UPPSALA
UNIVERSITET

IT 13 045
Examensarbete 30 hp
Juni 2013

A reliable SMPP implementation in Erlang

Mikael Laaksonen

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

A reliable SMPP implementation in Erlang

Mikael Laaksonen

Software failure is one of the main problems in computer systems. Especially in systems with demand for high availability, such as telecom systems. SMPP is a protocol for sending short messages between entities in the telecom network. This thesis describes the design and implementation of a highly fault tolerant and reliable SMPP server and client. The implementation is done in Erlang, a language that was designed for fault tolerance. After testing the fault tolerance, the developed software has been integrated into telecommunication systems in commercial use at Mobile Arts.

Handledare: Dragan Havelka
Ämnesgranskare: Karl Marklund
Examinator: Olle Gällmo
IT 13 045
Sponsor: Mobile Arts

Tryckt av: Reprocentralen ITC

Contents

Chapter 1

Glossary

API Application Programming Interface

BSC Base Station Controller

BTS Base Transceiver Station

CDMA Code Division Multiple Access

ESME External Short Messaging Entity

FSM Finite State Machine

GSM Global System Mobile

GUI Graphical User Interface

HLR Home Location Register

IAM Initial Address Message

IMS IP-based Multimedia Services

MC Message Center

MCA Missed Call Alert

MIB Management Information Base

MS Mobile Station

MSC Message Switching Centre

MTTF Mean Time To Failure

OTP Open Telecom Platform

PDU Prcoess Data Unit

PID Process ID

RX Receiver Mode

SM Short Message

SMPP Short Message Peer-to-Peer

SMS Short Message Service

SMSC Short Message Service Centre

SS7 Subsystem number 7

TRX Transceiver Mode

TX Transmitter Mode

UMTS Universal Mobile Telecommunications System

VLR Visitor Location Register

VM Virtual Machine

WAP Wireless Application Protocol

Chapter 2

Introduction

2.1 Motivation

We live in a digital age where almost everything we encounter in one way or another is directly or indirectly dependent on computer software and hardware. From the cars we drive, to banking or making a phone call. Even when we don't interact with them directly we rely upon computer systems to work as expected and to be available when we need them. Telecommunications is an area where high reliability is strived for. Here is often talked about *five nine*, that is 99,999% availability [15]. If translated into time this would equal to a downtime of maximum 5.26 minutes per year.

This thesis is a study of how the Short Message Peer-to-Peer protocol (SMPP) [12] can be designed and implemented in a highly reliable manner. I aim to address the following questions:

- What is the best practice for developing such an application using the programming language Erlang [2]?
- What coding paradigms are used and how can those be applied in other applications?
- There is no fault free software, therefor faults must be handled in some way. How should a programmer handle faults and what can be done so that the same fault does not occur again? How can the cause of be found?
- How are fault handling and error handling applicable in Erlang?
- How does fault handling and error handling affect design?

This thesis was conducted at Mobile Arts. A global IT company with its headquarter in Stockholm, Sweden. Mobile Arts develops software and platforms in IT/telecom industry. SMPP as a telecommunications protocol is in extensive use in the company's products.

2.2 Aims and objectives

The objective apart from addressing the questions posed above, is to implement a working prototype of a SMPP server and client. By the least this means that the server/client duo shall be able to:

- Initiate a session.
- Exchange messages allowing for SMS traffic.
- Integrate into existing Mobile Arts software.

In addition, more or less harsh faults will be provoked to the system trying to simulate real world situations as closely as possible. If there is time this will be done in a comparative manner towards another SMPP server/client setup. Future plans are for Mobile Arts to be able to use the implementation in production if it is able to achieve the desired qualities.

2.3 Related work

The SMPP protocol is an open specification. Many companies have developed their own implementations. Noteworthy is Logica's own additions to the standard which was given the name 4.0 which was customised for Japan [12]. There also exist several open source implementations. For example OSERL [4], which is written in Erlang.

Joe Armstrong's thesis *Making reliable distributed systems in the presence of software errors* [8] is related to this project. Armstrong introduces Erlang and shows its concepts and his ideas of how to make reliable software.

2.4 Outline of the thesis

Introduction This chapter will discuss the motivation and the objectives of the thesis.

Background This chapter provides a background to software reliability and basic telecom concepts with emphasis on the SMPP protocol. A subchapter will show SMPP's place in a larger telecom network. The programming language Erlang is introduced together with the database Mnesia. Reliability concepts in Erlang are introduced.

Design This chapter shows the design of the application. Requirements on reliability are presented. In the end is a discussion of how the chosen design leads to a highly reliable system.

Implementation This chapter describes the implementation of the applications. It is limited to practical issues such as tools and methodology but will also include a short discussion regarding the SMPP protocol in general terms.

Evaluation This chapter will evaluate and verify the implementation.

Conclusions The last chapter will present the conclusions, a summary and a discussion of future works.

Chapter 3

Background

3.1 Software reliability

3.1.1 Background

Software is omnipresent in today's society. From the programs controlling power plants ensuring that there is electricity in our homes to the gadgets and computers occupying those -software has undoubtedly become a factor in our everyday lives. As our dependency upon software systems has increased, the possibility of crisis from its failure has also increased. The need and demand of more complex software has increased more rapidly than the ability to design, test and implement them [15].

The effects from software failure can range from minor inconvenience, such as loosing unsaved text documents, to major economical loss or even death. Recent examples from Sweden's telecommunications industry are the major outages the operator Telia experienced in May 2012 [20]. One of the most serious reliability failures occurred for the IT-consultant Tieto affecting the Swedish pharmacy Apoteket, several municipalities and many other companies [17]. Software reliability is often regarded as one of the top factors for the customers' perception of the software's quality [15].

3.1.2 Definitions

Here is a list of basic terminology for software reliability [15]:

Software system An interacting set of software subsystems that are embedded in a computer environment.

Failure A failure is when the user perceives that the system fails to deliver the expected result. Failures are in some systems associated with different levels of severity.

Fault The cause of a failure is called fault.

Outage An outage is a failure in the dimension of time.

Error A discrepancy between the theoretically correct value and the computed/observed value.

Following from the definitions there is a difference between an error and a failure. An error that occurs could be fixed, overridden, or even unnoticed by the user. From a user's perspective the important aspect is that the software system does not violate the contract. It is then it becomes a failure. If it happens too often the software will be deemed unreliable.

In the field of software reliability there are four main ways to achieve the objective [15]:

(A) Fault prevention -to avoid by design that faults occur.

(B) Fault removal -to detect, by verification and validation, the existence of faults and eliminate them.

(C) Fault tolerance -to provide, by redundancy, service complying with the specification in spite of faults having occurred or occurring.

(D) Fault prediction -to estimate, by evaluation, the presence of and the occurrence and consequences of failures. This has been the focus of the research field software reliability modelling.

This thesis work will focus on fault prevention and fault tolerance. Fault prediction is what software reliability engineering has been concentrated on, it has an ANSI specification which defines it as [7]:

> *The probability of failure-free software operation for a specified period of time in a specified environment.*

The above definition quantifies reliability and therefor implies a statistical approach to software reliability engineering which is out of scope. However it should be mentioned that metastudies [18] have questioned the practical industrial value of fault prediction.

In this thesis a less strict definition will be used. Leaving out the statistical aspect, the goal becomes:

> *Creating fault tolerant software.*

Designing for fault prevention is in many ways the same as designing for fault tolerance. From here on when the term *reliability* is used, this is what is meant.

3.2 The SMPP protocol

3.2.1 Overview

The Short Message Peer-to-Peer Protocol (SMPP) [12] is a communication protocol at the application layer. This is one of the main layers in the OSI model

[5]. The company Aldiscon [1] created the specification which began as a closed protocol. In 1999 it became open and it has grown to become the industry standard for short message, machine-to-machine communication. SMPP supports any cellular technology such as GSM, UMTS or CDMA. In the project only the GSM network adaption will be considered.

Described as a client/server model, the client is called External Short Message Entity (ESME) and the server Message Centre (MC). The most common MC is a Short Message Service Centre (SMSC), which is the telecom node responsible for delivering and handling SMS:s. A typical ESME is a network SMS client such as a voice mail server (when sending new voicemail notifications with SMS:s) or a tele-voting server (which receives SMS:s from the general population voting for who is the best dancer).

3.2.2 Protocol data units

The SMPP protocol consists of a set of operations, each one has the form of a Protocol Data Unit (PDU). The operations are either a request or a reply, the latter indicates success or failure. For example the submit_sm request is replied with a submit_sm_resp. Operations are either MC specific, ESME specific or generic available. Operations are also associated with certain states or contexts. For example the submit_sm is unavailable to the ESME when it has no connection established to a MC. Operations are classified in the following groups.

Session management These operations are used to establish a SMPP session and provide means of handling unexpected errors. They are used for sending authentication data consisting of an identification and a password. All PDU:s beginning with *bind_* are sent from the ESME to establish sessions: bind_transmitter, bind_receiver, bind_transceiver, unbind and enquire. The PDU replied has the suffix *_resp*. The operation outbind is MC originated and is replied with a PDU for establishing a session. The generic_nack is sent by the MC or ESME to indicate a failure. An alert_notification is sent from the MC to the ESME to indicate that it is available.

Message submission These operations are used to submit messages from the ESME to the MC. With the corresponding reply PDU (adding *_resp*) they are: submit_sm, used for short messages; submit_data, a simpler form used for example in WAP messages; submit_multi, supporting short messages to multiple recipients.

Message delivery These operations enable a MC to deliver messages to the ESME. The deliver_sm is related to submit_sm and is used by the MC to notify the ESME that a submit_sm has been delivered to the recipient. It is replied with a deliver_sm_resp. The two other operations are data_sm and data_sm_resp.

Message broadcast These operations are designed to provide cell broadcast service within a MC. A broadcast ESME, wishing to broadcast a short message, can use this PDU to specify an alias, geographical areas and text of the short message.

Ancillary operations These operations are designed to provide enhanced features. Noteworthy are cancel_sm, to cancel a submit_sm and replace_sm, to update the content of a submit_sm stored at the MC.

3.2.3 The SMPP PDU

The general PDU format is a 16-octet header followed by an optional body. See figure 3.1. The header consists of Command length; Command id, to identify the PDU; Command status, used for error codes; and finally Sequence number. Each of the four fields in the header is four octets long.

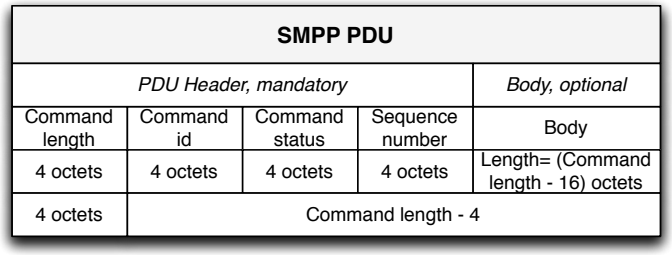

Figure 3.1: SMPP PDU

3.2.4 SMPP sessions

The SMPP protocol is an application layer protocol. The transport layer is commonly TCP, but X.25 or SSL are also used [12].

In the beginning of a session both the MC and the ESME are in the Open state, which means that there is no connection established. Most commonly the ESME will initiate the session sending a PDU requesting a session to the TCP/IP port that the MC is listening to. The sent operation will be a bind_receiver, bind_transmitter or a bind_transceiver. If the PDU is authenticated the two entities will change state to Bound_TX, Bound_RX or Bound_TRX depending on the type of session. In transmitter mode, Bound_TX, the ESME may send short messages to the MC for onward delivery to a mobile station. In receiver mode, Bound_RX, the ESME may receive short messages. In transceiver mode, Bound_TRX, the ESME may both receive and transmit short messages.

To gracefully tear down a session, either the ESME or the MC shall send the unbind PDU. After exchanging request and response, the two entities will enter the Unbound state. Figure 3.2 shows the setting up of a SMPP session with the ESME in transceiver mode.

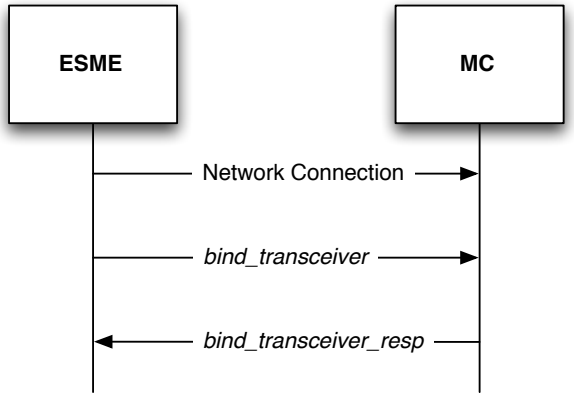

Figure 3.2: SMPP session initialisation

The SMPP protocol is asynchronous in the aspect that each PDU must not immediately be replied with a corresponding response PDU. The reason for this is to use the bandwidth more efficiently. To accomplish this each PDU has a sequence number set that is used to match request to response. Regardless of when an asynchronous response arrives, it can be immediately re-matched to the original request PDU.

Figure 3.3 shows a SMPP session with submit_sm:s and deliver_sm:s in an asynchronous order. The bind messages have sequence number one. When the bind_transceiver PDU is received the ESME transfers in to a transceiver state called Bound_TRX. It is now ready to send and receive SMPP messages. Two submit_sm:s are sent from the ESME client, bearing sequence number two and three respectively. Then the MC sends two submit_sm_resp:s acknowledging the previous two submit_sm:s with the corresponding sequence numbers. A deliver_sm is sent from the MC to the ESME with sequence number four. It is replied by the ESME. The final dialogue pair of messages are the unbind and the unbind_resp messages. They have sequence number five. The ESME enters the Unbound state. Eventually the session is closed.

3.2.5 Failures in SMPP

Failure handling mentioned in the SMPP specification is of two kinds. Problems related to the connection between ESME and MC, and failures related to PDU:s sent. A failure here follows the definition from above, it is a discrepancy between the expected outcome and the actual outcome. It is not necessarily caused by an error.

A connection failure is when the ESME or MC is unable to connect or when the established connection is lost. There are three probable causes for this:

- The network configuration is incorrect.
- The MC or ESME on the other side is unavailable.

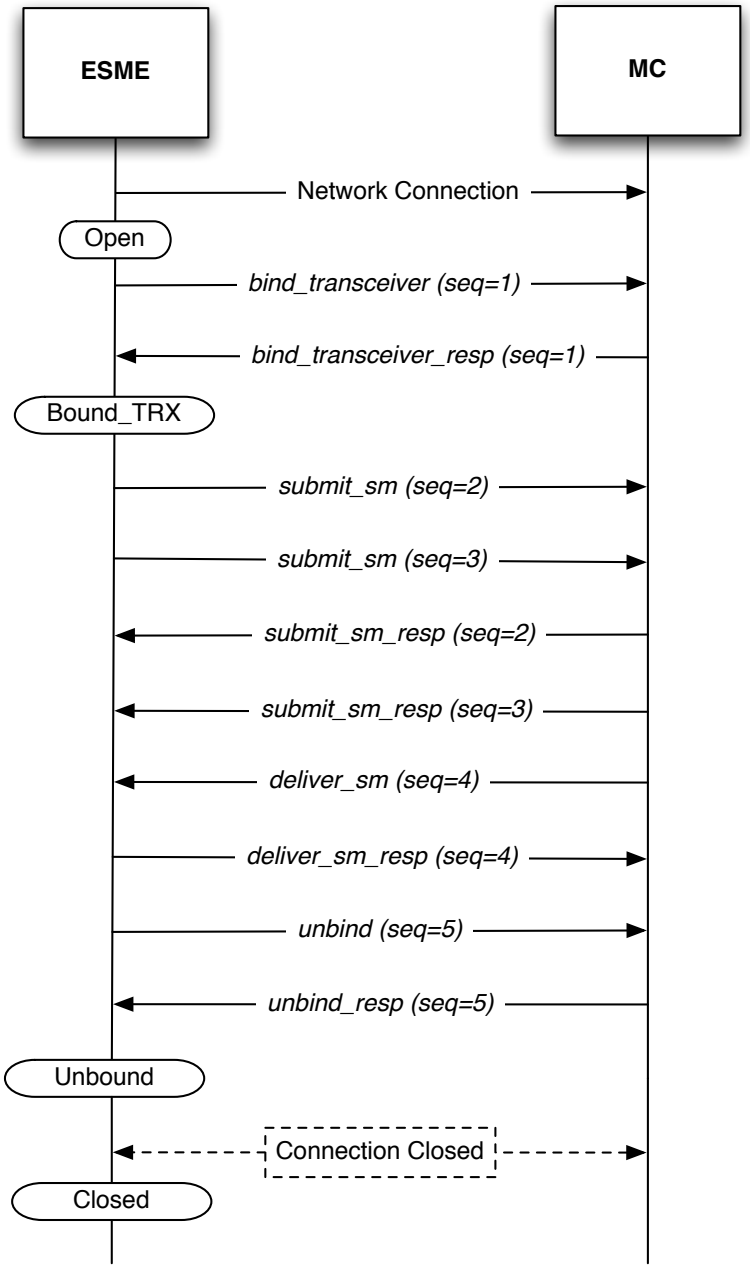

Figure 3.3: SMPP session example

- The network connection between the ESME and MC is unavailable.

The recommended approach to handle this is for the ESME to try to reconnect on regular intervals. When the issue is resolved the connection will be reestablished. It is also recommended for the MC to try to reestablish the connection if it has undelivered messages.

The other type of failure is operational failure. When the received PDU does not comply with the current state of the receiver or is malformed. The general recommendation is to reply with a response holding the appropriate error code. The ambition is to give as much help as possible to identify the error. If that is not possible, a generic_nack message is sent as a reply. Examples of these situations are when the command length is invalid, when the PDU is unrecognised, when the PDU is received in the wrong state or when the requested operation is disallowed.

3.3 SMPP in the context of telecom

This is a short example to show SMPP's place as a telecommunications protocol in a larger network. Figure 3.4 shows the network used in the example.

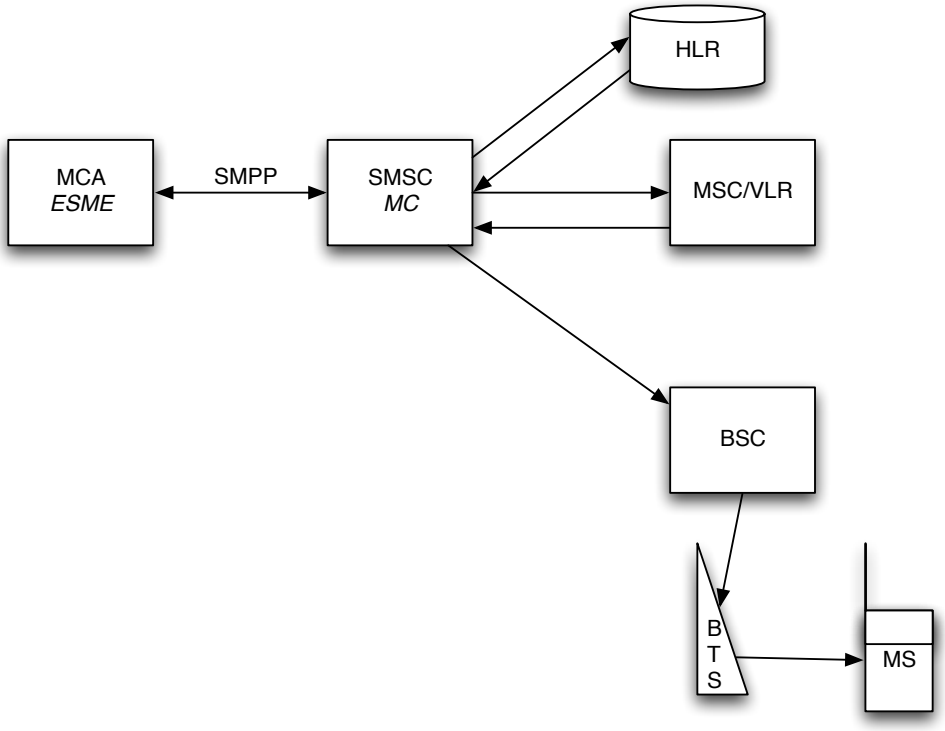

Figure 3.4: Network example

The ESME client will be a Missed Called Alert (MCA) node. When a call is placed from subscriber A's Mobile Station (MS) to subscriber B's MS and B does not answer, B receives a notification SMS telling about the missed call. This functionality is used by telecom operators to increase call throughput.

The network node Mobile Switching Centre (MSC) routes calls in the telecom network. When B does not pick up, the call setup message called Initial Address Message (IAM) is rerouted to the MCA node. The IAM will contain information about who called and who was called. Using this information the MCA will create a SMPP message submit_sm that will be sent to the receiver of the call. This message is sent between the MCA and the SMSC. The SMSC is the MC server.

To deliver the SUBMIT_SM to the MS the SMSC must locate the subscriber in the network. It will query the Home Location Register (HLR) for the Visitor Location Register (VLR) where the subscriber currently is. Each operator has at least one HLR, and it is the main database containing all mobile subscribers. Knowing the location the SMSC will try to send the message through the MSC/VLR and further to the radio network where the Base-Station Controller (BSC) and the Base Transceiver Stations (BTS) construct the radio network parts that functions as the external interface towards the MS.

3.4 The Erlang programming language

Erlang is a functional programming language designed at Ericsson Computer Science Laboratory. It was designed to meet the demands of high reliability and fault tolerance in the telecom industry. Noticeable features are [2, 8]:

- Garbage collection.
- Dynamic typing.
- Hot swapping (code can be changed without stopping the system).
- Soft realtime.
- Runs in a virtual machine.
- Support for distributed systems.

Further more it has strong support for concurrency and processes are lightweight. To have a system with thousands of concurrent process is considered computationally cheap [2, 8]. Distributed computing has strong support. The syntax is based on Prolog. For further reading start with the Erlang/OTP homepage [2] or one of the recently released books in the subject such as Programming Erlang by Joe Armstrong [9] or Erlang Programming by Francesco Cesarini and Simon Thompson [10].

3.5 The Mnesia database

Mnesia is a distributed database management system (DBMS) classified as a relational database. It was developed at Ericsson Computer Science Laboratory as a DBMS for Erlang and it is likewise written in Erlang. Likewise it was designed to meet the demands of high reliability and fault tolerance in the telecom industry [16].

Mnesia provides fault tolerance towards hardware and software errors through replication. A database table can be replicated on several nodes. Write operations will be performed on all replicas while read operations are performed on the local host for performance. However, the data location is transparent. If a host is unavailable during a write, it will later when available synchronise with the alive replicate through traversing of a transaction log and performing the necessary updates to the database. Mnesia can also separate data from garbage in case of a complete disc crash [16].

Requirements for the design were [16]:

- Fast real-time key/value lookup.
- Complicated non real-time queries mainly for operation and maintenance.
- Distributed data due to distributed applications.
- High fault tolerance.
- Dynamic re-configuration.
- Complex objects.

Other notable characteristics are:

- Three storage types –RAM only, persistent only and RAM *and* persistent storage.
- No type definitions.

Another benefit of using Mnesia when developing in Erlang is that the data format is the same. There is also a query language called Mnemosyne.

3.6 Reliability in Erlang

A set of key concepts is used to achieve reliability in Erlang. These are characteristics built into the language, the supporting libraries called Open Telecom Platform (OTP) and programming idioms.

Erlang's approach to reliability can be summarised with a quote from Armstrong's thesis [8]:

> *Since eliminating all (such) software errors for large software systems is an unsolved problem I think that the only realistic way to build large reliable systems is by partitioning the system into independent parallel processes, and by providing mechanism for monitoring and restarting these processes.*

First of all, there are no (complex) software systems without any bugs or errors. Even a mature system where most of the bugs have been eliminated, errors can still occur from outside input or due to hardware errors. Yet still, software updates and bug fixes might introduce new errors. This was a key assumption when the language was designed [8].

The second design philosophy was to organise tasks in a hierarchy in order of complexity. If a high level task fails, try to perform a simpler task. If that too fails, try to perform an even simpler task. If the task at the lowest level also fails the system will fail.

3.6.1 Processes and process links

Erlang is a concurrency oriented language, using processes as the atoms for concurrency. Processes in Erlang are strongly isolated. They share no state or memory and communication between processes is done solely by message passing. If one process crashes it has no effect on any other processes. In this way consequences of errors are limited to a single process. Each process is identified with its Process Id (PID).

Two processes can be linked together creating a process link. Links are used for monitoring. If a process terminates an exit signal will be sent to all its linked peers. The receiver may terminate as well or handle the exit in some way. This is used to build hierarchical program structures (see the supervisor behaviour below).

3.6.2 Let is crash

A function call has two possible outcomes. Either a value is returned or there is an exception. When a process encounters an error it should fail immediately. An error could for example be division by zero. The code evaluation is stopped and an exception is thrown. At the next level, the exception may be caught (using the construct try/catch) and corrected. If so, no harm is done. If not, the exception is thrown yet another level and so forth. If at the final level the exception is not caught the process will crash and send this information to all linked processes. At that point the error has become a fault, which might be perceived as a failure by the user. These are the mechanics that implement the hierarchy of tasks.

The guideline regarding exceptions is to separate the processes that try to accomplish tasks from the processes correcting errors. A process should have a clearly defined task. If it fails to accomplish this task it should fail immediately and the higher levels will correct it. This is what is meant by *let it crash* [8].

3.6.3 Don't program defensively

The idiom to not program defensively is related to the idea of letting a process encountering an error to fail fast. When writing functions one should not check the correctness of the input, but instead assume that the function will evaluate and return a result. The exception to this is input from the external world, for example received TCP packages. To not program defensively should be seen as a coding guideline when writing Erlang programs [8].

3.6.4 Let someone else do the error recovery

A process should try to accomplish its task, if it encounters an error and crashes, it should let some other process clean up. There are several advantages to this [8]:

- The process encountering the error and the process handling the error are separated.
- The code becomes more readable.
- The idiom works well in a distributed system.
- A system can be built and tested on a single node system, and porting the code to a distributed system calls for little changes of the code.

3.6.5 OTP

The Open Telecom Platform (OTP) is a set of libraries and tools for Erlang. It was originally designed for developing telecom software systems. The distribution contains compilers and tools, the Erlang run-time system, software libraries, design patterns, the Mnesia database and documentation. As of 1998 it is open source.

An overview of OTP is shown in figure 3.5. The Erlang run-time system is a virtual machine running on top of the native operating system. All Erlang processes are managed by the run-time system. As such, it offers many of the services traditionally provided by the operating system [8]. Running on the virtual machine are included applications such as the Mnesia database, a web server, etc. On top of the OTP layer are other Erlang applications or other software. The SMPP server and client from the developed would be placed here.

3.6.6 Behaviours and design patterns

Behaviours are central part of the OTP system. They are formalizations of common code patterns. The four standard behaviours are *supervisor*, *gen_server*, *gen_fsm* and *gen_event*. A behaviour consists of two parts. The generic part that is included in the OTP library. It implements the generic process work

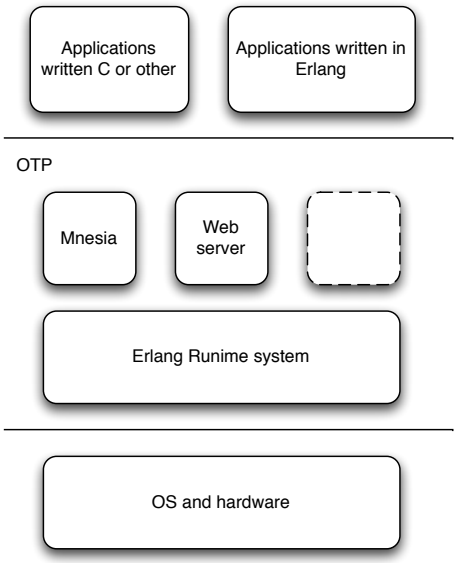

Figure 3.5: OTP overview

and error handling. The specific part consists of callback functions that will implement the logical functionality for the behaviour.

The gen_server implements the server in a client-server module. It can be viewed as central server managing a resource and several clients attach to it. The gen_fsm is a behaviour for finite state machines and the gen_event is a generic event handler.

The supervisor starts, stops and monitors child processes. The basic principle is that a child processes is linked to the supervisor process. If the child process crashes it will be restarted by its supervisor.

The most common design pattern using Erlang/OTP is the *supervisor tree* (also known as the *worker-supervisor pattern*). It is a hierarchical arrangement of supervisors and worker processes as shown in figure 3.6. The child process is either another instance of a supervisor behaviour or an instance of one of the other three behaviours. Such a process will be called a *worker*, as it performs the computational tasks.

A collection of workers and supervisors are called an *application*. The OTP system consists of several applications. It should not be confused with the more generic term *application* referring to a software program.

Remember the philosophy to break down a task into simpler and yet simpler sub tasks. The supervisor tree could be viewed as such. The idiom to *let someone else do the error recovery* could be applicated to supervisors, the supervisor being the someone else.

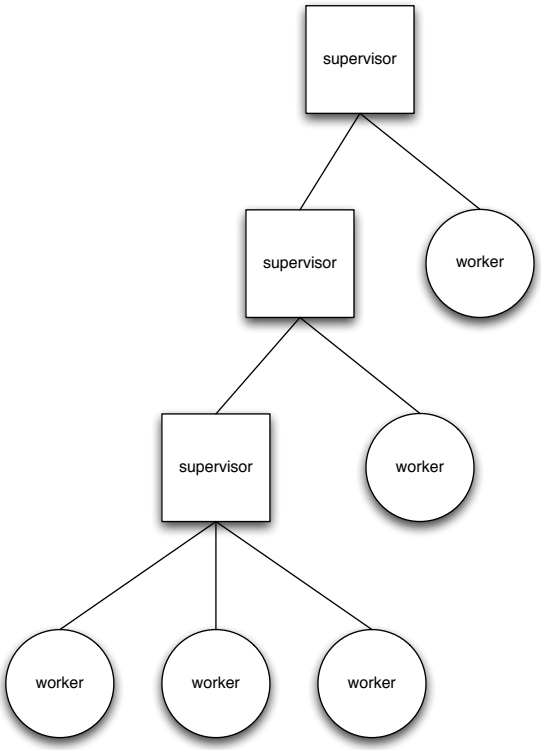

Figure 3.6: Supervisor tree

3.6.7 Discussion

Do Erlang and OTP provide a good platform for software reliability? In my opinion it does. The language and runtime environment provides features such as hot code swapping. And the OTP together with its design patterns and coding paradigms are designed to give the necessary tools.

The fundamental criteria of reliability have not yet been mentioned: To make a fault tolerant system you need (A) two separate computers and (B) the software distributed among them. The two systems must also be able to detect if the other one crashes. If one of the computers encounters an error and crashes the other should be able to immediately take over the operation. In Erlang this is solved through a strong support for distribution.

Chapter 4

Design

This chapter will present the design of the SMPP software developed during the thesis. First of all the requirements are presented. Emphasis is on reliability. Then the main modules and database designs are shown. Along side is a discussion about how the requirements are met.

4.1 Requirements

4.1.1 Functional Requirements

Task A highly fault tolerant and reliable SMPP client and server should be prototyped. It should follow the SMPP protocol version 5.0. This is the latest version from 2003.

Integration The client and server should integrate with existing Mobile Arts software. In other words, some or at least one server system and one client system should be able to use the implemented prototype to send and receive short messages.

The new version of the SMPP client/server must be capable to replace the old version used in Mobile Arts products without any disturbance or noticeable difference. It must also fulfill requirements to revert back to the old version during runtime.

4.1.2 Elaboration of Requirements for Reliability

The thesis shall investigate the implementation of a highly reliable software system. As stated in chapter 3 the project would aim to achieve reliability through fault prevention and by fault tolerance.

To concretise the task the following requirements are constructed:

- Fault tolerant to hardware failures [R1], the software system constructed

should be able to function after hardware failure.

- Fault tolerant to software failures [R2], internal or external events will lead to faults. The software system shall be designed to handle those faults. Also side effects of those faults should be minimal.

- Quality requirements [R3], the software system should have an acceptable level of service even if faults have occurred.

- Error identification [R4], the faults that occur must be documented in some way for debugging and fixes.

- Clearly structured code [R5], future maintainers should easily understand the software system.

4.1.3 Non-functional Requirements

Environment The prototype should run alongside the existing company environment. This means the common subsystems for configuration, counters, MIB files [3], alarms, system building, etc, should be used.

Language Erlang/OTP R13B04 should be used.

Timetable The implementation should be finished in 16 weeks and the almost ready draft of the written report should be handed to the supervisor after 18 weeks. It is suggested that the writing of the report is done in parallel with the implementation. Another suggestion is to keep notes or a diary to document the development process.

High reliability. The software should be designed to provide a very high level of reliability.

4.2 Overview

The SMPP system is divided into two logical units: SMPP client and SMPP server. The SMPP client implements the state machine that handles the client side of the SMPP protocol. The SMPP server implements a state machine that handles the server side of the SMPP protocol. These two are completely separate, and usually are used exclusively. The common part for both the client and the server, like parsing functions, encoding/decoding libraries are extracted to a common library.

4.3 ESME Client Application Design

4.3.1 Application architecture

The ESME client developed is called *ma_smpp_client*. The client is designed in the supervisor tree design pattern. Examining the hierarchy two levels of processes are made clear. The upper level is the supervisor level. The lower is the session level. The process hierarchy of the application is shown in figure 4.1. The upper level consists of four processes. *Ma_smpp_client_app* is the top-level application process. *Ma_smpp_client_sup* is the main supervisor responsible for starting the other process. *Ma_smpp_client_observer* is a watchdog process used to manage alarms. *Ma_smpp_client_manager* is a supervising process which starts each session.

Figure 4.1: Process hierarchy

The session level consists of session units. Each session unit represents one configured SMPP session. Every such session unit consists of three processes with distinct tasks. The supervising process is called *ma_smpp_client_session_monitor*. It is responsible for starting, monitoring and restarting the two actual worker processes, *ma_smpp_client_worker* and *ma_smpp_client_tcp*. The *ma_smpp_client_tcp* processes's task is to manage TCP sockets and send and receive data over

those. If it crashes it is restarted by the supervising process *ma_smpp_client_session_monitor*. So goes for all processes, if one crashes it will be restarted by the supervisor. The *ma_smpp_client_worker* implements the SMPP logic. It implements the correct protocol procedures for setting up and tearing down sessions and it carries the protocol logic for sending and receiving messages. In addition it performs the proper actions based on these events.

Initiation of SMPP sessions is based on a configuration where each session is saved together with authentication data and IP data. The top-level *manager* will read this configuration and start up session units, one per configured session. The worker process will call upon the tcp process to establish a session and eventually send a bind PDU. When the session is in a bound state other subsystems may use the ESME for sending and receiving short messages.

4.3.2 SMPP Client API

The ESME client is interfaced from other subsystems via the module *ma_smpp_client*. It exports:

- send_sync, for synchronous sending of SMPP messages from the calling process' point of view.
- send_async, for asynchronous sending of SMPP messages.
- deliver_resp, for sending response messages.
- get_systemIds_andIPs, is used by the calling process to know what sessions are bound in either transmission, receiver or transceiver states.

The module is deliberately made short and simple to hide the underlying processes and the underlying SMPP stack. This means that other developers using the application only need to understand a short and simple subset of the code.

It should be mentioned that it is the calling process that will perform encoding of the messages. The alternative is to have the SMPP application to do this. But that would potentially create a bottleneck.

4.4 MC Server Application Design

4.4.1 Application architecture

The MC server developed is called *ma_smpp_server*. As the client, the server is designed according to the supervisor tree design pattern.

The central process/module is the generic server *ma_smpp_server_packet* process implements the SMPP MC server logic, as from the SMPP Specification. It handles the protocol procedures for setting up and tearing down sessions and it implements the protocol logic for sending and receiving messages. In addition it performs the proper actions based on these events.

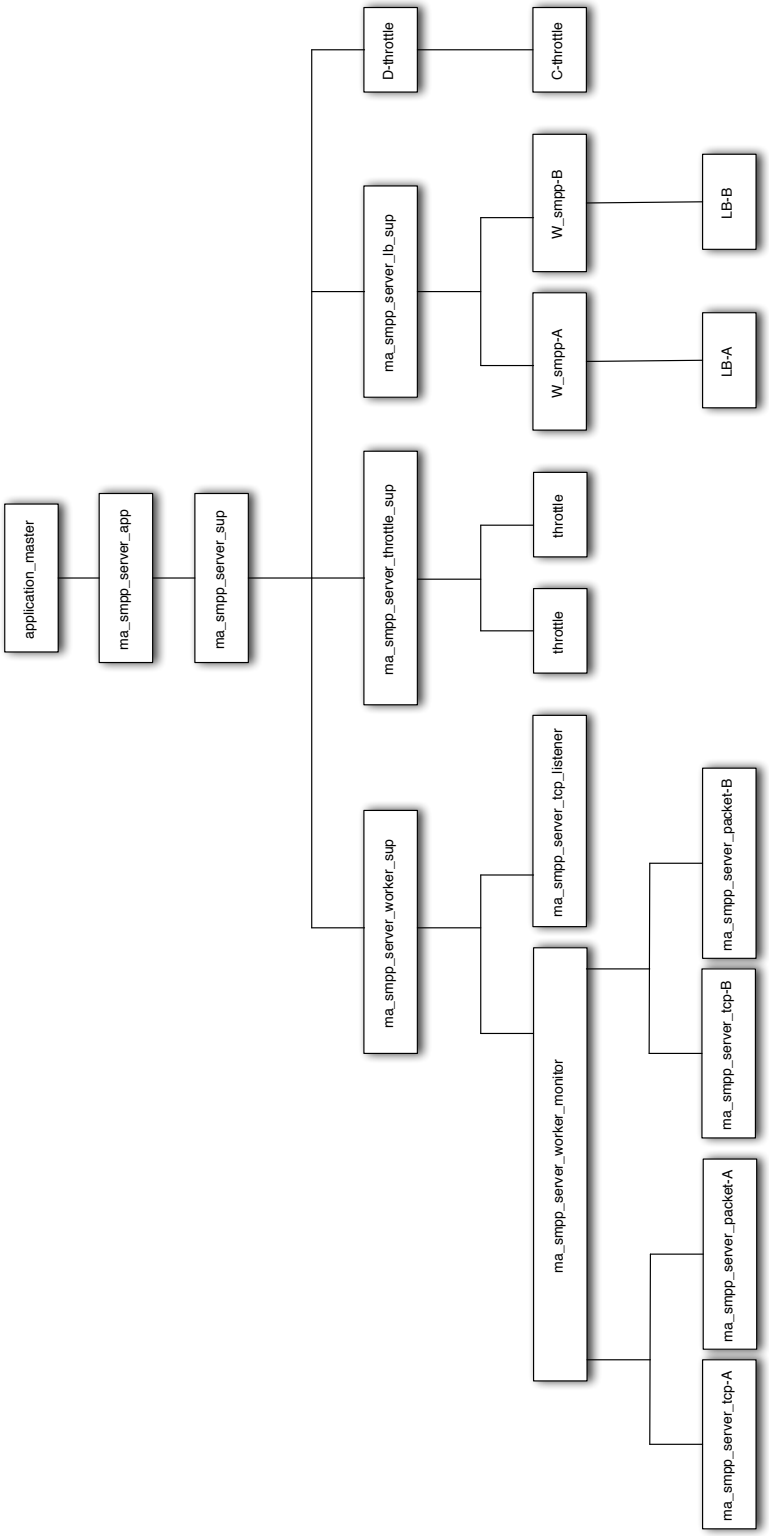

Figure 4.2: MC server process tree

When a client initiated session is setup, first of all the *ma_smpp_server_tcp_listener* will receive a TCP packet on its configured port. Doing so it will call upon *ma_smpp_server_monitor* to start the working pair: *ma_smpp_server_tcp* and *ma_smpp_server_packet*. If the session is authenticated it will enter a bound state ready to send and receive SMPP traffic. These processes described together construct the main SMPP logic and they are supervised by *ma_smpp_server_worker_sup*.

The other processes running in the application are related to throttle handling and to load balancing. The throttles are there to protect the server from overload. There is a global throttle that would apply for a whole SMSC node for example. Setting a maximum for the whole system. These are the *dthrottle* and its child process *cthrottle*. Then each SMPP session would have a throttle. These are the processes supervised by *ma_smpp_server_throttle_sup*.

Finally there are load balancing process called *LB*. There will be one per session. They have each one parent, *W_smpp_sessionname* that in turn is supervised by *ma_smpp_server_lb_sup*.

The figure 4.2 shows the server application with two connected ESME:s. There are two working pairs, two load balancers and two local throttles.

4.5 Discussion of the design

The similarities between the client and server are not coincidental. The structures, names and the idea of a working pair are seen in both the applications.

The processes implementing the stack logic are gen_servers. Though it would appear natural to implement a stack as a finite state machine utilising the gen_fsm behaviour. To my experience, my opinion is that the gen_fsm behaviour is harder to work with.

All processes spawned are monitored in some way. This is to avoid rogue processes that would become a memory leaks lowering the possibility for the software to run for a very long time.

Each process has a distinct task. And there is only one process per module. This helps to keep the complexity of the source code down.

Faults may occur everywhere in the application, that is why all the processes are supervised. Another way of looking at the two-layer design is to view it as three layers of supervisors. Where every underlying layer has a more specific task than the layer above. At the top layer is the application's main supervisor. Its task is to start the lower layers. Below is the *session_manager*, it is responsible for starting all of the session units. And in every session unit there is a *monitor* who supervises the worker pair. This means that all SMPP sessions are independent from each other. If for example *tcp A* experiences en error, it will crash and be restarted. But that will not affect the neighbouring process *tcp B*. This design gives a layered independence between the processes and the consequences of errors are isolated. After an error occurs, the process will be restarted. It gives a high tolerance against software faults, which satisfies requirements [R2] and [R3]. The design follows the idiom of *let it crash*.

There is very little internal communication excluding the start up of the supervisor tree and the initialization of links. It only occurs between the *ma_smpp_client_tcp* and the *ma_smpp_client_worker*. Minimising the process interaction minimises the complexity of the system. Keeping this kind of complexity at a minimum makes for less errors in the application.

The server's and the client's databases are interfaced through their respective interface module. Together with hot code swapping will allow for continuous operation. It is a part of achieving requirement [R2] and [R3].

The SMPP client and server are designed to run on distributed systems. To achieve fault tolerance against hardware failures [R1] the SMPP client/server must be deployed on two or more hosts. In this way, if the hardware of *node A* experiences a hardware failure *node B* will still function. The deployment on two or more hosts is achieved with a shared database and distributed configurations. The requirement [R1] will be met as soon as the ESME client or the MC server is deployed on two separate hardware systems. At the same time the tolerance for software errors [R2] and the quality of the software [R3] are addressed.

The try/catch construction is used sparsely. It is mostly used in encoding and decoding functions. This is because these functions take care of external input that might be erroneous. It is used so that the process trying to encode or decode a faulty PDU does not stop here. Instead the event is logged and each session keeps track of its failures.

Writing system logs and error logs to file is done in two different ways. The first type is file logging, where events are written in plain text to log files. How verbose the logs are can be controlled. Using a very verbose (debug level) logging is useful for finding errors, but it makes the logs hard to read. The other type of logging is event logging. Events and states are saved to an external database and are indexed and made searchable. This data is presented in a GUI. Both log types are based on existing infrastructure developed at Mobile Arts. They are accessible via macros and callback functions. All this addresses the requirement for error identification [R4].

Although the criteria for well structured code [R5] can be seen as subjective there are many rule of thumbs for good programming practice. Modules have intentionally been kept short, less than 600 lines of code in each module, except for *ma_smpp_client_worker*. Lines are no longer that 80 characters wide. This is close to the recommendations in [8]. Functions of simple nature have been moved to library modules and commonly used functions have been moved to yet other library modules. Each type of spawned process is kept in its own source code file. That is, no file spawns two different kinds of processes.

Inspired from a discussion on the Erlang mailing list [11], function names are verbose in their nature, while variables are one or two letters. Below is a short example of the style:

```
L2 = sort_msisdns_ascending( L )
```

However, this is a matter of style. It is up for discussion if this makes the code's intentions more clear or not. One argument is that keeping a style, whichever one may chose, leads to more readable code as the code will inherent

a certain structure. This attribute does in itself add to the readability and understandability of the code. If the requirement [R5] is fulfilled is for future developers to decide.

The quality of the service delivered is the sum of the other requirements. In this chapter has been shown that the requirements [R1-R5] have been addressed.

Chapter 5

Implementation

The master thesis was conducted at Mobile Arts, Stockholm, Sweden. The goal of the thesis is to design and implement a highly fault tolerant SMPP client and server prototypes and analyse the design. Additionally the prototypes should integrate well with the Mobile Arts' existing software. The purpose of this is to evaluate and test the project. A side-effect will be that the SMPP application is well integrated if it is to be used in future products.

5.1 Development methodology

Initially the project followed the Waterfall methodology [19]. The workflow pattern of the model suits the master thesis work well. There is a given specification and a defined task.

The Waterfall model is divided into five discrete phases starting with the requirement phase. In this phase the specification was studied together with relevant literature. It is followed by the design phase where the knowledge gained is used to create detailed requirements for the implementation phase. The implementation phase is when the design is carried over to code. Finally the Validation phase is used to validate that the requirements have been implemented correctly.

The phases are supposed to be discrete entities but in practice the work was intentionally driven to dissolve the boundaries and move away from the Waterfall model. Especially in the later phase a more iterative style of work was adapted. Implementation and validation was then done back-to-back to give more clear indications that the work was progressing in the right direction. Implementation tasks were isolated and managed through a spreadsheet system, the most important, blocking or fun task was chosen. New tasks were added if the need did arise. Others could be removed due to changes in priority or new understandings of the work at hand. Working in this manner meant that the development process became more agile.

5.2 Tools used

The tools used are simple in my opinion. Emacs was used for editing code together with the major mode that comes included in the Erlang/OTP distribution. For version control of the code base CVS was used. Later on for writing the report, Git was used to handle version control of .tex and text files. The report text was until the final stages written in the Markdown [13] markup language in order to focus on the writing. Even though LaTeXdoes this fairly well it still contains some distractions and overhead. The operating system has been OS X 10.7. For organising tasks and todo:s, an Internet based spread sheet software was used.

5.3 Implementation

After the initial requirement, study and design phases the implementation work began. The first application that was written was the SMPP client. To begin with the general application structure was created. As shown below.

```
ma_smpp_client/
              ebin/
              include/
              priv/
              src/
```

The ebin directory is for compiled beam files. The include directory is for header files. Priv is used for auxiliary files that incorporate into the product environment. Examples of these files are MIB instruction files and configuration files. In the src (source) directory the Erlang source code files are kept. The client application consists of approximately 30 different files.

When an initial prototype had been implemented, the first significant task was to set up a connection to a existing SMPP server. As this implies message sending of the bind-messages, a lot of groundwork already was done. After it had been accomplished the work focused on implementing the other messages and to begin to integrate the client. Here the waterfall model was left for the more agile approach. Implementation and redesign was done alongside as problems and issues were discovered. When the prototype had stabilised it was expanded for multi node systems. This introduced new problems and yet more iterations of re-design and implementation were done.

The SMPP server has the same file structure and layout. It too consists of approximately 30 source code files. The implementation pattern of redesign and coding was repeated when implementing the server.

Integration into the industrial environment was begun as soon as the prototype had reached a mature state. This was one of the goals for the thesis and the new SMPP client and server had been designed with this in mind. That is, there was a minimum of changes in function calls and in configuration files needed for the external applications making use of the protocol. The details of the integration will be further elaborated in chapter 6.

5.4 Discussion of the SMPP specification

The SMPP protocol specification is in my opinion easy to understand and to implement. The exception to this is the outbind operation. It was introduced to the public in SMPP version 3.4 in 1999. It is an operation that allows MC:s to establish a SMPP session by connecting to the ESME. Figure 5.1 shows the outbind traffic flow.

This could be used if the MC has outstanding messages to deliver to the ESME. It is initiated by the outbind PDU being sent from the MC to the ESME. If configured the ESME will reply with a bind_transceiver or bind_receiver message. Which in turn will be answered with the corresponding *-resp* to complete the session's initialisation.

The outbind PDU contains the system_id of the MC and a password. The specification is here ambiguous. If the ESME already has the MC's system_id in its configuration but with a conflicting password, it is not clear what should happen. An ESME can also have configured several MC's with the same system_id -how is the received outbind to be matched to the correct one. One solution to this would be to send the following bind-request on the same TCP connection. But if the same connection is to be reused or if a new TCP session is to be setup is unclear. The SMPP specification is very vague regarding the outbind operation. This opens up for different interpretations, which in turn introduces unreliability and instability to a SMPP ESME client application.

This could well be a bigger problem, but the outbind functionality is very seldom used according to expertise at Mobile Arts. They did not know of a single occasion that it had occurred in live telecom systems.

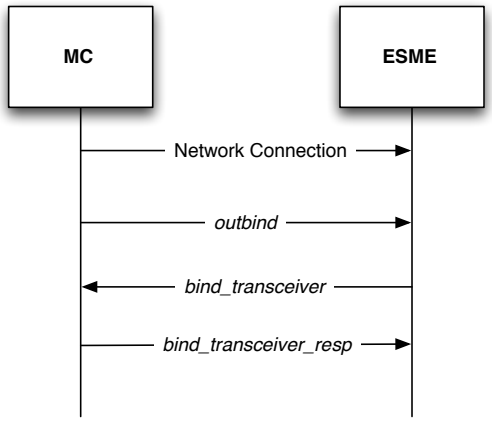

Figure 5.1: Outbind

Chapter 6

Evaluation

6.1 System integration

The purpose of the system integration testing is to make sure that the SMPP implementation has been carried out correctly. The integration with existing Mobile Arts products has gone well. The client and server fit well into the company's software products. The ESME client has successfully been integrated with the Missed Call Alert server (MCA). The MC server has been integrated with the SMSC server. All products mentioned above are developed and maintained at Mobile Arts. Figure 6.1 shows an overview of the MCA-SMSC setup.

The telecom kernel that constitutes the foundation of Mobile Arts' systems is fairly big and complex. Integration into this system was made early on to avoid problems later on. In order to do that well it was necessary to learn their building system, configuration system and also the company's internal rules and code conventions.

The MCA works as follows. When a phone call fails to be setup between two parties, the initiating message called Initial Address Message (IAM) is forwarded in the SS7 network to the MCA. This package contains information about who tried to call whom. The information is used to create a SMS that is sent to the target informing him or her of that they have missed a phone call.

Integration with the MCA and SMSC was done in a simulated environment. The MCA application part will use the ma_smpp_client to setup sessions, encode, send and receive SMPP messages. The SMSC will use the ma_smpp_server to handle authorisation and sending/receiving of SMPP messages. This has all worked to a very well.

6.2 Reliability testing

The purpose of the reliability testing is to ensure that the application meets the high demands for reliability and fault tolerance that is needed in the telecommunications industry. Except for the continuous testing with provoked faults

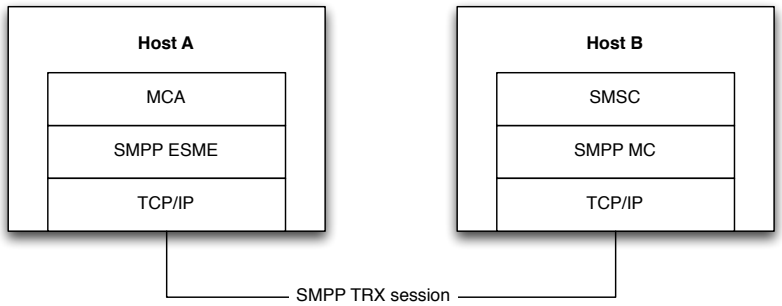

Figure 6.1: Session Example

during the development phase, a final set of tests was performed on the finished applications. The tests are divided into two categories, testing of internal errors and testing of external errors.

6.2.1 Test environment

The test environment was four virtual machines running in Virtualbox [6]. They were running Redhat 5.7 with Erlang/OTP 13B04 installed. On each of the machines a stripped down production system was installed. Two of the servers were configured to act as ESME clients (*esme-1* and *esme-2*) and two were configured to act as MC servers (*mc-1* and *mc-2*). The SMPP connections were configured for redundancy. So there is one connection from each of the clients to each of the servers. In total four connections. This is the basic redundant system configuration. The setup is shown in figure 6.2.

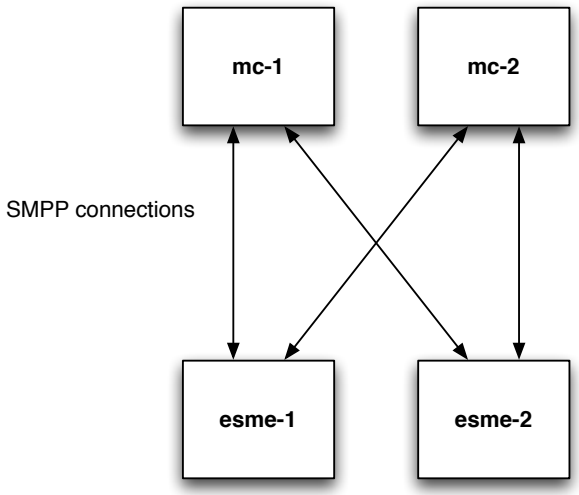

Figure 6.2: Fault tolerant test environment

6.2.2 Testing internal errors

Internal errors have been tested by sending incorrect and malformed data over the TCP connection. They have also been provoked by deliberately killing working process. These tests were performed on both the ESME clients and the MC servers.

Performing the test with sending in deliberately bad data gave the expected results. No crashes did occur. If the packet received was decodable to the extent that it was recognised as a SMPP PDU it was replied with a *generic_nack* with the status set to ESME_RINVCMDID (invalid command id) or if the command length was incorrect the status was set to ESME_RINVCMDLEN (invalid command length). These are the actions specified in the specification.

To intentionally crash processes is a way to simulate an unexpected error or crash in that part of the system. Each process has a test function that will throw an exception when called. This will cause the process to terminate immediately. Testing this on each process in the application had the desired and expected results. The supervisor would be informed of the dead child process and restart it. Crashed supervisors would also be restarted. Lost SMPP connections would as soon as possible be reestablished. This is according to the error handling in the SMPP specification.

6.2.3 Testing external errors

External tests have been performed to simulate external error situations that might occur to the system. Loss of network connection was performed by closing down the network connections using the program IPtables. IPtables was used to stop all incoming traffic and all outgoing traffic. The tests were run in a virtual environment where this method simulates a lost connection.

The other test was to simulate a major crash of the system. This was done using the program Kill along with the KILL flag (-9). This is a non-catchable, non-ignorable kill from the operating system. It is likened to turning of a computer with the power button. It is a very harsh way of stopping any program.

To stop a running Erlang system in production mode it is not enough to kill the virtual machine process. You must kill the heartbeat process monitoring the virtual machine process otherwise it will only be restarted again.

Performing the *kill -9* tests went as suspected. When a client or a server was killed, the SMPP traffic did still run normally from the redundant nodes. Upon restart the connections were quickly reestablished and the traffic continued to flow normally without any failures.

6.2.4 Multiple SMPP software

One of the requirements was that the SMPP software should be able to revert back to the old version during runtime. This is possible as the two implementations can exist independently of each other and have established connections. To change which versions is used the calling modules could be recompiled and

exchanged. The support for hot-code swapping allows this to be done with minimal disturbance.

6.3 Discussion of reliability and design

The tests with errors introduced internally and externally showed that the system continued to work properly and as expected. The system would deliver its service to the user, a SMPP connection able to send and receive traffic. The user does not experience a failure even though errors have occurred. The tests show that the desired fault tolerant and reliability characteristics are achieved,

Returning to the questions posed in chapter 2. How should faults be handled? What coding paradigms are used? How are fault handling and error handling applicable in Erlang? How does this affect the design?

Compare Erlang to the object oriented scripting language Ruby's error handling [14]. It shares this with many of the modern object oriented languages such as Java. Errors and the information about them should be packaged into an object called *exception*. The exception is propagated onto the call stack until the runtime system finds code that handles the error.

Erlang on the contrary allows errors, by letting the calling process crash to immediately restart it. There is a very big difference between these two approaches to error handling. The benefits of the latter are:

- Simpler code, the source code is not cluttered with error handling statements.
- Not all errors are foreseeable. In a complex software system it will never be possible to predict all potential pitfalls.
- Not hiding an error forces the developer to act notice the error and eventually make a conscious choice to act or not to act upon it. In my opinion the biggest danger with exceptions is that they can shadow the real cause of error.

Erlang also has exception support. But using the *let it crash* coding paradigm is such a powerful tool, that exceptions very seldom are used.

To avoid that the same errors occur again there is traceability in the system. It consists of the Erlang virtual machine's built in debugging information such as stack traces from crashed processes together with the Mobile Arts own log support.

Chapter 7

Conclusion

7.1 Summary

The purpose of this thesis was to investigate how a highly fault tolerant and reliable SMPP server and client could be written in Erlang. I have implemented such a client and server. In the process much thought was given on how to design reliable software. During testing errors where introduced into the systems trying to provoke failures. The tests show that the desired reliability attributes of the software are achieved.

The programming language Erlang proved itself to be very useful for designing and implementing a reliable software system. The support for distributed systems provides an easy access to the fundament for reliability, two computers in connection to each other running the same software. Good language and library support is not enough. The code paradigms and design philosophies are of good help. But even with all this, it always comes down to the developers and designers to use the tools correctly.

To summarise, the development of the SMPP client and the server in Erlang has gone well. The functionality is ready for industrial usage and the client application has already been integrated into Mobile Art's telecommunication systems in commercial use. Further, the software was delivered on time.

7.2 Problems encountered and experience gained

To design a reliable application is difficult. Much time is spent figuring out how logic and process layout should be constructed to provide the correct behaviour. There were many iterations of design ideas as well as prototyping and discussions with my supervisor and other persons familiar with the problem domain before the final design was reached.

To understand and implement the SMPP protocol was a simpler task when compared to the design task. The specification is well structured and in my opinion easy to understand. However, sometimes it lacks in detail, as with the

case of the outbind operation (discussed in chapter 5).

Another problem encountered is in the nature of Erlang and the parallel events that occurs when having multiple SMPP sessions. When a crash or undesired event occurred it could be difficult to debug. Code that worked well for a single process could behave differently running in a multi session mode. This could either be caused by bugs in the code or by race conditions, which are not necessarily bugs, but rather an issue over limited resources. These could be very hard to reproduce and to find. Further more the logs and other debug tools got cluttered with many parallel processes writing to them.

7.3 Suggestions for future work

When a software application has met its initial requirements there will always be room for improvement. These are my thoughts for improving the reliability and fault tolerance of the SMPP client and server.

7.3.1 Analyse with statistical models

The SMPP application has fundamental support for gathering statistics of errors and performance. It would from this be possible to conduct a statistical analysis of a more traditional software reliability engineering approach. The results of such a study would be interesting. Especially if the SMPP application is to be used in live systems with high amounts of traffic.

7.3.2 Comparison to other implementations

There was an initial plan to compare my implementation with another SMPP implementation. Due to lack of time towards the end of the project it was decided to be removed.

A comparison to measure pure performance would be interesting. Especially if the two implementations were to be considered equivalent in functionality, but implemented in different languages. That would be to compare Erlang to the other language. To test and compare the reliability of the implementations would be worthwhile of an investigation, though that would be hard to quantify.

7.3.3 Missing functionality

Not all parts of the SMPP protocol have been implemented due to lack of time. Notable is outbind support in the MC server. It is at the time of writing not implemented. However this functionality exists in the ESME client.

References

[1] Aldicson. http://en.wikipedia.org/wiki/Aldiscon. Accessed: 2013-04-16.

[2] Erlang.org. http://www.erlang.org. Accessed: 2013-04-22.

[3] Management information base. http://en.wikipedia.org/wiki/Management_information_base. Accessed: 2013-04-23.

[4] Oserl. http://sourceforge.net/projects/oserl/. Accessed: 2013-05-06.

[5] Osi-model. http://en.wikipedia.org/wiki/OSI_model. Accessed: 2013-04-23.

[6] Virtual box homepage. http://https://www.virtualbox.org. Accessed: 2013-05-11.

[7] Standard glossary of software engineering terminology. ANSI/IEEE, 1991. STD-729-1991.

[8] Joe Armstrong. *Making Reliable distributed systems in the precense of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[9] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[10] Francesco Cesarini and Simon Thompson. *Erlang programming*. O'Reilly Media, 2009.

[11] Armstrong et al. Erlang google group, variable naming conventions. https://groups.google.com/forum/?fromgroups#!forum/erlang-programming. Accessed: 2013-04-22.

[12] SMS Forum. Short message peer-to-peer protocol specification 5.0. http://docs.nimta.com/smppv50.pdf, 2003. Accessed: 2013-04-23.

[13] John Gruber. Markdown specification. http://daringfireball.net/projects/markdown. Accessed: 2013-04-22.

[14] Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2004.

[15] Mikael R. Lyu. *Handbook of Software Reliability Engineering.* IEEE Computer Society Press and McGraw-Hill Book Company, 1996.

[16] Hakan Mattsson, Hans Nilsson, and Claes Wikstrom. *Mnesia, A distributed robust DBMS for telecommunications applications.* Computer Science Laboratory, Ericsson Telecom AB, 1999.

[17] Richard Oehme. Reflektioner kring samhällsskydd och beredskap vid allvarliga it-incidenter. Technical report, Myndigheten för samhällsskydd och beredskap, 2012.

[18] Apoorva Singhal and Ankur Singhal. *A Systematic Review of SoftwareReliability Studies.* University School of Information Technology, Delhi, 2011.

[19] Ian Sommerville. *Software Engineering.* Addison Wesley, 7 edition, 2004.

[20] Clas Svahn. Dagens nyheter. `http://www.dn.se/nyheter/sverige/telias-mobilnat-utslaget`, 2012. Accessed: 2013-04-23.