing between the applications in the AXD 301 and the AXE-10 [24].

The supervision of the communication is performed by the use of this protocol but it is not a reliable transport protocol, thus messages can be lost.

2.4.2 The Switch Device Management Protocol (SDM)

The purpose of this protocol is to let an AXE-10 (master) configure and manage physical access points available on the MG's in the network and controlled by the ML connected to the master [22].

In order for the master to have a logical view of the network and a way to control the available resources available to it, this protocol is used to build a mapping between the access points in the MG's and a logical identity. This identity is maintained by the slave and is later used in the CM protocol [22].

2.4.3 The Connection Management Protocol (CM)

The connection management protocol allows the master to establish, manage, release connections of type ATM or IP and to start or stop tones on telephones connected to an MG. In other words it is used to control connections between the MGs. The access points (in the MG) must have been defined earlier by the SDM-protocol or the BRM-protocol and each physical termination in every MG will then be associated with a switch device identity [27].

When the master wishes to set up a connection on a termination in a MG it will use the switch device identity in its message to the slave. The slave will then use its mapping-table to see which termination and which MG that is referred to [27]. After the request from the master has been performed the slave will report the result of the request to the master in a response message.

Chapter 3

Problem Description

The task of this thesis was to implement a simulation tool which should terminate ISUP signaling. It would then be used for testing some existing and future products at Ericsson.

3.1 Motivation

Ericsson has developed a tool called xAPP which is used to simulate part of the AXE-10 behavior. This is done in order to minimize cost and also to decrease the time spent in test preparation. Several parameters can be set in xAPP to for e.g. controlling the call rate.

A limitation with this sort of software simulation was that xAPP could not handle and terminate any tunneled ISUP messages from the MGs. It was therefore necessary to develop an extension to xAPP, which could analyze ISUP signaling.

3.2 The Assignment

The assignment was to develop a tool which analyzes and terminates incoming ISUP signaling. The resulting application should interact and make use of xAPP and thereby simulate the behavior of an AXE-10. The implementation should be done in Erlang and should be tested on a PC where the PC would act as the AXE-10 exchange. A real ML and MGs would be connected to the PC containing an E1/T1-card for reception and transmission of ISUP messages.

As can be seen in Figure 3.1 the tool created was to be integrated into xAPP by communicating with the modules trhAXE and xAXE to control the ML.

To accomplish the task, an existing SS7 protocol stack was available. The stack had all the necessary layers for receiving and transmitting ISUP messages through the E1/T1-card.

To generate ISUP signaling a traffic generator of type Spectra was used. It was connected to the E1/T1-card in the PC using two separate trunks (links). The traffic

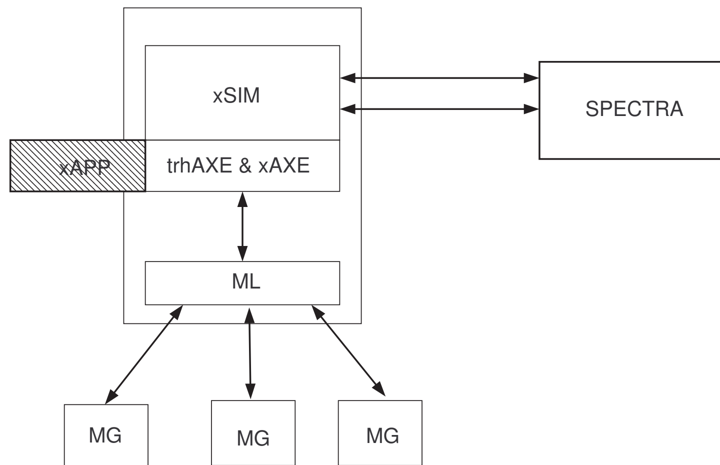

Figure 3.1: xSIM in the system

generator would simulate two signaling points, or telephony users, wanting to establish calls between them. To simulate these scenarios the simulation tool had to terminate ISUP signaling coming to it from the traffic generator. By interpreting the data in the ISUP messages it could then send commands to the ML which in turn transformed the commands into H.248 standardized messages and delivered these to the involved MGs.

The goal of the simulation tool was to handle establishment and release of 100 calls/s. The tool should also be robust and stable enough to be tested during several days. Basically the assignment included the following:

1. install and configure the SS7 protocol stack,
2. write a C-interface which interacts with the API of the SS7 protocol stack and is reusable,
3. write an Erlang program which interacts with the C-interface and the existing tool xAPP to simulate a part of the AXE behavior,
4. use the Spectra to generate ISUP traffic, and
5. create a GUI for the tool.

3.3 Timeplan and Method

The thesis was divided into seven milestones (MS) in order to be able to keep track of the work and make sure the timeplan was followed. The seven milestones are listed below.

MS1 Be able to receive and respond to a message from the Spectra traffic generator.

MS2 The first six chapters from the book ”Concurrent programming in Erlang” and the ISUP specification should have been read.

MS3 A first version of the ISUP protocol should have been implemented and integrated with existing tool. It should be possible to establish one call.

MS4 Implementation should be improved so it is possible to handle 100 calls/s.

MS5 A GUI should be implemented.

MS6 The scientific study should be finished.

MS7 User manual covering the implemented functionality and master thesis should be finished. The result should be presented at Ericsson.

Theses milestones were further divided into smaller goals which were set up in order to accomplish the milestones' goals.

The first three milestones had to be done concurrently. In order to reach MS3, the ISUP specification and the book had to be read several times. MS3 was divided into smaller steps as can be seen in the list below.

MS3.1 Read documentation about the SS7 protocol stack and its API and configure the stack to suit the requirements.

MS3.2 Write a C-interface which interacts with the API of the stack.

MS3.3 Read documentation about the Spectra traffic generator and configure the Spectra traffic generator to match the configuration of the stack.

MS3.4 Be able to send ISUP messages from the Spectra to the stack and back by using the C-interface written.

MS3.5 Implement the Erlang program which should communicate with the C-interface.

MS3.6 Integrate the Erlang program with already existing tool.

MS3.7 Be able to set up one call.

The next milestone (MS4) involved some optimizations and also some changes to the existing implementation. It was also divided into some smaller steps.

MS4.1 Change and optimize the C-interface to be able to handle 100 calls/s.

MS4.2 Optimize the Erlang program.

MS4.3 Understand how to generate and create tests for 100 calls/s continously over several days. Create such a test.

MS4.4 Remove memory leaks and decrease CPU usage.

MS5 was straight-forward, implement a Graphical User Interface. This involved reading documentation about Erlang's graphical module in order to accomplish the implementation. MS6 and MS7 dealt with writing of the user manual and the scientific in-depth study and were not divided into any smaller steps.

Chapter 4

Erlang in Telecommunication Applications

Erlang was developed at Ericsson as an experiment to investigate if there was some better way to program telecommunication systems than the ones that already existed. This chapter will first give an overview of the programming language Erlang and then investigate in what ways Erlang is adapted for developing telecommunication applications.

4.1 Introduction to Erlang

Erlang is a declarative functional programming language with single assignment variables. In total is the language quite small and thus easy to learn and use.

4.1.1 History

Erlang was developed by Joe Armstrong and his colleagues at the Ericsson Computer Science Laboratory. Ericsson wanted to investigate if there were better ways to program telecommunication systems than the existing ones and thus they conducted several experiments which led to the birth of Erlang. Some of the criteria were that they wanted to program telecommunication systems with less effort and fewer errors than with using currently available technology [2].

The development group had access to a lab with a lot of hardware. Among other things they had access to a small telephony exchange connected to a Unix machine which acted as a controller. The group programmed the exchange to handle ordinary telephony services, such as connecting two subscribers calling each other, in every possible programming language. The results from the experiments showed that declarative languages were a lot shorter and easier to understand than imperative languages [2]. A declarative programming language differs from an imperative in the way that a declarative language defines "what" rather than "how". As an example Prolog is declarative in where the programmer states relationships and then asks questions about those, without defining how to compute the answers [26]. Unfortunately, declarative languages most

often did not have sufficient features for concurrency control and error handling facilities necessary [2].

The conclusions the group drew from the experiments were that Prolog with added facilities for concurrency and improved error handling would be a proper language to work with. The development group continued by programming a lot of different telephony features in Prolog. In the end a Prolog interpretator was created which added the notion of a process and facilities for error detection and recovery to Prolog [2].

Between 1986 and 1988 the members of the science lab worked very closely together with a group at Ericsson, that wanted to use the results from the experiments for software development, to evolve the language. During this time the Prolog interpretator changed rapidly and somewhere along the way it was named Erlang. By 1988 Erlang was a good language for prototyping telephone exchanges. Unfortunately it was too slow to be used for product development. This lead to the development of different abstract machines and compilation techniques to optimize Erlang. After some time Joe's Abstract Machine (JAM) was invented. When the JAM machine eventually was delivered it was decided that Erlang needed to be even faster than it was with the JAM machine in order to be suitable for product development [2].

During the period 1989 to 1994 some new people joined the Erlang group at the science lab. Then distribution primitives were added to the language. Between 1992 and 1996 Erlang was very successful in smaller projects, but it was still too slow. Therefore in 1992 work started to build the BEAM (Bogdans's Erlang Abstract Machine) compiler [2].

Later on Erlang grew as a language and number of users internally at Ericsson also grew. Erlang was used successfully in some larger projects where one was the AXD 301. By now the Ericsson system came with an extensive toolkit consisting of a wide range of tools useful for building telecommunication applications, including cross-compilers for interfacing Erlang to foreign language applications as ASN.1 interface compiler, SNMP toolkit, HTTP-server etc. Many real-time applications also need access to consistent data for a long period of time. To accomplish this Mnesia was designed, which is a real-time distributed database written entirely in Erlang [2].

After some time the requirements on the operating systems, the hardware platform and also the language changed. Users expressed desire of having the operating system, the hardware platform and the language delivered in one package. To comply to the users' wishes the Open Telecom Platform (OTP) was developed. Extensive libraries were also created to solve common application problems [2].

4.1.2 Overview of the Erlang Language

Erlang looks like a single assignment functional language. In such languages variables can only be assigned a value once, but there is a difference between Erlang and a strict functional language. Erlang was designed to control soft real-time applications where strict sequencing of events is very important, which is a trait from imperative programming. In a pure functional language calling the same function twice with the same arguments will produce the same result while in Erlang where a function call may result in a hardware action it can not be guaranteed that a hardware action will produce the same result twice [6]. To summarize, Erlang is a declarative language where a subset of it is a dynamically typed and strict functional language [3].

Erlang can be described by the following four statements:

1. everything is a process and these are strongly associated,
2. process creation and destruction are light-weight operations,
3. message passing is the only way to communicate between processes, and, finally,
4. processes execute their code and as they are supposed to or fail.

4.1.3 Sequential Programming

The sequential programming in Erlang deals with the strict functional programming part of Erlang. This part has no side-effects and is dynamically typed.

4.1.3.1 Data Types

Erlang has only a few data types. These belong either to the group of constant data types or the group of compound data types. The constant data types can not be split into smaller subtypes. Atoms belong to this class. Compound data types on the other hand group together other data types. Among these, tuples and lists can be found. Furthermore variables also exists, but not in the sense as in more conventional programming languages such as C [3].

Erlang also introduces the notion of terms. A ground term or term in Erlang is defined to be either a primitive data type, like an atom, or a tuple or list of terms [5].

Constant Data Types
As stated before these can not be split into smaller subtypes. Among others, numbers and atoms belong to this class [3]. Numbers are ordinary integers and floats, while atoms are constants with names. These must begin with a lower case letter and end with a non-alphanumeric character except in those cases when an atom is quoted. A quoted atom is contained by the characters `'` and `'`, then any character may be included in it.

Compound Data Types
There are two different data types which can group other data types. These are tuples and lists.

Tuples are enclosed by curly brackets and their elements are separated by commas. They are used for storing a fixed number of elements and are similar to structures and records in other programming languages. The example below shows a tuple of size four. Each element in the tuple can be of type constant or compound [5], as can be seen in the example below.

```
{a, b, {1, 2}, 'malin'}
```

Terms enclosed by square brackets are called lists. These are used for storing a variable number of terms and is indexed in the same way as tuples. Index starts at one and not zero as in conventional arrays. The example below shows a list of size four.

```
[1, 2, a, 'malin']
```

The first element in the lists is referred to as the head and the rest as the tail. The notation `[Head | Tail]` can be used to retrieve the Head and the Tail from the list:

```
[Head | Tail] = [1, 2, a, 'malin']
Head = 1
Tail = [2, a, 'malin']
```

4.1.3.2 Pattern Matching

The most essential part of sequential Erlang is pattern matching. It provides the basic mechanism for assigning values to variables. As mentioned before, when a variable has been bound to a value it can not be changed. This is called "bind once" or "single assignment" in contrast to imperative languages which use destructive assignment. Pattern matching is performed in assignment statements, when calling a function and when evaluating other primitives such as `case` and `receive` [5].

A pattern is defined as a term except from that it also can contain variables. A pattern where all primitives are different is said to be a primitive pattern.

When performing pattern matching a term is compared with a pattern, and the pattern is said to match if the term and pattern have constants occurring at the same positions and are of the same shape. Any variables in the pattern will be bound to the corresponding items in the term.

Assignment Statements
In the statement `Pattern = Expression`, `Expression` will be evaluated and its result matched against `Pattern` [5].

```
{A, malin, B, List} = {12, malin, [a, b], 77}
```

Above, will the pattern match against the right hand side because both sides have the same structure, that is tuples, and the same number of term along with the constant `'malin'` at the same position. The variables `A`, `B`, `List` will be bound to `12`, `[a, b]` and `77` respectively.

Function Calls
When a function call is evaluated the arguments of the call are matched against the parameters given in the function definition, which is called a function head (clause head). If they match the code in the body of the function is evaluated. If no clause head matches an error is generated and the function is said to fail. Any variables in the clause heads of the function will be bound when one of the clause heads is matched [5].

```
getAge (Var1, Var2, 5) ->
     {Var1+Var2, 5}.
```

When calling `getAge()`, declared above, with the function call `getAge(1, 2, 5)`, `Var1` and `Var2` will be bound to 1 and 2 because the call matches the pattern in the clause head. On the other hand, if making the function call `getAge(1, 2, 3)` an exception will be generated because the call did not match any clause head.

4.1.3.3 Module System

Erlang has a module system which allows large systems to be divided into a set of smaller modules. Each module has its own name space and thus the same function names can be used in different modules [5].

```
-module(agemodule).
-export([getAge/1]).
-import(namemodule, [getName/1]).

getAge(malin) ->
     23;
getAge(_) ->
     0.

getAll(Age) ->
     getName(Age)
```

If a function is to be available outside the module it must be defined in an export-clause and can then be called with `Modulename:Functionname()`. In the example above, `getAge()` is exported and can be called with `agemodule:getAge()`. Functions can also be imported by using the import-clause and can thereafter be called using only the function name.

4.1.3.4 Functions

Each function is built upon a number of so called clauses. These are separated by semicolons and each one is composed of a clause head, an optional guard and a body.

```
getAge(malin) ->
     23;
getAge(_) ->
     0.
```

A clause head consists of a function name followed by a number of arguments where each is a pattern. When a function call is performed, the call will be sequentially matched against the set of clause heads which makes up the function [5].

A clause guard is a condition which has to be fulfilled before a clause is chosen. A guard can either be a simple arithmetic test or a sequence of tests separated by commas as can be seen below. But it can never be a user-defined function [5].

```
getName(Age) when Age == 10 ->
     jonas.
```

The body of a clause contains a sequence of one or more expressions which are separated by commas. These are evaluated sequentially and the value of the last expression is returned from the function. A function can only return one term.

4.1.3.5 Conditional Primitives

Erlang has two primitives which can be used for conditional evaluation. Those are the `if` and the `case`.

The `case` primitive consists of several patterns and optional guard tests. An expression will be sequentially matched against all patterns until a match occurs and the guard test evaluates to true. When a match is found the body belonging to the pattern will be

evaluated. If a match is not found a run-time error will be generated. The `if` primitive consists of several guards, where each of the guards will be evaluated sequentially until one of them succeeds and the following body will be evaluated. Just as with the case primitive, this primitive will either find a matching pattern or generate a run-time error.

4.1.3.6 Other Features

Erlang also includes higher order functions and list comprehensions.

Higher order functions are functions that take other functions as input arguments. For example `map()` is a higher order function which takes a function and a list as input arguments. It then applies the function on every element in the list [10].

4.1.3.7 Other Data Types

Furthermore Erlang also supports binaries, bit syntax, records and macros.

Binaries are memory buffers that are used to store untyped data. By using bit syntax it is possible to construct binaries and to pattern match on the contents of binaries. To create a binary `<<...>>` is used where values in the binary are separated by commas. Every value is represented by eight bits and thus 255 is the highest value that can be represented. The number of bits can be changed by using the syntax `Value:Numberofbits` [5].

Records are equivalent to structs in conventional programming languages. A record will associate specific elements in a tuple with names.

4.1.4 Concurrent Programming

Concurrent programming is accomplished in Erlang by using processes. The processes are light-weighted so it is not costly to spawn a new process or destroy an existing process. Because Erlang does not have any shared memory the only way for processes to communicate are through message passing.

4.1.4.1 Process Creation

A process in Erlang is a self-contained, separate unit of computation and several processes can exist concurrently. There is no inherent hierarchy among the processes created, however the programmer may create such a hierarchy explicitly [5].

The function `spawn()` is used to create a new process. It will return a Process Identifier, or PID, to the process making the call. The PID will be known to that process only. Sometimes other processes also needs to communicate with not known processes and thereby Erlang provides a global registry into where PIDs can be registered and associated with a name.

4.1.4.2 Inter Process Communication

As stated earlier, the only way for processes to communicate with each other is by message passing. To send a message `!` is used as shown below.

```
Pid ! Message

receive
     Message1 [when Guard1] ->
          Seq1;
     .
     .
     .
     MessageN [when GuardN] ->
          SeqN
end
```

Messages are received by using the receive clause as shown above. Every process has its own mailbox and all messages sent to it will be stored in the mailbox in the order of arrival. The receive clause will match each message in the mailbox against all patterns (here `Message1` to `MessageN`) in the receive clause sequentially. Upon a match and the guard evaluates to true the body following will be executed and the message is removed from the mailbox [5].

Moreover the receive primitive is blocking until it finds a message that will match. Therefore, to prevent infinite blocking to occur the `after` primitive is provided [5].

```
receive
     Message1 [when Guard1] ->
          Seq1;
     after Time ->
          Actions
end
```

If no message has arrived and matched within `Time` milliseconds a timeout will occur and the sequence Actions will be evaluated.

4.1.5 Error Handling

Because telecommunication applications need to have only minimal downtime it is important that they have the means to easy handle run-time errors. Run-time errors occur in Erlang for example if a match fails or a Built In Function (BIF)[1] is called with the wrong type or number of arguments [5].

All Erlang programs are compiled to virtual machine instructions and then executed by a virtual emulator. If the emulator detects an abnormal condition it will generate an exception. There are six different types of exceptions, which are listed in Table 4.1.5 [5]. Erlang has several methods of detecting and monitoring such errors.

4.1.5.1 Catch and Throw

The catch primitive is used to catch exceptions in order to prevent a process from dying. If an expression is guarded by catch it will upon failure return the tuple `{'EXIT',`

[1]BIFs are built in functions which perform operations that are impossible or inefficient to program in Erlang itself

Table 4.1: Exceptions in Erlang

Type of Error	Reason
Value Errors	"Divide-by-0". Argument to function has correct type but wrong value
Type Errors	Generated when BIF is called with argument of wrong type
Pattern Matching Errors	When no match is found for example in function heads
Explicit Exits	Exit(Why) is called
Error Propagation	If a process receives an exit signal it dies and propagates the signal to all processes it is linked to
System Exceptions	Run-time system terminates a process if it runs out of memory or there is some inconsistency in some internal table

`Reason`} where Reason is the reason for failure. If no failure occurs it will return the result of the evaluation of the expression.

By using throw in conjunction with catch one may return user defined errors back to the catch clause, given as input argument to `throw`. It is not possible to evaluate a throw clause by itself, this would generate an error [5].

4.1.5.2 Process Links and Monitors

Process links and monitors are used to inform other processes about processes terminating.

Process links are used to group sets of processes. Two processes can be linked together by using the BIF `link(Pid)`. When a process linked to some other process terminates it will terminate with an exit reason. This is emitted as an exit signal to all processes linked to the terminated one. The standard behavior of processes when receiving an exit signal is to terminate. However, if a process is set to trap exit signals, by setting the flag `trap_exit` to true first, it will not terminate but instead receive a message in the form {`'EXIT', Pid, Why`}. Thereby it may continue to execute even though other processes in the group have died [10]. Process monitors on the other hand are used to monitor or watch pairs of processes. A monitor is created by a process by calling the BIF `erlang:monitor(process, Pid)`. If process `Pid` terminates the process monitoring it will receive a message informing that the process has terminated and the reason for termination. This is useful in a client-server model where all clients should die if the server dies, but not the opposite way [10].

4.1.6 Distributed Programming

There are many reasons for writing distributed applications. An application that has been split into smaller parts can be run in parallel on different nodes or machines. Together they will solve the problem or supply services faster. A system can also be made more reliable and fault-tolerant by letting many nodes cooperate so that failures would not affect the operational behavior. Moreover, the performance of a system may be improved by adding more nodes [3].

Erlang provides the notion of a node to be a complete, self-contained Erlang system. Several nodes can be executed in a host operating system. A new node is created by the primitive `spawn` and monitors can be used to watch processes as explained earlier. Each node is assigned a name and can connect to each other to build a distributed Erlang system. Distribution is a special case of concurrency where it is possible to spawn new processes at other nodes instead of locally, it is thus also possible to communicate with those processes with only a slightly different primitive, in comparison to the ordinary communication primitive [5]. This transparency makes distributed systems easy to implement and is a big advantage of Erlang.

4.1.7 Ports

Erlang uses so called ports to communicate with programs written in other programming languages. The process creating the port is the one that owns it and this is the only process which can use it.

One may send data to the port just in the same way as messages are sent between processes using the `!` only in a slightly different manner. Furthermore, data is received in the same way from the port as from processes.

4.1.8 Dynamic Code Change

In other programming languages the system must be stopped and then re-started in order to change the executing code. This is not convenient when dealing with soft real-time control systems which need to have the smallest downtime possible.

In Erlang two versions of code is supported for every module in the system and all processes share the same code. If a function is called with a fully qualified name, i.e. `Module:Function`, the latest version of the function will be called, otherwise the currently loaded (executing) version will be used. Thereby code can be changed and put into the system without stopping it [5].

4.1.9 Generic Behaviors

To help with implementation of applications certain abstractions of common programming patterns are provided by Erlang. These are called behaviors. There exist four different behaviors [10]:

gen_server Pattern for the client–server model.

gen_event Pattern for event handlers.

gen_fsm Pattern for a finite state machine.

supervisor Pattern for a supervisor in a supervision tree.

These hide some details that are needed in the implementation of the behavior and also provides the programmer with a generic pattern which must be used. The most used behavior is the client-server model [10]. Basically, the pattern defines certain callback functions that must be implemented by the application using the behavior, much in the same way as interfaces are used in Java.

4.1.10 Supervision Hierarchies

A basic concept in Erlang is the notion of supervision trees, which is a process structuring model based on workers (children) and supervisors. Workers are the processes which makes the work, i.e. the computations, while the supervisors monitors the behavior of the workers. If a worker crashes, the supervisor has the authority to restart the worker [10].

In order to create a supervisor process one need to implement the supervisor behavior, i.e. its defined callback modules have to be implemented. Many of the child processes or workers are also implemented using behaviors. The behaviors used for this are the `gen_event`, `gen_fsm` and `gen_server` which were described in Section 4.1.9.

In the supervisor it is specified what strategy to use in the restart of processes and at what rate processes should be restarted. There are three types of models for the restart [10]:

one-for-one If one process is terminated it is restarted

one-for-all If one process dies all the other children are terminated and together all processes are restarted

rest-for-one If a child process terminates the rest of the children processes (those after the terminated one in start order) are terminated and then all terminated processes are restarted

In order to prevent a child process to be restarted all the time and still dying from the same fault, a limit of how many restarts that can be performed within a certain time is specified. If a child process has exceeded this limit the supervisor terminates all other children (if not already done) and then terminates itself [10]. The supervisor at a higher level will then notice the termination and either restart the supervisor or terminate itself.

4.1.11 Open Telecom Platform (OTP)

OTP stands for Open Telecom Platform and is a middleware designed for building and running telecommunication systems. It consists of an Erlang run-time system, a number of ready-to-use components and a set of design principles for Erlang programs [10].

An example of a component in OTP is the distributed database management system Mnesia which is specially appropriate for telecommunication systems and other soft real-time applications. Some other components in OTP are an SNMP agent, support for C and Java interfaces and also an HTTP server and client.

4.2 Erlang and Telecommunication Applications

The intention behind Erlang was to find a better way of programming telecommunication systems. Erlang was therefor built upon a set of requirements in order to be adapted for programming of telecommunication systems. But how suitable is Erlang really for use in such systems?

4.2.1 Requirements on Telecommunication Applications

Telecommunication software are usually very large systems and the demands from today's customer or consumer are a low-cost, easy to use system which is always available and reliable. Furthermore, the industry also thrives for as high availability as possible [20]. It is not uncommon to allow only minutes of downtime per year, and that includes scheduled maintenance such as applying patches and function growth.

In order to implement a high availability system demanded by customers or consumers leads to some specific technical challenges. Most downtime is caused by system and application software failure, therefore it is necessary to make the system or application fault-tolerant so it can recover from run-time failure [20]. Moreover, to ensure high availability an application must be maintainable, including debugging of existing systems and adding of new features, and able to adopt dynamically to both hardware and software upgrades. Additional requirements on application software for telecommunication systems are that the application needs to handle interactions between several components, i.e. hardware, software, networks and operating systems. Furthermore, systems need to respond and react without delay to new requests and they also need to be scalable, since they need to adapt to handle increased demand. If a system is overloaded its performance should downgrade gracefully [20].

These thoughts are also supported in [9] by Däcker (CS-lab), where requirements on a programming technology for telecommunications switching systems are stated. His requirements are listed below.

1. Handling a very large number of concurrent activities
2. Actions to be performed at a certain point in time or within a certain time
3. Systems distributed over several computers
4. Interaction with hardware
5. Very large software systems
6. Complex functionality such as feature interaction
7. Continuous operation for many years
8. Software maintenance without stopping the system
9. Stringent quality and reliability requirements
10. Fault tolerance both to hardware failures and software errors

The requirements can be motivated by looking at several characteristics of a telecommunication switching system. Such systems need to handle many concurrent activities efficiently because a typical switch would probably have many tens of thousand of people simultaneously interacting with it [3].

Switching systems are also soft real-time systems because some operations are strictly enforced and needs to be succeeded within a time limit or else they will be aborted, and some operations are able to be repeated if not succeeded within the time limit. It is therefore important to be able to manage several timers in an efficient manner [3].

Distribution is common within telecommunication systems and thus it is important that it is easy to switch from a single-node system to a multi-node distributed system. In fact most of the telecommunication systems are distributed naturally in order to achieve robustness and high availability [3].

A switch contains a lot of hardware which must be controlled and monitored. The programming language should make it possible to create efficient device drivers to accomplish this, which implies that good support for hardware interaction is necessary [3].

Requirement number six is not unique to the telecommunication area, but is probably a lot more common as these systems are very large and a lot of software is making very complex environments where numerous add-on services, or features, are working on the same resource. To explain complex features imagine a scenario where user A calls user B, but B's phone is busy. If B has the feature "Call Forwarding on Busy" activated, the call should be forwarded, but how should the software handling the transmission of the busy tone act? Should it reply with a busy tone or not? This is a simple example but the complexity grows rapidly as the telecommunication systems have support for hundreds of such features and most of them are working on the same resources but with different goals. And this at the same time [15]. Along with requirement seven, it is clear that run-time upgrades are crucial to keep a healthy system with as little faults as possible, but at the same time have new functions added as customers and users demand them [3]. These requirements are also connected to requirement number five and eight which states that the systems are very large and need to be maintained without stopping them.

The most important requirements are the two last ones that deals with fault-tolerance and quality in the presence of errors. Because a switching system must have as little down-time as possible, preferably none, it is important that the programming language supports extensive error handling to deal with both software and hardware faults, while the system still provide its users with an acceptable level of service [3].

Armstrong, one of the creators of Erlang, argues in [3] that there are six requirements on the underlying operating system and programming language of the system:

R1 System must support concurrency

R2 Error encapsulation

R3 Fault detection both locally and remotely

R4 Fault identification, i.e why a fault occurred

R5 Code upgrade while system still is running

R6 Stable storage which survives a system crash

4.2.2 Overview of Erlang Design

When developing and examining Erlang both technical and organizatorial requirements from the industry had to be met [4]. At first Erlang was an experiment to investigate if declarative languages were suitable for programming large telecommunication switching systems. In the end Erlang was found to be suitable both for this and a wide range of industrial embedded real-time control problems.

When dealing with switching software it is important that they have response times in the order of milliseconds. Erlang is designed for programming so called soft real-time systems, i.e. systems where not all timing deadlines have to be met. Furthermore, control systems can not be stopped for software maintenance. To solve this Erlang provides mechanisms for upgrading software while the system still is running.

Erlang has an abstract machine which allows change of program code in a running system. The abstract machine also allows Erlang to compile to code which can be executed on any of a large number of different operating systems, making Erlang very portable.

Applications of these kinds are best modeled by concurrent processes which can handle many tasks simultaneously. Processes are the basis of Erlang, which is designed for running in a distributed multi-node environment and thus provides lightweight processes, which are not expensive to create or destroy. Moreover the Erlang processes are not dependent on the operating system so no concurrency dependencies are derived from the operating system [4].

In real-time systems long garbage collection delays are unacceptable therefore applications implemented in Erlang uses bounded-time garbage techniques which are handled automatically [4].

4.2.3 Erlang used in Telecommunication Applications

Since the language and the middleware was built with telecommunication applications in mind, a number of unique features exist in the language which are not seen in other popular languages. This section will take a closer look at these features and explain why and in what way they are suitable in a telecommunication application environment. Some have already been discussed, but not in detail. This is followed by a discussion regarding how suitable Erlang is for telecommunication applications.

An overview of how Erlang matches the requirements on telecommunication switching systems can be seen in Table 4.2.3.

4.2.3.1 Concurrency

Concurrency is very important in telecommunication systems. The systems are very large and should manage to handle many concurrent activities. An example of this is to look at a telecommunication switching system in where many connections from different subscribers need to be handled at the same time and within a reasonable time limit. In those systems it is important to have a response time in the order of milliseconds [3].

Erlang provides concurrency by using processes as described in Section 4.1.4. The processes are light-weighted and thus cheap to create and destroy, which will improve

performance. Moreover the processes do not have any dependencies on the underlying operating system. Even if handling 1000 active Erlang processes the operating system will still experience it as only one process [3]. Thereby Erlang can handle many concurrent activities by using processes for every activity without being dependent on operating system limitations regarding the number of processes or their size.

In a comparison with Java it is shown that Erlang processes are faster to create and destroy than Java threads [11]. The test was conducted with three different Pentium computers used and three different operating systems. Erlang was able to spawn up to 20 thousand processes while Java could only spawn up to 1600 threads [11].

Furthermore, the message passing between the Erlang processes were also faster than between the Java threads [11]. This implies that Erlang processes are quite efficient and thus using Erlang in telecommunication systems would mean that the concurrent tasks would be handled in a sufficiently efficient manner.

4.2.3.2 Interoperability

There are two interoperability mechanisms in Erlang. One is distributed Erlang and the other one is ports [10].

A distributed Erlang node is created by giving it a name, it can then connect to or monitor other nodes. As described in Section 4.2.3.3 message passing and error handling between processes at different nodes are transparent. The distribution method is implemented using TCP/IP sockets, and it is primarily used for communication between two Erlang programs but can also be used for communication between Erlang and an external program written in e.g. C [10].

Ports are another method for communication with the outside world. They provide a byte-oriented interface to an external program written in some other programming language. The actual implementation of the port mechanism depends on the platform, but in Unix pipes are used [10].

To help programmers there are already implemented interfaces for C and Java as libraries. These provide some mechanisms for encoding, decoding and also for creating Erlang terms in the corresponding language. Thus, it is easy to create drivers (e.g. in C) for controlling hardware and integrate those into the Erlang system.

4.2.3.3 Distribution

Many telecommunication switching systems are distributed over several machines and thus support for distributed programming is necessary.

Because processes in Erlang are so fundamental, distribution comes naturally. As mentioned earlier, Erlang has something called nodes, which actually are complete, self-contained Erlang systems on the same or on other separate computers. Every node has its own name and can connect to other nodes. Because process identifiers are used, the message passing between processes at different nodes and also links and monitors are transparent. The nodes have also the ability to monitor each other for errors and restart nodes that crashes.

4.2.3.4 Fault-tolerance and Error Handling

Telecommunication systems need to be reliable and available for use all the time. To fulfill these requirements it is of importance to the systems that they are fault-tolerant and can handle run-time errors.

In [3] Armstrong suggests a strategy for creating a fault-tolerant software system. The strategy is divided into three parts:

1. The software is organized into a hierarchy of tasks the system has to perform. Tasks are ordered by complexity, where the top level task is the most complex.
2. Try to perform the top level task.
3. If an error is detected an attempt is made to correct it. If that failed the task is immediately aborted and a simpler task is started instead.

In order to use this strategy it is important that there exists a strong encapsulation method for error isolation. The isolation would prevent errors from propagating and affecting other parts of the system [3]. Moreover, it is not possible to correct an error if it is never detected.

Erlang provides error isolation by using processes. As soon as two processes share any common resource the possibility exists that an error in one of the processes will corrupt the shared resource. That is a reason to why Erlang processes do not share any memory, but relies on message passing instead [3].

Error detection in other processes are handled by monitors and process links. Monitoring will detect generated errors in the processes while process links group processes together. If an error would occur in a linked process the other processes would also get the generated error or exception (see Section 4.1.5).

Included in OTP is the supervisor-module. This can be used to organize processes into a tree-structure with supervisors and workers as described in 4.1.10. The supervisors can restart the worker processes if they have crashed.

To handle errors locally in a process the primitives catch and throw can be used to catch exceptions generated by function calls and BIFs.

Erlang together with OTP is well suited for building fault-tolerant systems both due to its natural error encapsulation and to its features for error detection and error handling. Thus Erlang will fulfill several requirements stated by both Armstrong [3] and by Däcker [9].

4.2.3.5 Dynamic Code Change

Customers and users want telecommunication systems that are reliable and available at all time. It is therefore not optimal to shut the system down every time maintenance are performed or a code upgrade must be done. A telecommunication system is expected to be running for years with as little downtime as possible or even no downtime at all.

Erlang provides means of changing the code dynamically as described in Section 4.1.8. Basically Erlang supports two versions of every module in the system, one currently running and one new version. If a function is called with its full name, i.e. `Module:Functionname`

the newest version will be used. On the other hand, if called only with the function name the currently loaded (executing) version will be used. In that way new code could be added to a module and then be used in the running system without shutting it down.

In comparison to other conventional programming languages which has to be recompiled and restarted Erlang has a big advantage with this feature. By using it to upgrade and maintain systems the total downtime will be largely minimized.

4.2.3.6 Stable Storage

For telecommunication systems it is very important to have some kind of static and stable storage. This is particularly important after a crash of a system in order for the system to be able to retrieve essential data when being restarted, data that otherwise would have been lost. It is also necessary for storing data that needs to be stored for several years. Erlang provides two solutions for this in its many libraries that are included in OTP. Both the Mnesia database and the dets table can be used for storing these types of data [3].

Mnesia is a distributed database management system (DBMS) appropriate for telecommunication applications and other Erlang applications, which has need of continuous operation and soft real-time operations. Mnesia is implemented in, and very tightly connected to Erlang. In order to meet the requirements on data management in telecommunication systems it has a broad range of features not normally found in traditionally databases [10].

Mnesia was designed with the following requirements in mind:

1. Fast real-time key/value lookup
2. Complicated non real-time queries mainly for operators and maintenance
3. Distributed data due to distributed applications
4. High fault tolerance
5. Dynamic re-configuration
6. Complex objects

Fast real-time queries and high fault tolerance together with abilities of dynamical re-configuration makes Mnesia very suited for development of telecommunication applications. Mnesia supports both the soft real-time requirement that telecommunication applications need to meet and also dynamic re-configuration which would minimize the downtime, because it is implemented in Erlang [10].

The unique features of Mnesia includes:

- Database schema can be dynamically re-configured at run-time
- Tables can have properties such as location, replication and persistence
- Tables can be moved or replicated to several nodes to improve fault tolerance without disturbing access to them

– Table locations are transparent to the programmer, they are accessed through their table names

– Database transactions can be distributed and large number of functions can be called within one transaction

– Several transactions can be performed concurrently, the DBMS synchronizes so that no two processes will manipulate the same data simultaneously

– Transactions can be assigned to be executed on several nodes or on none

Dets is a disk based version of the module ets[2] and saves terms in files on disk. It is possible to insert, delete and search for specific terms in a file. This implementation is used as the underlying file storage mechanism of the Mnesia DBMS [10].

It is important to realize that a single lookup operation in a dets table might involve a series of disk seek and read operations. Thus the operations in this module are much slower than those in the ets module. It is also worth noting that if several Erlang processes opens the same file they will share it and the module is not yet concurrency safe [10].

4.2.3.7 Other Features

There are some other advantages with Erlang as a programming language. These are its module system and its ability to handle soft real-time requirements.

The module system Erlang uses is described in Section 4.1.3.3 and it may be used to divide larger programs into smaller units.

In order to handle soft real-time requirements Erlang has features for handling timeouts, which are used in conjunction with the `receive` primitive. When executing a receive-clause, situations where a message never is received may occur and these can be handled by a timeout. This is done by using the `after` primitive which generates an timeout after a specified amount of time and exits the receive-clause.

4.2.3.8 Efficiency

Many things are adapted in Erlang to suit development of telecommunication applications, but there is also disadvantages. One problem is that Erlang is not very efficient. Other languages like C are much faster than Erlang. Performance measurements have been made [8] where several different benchmarks were used. When comparing Erlang's performance with C's performance C was faster in all the benchmarks, as can be seen in Figure 4.1.

Erlang was adapted for telecommunication applications which often need the ability to handle many concurrent activities. Therefore Erlang was developed to suit concurrent programming. In order to make a fair comparison Erlang should then be compared in the area of concurrency. No concurrency benchmark existed in C so instead Erlang was compared to Java in this area. In the comparison Erlang was much better both regarding memory use and speed [8]. The comparison of the two languages can be seen in Figure 4.2, where the concurrency result is the third bar from the top.

[2] ets tables can store very large quantities of data in the Erlang run-time system and provides constant access time to the data

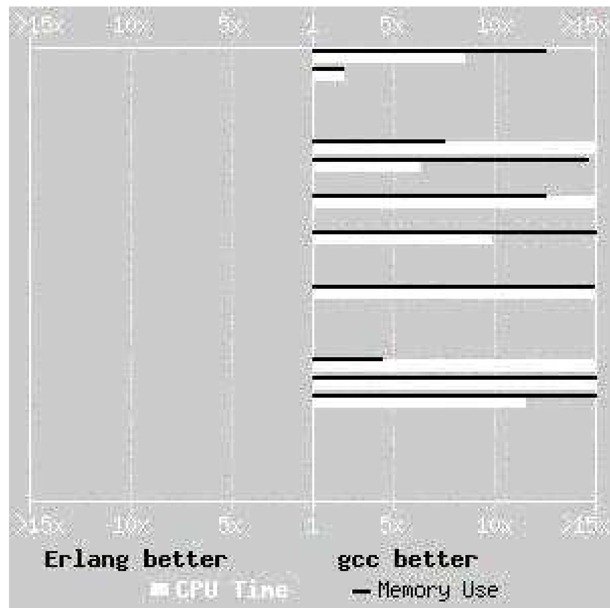

Figure 4.1: C compared to Erlang

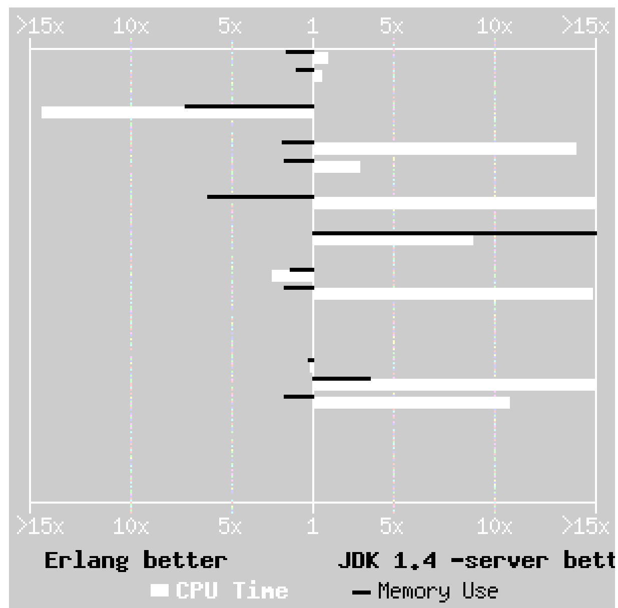

Figure 4.2: Java compared to Erlang

In [7] it is argued that large systems as telecommunication systems will not be entirely implemented in only one programming language but in many languages. For example tasks that demand efficiency is perhaps better to implement in C. An example is the AXD 301 which was implemented using both Erlang and C. The AXD 301 control system, release 3.2, consists of roughly one million lines of Erlang source code, 400000 lines of C/C++ code and also 13000 line of Java code and each language is used in the area where it fits best. Practically all the complex work is done in Erlang [25]. In [9] the author reasons in the same manner, that the future telecommunication systems cannot be programmed with one language using one methodology.

4.2.4 Discussion

Much of the material written on this subject were written by people working at the CS-lab at Ericsson and thus have been participating in the development of Erlang. Perhaps these were not the most objective people when it comes to evaluating Erlang as a programming technology for development of telecommunication systems, due to their subjectiveness. On the other hand, they have a lot of experience with development of telecommunication systems and can surely use this knowledge to form the language so it best suits the requirements upon telecommunication applications.

According to other theses and reports read [3, 7, 9], Erlang provides a nice programming methodology used for developing telecommunication systems. The language is small, easy to learn and is a high-level declarative language. Faults dealing with memory allocation, management and deallocation can not occur, since no shared memory exists.

Erlang has a good error handling and fault detection methodology together with distributed programming which is very transparent. This is referring to the handling of processes which can be communicated with easily, regardless of being on a local or remote machine.

Furthermore, the dynamic code changing is also an advantage in dealing with telecommunication systems due to the requirement of a low down-time of the system.

Erlang/OTP also provides with a stable storage in the distributed database management system Mnesia. Because Mnesia is implemented in Erlang it also has the ability to be upgraded by a dynamic code change.

The only disadvantage mentioned is that Erlang is not a very efficient programming language. Thus when dealing with efficiency demanding tasks Erlang is not the best language to use. As can be seen in performance measurements C is much faster than Erlang as was described in 4.2.3.8. On the other hand concurrency is not compared between the two languages. Erlang is instead compared to Java regarding concurrency, and here it is shown that Erlang performs better both in speed and memory usage.

Because Erlang is not very efficient it is not wise to use it in implementations of time-critical parts in a telecommunication application. It would be better if Erlang is to be used in conjunction with other more efficient languages such as C, where Erlang would handle the concurrent activities while C handles the time-critical parts and hardware interactions. Furthermore, many requirements put on telecommunication applications also applies to other applications such as distributed systems and also Internet services and Erlang would probably be well suited for developing those types of applications too.

Table 4.2: Erlang matched against requirements on telecommunication switching systems

Requirement	Erlang feature
Handling of a very large number of concurrent activities.	Concurrency is provided through light wight processes. A typical program can handle tens of thousands concurrent processes in one node.
Actions need to be performed at a certain point in time or within a certain time.	Erlang handles soft real time and has primitives for dealing with timeouts.
Systems may be distributed over several machines.	A system in Erlang may contain nodes on many different machines running on different operating systems over a network.
Interaction with hardware.	Erlang can easily communicate with hardware drivers.
Very large software systems.	In Erlang a system can be divided into several modules.
Complex functionality such as feature interaction.	Depends on the application.
Continuous operation for many years.	Depends on the application.
Software maintenance without stopping the system.	Erlang permits dynamic code change.
Stringent quality and reliability requirements.	Depends on the application.
Fault tolerance both to hardware failures and software errors.	Erlang has mechanisms to catch and contain errors and to design supervision structures.

Chapter 5

Accomplishment

The purpose of the thesis was to create a simulation tool which simulates part of the AXE-10 behavior together with already developed tools and a traffic generator. The work was divided into several steps as stated in Section 3.3 and described into more detail in this chapter.

5.1 Development

As mentioned (see Section 3.3), the work was divided into seven milestones (MS1-MS7). Some of these were further divided into smaller sub-milestones.

5.1.1 MS1-MS3

The first three milestones (MS1-MS3) were performed concurrently. The first step was to read all the documentation about the SS7 protocol stack, install it and to configure it correctly. A lot of time was spent in this phase because the stack was quite complex. During the same phase the ISUP specification was read in order to understand all features of the stack.

While configuring and setting up the stack the C-interface was also implemented. In this first version of the C-interface it was only one process which handled all communication between the stack and the Erlang program. To be able to test the stack and the C-interface the traffic generator was configured and used. The first test conducted was to generate ISUP traffic to establish a call between the traffic generator and the SS7 protocol stack using the C-interface. In this case the traffic generator simulated the calling subscriber while the stack together with the C-interface simulated the called subscriber. All intelligence handling the ISUP signaling was implemented in the C-interface.

When the stack, the C-interface and the traffic generator was functioning the Erlang program was implemented. During this phase the intelligence regarding the ISUP handling was moved from the C-interface to the Erlang program. Moreover, the C-interface was changed into being only an interface between the Erlang program and the stack.

The work continued by analyzing the already existing tool to establish where xSIM was to be integrated. Furthermore, it was also necessary to study the protocols used for controlling the ML. The protocols most studied were the SDM protocol and the CM protocol (see Section 2.4).

After the implementation was done it was tested by using the traffic generator. The test run was to set up a telephone call, let it continue for some time and then tear it down.

5.1.2 MS4

When the application had been implemented, integrated and tested so that it managed to set up a call the next step was to optimize it. The goal of this step was to optimize and adapt the application so that it would manage to handle 100 calls/s.

First the C-interface needed some modifications. Instead of only one process handling all communication between the stack and the Erlang program two threads were added. The reason for this was to simultaneously handle the traffic coming from the stack and going to the Erlang program and the traffic coming from the Erlang program going down to the stack, and by that increase the performance.

During the optimization some memory leaks were discovered and also some optimization problems. The application had apparently some inefficient code that took too much CPU performance. This code was removed along with the memory leaks.

Due to the memory leaks the structure of the C-interface had to be changed. The initial purpose was to make the C-interface as general as possible in order for it to be reusable when using layers other than ISUP. Unfortunately, the complex design of it resulted in that the memory leaks were hard to remove without changing the implementation. It was then decided that the best would be to change the implementation of the C-interface to make it easier to maintain and to understand. The whole C-interface would still be reusable in terms of using the ISUP layer, and some parts of it could be reused in other implementations towards the stack.

5.1.3 MS5

During this phase some reading was done to understand how the graphical module in Erlang works. Some errors were also discovered in the current implementation of which all were corrected. During the development of the GUI, improvements to the whole system were also added.

In this phase a lot of testing and correction of errors were performed. In the end, when the implementation was satisfactory, a large test was performed in where the simulation tool would be running for several days terminating ISUP traffic at a rate of 100 calls/s. The test confirmed the application to be robust and stable. More details about the test are described in Section 6.4.1.

5.1.4 MS6-MS7

During milestones six and seven lots of time was spent in reading other reports and literature. There was some difficulties of finding written material which was not written

by current or old employees at Ericsson, but in the end some objective reports and articles were found.

Chapter 6

Results

The aim of this chapter is to describe the resulting application xSIM. The configuration of the stack and also the traffic generator's configuration will be described because they are part of the result.

6.1 The Application xSIM

xSIM was composed of two different parts; the Erlang program and the C-interface between the Erlang program and the SS7 protocol stack.

The already existing tool together with the SS7 protocol stack and the traffic generator belongs to the system as well (see Figure 6.1).

6.1.1 The Structure of the Erlang Program

The Erlang program is basically divided into four different parts. One is a generic server (xSIM in Figure 6.1) which controls the whole application, and one is a process which starts and handles the already existing tool (trhComm in Figure 6.1). The third parts are all the call processes (p1 to p4), which are created by the generic server, and each process represents a requested call or connection. The fourth part is a GUI which is intended to start the whole application and also manage some statistics.

Figure 6.1 shows what the system's structure looks like. Here the server, xsim, handles all communication to and from the C-program. The server creates the trhComm-process and also call processes when necessary (upon reception of an IAM). Therefore, if the call processes (p1 to p4 in Figure 6.1) wants to send messages to the C-program they must do this through the server.

The trhComm-process is the process that starts the necessary parts of the already existing tool (trhAXE and xAXE) and keeps it available. These are the parts that are used in order to communicate with the ML. Moreover, the process also initializes the ports to be used in the ML using the SDM protocol. The call processes communicate directly with trhAXE and xAXE in order to send requests to the ML and need thus, not go through the trhComm-process to accomplish this. The call processes uses the CM protocol to send requests to the ML to establish paths between MGs.

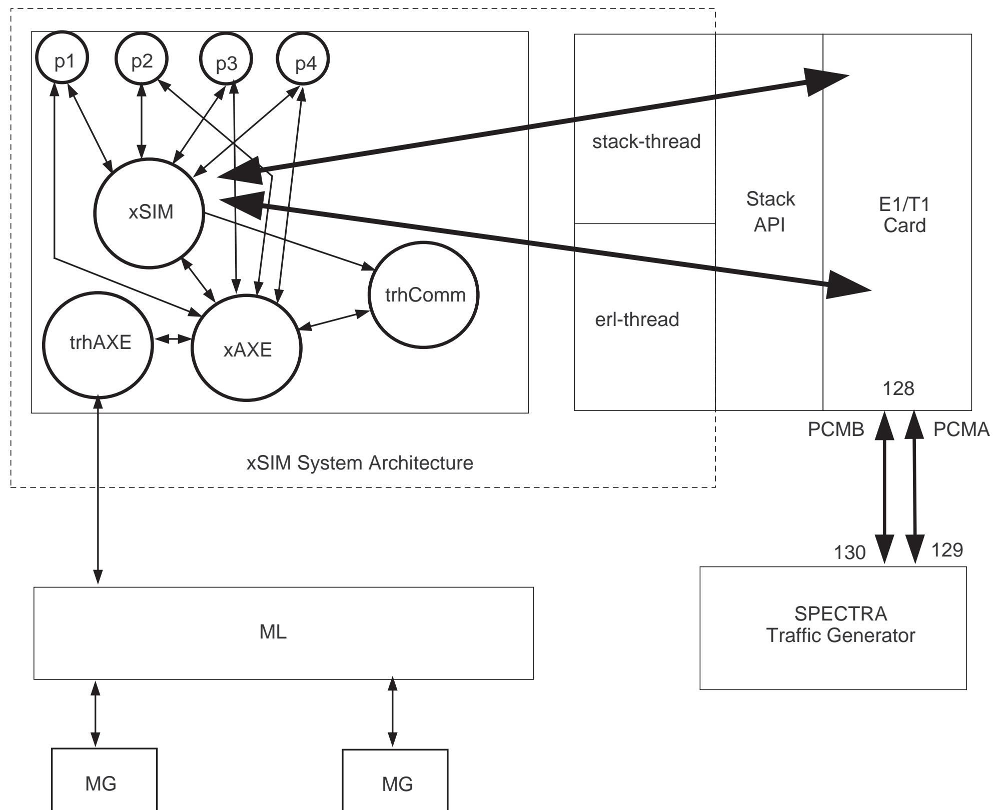

Figure 6.1: Overview of xSIM

The call processes handle all data analyzing of the ISUP messages and also control of all requests to the ML. The only critical decision the server makes, besides forwarding messages between the call processes and the C-interface, is to create new call processes when it receives an IAM from the C-interface.

6.1.2 The Structure of the C Program

The C-program is divided into two threads. One which handles the communication from the the stack and to the Erlang program, referred to as the stack-thread, and one handling communication coming from the Erlang program to the stack, called the erl-thread, as can be seen in Figure 6.1.

In order for the C-interface to be used more, a static structure is used which contains a section for each user-defined function written to handle commands from the Erlang program. Each section in the structure contains both a string with the command and also a function pointer to the function. Basically the erl-thread receives messages from the Erlang-program on file descriptor 3 and decodes the message. It then looks at the first element of the message which it compares to the commands in the static structure in order to find the correct function pointer. The correct function is called with the rest of the message received as a parameter.

The stack-thread on the other hand handles callbacks from the stack when messages are received from it. The thread builds the messages to be sent to the Erlang-program, according to the protocol described in 6.1.3, and sends them to the Erlang-program by using file descriptor 4. A C-library named `erl_interface` in OTP is used to build the Erlang messages and to encode and decode them.

Intentionally, the C-interface lacks any kind of intelligence regarding the analysis of the ISUP signaling received from the stack. By letting the C-interface only translate messages using the defined communication protocol (see 6.1.3) and performing commands issued by the Erlang program, the C-interface could be reused by any Erlang program wanting to terminate or analyze ISUP signaling.

6.1.3 The Communication Protocol

In order for Erlang to communicate with C and vice versa a protocol was created. The intention of the protocol was to make it easier for the Erlang program to interpret what data was being delivered to it and to have a standard of how to control the C-interface.

All callback-functions from the stack have several parameters provided to it. These are either special C-structures with varying number of elements of different types, a pointer to some memory location or just an ordinary value. In order for the Erlang program to easily see what value belongs to what parameter, every value is marked with a tag. Thus every value in the arguments, regardless if being an element in a structure or an integer, is assigned a specific tag and put into a tuple as can be seen below.

```
{cause, 0}
```

This argument is called cause and has value 0, thus it is given the tag `cause`. Pointers, on the other hand, are also put into a tuple with a tag, but because their values probably are several, these were put into a list. To separate pointers from ordinary values all tags being pointers have the suffix '`_p`'.

```
{list_p, [1, 2, 3]}
```

All structures are also tagged but with the suffix '_sp' and put into a tuple with all its elements inside a tuple.

```
struct person {
     ssn[10] = {8, 2, 0, 3, 2, 6, 1, 2, 3, 4};
     age = 23;
}

Will produce the following:
{person_sp, {{ssn_p, [8, 2, 0, 3, 2, 6, 1, 2, 3, 4]},
             {age, 23}}}
```

In this way all the arguments with their corresponding values are translated and then sent to the Erlang program. Because the Erlang program receives valid Erlang terms it is possible to directly pattern match on and use the data received.

6.1.4 The GUI

There are no configuration opportunities available in the GUI. The reason for this is because the Spectra can not be controlled remotely, and thus all configurations of the Spectra has to be done manually.

Because all decisions regarding the traffic load, i.e. calls/s, and also what kind of ISUP messages that are transmitted, are to be configured and decided in the Spectra, no configuration choices were added to the GUI or xSIM.

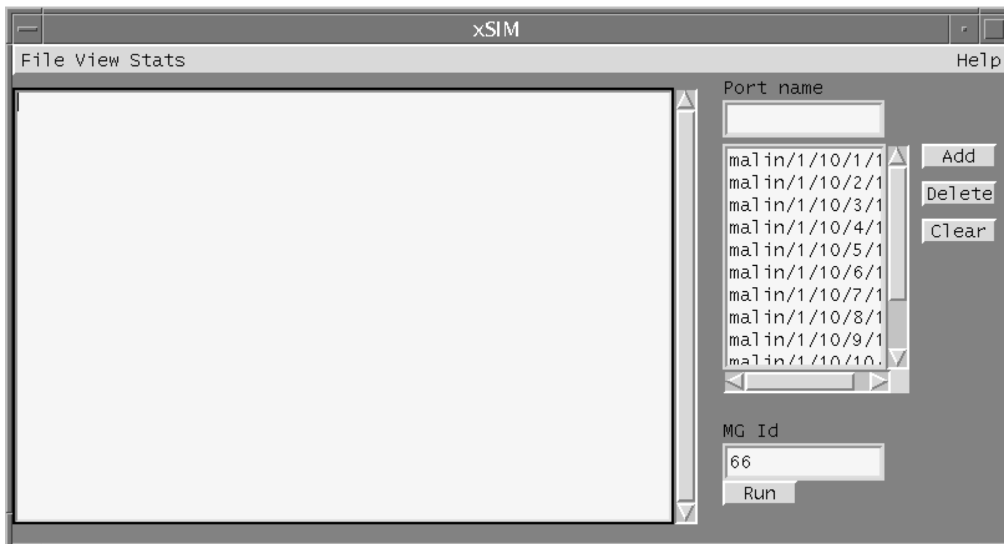

Figure 6.2: The GUI for the Application xSIM

As can be seen in Figure 6.2 the GUI has three different sub menus. These are the File menu, the View menu and the Stats menu. Under the menus there is a large window in

which information, such as statistics or other things, will de displayed. To the right of the display is a configuration area in which the Run-button is used to start the whole application xSIM with the given configuration.

When the GUI is started there are default values of the ports and the MG identity given in the configuration. These default values have been manually configured in the ML. If the values are to be changed it is important to make sure that the new values match the configuration in the ML. In the default configuration there are 12 ports defined and the MG identity is 66. New ports can be added to the list by using the Add-button or deleted by using the Delete-button. A Clear-button is also provided to clear the whole list. The MG identity is changed by simply writing a new number in the "MG Id" entry.

6.1.4.1 The File Menu

The File menu contains two choices; Exit and Save. Exit will stop the whole application while Save will write all information in the display to a given file. When clicking Save a pop-up window will appear in where a filename or a whole path to the logfile can be entered.

6.1.4.2 The View Menu

This menu has only one submenu called Filter. When clicking Filter a new window will appear in which the user may decide which types of messages that should be filtered by marking radiobuttons. These types of messages will then not appear in the information display. Default is that no message is being filtered.

6.1.4.3 The Stats Menu

Under this menu the user may choose between two different submenus, Failures and Messages. Both of these contains different choices for retrieving statistics. Under Messages the user may for example retrieve statistics of how many IAM-messages that has been received by the application xSIM.

6.2 The Configuration of the SS7 Protocol Stack

To be able to simulate two subscribers calling each other the application must act as a forwarding layer. The traffic generator would be simulating the two subscribers and the stack would be a signaling point itself.

The configuration of the stack is shown in Figure 6.3 and shows how the application xSIM (together with the stack) represents signaling point 128. The Spectra is simulating two signaling points, 129 and 130. ISUP signaling is using timeslot 16 on trunk 1 going to SP 129 while it uses timeslot 17 on trunk 2 going to SP 130.

As mentioned in Section 2.2 is the stack composed of two parts, the front end (FE) and the back end (BE). Between the FE and the BE there is an internal logical trunk called PRA and from the FE to the Spectra are two external trunks called PCMA and PCMB connected (see Figure 6.3).

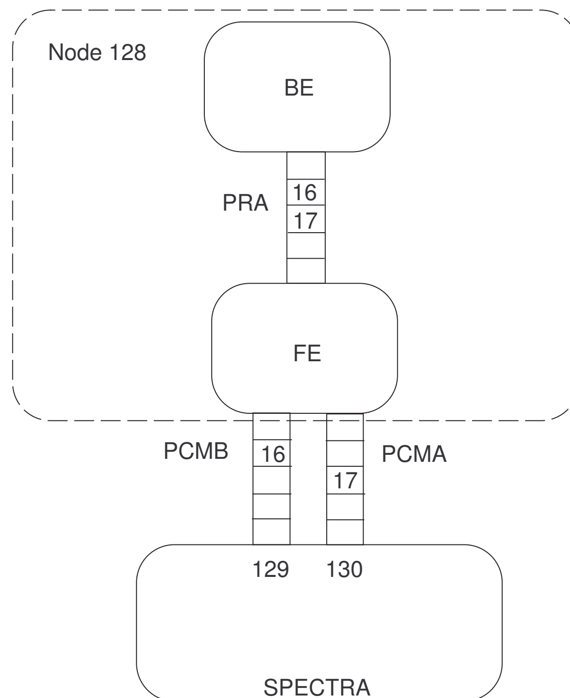

Figure 6.3: The Configuration of the stack and the Spectra

The configuration was mapped so that the logical signaling data link (SDL) in the BE going to node 129, was 0, and going to 130, was 1, with the logical hardware selection number (HSN) 0 and 100 respectively. These signaling data links were mapped onto timeslots 16 and 17 in the PRA-trunk.

The most important configuration in the FE is the one that maps the timeslots of the PRA trunk onto the trunks PCMA and PCMB. In order to separate the traffic coming to and going from the two nodes 129 and 130, their signaling data links were mapped onto different timeslots of the PRA-trunk. The first 16 timeslots of the internal trunk were then mapped onto the first 16 timeslots of the PCMA-trunk. Thus the ISUP traffic destined for node 129 would be flowing in timeslot 16 on the PCMA trunk. In the same way was the last 15 timeslots of the internal trunk mapped onto the PCMB trunk and the traffic going to node 130 would use timeslot 17 of the PCMB trunk.

6.3 The Traffic Generator

In order for the system to function correctly, the traffic generator had to match the configuration of the SS7 protocol stack. This was done by letting the traffic generator simulate two signaling points, node 129 and node 130.

In the traffic generator the network was configured as in Figure 6.3, i.e. the Spectra would be two nodes and the adjacent node to both of them would be node 128 (the stack/xSIM). It was also defined what timeslots that were to be used, which naturally were the same as in the stack, timeslot 16 for node 129 and timeslot 17 for node 130.

6.4 Tests

In order to test the simulation tool thoroughly a test was set up in the Spectra to establish 100 calls/s between two telephony users. The calls would then be running for a while until all 100 calls are released and then 100 new calls are established.

This test was ongoing and running during about 84 hours. Some errors occurred during this time but the simulation tool handled these correctly, by releasing faulty calls, and continue executing. About 350000 calls were established during this session.

6.4.1 Layout of the Test

The test that was used was divided into two parts, one part that would represent a subscriber calling someone, and one representing a subscriber that was called. Node 129 simulated the first part of the test while node 130 simulated the other part.

The calling subscriber was structured in the following way:

```
Send IAM (set up call)
Wait for ACM
Wait for ANM (call is answered)

Let call continue some time

Send REL (release call)
Wait for RLC (confirmation)
```

The called subscriber, on the other hand, started by waiting for the reception of an IAM:

```
Wait for IAM
Send ACM
Wait for subscriber to answer the call
Send ANM

Let call continue

Wait for REL
Send RLC
```

The waiting for an answer part was here simulated by letting the traffic generator wait for a period of time before sending the ANM.

Chapter 7

Summary and Conclusions

The main goal of the thesis was to implement a tool which terminates and analyzes ISUP signaling. The original goal was reached, but due to time restriction not all ideas regarding the tool was implemented. Moreover, an SS7 protocol stack was configured and interfaced with the purpose of receiving ISUP messages and also a traffic generator was used to transmit those messages. The tool was written in Erlang and C because the protocol stack, containing an API implemented in C, needed an interface.

The work was finished in the end but I do feel that too much time was spent on configuring the SS7 protocol stack and could instead have been spent on designing the implementation.

7.1 Problems during the Development

During the time at Ericsson a lot of time was put into reading documentation in order to understand what to do and how to do it.

There was a lot of documentation available for the SS7 protocol stack and it was not easy to grasp and understand. One reason was that there were so much documentation that it was difficult to find the correct one, and another reason was that a lot of terms used in the documentation are related to the telecommunication area in which I was not familiar.

In order to configure the protocol stack even more time was spent on retrieving information about how telephony systems function. In the end after a lot of reading the protocol stack was configured as intended.

Many other problems with the stack regarding the configuration arose during the continuous work, but all of those problems were solved.

7.2 Limitations and Restrictions

In the beginning the intention was to make the C-interface as general and dynamical as possible in order to allow it to be used for several protocols. The idea was to let all the

layers of the protocol stack be available for use by dynamically defining what layer the C-interface would implement.

Unfortunately, time was limited and due to poor and difficult documentation of the protocol stack, too much time was spent in configurating and understanding it. Therefore, it was decided to only implement the C-interface to handle the ISUP layer so that I would be able to finish the work in time. Due to the initial design some parts of the implementation are still reusable if improving the C-interface to apply to other layers as well.

Because the traffic generator used for generating ISUP signaling could not be remotely controlled no controlling features could be implemented in the application or its GUI. It would otherwise have been nice to be able to set, e.g. the call rate in the GUI instead of manually setting it in the traffic generator.

Another limitation is that the server handling all communication with the C-interface could be a potential bottleneck when dealing with high call rates, because it handles all communication between the C-interface and the call processes.

7.3 Future Work

There are several improvements that could be added to the simulation tool to e.g. make it more dynamic.

7.3.1 Dynamic and General Solution

By changing the C-interface to apply for all layers in the protocol stack, the application would be more reusable than the current version. Those kind of changes would also promote changes in the Erlang program in order for it to be more dynamic and reusable.

A framework for protocol stack interfaces implemented in Erlang is described by Anderson and Kvisth in [1], which could be used in order to improve the simulation tool. What the framework suggests is that the implementation would be split into two parts, where one is stack-dependent (the ”lower” part) and one is not (the ”upper” part). As in xSIM a communication protocol would be used in order to send requests to the stack and receiving indications from the stack. Much in the same way in the implemented C-interface would there also be need for some encoding and decoding functions to interpret the Erlang terms being sent to it and also written functions for handling requests to the stack and callbacks from the stack.

7.3.2 Controlling the Traffic Generator

Another improvement would be to use a traffic generator that could be controlled remotely by issuing Erlang or C commands, e.g. an AXE-10. An AXE-10 would allow the Erlang program to configure and control the AXE-10 by issuing commands in Erlang terms. Consequently features for controlling the call rate and other parameters could be added to the application.

7.3.3 Error Handling

It would be valuable to add error detection regarding the communication between the C-interface and the Erlang program. Currently, if the C-interface receives a primitive (message) from the Erlang program it does not recognize it will simply ignore it. Further improvements could be to add error detection codes which can be sent to the Erlang program in order for it to realize what error has occurred and maybe be able to correct it.

Chapter 8

Acknowledgments

I would like to thank my internal supervisor Ola Ågren for helping me with my report by correcting grammar. I would also like to thank my external supervisor at Ericsson, Karl Olsson, for helping me by introducing me to what to do and for always having time to answer my questions. Another big thanks goes to Magnus Hammar who made this master thesis possible and for continuously following up my work and answering all questions I had. Another thank you goes to my brother, Tomas Pihl, for explaining things for me, buying me lunch and for reading and commenting on my report.

Also, a big thanks to all people at the test department for interesting discussions at the coffee breaks and for their kindness.

A special thanks to Folke for the champagne at Café Opera.

References

[1] Peter Andersson and Markus Kvisth. A General Protocol Stack Interface in Erlang. Master's thesis, Uppsala University, Computing Science Department, March 2000.

[2] Joe Armstrong. The development of Erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, New York, NY, USA, 1997. ACM Press.

[3] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institue of Technology, Stockholm, Sweden, December 2003.

[4] Joe Armstrong and Thomas Arts. Erlang and Its Applications. In *Workshop on Constraint Programming for Time Critical Applications*, pages 27–28, Linz, Austria, 1997.

[5] Joe Armstrong, Mike Williams, Claes Wikström, and Robert Virding. *Concurrent Programming in Erlang*. Prentice Hall, London, 1994. ISBN 0-13-285792-8.

[6] Joe L. Armstrong and S. R. Virding. Erlang - an experimental telephony programming language. In *XIII International Switching Symposium*, June 1990.

[7] Thomas Aronsson and Johan Grafström. A Comparsion between Erlang and C++ for Implementation of Telecom Applications. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, November 1995.

[8] Computer Language Shootout Benchmarks. Erlang Benchmarks — Gentoo. Webpage, Last visit January 2006. `http://shootout.alioth.debian.org/gp4/erlang.php`.

[9] Bjarne Däcker. Concurrent functional programming for telecommunications: A case study of technology introduction. Technical report, Department of Teleinformatics, Royal Institue of Technology, Stockholm, Sweden, 2000.

[10] Erlang/OTP Group. Erlang/OTP R10B Documentation. Webpage, Last visit January 2006. `http://www.erlang.se/doc/doc-5.4.12/doc/index.html`.

[11] J. Halen, R. Karlsson, and M. Nilsson. Performance Measurements of Threads in Java and Processes in Erlang. Webpage, Last visit January 2006.

[12] Pälvi Isoaho. *ISUP ITU R8A - Functional Specification*. Ericsson Infotech AB, Karlstad, Sweden, 2004. Specification of ISUP layer.

[13] Signalling System No. 7 – ISDN User Part general functions of messages and signals. Standard, ITU–T – Telecommunication Standardization Sector of ITU, 1999.

[14] Signalling System No. 7 – ISDN User Part signalling procedures. Standard, ITU–T – Telecommunication Standardization Sector of ITU, 1999.

[15] Yinghua Jia and Joanne M. Atlee. Run-Time Management of Feature Interactions. In *6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, pages 27–28, University of Waterloo, Waterloo, Ontario, Canada, 2003.

[16] D. Richard Kuhn. Sources of Failure in the Public Switched Telephone Network. *IEEE Computer*, 30:31–36, 1997.

[17] Luis Calvo Lopez. *ISUP ITU R8A - Configuration File Description*. Ericsson Infotech AB, Karlstad, Sweden, 2001. Configuration of ISUP layer.

[18] Luis Calvo Lopez. *ISUP API R8 - Functional Specification - API*. Ericsson Infotech AB, Karlstad, Sweden, 2002. Specification of API to ISUP layer.

[19] Function specification: Media gateway. Internal document at Ericsson, 2006.

[20] J. H. Nyström, P. W. Trinder, and D. J. King. Evaluating distributed functional languages for telecommunications software. In *ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 1–7, New York, NY, USA, 2003. ACM Press.

[21] Packetizer, Inc. H.248 Standards. Webpage, Last visit January 2006. `http://www.packetizer.com/voip/h248/standards.html`.

[22] Switch Device Management (SDM) Protocol Messages. Internal document at Ericsson, 2002.

[23] Performance Technologies. SS7 Telephony Tutorial by Performance Technologies. Webpage, Last visit January 2006. `http://www.pt.com/tutorials/ss7/index.html`.

[24] TRH Protocol – Message and Element List. Internal document at Ericsson, 2001.

[25] Ulf Wiger. Four-fold Increase in Productivity and Quality. Technical report, Ericsson Telecom AB, Stockholm, Sweden, 2001.

[26] Wikipedia, the free encyclopedia. Webpage, Last visit January 2006. `http://www.wikipedia.com`.

[27] Paul Wilson. Connection Management (CM) Protocol Messages. Internal document at Ericsson, 2004.

Appendix A

Abbreviations and Acronyms

A.1 ISUP Protocol

ACM Address Complete Message (see Section 2.1.1.2 on page 6)
ANM Answer Message (see Section 2.1.1.3 on page 6)
CCR Continuity Check Request (see Section 2.1.1.8 on page 7)
CON Connect Message (see Section 2.1.1.9 on page 7)
COT Continuity Message (see Section 2.1.1.10 on page 7)
CPG Call Progress (see Section 2.1.1.11 on page 7)
IAM Initial Address Message (see Section 2.1.1.1 on page 6)
ISUP Integrated Service Digital Network User Part (see Section 2.1 on page 5)
REL Release Message (see Section 2.1.1.4 on page 6)
RES Resume Message (see Section 2.1.1.7 on page 7)
RLC Release Complete Message (see Section 2.1.1.5 on page 7)
SGM Segmentation Message (see Section 2.1.1.12 on page 7)
SUS Suspend Message (see Section 2.1.1.6 on page 7)

A.2 Telephony Systems

CM Connection Management (see Section 2.4.3 on page 17)
MG Media Gateway (see Section 2.3.3 on page 16)
MGC Media Gateway Controller (see Section 2.3 on page 15)
ML Mediation Logic (see Section 2.3.2 on page 16)
MTP Message Transfer Layer (see Section 2.2 on page 10)
PSTN Public Switched Telephone Network (see Section 1.1 on page 1)
SDM Switch Device Management (see Section 2.4.2 on page 17)
SP Signaling Point (see Section 2.1 on page 5)
SSP Service Switching Point (see Section 2.1 on page 5)
STP Signal Transfer Point (see Section 2.1 on page 5)
SCP Service Control Point (see Section 2.1 on page 5)
SS7 Signaling Systems no. 7 (see Section 2.2 on page 10)

TCAP Transaction Capabilities Application Part (see Section 2.1 on page 5)
TRH Transport Handler (see Section 2.4.1 on page 17)
TUP Telephone User Part (see Section 2.2.1 on page 10)