Department of Computer Science

Peter Hjärtquist and Erik Möller

Robustness Testing of a CPP emulator

Computer Science

C-level thesis (15hp)

Date / Term:	08-06-03
Supervisor:	Donald F. Ross
Examiner:	Martin Blom
Serial Number:	C2008:05

Robustness Testing of a CPP emulator

Peter Hjärtquist and Erik Möller

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

Peter Hjärtquist

Erik Möller

Approved, 080603

Advisor: Donald F. Ross

Examiner: Martin Blom

Abstract

During the last few years, Ericsson has developed an emulator for the telephone system called CPP (Connectivity Packet Platform) with TietoEnator as a subcontractor. The emulator is called CPPemu and emulates the hardware used in the network nodes in CPP. This means that the same software that may be run on a node in a CPP network may be run on the emulated hardware in the emulator. TietoEnator would like to examine whether the emulator may be used for testing software instead of running tests using physical hardware. For this experiment, a particular event will be tested in the CPP emulator.

A fail-over procedure, which works in a physical CPP node, will be verified to work in the CPP emulator. A fail-over may be defined as

> *The failure and automatic replacement of part of a system such that the user does not notice the failure and is not affected by it. The part which has failed is replaced by a backup part.*

This experiment consisted of three majors steps, namely (i) configuring the emulator, (ii) creating a network by setting up a redundant network (one main link and one backup link) between two emulated nodes and finally (iii) testing the fail-over procedure. The fail-over was tested by generating and sending traffic through the network, triggering the fail-over by ejecting the board which is currently sending and receiving traffic and inspecting the log files to evaluate if the fail-over works as intended. The results of the experiment indicated that the system performed as expected.

Contents

List of Figures

List of Tables

1 Introduction

During the last few years, Ericsson have developed an emulator for the telephone system called CPP (Connectivity Packet Platform) with TietoEnator as a subcontractor. The emulator is called CPPemu and emulates the hardware used in the network nodes in CPP. This means that the same software that may be run on a node in a CPP network may be run on the emulated hardware in the emulator. TietoEnator would like to examine whether the emulator may be used for testing software instead of running tests using physical hardware. For this experiment, a particular event will be tested in the CPP emulator.

A fail-over procedure, which works in a physical CPP node, will be tested and verified in the CPP emulator. A fail-over may be defined as

> *The failure and automatic replacement of part of a system such that the user does not notice the failure and is not affected by it. The part which has failed is replaced by a backup part.*

This means that a failure of a component will be modelled and the automatic recovery procedure will be evaluated. In a hardware CPP node, there are fail-over capabilities for the important components, such as communication boards and links (see chapter 3). In this experiment, the fail-over of an active broken communication board will be modelled in the CPP emulator. The hardware CPP node consists of a rack containing a number of boards used for specific purposes. Some boards provide network connections while other boards provide administration access to the node. The board that will fail is a network connection providing board. The failure is modelled by ejecting the board from the rack. When the board fails, a backup board will continue the transfer instead of the failed board. Thus, the data will get to its destination despite the failure. The evaluation of whether the fail-over procedure works or not will be performed by comparing the amount of sent data versus received data.

To accomplish this task there are a number of subtasks which must be performed.

1. Set up a network between two nodes inside separate emulators. Boards 6 and 7 are both network connection boards and will be connected to each other as shown in Figure 1-1.

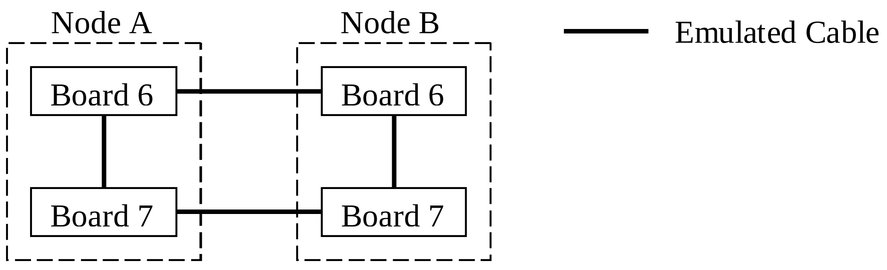


Figure 1-1 Emulated network configuration

2. Set up the logical network between the nodes.

   There are a number of software components, which are used to set up the logical part of the network. The main component is the signalling point (SP) which is used for routing of data. All SPs will be able to send data to adjacent SPs, and the SP1 and SP4 will be able to communicate via the other SPs.

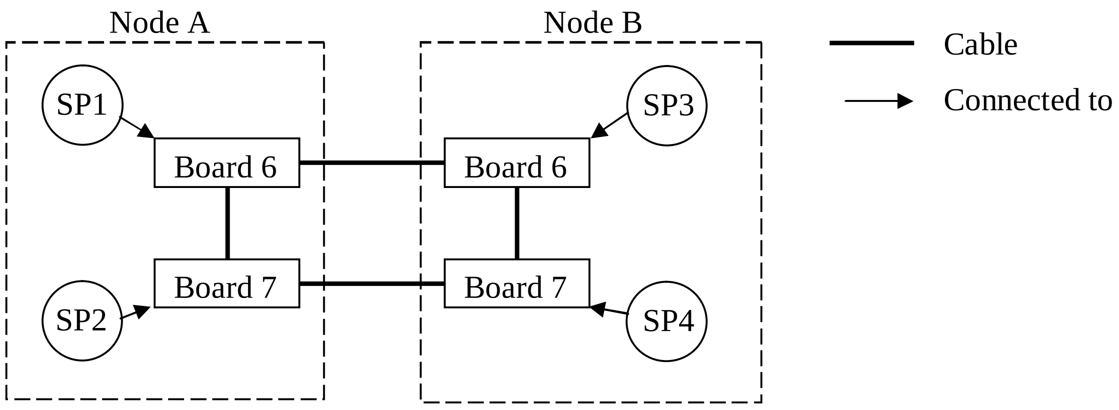


Figure 1-2 Basic CPP components

3. Send data between SP1 and SP4.

   Data will be sent in both directions between SP1 and SP4 via SP2. This path is chosen because the path via SP2 is configured with a higher priority than the path via SP3. The data is a simulation from data found in a real in-use system.

4. Model a failure

   The modelled failure is the ejection of board 7 on Node A. This will force the data sent between SP1 and SP4 via SP2 to be rerouted and will instead pass by SP3.

5. Evaluate the results

   Compare the amount of sent data versus the amount of received data.

The tools that will be used are:

- CPPemu [section 2.4.7], used to emulate the hardware
- LOCO [section 2.4.4], used to generate and send traffic over the network
- MO-Shell [section 2.4.8], used to manipulate the logical setup of a node
- CETP [section 2.4.3], a scripting tool used to automate the process

The experiment is divided into two subtasks.

1. Examine what happens when a link between two connected signalling points fails.
2. Automate the ability to eject and insert boards in the emulator.
   (only if there is sufficient time left in the project)

The dissertation is divided into five chapters:

- Chapter 1 is an introduction to the experiment.
- Chapter 2 describes the background and an introduction to the interfaces used when working with the experiment.
- Chapter 3 describes the approach used for the experiment.
- Chapter 4 presents the result of the experiment.
- Chapter 5 is a discussion and evaluation of the result of the experiment.

2 Background

2.1 Introduction

During the past years, Ericsson has developed an emulator for CPP (see section 2.4.6), with TietoEnator as a subcontractor, meant for testing the software used in CPP. TietoEnator would like to examine whether the emulator can be used for testing software instead of running tests using physical hardware. A positive outcome implies several advantages for the company:

- Maintenance of the software in the emulator leads to cost savings in comparison to continuously buying new hardware.
- There is no need to have physical access to the hardware to be able to test different soft- and hardware configurations.
- Access to the hardware is limited as oppose to the emulator, which is always available from every employee's workstation.

The emulator is an excellent tool for debugging:

- It is possible to take a snapshot of the current state when executing the emulator. It is easy to proceed with previous work and the same operations can be run an unlimited number of times, optionally using different parameters, to detect errors.
- During execution, there is an option to step forward one instruction at a time and simultaneously monitor the registers in every emulated processor.
- The emulator is very suitable for negative testing since it consists of software and no serious damage can occur. Negative testing means testing limits the application using unexpected input. For example, when a text string is keyed in when the application prompts for an integer.

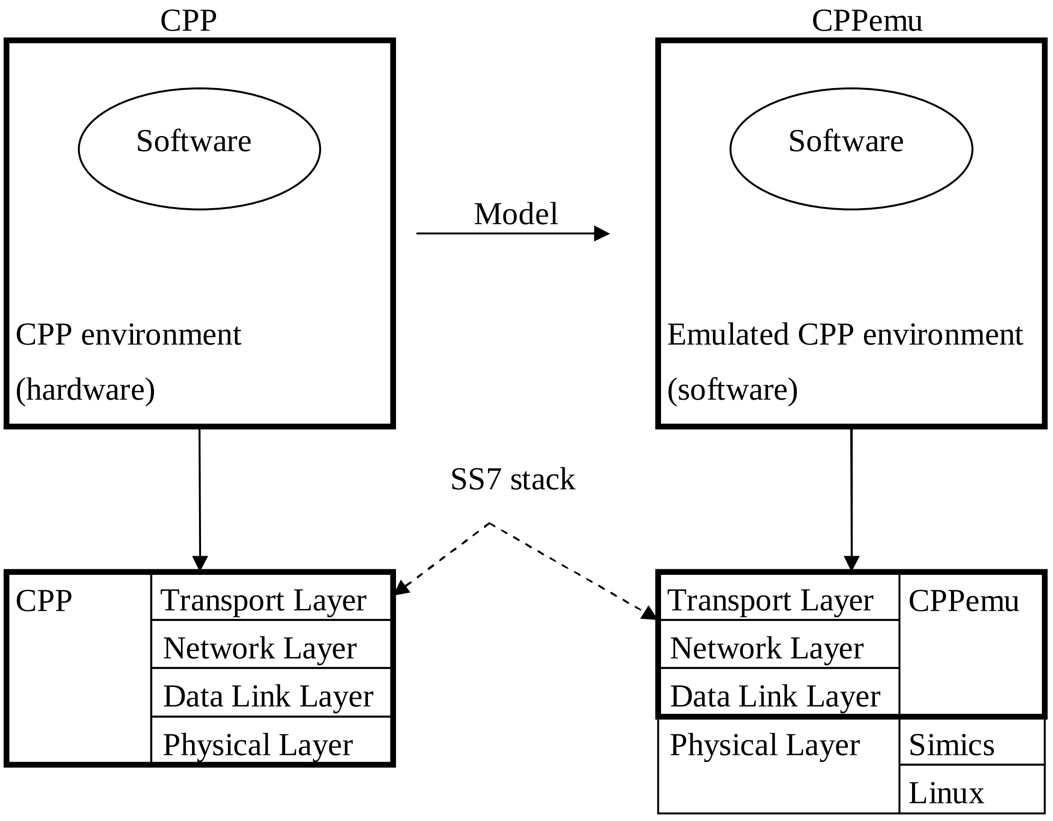


Figure 2-1 Mapping of a CPP node to an emulated CPP node

Each CPP node is built upon the SS7 protocol stack. As illustrated in Figure 2-1 the main difference between the emulated and the physical CPP node is that the emulated node does not handle the physical layer by itself. Simics and Linux handle the physical layer, meaning the physical wires and the transference of data from one host to another.

It is required to test the emulator properly to ensure that testing in the emulator works just as well as in the real CPP environment. If there is an unknown bug in the hardware system this implies that the implementation of the emulator does not include this bug and therefore not represent the real environment.

2.2 Purpose

The purpose of the experiment is to:

- Create a network between two emulated CPP (see section 2.4.6) nodes running on the same Linux host.
- Run traffic through the network.
- Examine whether the traffic finds a new path in the network when a link fails.

CPPemu is a representation of the hardware in a physical (hardware) CPP node. CPP is an abbreviation for Connectivity Packet Platform [section 2.4.6] which is a connection node in the telecom network. Several of these nodes are connected in a network used as a carrier for transporting data between mobile phones and service providers.
The data travels through physical links in the network between the nodes. On top of the hardware configuration there are software implemented signalling points (SP) that handle the traffic flow, as illustrated in Figure 2-2.

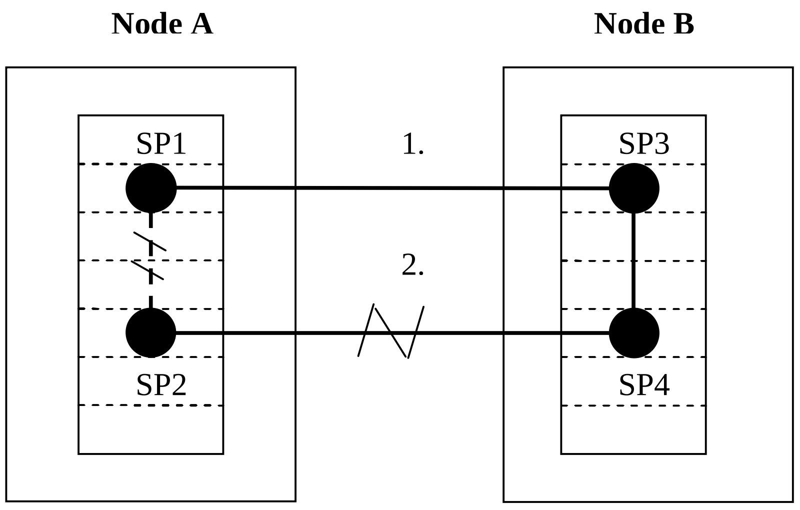


Figure 2-2 Finding a backup path when the active path fails (fail-over)

The scenario for the test of the fail-over is:

1. Send traffic from SP1 to SP4 via SP2 and examine what happens when all links connected to SP2 fails (Figure 2-2).
2. If the fail-over works properly, the traffic should find a new path from SP1 to SP4 through SP3.

2.3 Subprojects

2.3.1 Task 1

This task is to examine, using CPPemu [section 2.4.7], what happens when a link between two connected signalling points fails. This task is divided into further subtasks:

- Set up a redundant network (one main link and one backup link) between two nodes using two emulators.
- Construct test scripts in TCL [2] for the network.
- Generate and send traffic through the network using LOCO [11].
- Test the fail-over by ejecting a board which is currently sending or receiving traffic
- Inspect the log files to evaluate if the fail-over works as intended
- Write a user guide containing information about how to reproduce the experiment.

2.3.2 Task 2

The second task to be performed, if there is time left in the project after completion of the main task (Task 1), is:

- Automate the ability to eject and insert boards in the emulator, which would relieve future testing to run a test-script instead of unplugging a board manually.

2.4 Application Program Interfaces

2.4.1 Overview

The CPP emulator runs on the Simics (see section 2.4.5) a simulator which runs in a Linux host. CPPemu (see section 2.4.7) contains various types of boards, all of which uses OSE (see section 2.4.12) as operating system. In a physical (hardware) node, the board is a pluggable module with its own memory and processor. There are different types of boards which serve different purposes and provide connections such as broadband or IP. In the emulator, all boards are software models of physical (hardware) boards.
In OSE (see section 2.4.12), there is a software tool called LOCO (see section 2.4.4) for generating network traffic. OSE also provides an interface to the Mo-shell (see section 2.4.8), which is used to configure the logical part of the network.

The Mo-Shell runs on a Unix (see section 2.4.11) host and connects to the OSE inside the emulator using a telnet session. MO-Scripts are scripts used to automate MO-Shell commands. Mo-Scripts consist of Mo-Shell commands but with a different syntax, see section 3.5.3 for examples of the syntaxes. The Tcl [section 2.4.2] is a scripting language commonly used for test, that runs on a Unix host. On top of Tcl runs CETP [section 2.4.3] which is a software framework used for testing. The CETP connects to the OSE using a telnet session when testing the CPP nodes.

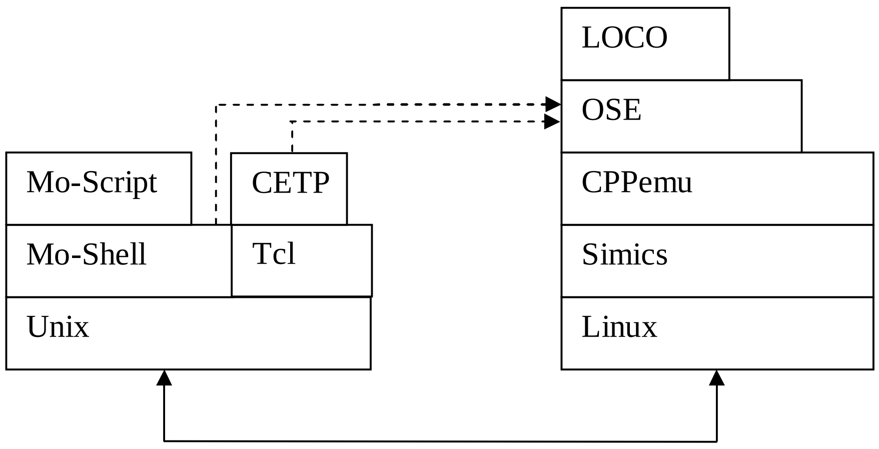


Figure 2-3 Overview of system interaction

2.4.2 Tcl

Tcl [2] is a scripting language often used for testing. It is easy to use but the performance is not optimal, since it is an interpreted language and therefore suffers from runtime overhead. For our purpose, optimal performance does not have a high priority. It is not a requirement to use Tcl for this experiment but it is practical as many software testers at TietoEnator use Tcl. This makes it easier to maintain or reuse the test scripts written for this experiment.

2.4.3 CETP

CETP, Common Expect Test Platform, is a test platform developed at Ericsson. It is based on the open source test suite called Expect [16]. It provides functions to connect to CPP nodes via both MO-Shell [section 2.4.8] and telnet. CETP makes it easy to run commands and by using the built-in output examination function in Expect it is also easy to verify that the command was run successfully.

2.4.4 LOCO

LOCO [11] is a network traffic generation tool, which is used for generating a specific load of network traffic when running tests. It includes such functionality as logging and statistics. LOCO is run from OSE.
LOCO is a tool developed internally at Ericsson with TietoEnator as subcontractor and was created in order to generate traffic for testing the CPP and is used in the CPP emulator as well. The tool supports a number of possibilities to make the generated traffic resemble the traffic found in in-use signalling networks:

- The data packet size.
- The data sending intensity.
- Number of data packets to generate.

2.4.5 Simics

Simics is a hardware simulator developed by Virtutech [1]. As well as all popular computer architectures, for example x86, amd64, Alpha, MIPS, EM64T, IA-64 and PowerPC, Simics can simulate any electronic system, such as routers or avionics [1]. One of the major uses for Simics is to be able to write and test software for specific hardware systems before the hardware is produced. This can be achieved since creating a model of the hardware does not need the actual hardware but only the specifications. Simics simulates the hardware in a physical CPP node.

2.4.6 CPP

CPP [10] is an abbreviation for Connectivity Packet Platform which is a connection node in the telecom network. The interconnection of such nodes creates a network used as a carrier for transporting data between mobile phones and service providers. A node consists of one rack which in turn has, depending on model, up to 24 or 28 pluggable boards connected. There are several types of boards which serve different purposes, for example the Exchange Termination Board which provides hardware connections. A physical CPP node implements all layers from transport layer to physical layer in the SS7 [14] stack. The CPP platform provides robustness in many aspects as with redundant switches, power, links between racks and signalling links [9].

2.4.7 CPPemu

CPPemu [10] is an abbreviation for Connectivity Packet Platform Emulator that is an emulator for CPP. It uses Simics as a simulator for the hardware and can simulate the hardware for an arbitrary number of nodes optionally connected to each other via broadband or IP (Internet Protocol) connections. Since there is no CPP hardware involved in the emulation, Simics simulates all connections that can exist between nodes in the same emulator or between nodes running on different hosts and emulators connected through the Simics environment.

Physical Layer:	MTP1	ATM	IP
Data Link Layer:	MTP2	SAAL	SCTP
Network Layer:	MTP3	MTP3B	M3UA
Transport Layer	SCCP		

Table 1 SS7 layers in CPP with respective protocols

The emulated CPP implements layers from transport layer to data link layer in the stack (shaded area in Table 1). A physical node also takes care of the physical layer with the transfer of binary digits in the network. When the emulator is running, it passes this task to the simulator that in turn uses a Linux host which transports the data to the destination.

2.4.8 Mo-Shell

The Mo-Shell [11] is a command line interface used for running commands and administrating the Managed Objects (MO) in OSE. MO-Shell was created and is maintained internally at Ericsson. The Mo-Shell uses remote connections directly to OSE on a physical CPP node or via forwarded ports in the emulator to a node inside CPPemu. The Mo-Shell uses both the http, ftp and telnet protocols when it connects to the node.

2.4.9 Mo-Script

The Mo-Script [12] is a scripting language used for creating scripts that is triggered from the Mo-Shell. It is used to automate commands instead of manually running every single command in the Mo-Shell. The Mo-Script is commonly used for managing logical networks. In this experiment, it will be used to create the logical configuration of the network.

2.4.10 Linux

Linux is an open source operating system kernel created in 1991 by Linus Torvalds [8]. Common Linux distributions use applications from the GNU Project [5] created by Richard Stallman in 1984. Most of these applications and the kernel are distributed under the GNU GPL [6] license which implies that everyone can download, modify and redistribute source code for non-commercial purposes. In this experiment, Linux was used on the servers running CPPemu.

2.4.11 Unix

Unix [7] is an operating system created in 1969 by AT&T. Since then, Unix has developed into several versions where some are free and some are not. The variant used in this experiment is called Solaris 9, which is developed and maintained by Sun Microsystems. The Solaris machines were used for running Mo-Shell since the Mo-Shell is currently only operational on Unix.

2.4.12 OSE

OSE [10] is an abbreviation for Operating System Embedded which is a realtime operating system created by the Swedish firm ENEA [3]. OSE is used in the boards in the CPP emulator. OSE provides interfaces towards the user in terms of a terminal for entering commands and an interface for the Mo-Shell [11].

2.4.13 ClearCase

ClearCase [12] is a revision control software tool which can handle both source code and binary files. ClearCase forms the basis of version control for many large and medium sized businesses and can handle projects with hundreds or thousands of developers [4]. In order to access files a view has to be set. This view defines which files and version of the files that will be shown to the user.
The use of ClearCase in this experiment is very limited, as it is only required in order to access the CPPemu binaries, Mo-Shell, test utilities and a few configuration files.

2.5 Existing Systems

This experiment did not require creation of new configurations of the network for the emulator, but rather modifying already existing configurations. There were already existing configuration files of the network that contained several more paths and signalling points than were required to perform the first task. Finally, the approach taken was to work with the configuration files and suit the parts required for the experiment.

2.6 Summary

TietoEnator wants to examine whether the CPP emulator can be used for testing software instead of running tests using physical hardware. A positive outcome implies many advantages for the company:

- Maintenance of the software in the emulator leads to cost savings in comparison to continuously buy new hardware.
- There is no need to have physical access to the hardware to be able to test different soft- and hardware-configurations.
- Access to hardware is limited in difference to the emulator, which is always available from every employee’s workstation.

The experiment is divided into two subtasks.

1. Examine what happens when a link between two connected signalling points fails.
2. Automate the ability to eject and insert boards in the emulator. (only in case of time left in the project)

The experiment involves several different tools of which the main tools are:

- LOCO, which is a network traffic generation tool, is used for generating a specific load of traffic when running tests.
- CPPemu is an emulator used to emulate a physical CPP node. The emulated CPP node involves layers from transport layer to data link layer in the stack.
- The Mo-Shell is a command line interface used for running commands and administrating the Managed Objects (MO) in OSE.

These tools will be used together to run traffic over a network for testing of fail-over. For further information, see chapter 3.

3 Experiment

3.1 Introduction

Chapter 3 contains a detailed description of the experiment for testing a fail-over. A fail-over may be defined as

> *The failure and automatic replacement of part of a system such that the user does not notice the failure and is not affected by it. The part which has failed is replaced by a backup part.*

In order to test a fail-over, two emulators have to be launched and connected to each other by means of (at least) two connections. The desired network has to be configured and loaded onto the emulators and each emulator may model an arbitrary number of nodes. The next step is to run traffic over the network and test a fail-over by ejecting a board, thus removing a connection, in one of the CPP emulators [section 3.3.9] to model a failure. The test is run from CETP (Common Expect Test Platform) [section 3.6.3], which uses TCL-scripts (Tool Command Language) [section 2.4.2] to perform the test. The CETP reads its configuration file and the TCL-scripts and then launches the test sequence.

CETP and the CPP emulators may be run on several different host machines, for example using Linux or Solaris machines. This chapter starts with instructions how to connect to a host machine and basic operations, such as how to start the emulator and add nodes to it. Further, more details about each phase are presented, for example, how to configure the CPP emulator [section 3.2], and how to set IP address [section 3.3.2] and port forwarding [section 3.3.3].

After the details for the CPP emulator configuration are described, descriptions and explanations on how to use the basic components in the network configuration are presented in section 3.4. Firstly, an abstract description of a signalling point and its corresponding components is presented. More detailed explanations about how to use MO-shell and MO-

scripts are presented in sections 3.5.2 and 3.5.3 respectively. The logical network configuration was defined using MO-Scripts.
The final part of the chapter describes how the tests were performed. First, generation of traffic with LOCO [section 3.6.2], and subsequently how the testing was automated with configuration files and TCL-scripts.

In the Figure 3-1, a simplified overview of the complete setup is shown.

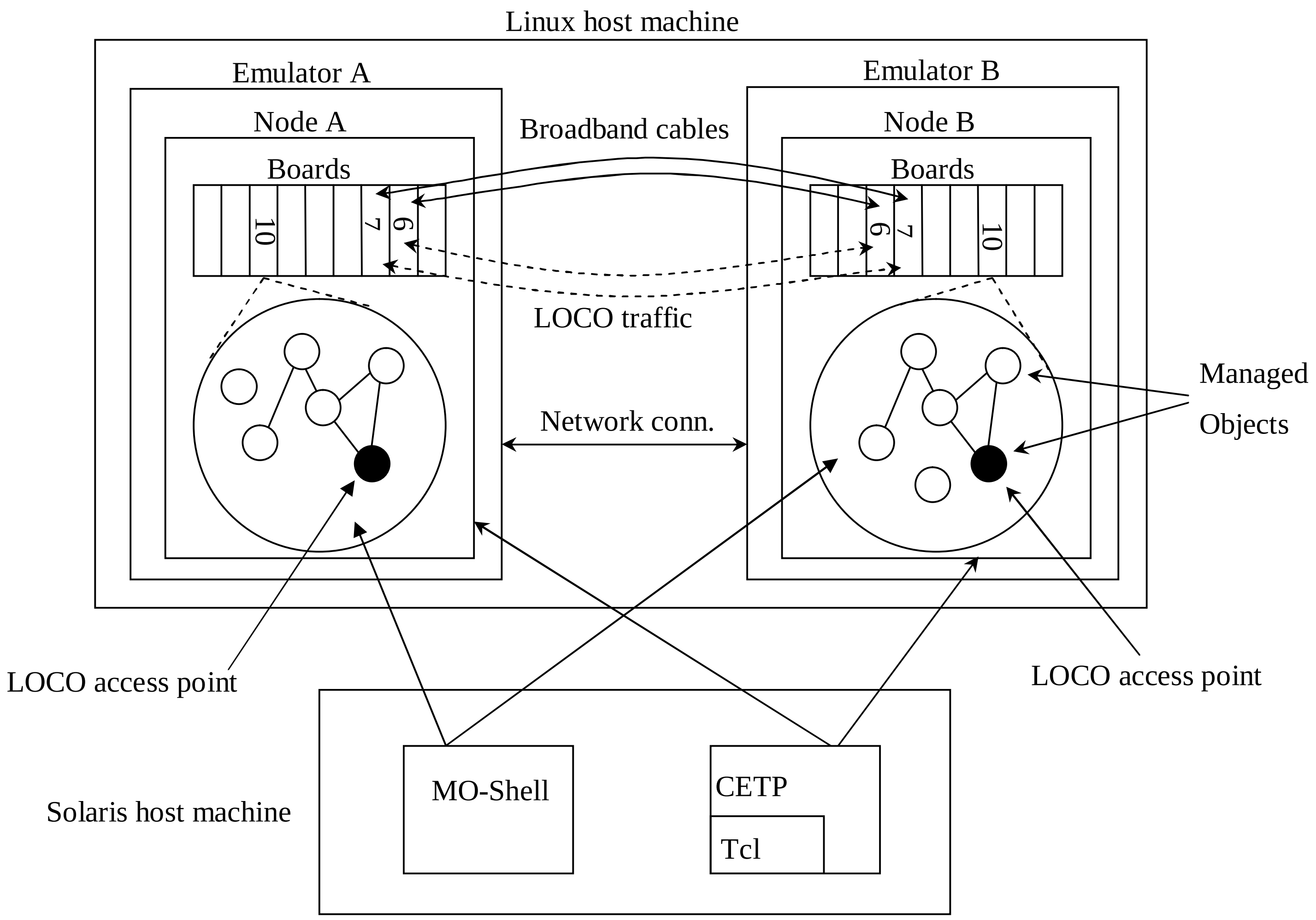


Figure 3-1 Relationship and connections between components and applications

In Figure 3-1, the following components and applications are shown:

- Host machines:

    - Linux:

      This is the host machine that runs both emulators
    - Solaris:

      This is another host machine used for running MO-Shell and the CETP scripts
- Emulators:

  These CPP emulators run the nodes. The emulators are built upon Simics.
- Nodes

  There are two nodes in this setup. They are named NodeA and NodeB respectively. Each is run from inside an emulator.
- Boards
    - 6 and 7

      These boards are broadband boards (ETM-4) which provide broadband connections.
    - 10

      This is the main processing board which controls the node.
- Broadband cables:

  These cables are emulated and used to connect the boards in the nodes. In addition to the cables shown in the figure, board 6 and 7 are connected within the node, but these cables have been omitted from the figure. The data transferred on these cables are transferred via the network connection between the emulators.
- LOCO traffic and LOCO access point:

  LOCO traffic (network traffic generated by a tool named LOCO) is sent between the nodes. The entry and exit points for the LOCO traffic are the LOCO access points. Although not shown in figure, LOCO traffic may also pass by or go between boards 6 and 7 within the same node.
- Network connection:

  The emulators are connected to each other via a network socket. This connection enables the nodes inside the emulators to communicate with each other.
- Managed Objects:

  Managed objects are software objects representing hardware or software objects on the node. All managed objects reside on board 10.

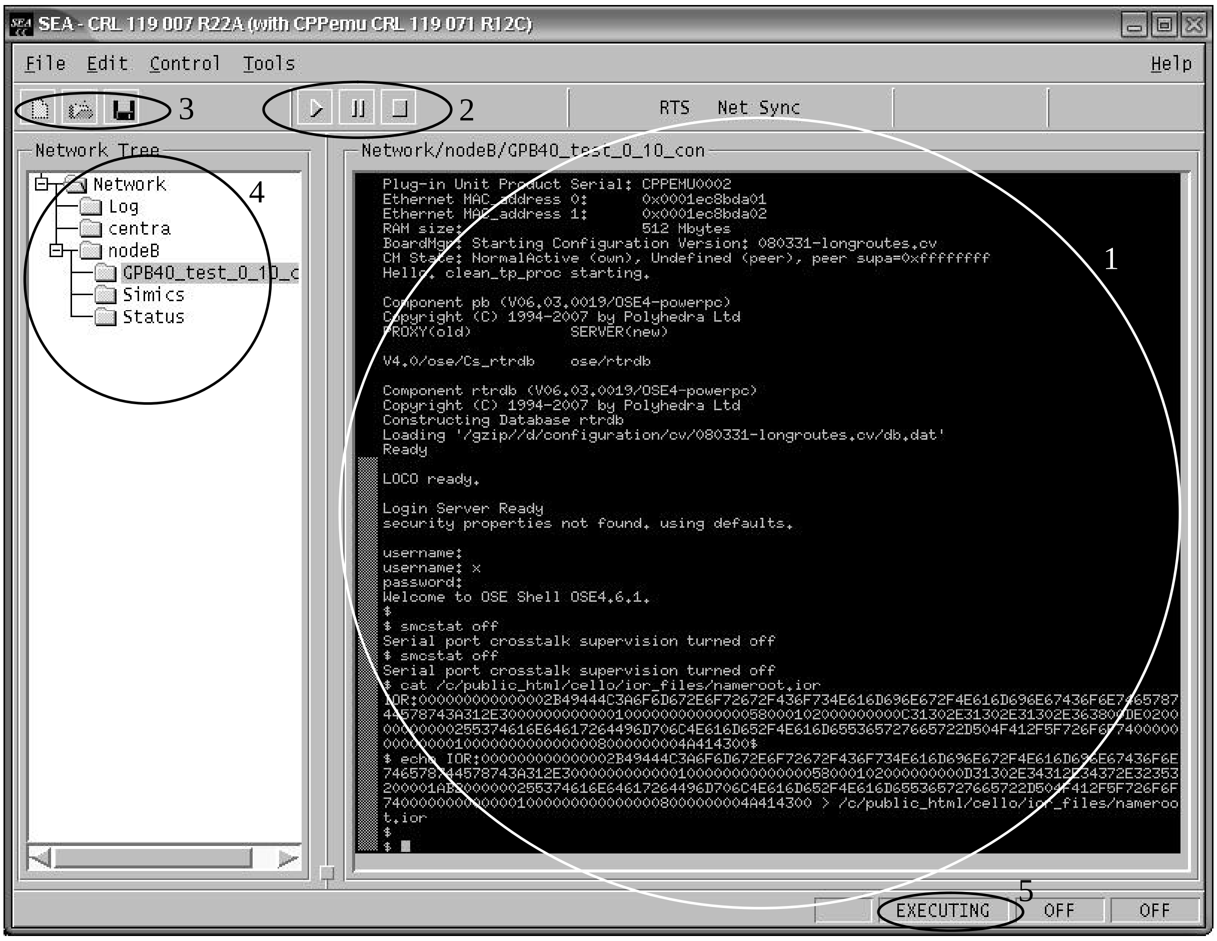
- MO-Shell:

  MO-Shell connects to board 10 (not shown in figure) and is used to add, modify or delete managed objects.
- CETP and TCL:

  CETP is a test platform written in TCL. It is used to run the test suites and test cases.

Each of these components will be explained in detail in this chapter.

Figure 3-2 shows the graphical interface of the CPP emulator. Figure 3-2 may be useful to refer to when reading the chapters 3.2 Configuration of the CPP emulator and 3.3 Details for the CPP emulator configuration.



```
Plug-in Unit Product Serial: CPPEMU0002
Ethernet MAC_address 0:      0x0001ec8bda01
Ethernet MAC_address 1:      0x0001ec8bda02
RAM size:                    512 Mbytes
BoardMgr: Starting Configuration Version: 080331-longroutes.cv
CM State: NormalActive (own), Undefined (peer), peer supa=0xffffffff
Hello, clean_tp_proc starting.

Component pb (V06.03.0019/OSE4-powerpc)
Copyright (C) 1994-2007 by Polyhedra Ltd
PROXY(old)            SERVER(new)

V4.0/ose/Cs_rtrdb    ose/rtrdb

Component rtrdb (V06.03.0019/OSE4-powerpc)
Copyright (C) 1994-2007 by Polyhedra Ltd
Constructing Database rtrdb
Loading '/gzip//d/configuration/cv/080331-longroutes.cv/db.dat'
Ready

LOCO ready.

Login Server Ready
security properties not found, using defaults.

username:
username: x
password:
Welcome to OSE Shell OSE4.6.1.
$
$ smcstat off
Serial port crosstalk supervision turned off
$ smcstat off
Serial port crosstalk supervision turned off
$ cat /c/public_html/cello/ior_files/nameroot.ior
IOR:000000000000002B49444C3A6F6D672E6F72672F436F734E616D696E672F4E616D696E67436F6E7465787
44578743A312E3000000000000001000000000000000580001020000000000C31302E31302E31302E363800DE0200
000000000255374616E64617264496D706C4E616D652F4E616D6553657272665722D504F412F5F726F6F7400000
0000000010000000000000000080000000004A414300$
$ echo IOR:000000000000002B49444C3A6F6D672E6F72672F436F734E616D696E672F4E616D696E67436F6E
7465787444578743A312E30000000000000001000000000000000580001020000000000D31302E34312E34372E32353
200001AB000000025537461 6E64617264496D706C4E616D652F4E616D6553657272665722D504F412F5F726F6F
7400000000000001000000000000000080000000004A414300 > /c/public_html/cello/ior_files/nameroo
t.ior
$
$ █
```

Figure 3-2 Graphical user interface for the CPP emulator

1. The OSE [section 3.3.12] console for board 10. This is used to control and run commands on the node. This same software is used on a physical (hardware) CPP node. The Simics [section 3.3.12] console will be visible in this window when it is used. The Simics console is used to control the environment for the node. For

example, connect a node to another node. The Simics prompt is only available when the emulator is paused.

2. Start, pause or terminate a node in the network.
3. New, open or save a configuration file.
4. The network tree for the node: It is possible to choose which console to use, Simics or the OSE for any board connected to the node. There is a choice to inspect the status for every board that is connected when enabling the status window [Figure 3-4] for the node. The IP-address of the host machine can be viewed from the network tree.
5. This field indicates the status of the CPP node emulation. The status can be either starting, executing, halted (paused) or terminated.

3.2 Configuration of the CPP emulator

3.2.1 Introduction

To perform the failover test certain steps are required in order to set up the environment. Two emulators will be used, both running on the same host machine. In each of the emulators, one CPP node will be emulated, i.e. two emulated CPP nodes will be used. The emulated CPP nodes are called NodeA and NodeB.
To set up the emulated nodes, the following steps are performed for each node:

1. Connect to a host machine which will run the emulator for the node
2. Starting an emulator
3. Add the emulated CPP node
4. Create an ETM-4 board in slot 7
5. Prepare for the MO-shell
6. Connect to the emulated CPP nodes using MO-Shell
7. Run the logical network configuration scripts (MO-Scripts)
8. Connect the emulators to each other

When these steps are completed the testing can begin, see section 3.6.
Each step will be shown and explained in chronological order.

3.2.2 Connecting to a host machine

Most work in this experiment is done on remotely located hosts. To connect to such a host, ssh is used. Ssh is run by:

ssh –X <host machine>

The first parameter, -X, is required for X11 forwarding which is used for showing the GUI[1] on the local machine from programs run on the remote machine. Upon connecting, ssh will prompt for password. Once connected, the correct view in ClearCase [section 2.4.13] has to be set to be able to run the emulator. This is done by running:

ctv cppemu

This will allow the user to reach the necessary files inside the ClearCase VOB [13] library.

3.2.3 Starting an emulator

There are dedicated host machines for running the CPP emulator and thereby the emulator has to be started from one of those. Logging in to the host machine is described in section 3.2.2. It is possible to run the emulator without the GUI, but this is not recommended for this experiment. Once connected and the correct ClearCase [section 2.4.13] view is set, CPPemu [section 2.4.7] is started by simply running:

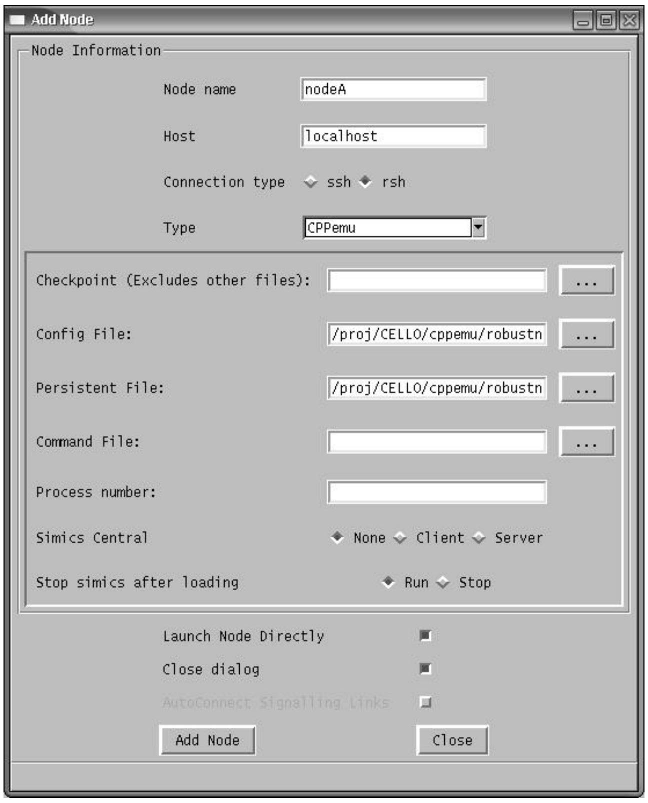
cppemu

Since two nodes are required for this experiment, two emulators are required. The second emulator is started in the same way as the first one, optionally using two separate ssh connections.

3.2.4 Adding the nodes

When the emulators have started, the next step is to add nodes to them. This is achieved by clicking Edit and then Add Node in the menu bar. The window showed in Figure 3-3 will then pop up:

[1] Graphical User Interface



Add Node

Node Information

Node name: nodeA

Host: localhost

Connection type: ssh [✓ rsh]

Type: CPPemu

Checkpoint (Excludes other files):

Config File: /proj/CELLO/cppemu/robustn

Persistent File: /proj/CELLO/cppemu/robustn

Command File:

Process number:

Simics Central: [✓ None] Client Server

Stop simics after loading: [✓ Run] Stop

Launch Node Directly ✓

Close dialog ✓

AutoConnect Signalling Links

Figure 3-3 Adding a node in CPP emulator

In this window, the name of the node and its .cppemu and .persistent files are entered. The files used for node A are:

cppemu: cello517.cppemu

Persistent: cello517.persistent

For node B, the files are

cppemu: cello518.cppemu

Persistent: cello518.persistent

These files are created by cloning physical CPP nodes (named cello517 and cello518). Cloning nodes [10] is not part of this experiment.

3.2.5 Creating ETM4 board in slot 7

The cloned CPP nodes used for this experiment, cello517 and cello518, have only one ETM4 board. For this experiment, two boards of this kind are required, since one of the two connections between the nodes will be unplugged, i.e. one board will be ejected, and if only one board were to be used, both connections would be lost when the board was ejected. Creation of the emulated hardware is performed in the status window in the emulator. Clicking the “Create Board” button will show a window where attributes such as board type, revision, position can be configured. As an ETM4 board was required, a board of this type was created. The software required for this board is configured in the MO-Shell. When the software is set up for the board, it is required to save a new version of the configuration for board 10 and restart board 10 to redistribute software to all boards, including the newly created board in slot 7.

3.2.6 Preparing for MO-shell

Advance preparations are required to be able to connect to a node with MO-Shell, see Port Forwarding (3.3.3). When connecting to a node, MO-shell uses Corba [12] that is a tool used to simplify the running of remote procedures. The use of Corba is required if there is more than one emulated CPP node on a single host. All nodes need to have unique IP addresses and different ports forwarded. Corba fetches an IOR file from the CPP node. The IOR file contains information about where to contact the Corba service on the CPP node. This file has to be rebuilt using a Simics command. The command uses information about the host machine as the IOR file contains information fetched from the emulated CPP node which only works if the host machine and the emulated node has the same IP address configured. Rebuilding this file is done by running:

```
cppemu-rebuild-nameroot-ior host-ip=<host machine IP>
                                        host-port=<corba port>
```

in the Simics prompt. The Corba port is the port forwarded [section 3.3.3] to the default port on the emulated CPP node.

If there is only one emulated CPP node on the host machine, the IP address on the node can be set to the same address as the host machine. If this is done it is possible to connect to the CPP node without Corba, by specifying corba=no, when running MO-Shell.

For this experiment, two emulators were run on the same host machine. This forces the IOR[section 3.5.2] file to be rebuilt. The commands run for node A are:

```
cppemu-connect-real-network-port 10.10.10.67 5000
cppemu-rebuild-nameroot-ior host-ip=10.41.47.252  host-port=5834
```

The commands run on node B are:

```
cppemu-connect-real-network-port 10.10.10.68 6000
cppemu-rebuild-nameroot-ior host-ip=10.41.47.252  host-port=6834
```

3.2.7 Connection with the MO-Shell

The MO-Shell requires a Unix host to run. One MO-Shell connection per emulator is required so two remote connections to a Unix host have to be made. These are performed as described in (3.2.2) to the host named seksux047. Once this is done, run:

c5alias

in order to set up an alias for the MO-Shell command. Connecting to the node inside the CPP emulator is done by, in different shells, running:

moshell –v http_port=5080,telnet_port=5023,ftp_port=5021 sekslx089

for node A and

moshell –v http_port=6080,telnet_port=6023,ftp_port=6021 sekslx089

for node B. When connecting to a CPP node with MO-Shell, the command

lt all

must be run [section 3.5.2]. It is now time to run the MO-Scripts that will create the signalling points required for the network configuration. For both nodes, there are a couple of scripts, written specifically for this project, to run, which for each node creates:

- Signalling points
- Routes to adjacent signalling points
- Routes to the non-adjacent signalling points
- The necessary software for the ETM4 board which is located in slot 7

The script that creates board 7 is the same for both nodes.

3.2.8 Connecting emulators together

The interconnection between the CPP nodes is done via the ETM4 boards and the cables connecting the boards. Connecting two ETM4 boards together is performed as described in Connecting Nodes Together Using Broadband [section 3.3.10] by on NodeA running:

cppemu-connect-et local=ETM4_test_0_6_phy2 remote=ETM4_test_0_7_phy2 host=10.41.47.252 port=<node A MPH>

cppemu-connect-et local=ETM4_test_0_6_phy1 remote=ETM4_test_0_6_phy1 host=10.41.47.252 port=<node B MPH>

cppemu-connect-et local=ETM4_test_0_7_phy1 remote=ETM4_test_0_7_phy1 host=10.41.47.252 port=<node B MPH>

and on nodeB running:

cppemu-connect-et local=ETM4_test_0_6_phy2 remote=ETM4_test_0_7_phy2 host=10.41.47.252 port=<node B MPH>

All four commands are run in the Simics prompt. These commands will connect, by using emulated broadband cables, board 6 with 7 internally on both nodes; connect board 6 on node A with board 6 on node B and the same procedure with board 7.

3.3 Details for the CPP emulator configuration

3.3.1 Introduction

This section includes more details about the different phases in the CPP emulator configuration.

3.3.2 Set IP-address

The main processor, board 10, has an IP address that is possible to change.
This can be done by using a built-in cppemu command in the Simics prompt:

cppemu-change-ip ip=<new IP-address> cv=<new CV>

This command will set the new IP-address, create a new CV [section 3.3.4], set the new CV to be loaded upon next start-up and finally restart the board.

3.3.3 Port Forwarding

When connecting to a physical CPP node using the MO-shell, it is possible to connect directly to the node using its IP-address and the standard ports for HTTP, telnet and FTP traffic. Connecting to a node inside an emulator is not quite as simple as this. As the emulated CPP node has no direct contact with the network, the emulator has to forward the required ports, i.e. redirect incoming and outgoing traffic, to the emulated CPP node. This is achieved by running the following command in the Simics prompt:

cppemu-connect-real-network-port <IP-address of the CPP node> <port base>

The specified IP address is the IP address of the emulated CPP node. As there are several ports required to be forwarded, instead of specifying each port, one only has to specify the port base and the emulator will forward all required ports using the port base argument as an offset for the standard ports. To reach the telnet server on the CPP node where the emulator uses 4000 as port base one would connect to the host using port 4023.

3.3.4 Configuration Version (CV)

All settings set using the command prompt in OSE [section 3.3.12] or the MO-shell [section 3.5.2] are stored in the memory of board 10. To prevent losing the settings upon next reboot a configuration version (CV) has to be saved. This is done by running:

cv mk <name of CV>

in the prompt of board 10. Now, a new CV is created, but it will not load when the board is restarted. To force this CV load upon next start-up, one has to run this command:

cv set <name of CV>

Now the new CV will load next time the node is started. To retain this CV after a complete restart of the emulator, the saving of a new persistent state is required. If this is not done, the new CV will be lost and the old CV will load upon next start-up.

3.3.5 Set Mac-addresses

All newly created boards with an Ethernet connection have the same mac-address (hardware address) on the Ethernet connection. To be able to use the Ethernet connection the mac-address has to be changed. This is easiest achieved by running

cppemu-set-macs

in the Simics prompt. This will set unique mac addresses for all Ethernet interfaces on the node.

3.3.6 Cppemu Files

The cppemu files are used to tell the emulator which boards are used, their position and whether they are connected or not. This file is required when creating a node. Cppemu files have ".cppemu" as file name suffix. As it is possible to create and delete boards in the emulator, it is also possible to save a cppemu file for the current configuration. This is done by saving a new persistent state. To create a completely new cppemu file, use the cppemu-configurator [10] or clone a physical node [10].

3.3.7 Persistent State Files

The persistent state files store the software and configuration, such as CV [section 3.3.4], for each board. For each saved persistent state there is one persistent state file per board and one index file which has the file name suffix of ".persistent".

When specifying the persistent state one specifies the index file. This file will include all persistent state files belonging to that state. Saving a new persistent state is easiest done by choosing “Save persistent state” in the menu in the emulator. Saving a persistent state will also create a cppemu file [section 3.3.6] which can be used with this persistent state.

3.3.8 Restarting the Node or a Board

Under certain circumstances, the node or board must be restarted.-There are a couple of ways of doing this. In order to restart a single board, either eject and insert the board (see 3.3.9) or run:

restartObj me cold

in the OSE prompt of the board. To restart all boards, i.e. the whole node, one can perform the same procedure, but for board 10 (see 3.3.11).

3.3.9 Insert/Eject Board

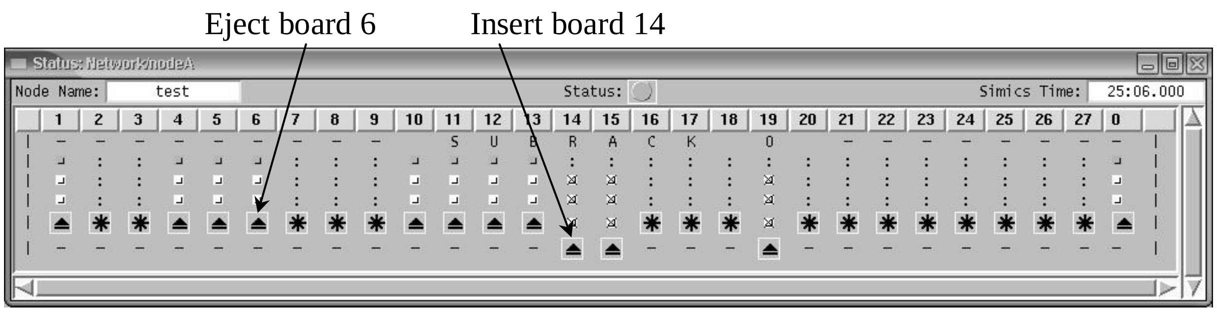


Figure 3-4 Eject or insert a board in the status window of CPPemu

In the emulator, it is possible to eject a board when it is in use, just as it is possible to do that on a physical node. This can be done by clicking the button for ejecting a board in the status window (see Figure 3-4) in the emulator or by running

cppemu-unplug-board <board>

in the Simics prompt. The insertion of a board is performed either by clicking the button for inserting a board or by running:

cppemu-insert-board <board>

in the Simics prompt.

3.3.10 Connecting Nodes Together Using Broadband

The ETM4 boards provide the broadband, ATM [14], connections. ATM is an acronym for Asynchronous Transfer Mode, which is a protocol in the SS7 stack. These boards have two physical connections and can therefore be connected to up to two other boards.
To connect two boards using an emulated broadband connection (ATM), the following command is run in the simics prompt:

```
cppemu-connect-et local=<ETM board>_phy<port>
                  remote=<ETM board>_phy<port>  host=<host IP>
                  port=<MPH port>
```

where the parameters named local and remote specify which ETM4 boards and which port on each board that will be connected. Host IP is the IP address of the host machine running the emulated CPP node to which the other end of the cable will be connected. The MPH (Message Protocol Handler) port is a network port on the remote host where Simics listens for connections of this type. The port number can be found in the Control Center by choosing the option with the same name as the node in the network tree [Figure 3-2]. The MPH port number is node specific and different for every node started on the same host, even when starting the nodes in the same emulator. The port number is different for the nodes upon every restart of the emulator.

3.3.11 Board 10

The board in slot 10 is an ordinary general-purpose board. This board is the main processor in the rack and controls the other boards. It is board 10 that defines what software will be available on the other boards and on startup of board 10, the software is broadcasted to the other boards. Ejecting and inserting this board will restart the whole node.

3.3.12 OSE

OSE [section 2.4.12] is the operating system running on the boards. As there are no displays on the boards one needs to connect either with a console cable directly to the board or with telnet or ssh via the IP network. In the emulator it is easy to connect directly to the console via the menu bar by clicking Tools, choosing node, clicking CPPemu Console and in the popup window choosing which board to connect. For this experiment, the OSE prompt will mainly be used for creating and setting new CVs [section 3.3.4] and controlling LOCO [section 3.6.2]. The emulator must be executing to be able to run OSE commands.

3.3.13 Simics

Simics [section 2.4.5]is the simulator upon which the emulator runs. Simics provides a prompt from which Simics commands may be run. The prompt is only available when the emulator is paused while OSE is available only when the emulator is executing. Some CPPemu specific Simics commands will start the emulator while they themselves are executing to be able to run commands in the OSE prompt on board 10. Pausing and starting the emulator is done by clicking the “pause” and “play” buttons in the user interface of the emulator.

3.4 Network configuration basics

3.4.1 Introduction

This section introduces the network configuration created in the MO-shell. It is difficult to understand the MO-scripts [section 3.5.3] without having read section 3.4. This section describes the basic components and how they interact with each other on a higher level of abstraction, to provide a better understanding when reading section 3.5 (Configuring network using MOs). Section 3.5 contains more details of the network configuration that is not covered in this section. The software configuration is the same for both a physical node and an emulated node.

3.4.2 Components

To configure the network there are some basic components that have to be included for the configuration to work in the CPP platform:

- First of all a signalling point has to be created. The signalling point is responsible for message routing between originating point and destination point.
- The signalling link is a logical link that connects two signalling points.
- The signalling link set is the number of parallel signalling links that are connected to the same signalling points.
- The signalling route is defined as a path assigned to carry traffic from an originating point to a destination point.
- The signalling route set contains all the signalling routes for message traversing from the originating signalling point to the destination signalling point.

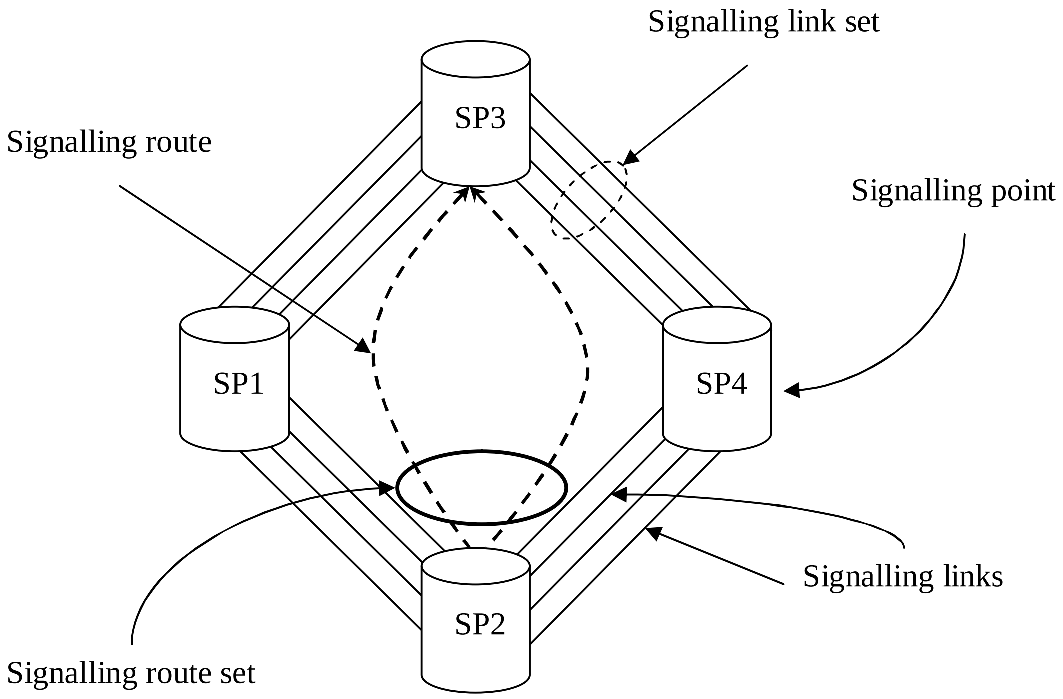


Figure 3-5 Components in the network configuration

The network configuration in Figure 3-5 consists of four signalling points that are connected to each other. Every pair of adjacent signalling points is connected with a link set containing four links. There is one signalling route set between SP2 and SP3 that contains two different routes. One route from SP2 via SP1 to SP3 and one route from SP2 via SP4 to SP3. Only one route set may exist between one starting SP and one destination SP, but there may be many different routes between them as in Figure 3-5.

In both the hardware CPP system and the CPPemu (the emulated system), one reason for having many logical links is that if a link set only contained one logical link and that link suffers from a software failure, such as problems with counters or timers, the physical link (both hardware and emulated) also fails, because there are no logical links to use as backup. Having several logical links makes it possible to redistribute the traffic to the other, still functional, logical links.

From now on in the report, the names of the signalling points 1, 2, 3 and 4 are replaced with the names that have been used for the actual configuration, namely 2-280, 2-290, 2-180 and 2-190 respectively.

3.4.3 Emulated Hardware Configuration

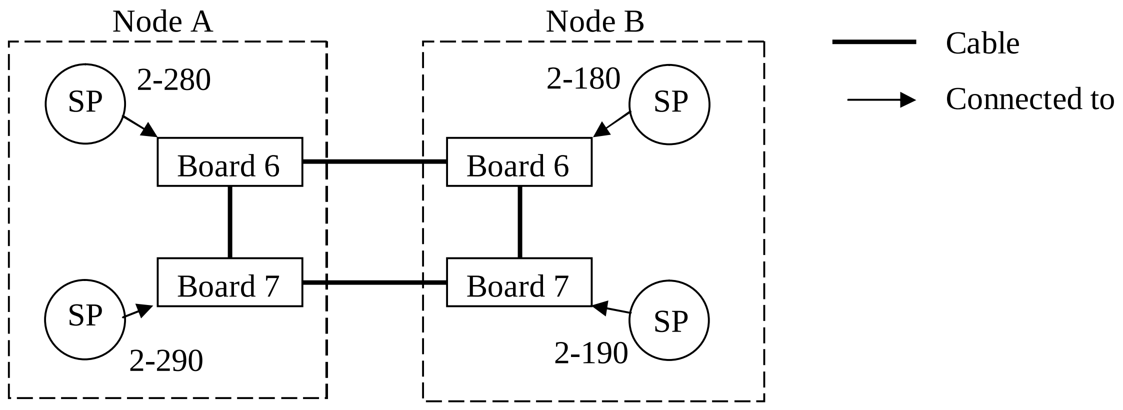


Figure 3-6 Basic CPP components

The hardware configuration setup is the same for the emulated nodes and for the physical CPP nodes. The difference is that the links and boards are software implemented in the emulator instead of physical cables and boards as in the physical CPP nodes. Figure 3-6 shows how the emulated links are connected in the configuration. The naming of the signalling points follows a standard that has been used at TietoEnator and it is necessary to be

conversant with these names (2-180, 2-190, 2-280, 2-290). There is a possibility that fail-over will be tested for other types of connections in the future. The prefix “2” in all the names of the signalling points indicates that the configuration is for a national network.
First, there are two ETM4 boards, board 6 and 7, which are connected to each other within the same node. Every ETM4 board may only have two physical links connected. There is a physical link between board 6 in node A and board 6 in node B. Finally, there is a physical link between board 7 in node A and board 7 in node B.

3.4.4 Software configuration

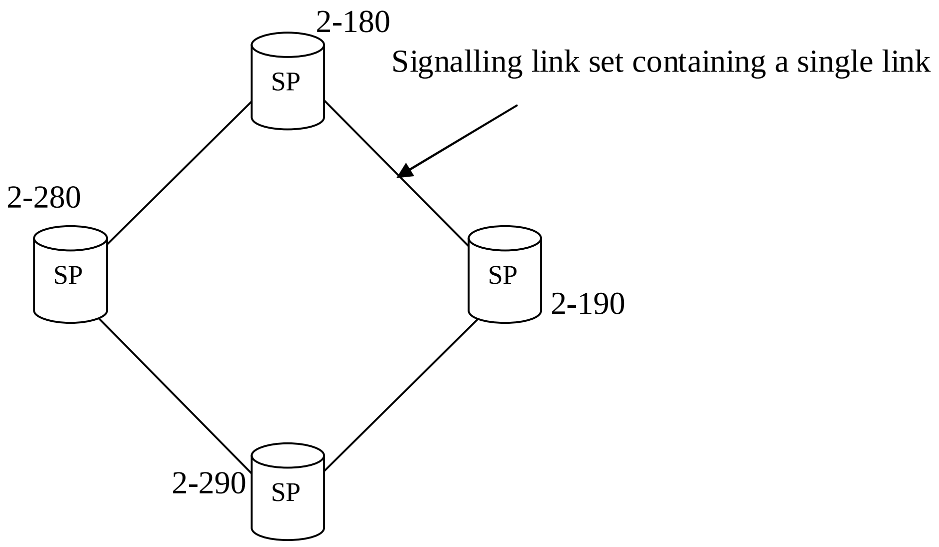


Figure 3-7 Logical configuration for the Signalling Points

The software configuration shown in Figure 3-7 contains only components which are the absolutely necessary for the creation of a functional network. The signalling points are connected to each other using a signalling route set, signalling route and a signalling link set which contains one single signalling link.
A signalling route set may only exist between one starting point and one destination point. There is, however, the possibility of having several different routes between the same pair of signalling points. The signalling link set can only exist between two adjacent signalling points. The link set may contain several different links but there is a maximum number of links that may be created within one link set. The number of links that may be created depends on which type of connection that is used.

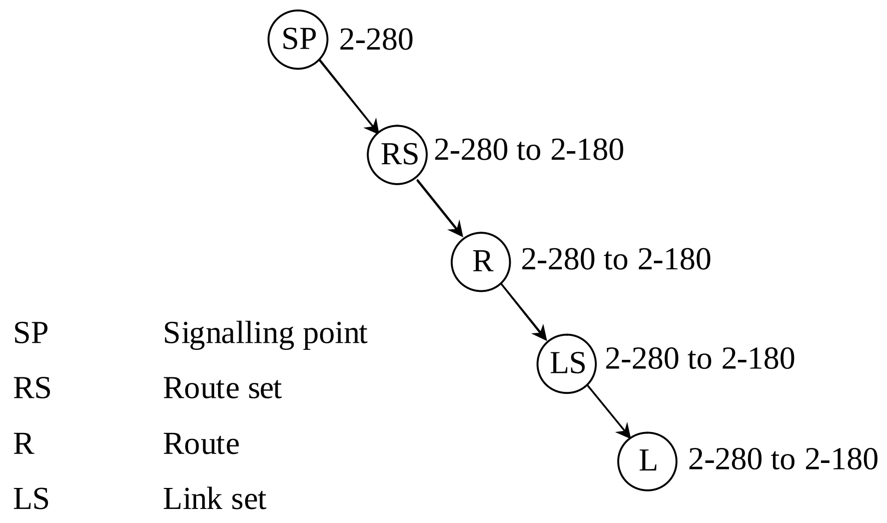


Figure 3-8 One-directional path from 2-280 to 2-180

The path between SP 2-280 and SP 2-180 is shown in Figure 3-8. Firstly, an SP 2-280 is connected to a route set to SP 2-180. The route set is connected to a route between SP 2-280 to SP 2-180. The route is connected to a link set and the link set is connected to a link. The link set could be connected to many links, but for testing of a fail-over it is not necessary. Therefore, the configuration is kept as simple as possible. This setup represents the configuration on Node A where the SP 2-280 exists. This configuration is only unidirectional, which means that SP 2-280 has a path to SP 2-180 but SP 2-180 does not have a path to SP 2-280. To make this configuration bidirectional the same procedure has to take place on Node B for SP 2-180. When the configuration is completed for SP 2-280, SP2-290, SP 2-180 and SP 2-190, it is then possible to send traffic between every adjacent pair of signalling points in the configured network. To be able to send traffic for a longer distance, additional configuration is required.

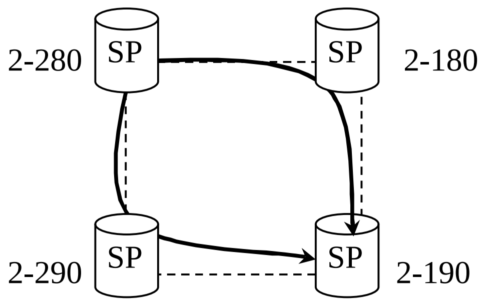
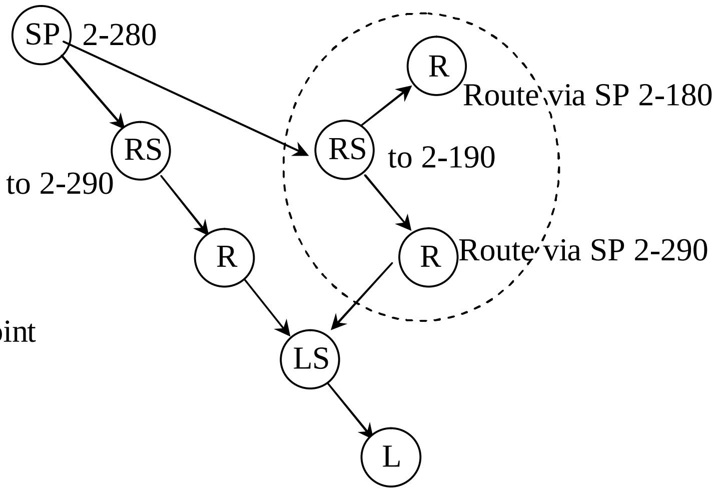


Figure 3-9 Route set for SP 2-280 to SP 2-190

For this experiment, a route set from SP 2-280 to SP 2-190 is configured. The route set contains two different routes, one route from SP 2-280 via SP 2-290 to SP 2-190 and another route that goes from SP 2-280 via SP 2-180 to SP 2-190.

When configuring the routes there is a priority attribute for each route. The route via SP 2-290 was set to priority one and the route via SP 2-180 was set to priority two. This means, when starting to send traffic from SP 2-280 to SP 2-190, the traffic travels via SP 2-290 because of the priority attribute of the route.

The configuration of the route set from SP 2-280 to 2-190 consists of three components to be added to the existing configuration. Two routes from SP 2-280 to SP 2-190 and a route set for the different routes.



SP	Signalling point
RS	Route set
R	Route
LS	Link set

Figure 3-10 Setup for a route via an SP

The Figure 3-10 shows how the components utilize each other in a configuration of a route from SP 2-280 via SP 2-290 to SP 2-190. Section 3.5 describes what the actual configuration looks like. The SP 2-280 has a route set and a route to SP 2-290. The route utilizes a link set with one link. The circumscribed objects in the figure are the only objects that have been added to the existing graph from Figure 3-8, a route set, and two routes from SP 2-280 to SP 2-190. One route goes via SP 2-290 and the other route goes via SP 2-180. The route via SP 2-290 is connected to the same link set as the route from SP 2-280 to SP 2-290 and therefore specifies the traffic sent from SP 2-280 to SP 2-190 will travel via SP 2-290. When traffic is sent from SP 2-280 to SP 2-190, a route is chosen either via SP 2-290 or via SP 2-180. The link set referred to by the chosen route specifies which path to choose.

The same procedure will be applied to configure the other route from SP 2-280 via SP 2-180 to SP 2-190. When the routes are created, a priority attribute is set for each route. When the traffic is sent between SP 2-280 and SP 2-190 the traffic will choose the route with the highest priority, i.e. the lowest value of the priority attribute (see 3.5.3).

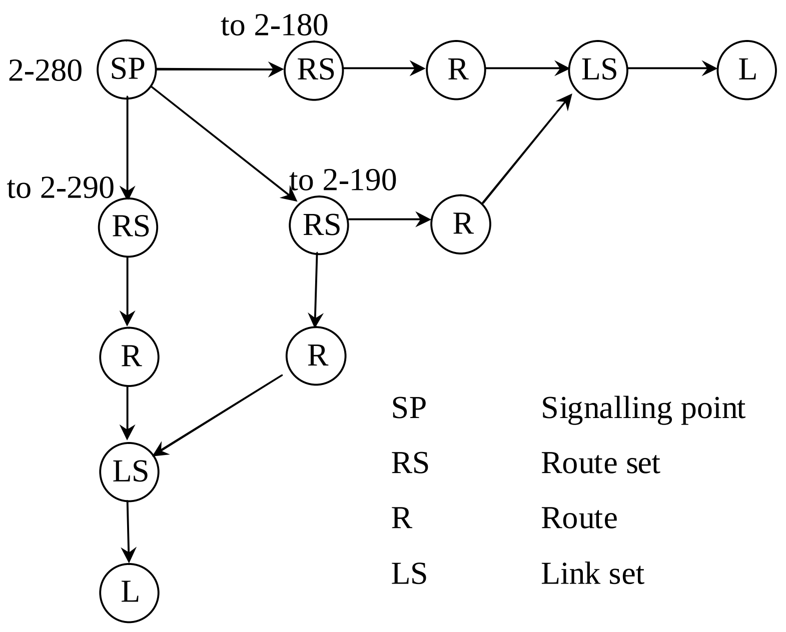


Figure 3-11 Network components at SP 2-280

The entire graph with the components for SP 2-280 is shown in Figure 3-11. SP 2-280 is connected with a route set to SP 2-180, SP 2-290 and SP 2-190. The difference with the route set SP 2-280 to SP-190 compared to the remaining route sets is that this route goes via SP 2-

180 or SP 2-290 depending on which link set the route with the highest priority in the route set 2-280 to 2-190 points to. This representation of the configuration for SP 2-280 is only unidirectional, which means that SP 2-280 is connected to SP 2-290, SP 2-180 and SP 2-190 but only from the view of SP 2-280. It is not possible to send traffic on a unidirectional connection.

Given that there are configurations for SP 2-290 and SP 2-180 to adjacent signalling points the same procedure has to be completed for SP 2-190 to make the representation of the configuration bi-directional and enable the route set from SP 2-280 to SP 2-190.

Configurations are required for every signalling point in the network in order to be able to send traffic in all directions. It is possible to create a configuration for just SP 2-180 to SP 2-280 but then it is not possible to send traffic between any other signalling points than SP 2-280 and SP 2-180.

The configurations for SP 2-290 and SP 2-180 contain only routes to adjacent signalling points. SP 2-290 and SP 2-180 have no routes to non-adjacent signalling points. That means that fail-over may only be tested when sending traffic between SP 2-280 and SP 2-190. To disable a route, a board has to be ejected which disables an SP [Figure 3-6]. If the traffic is sent between two adjacent signalling points, it is not possible to eject a board, because the sending or receiving SP then becomes disabled.

In this experiment, the traffic is sent from SP 2-280 via SP 2-290 to SP 2-190. If board 7 on the A Node [Figure 3-6] is ejected the route is disabled. The traffic has to find another route. The route set between SP 2-280 to SP 2-190 contains one more route, which is the one via SP 2-180. If the fail-over works properly, the traffic will choose the route via SP 2-180 when the route via SP 2-290 fails.

3.5 Configuring network using MOs

3.5.1 Introduction

MO is an acronym for Managed Object. MOs have a very broad scope of use and are the logical parts of a node. This includes the components in section 3.4.2 and the software that will be available on the boards. The network consists of several objects and there are often several thousands of MOs on a node.

3.5.2 The Mo-Shell

The MO-Shell is a tool used for creating, modifying or deleting MOs. It was developed internally at Ericsson. When connecting to a CPP node, physical or emulated, the MO-Shell uses FTP, Telnet, HTTP and optionally Corba [12] to communicate with the CPP node. Connecting to a physical CPP node is done by running:

moshell <IP or hostname of CPP node>

in a shell on a Unix host. When connecting to an emulated node one cannot use the standard ports used by the protocols because these ports require administrator rights on the host machine. This requires that some non-standard ports (e.g. FTP, Telnet, HTTP) be forwarded from the host machine to the standard ports on the emulated CPP node (see Port Forwarding 3.3.3) using a port base address.. Connecting to an emulated CPP node is done by running:

m*oshell –v http_port=<port>,ftp_port=<port>,telnet_port=<port>*
<IP or hostname of host machine> corba=<yes or no>

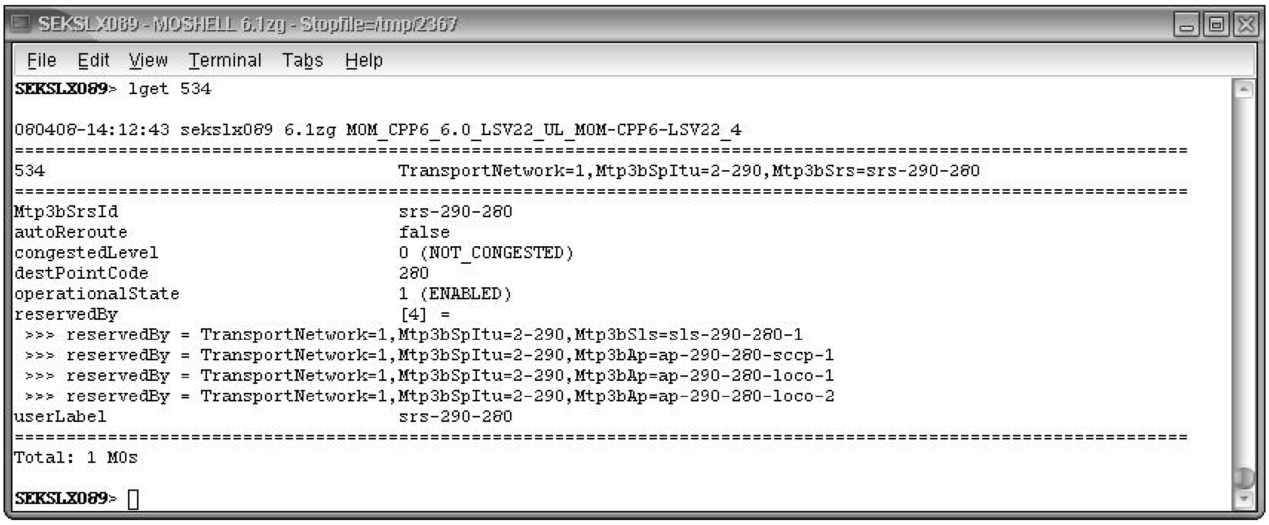
The specified ports are the ones forwarded to the standard port for each protocol. The IP address or hostname is the host machine running the emulator.

When the MO-Shell has connected to a node, it provides the user with a prompt where the MO-Shell commands are run. The first command to run is always “lt all” (Load type all). This command must be run upon every connection to build a table with all MOs and their proxy IDs. The proxy ID is a numeric identifier for an MO that is unique within the node. Most MOs are children to another MO. The LDN (Local Distinguished Name) is a textual string that contains the name of the MO and the name of all its parents as a comma-separated list. An MO can be identified either by its proxy ID or by its local distinguished name, LDN. An example of a LDN is shown in Figure 3-13. The LDN can be viewed as a comma-separated list of MOs that describes the branch of the MO tree to which the MO belongs. The leftmost MO is the top node in the tree and the MO to the right is a sub node (child) that in turn has its own sub nodes.

The most useful MO-Shell commands are:

Command	**Description**
lst <string or PID>	Lists all MO:s containing <string> in LDN or having a proxy ID of PID.
lget <string or PID>	Fetches more information, i.e. all attributes, about MO:s containing string or having a proxy ID of PID.
lset <string or PID> <attribute> <value>	Change the value of attribute <attribute> to <value> on MO having a proxy ID of PID or a LDN that matched the string.
cr <LDN>	Create an MO specified by the LDN. The command will prompt for values for the attributes.
del <PID>	Delete MO having a proxy ID of PID.
trun <MO-Script>	Run MO-Script.

Table 2 Useful MO-Shell commands


SEKSLX089 - MOSHELL 6.1zg - Stopfile=/tmp/2367
File Edit View Terminal Tabs Help

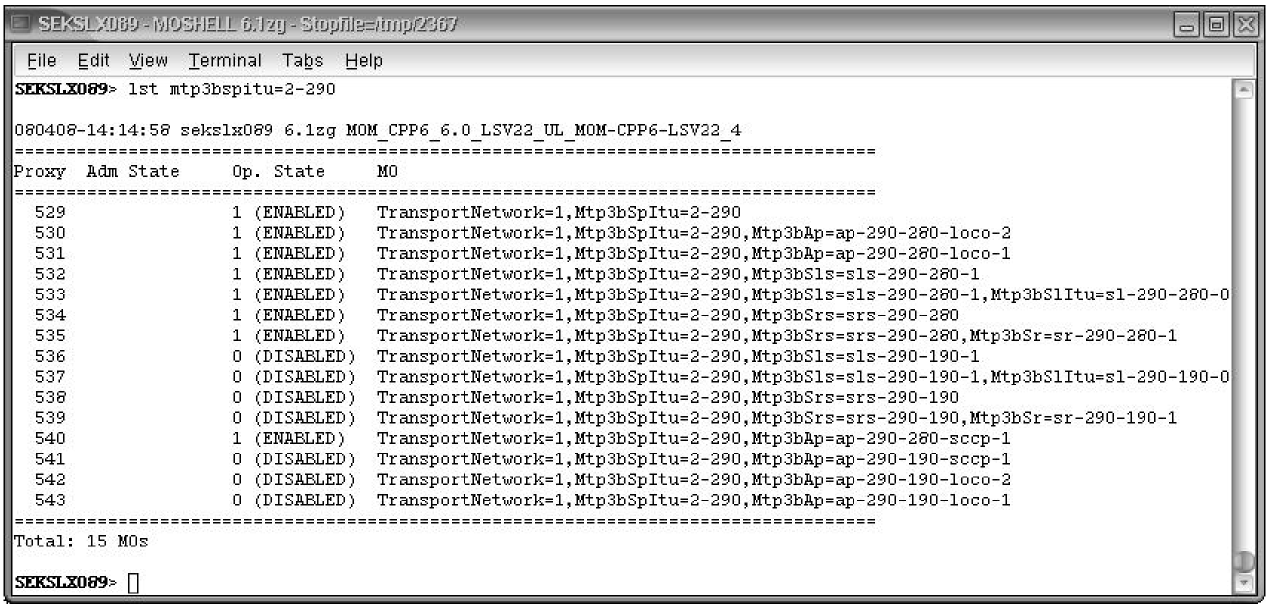
```
SEKSLX089> lget 534

080408-14:12:43 sekslx089 6.1zg MOM_CPP6_6.0_LSV22_UL_MOM-CPP6-LSV22_4
===================================================================================================
534                                     TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bSrs=srs-290-280
===================================================================================================
Mtp3bSrsId                              srs-290-280
autoReroute                             false
congestedLevel                          0 (NOT_CONGESTED)
destPointCode                           280
operationalState                        1 (ENABLED)
reservedBy                              [4] =
 >>> reservedBy = TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bSls=sls-290-280-1
 >>> reservedBy = TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bAp=ap-290-280-sccp-1
 >>> reservedBy = TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bAp=ap-290-280-loco-1
 >>> reservedBy = TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bAp=ap-290-280-loco-2
userLabel                               srs-290-280
===================================================================================================
Total: 1 MOs

SEKSLX089> 
```

Figure 3-12 MO-Shell showing information about a signalling route set

Figure 3-11 shows the output from the “lget” command for the signalling route set from SP 2-290 to SP 2-280. In this output, all attributes and the LDN of the children, three APs and one link set, of the specified MO are displayed. The attributes are different for each MOs, but one attribute is common to all MOs, namely “reservedBy” which tells the LDN of all MOs having this MO as parent. Two important attributes are the “destPointCode”, and “operationalState”, which tells the identity of the destination signalling point, and whether the MO is currently useable. In this case, whether it has a connection with the route set at the destination signalling point, i.e. a bi-directional connection, or not. The latter attribute cannot be set by the user but will be changed whenever a connection is established or lost. For a description of the other attributes, refer to the managed object model (MOM) [15].


SEKSLX089 - MOSHELL 6.1zg - Stopfile=/tmp/2367
File Edit View Terminal Tabs Help

```
SEKSLX089> lst mtp3bspitu=2-290

080408-14:14:58 sekslx089 6.1zg MOM_CPP6_6.0_LSV22_UL_MOM-CPP6-LSV22_4
===================================================================================
Proxy  Adm State      Op. State     MO
===================================================================================
  529                1 (ENABLED)   TransportNetwork=1,Mtp3bSpItu=2-290
  530                1 (ENABLED)   TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bAp=ap-290-280-loco-2
  531                1 (ENABLED)   TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bAp=ap-290-280-loco-1
  532                1 (ENABLED)   TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bSls=sls-290-280-1
  533                1 (ENABLED)   TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bSls=sls-290-280-1,Mtp3bSlItu=sl-290-280-0
  534                1 (ENABLED)   TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bSrs=srs-290-280
  535                1 (ENABLED)   TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bSrs=srs-290-280,Mtp3bSr=sr-290-280-1
  536                0 (DISABLED)  TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bSls=sls-290-190-1
  537                0 (DISABLED)  TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bSls=sls-290-190-1,Mtp3bSlItu=sl-290-190-0
  538                0 (DISABLED)  TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bSrs=srs-290-190
  539                0 (DISABLED)  TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bSrs=srs-290-190,Mtp3bSr=sr-290-190-1
  540                1 (ENABLED)   TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bAp=ap-290-280-sccp-1
  541                0 (DISABLED)  TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bAp=ap-290-190-sccp-1
  542                0 (DISABLED)  TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bAp=ap-290-190-loco-2
  543                0 (DISABLED)  TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bAp=ap-290-190-loco-1
===================================================================================
Total: 15 MOs

SEKSLX089> 
```

Figure 3-13 MO-Shell showing the MOs related to signalling point 2-290

Figure 3-13 shows the MOs created for SP 2-290. In the leftmost column, the Proxy ID is displayed and the next two columns show whether the MO is enabled or not. In this example, there is no connection between SP 2-290 and SP 2-190, hence the disabled MOs. The rightmost column shows the LDN for each MO, which in turn shows which MOs are children to other MOs, by having the closest parent as prefix in the LDN. For example, the signalling link (Proxy ID 533) is a child to the signalling link set (Proxy ID 532).

The MO with Proxy ID 533 contains the substring “TransportNetwork=1,Mtp3bSpItu=2-290,Mtp3bSls-290-280-1” in its LDN and is therefore a child to the MO that has this string as its entire LDN, this is the MO with Proxy ID 532.

3.5.3 The Mo-Script

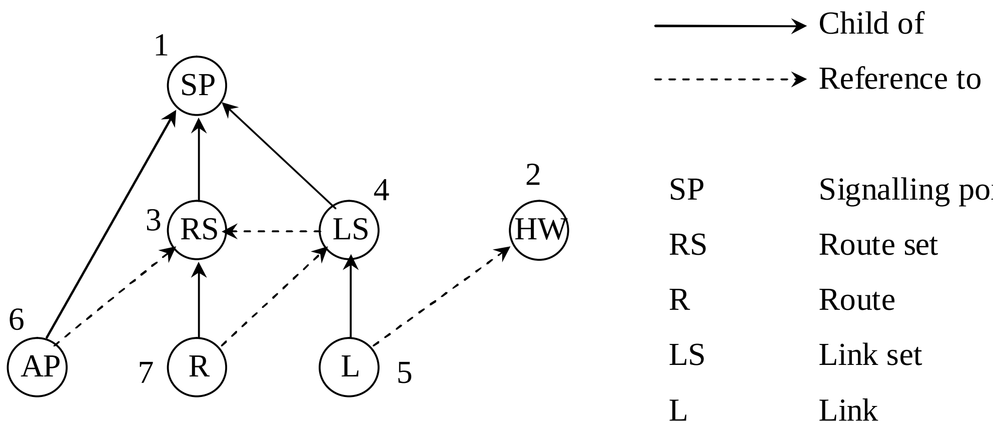
To automate the running of MO-Shell commands, an MO-Script is used. When writing an MO-Script, ordinary MO-Shell commands are not used but instead another syntax is used in MO-Script. For route sets, there is an attribute named ‘destPointCode’ that determines the destination SP for the route set. If the destination SP is to be changed to ‘190’ on the signalling route set with routes from SP 2-290 to SP 2-280 one would run:

```
lset TransportNetwork=1, Mtp3bSpItu=2-290,Mtp3bSrs=srs-290-280
                                        destPointCode 190
```

in the MO-Shell, and in the MO-Script, this would be:

```
SET
(
   Mo ManagedElement=1,TransportNetwork=1Mtp3bSpItu=2-
                                              290,Mtp3bSrs=srs-290-280”
    exception none
    destPointCode Integer 190
)
```

This example demonstrates the difference between an MO-Shell command and an MO-Script. The name of the route set, “Mtp3bSrs=srs-290-280”, implies that the route set starts at SP 2-290 and ends at SP 2-280. It is not recommended to run the commands in the example since the name of the route set will not be changed and therefore still imply that the route set ends at SP 2-280 while it actually ends at SP 2-190. The prefix, “2-“, only specifies that it is a national net and is therefore omitted in the destination point code. In this experiment, one MO-Script for each signalling point (including routes to adjacent SPs), one for each of the routes that goes to a non-adjacent SP and one script for creating the necessary software on board 7 have been created. This gives a total of seven different MO-Scripts. When creating MO-Scripts, the statements must be executed in a particular order. This is because each MO used in this experiment requires a parent. The parent of the signalling point is not included in the figures or scripts since it already exists and is of no significance for this experiment. The relationship for MOs on one node connecting two adjacent signalling points together is shown in Figure 3-14. This figure differs from the figures of the configuration shown in section 3.4 (Network configuration basics) because Figure 3-4 shows the actual configuration with dependencies and relations. Section 3.4 discusses the basic parts of the configuration, and how they interact instead of the implementation details for the actual configuration.



SP	Signalling point
RS	Route set
R	Route
LS	Link set
L	Link
HW	Hardware

Figure 3-14 Configuration of an adjacently connected SP

The order of the statements required to create an SP with MO-Script with connection to one adjacent SP is:

1. Signalling point (SP)

   The signalling point is the top MO and must be created first.

2. Hardware connecting MOs (HW)

   There are a few MOs used to enable access to the hardware connections on the boards. These MOs are VclTp, Aal5TpVccTp and NniSaalTp. A set of these MOs exists for every hardware connection on every board. A description of these MOs is not required for an understanding of this experiment and has therefore been omitted.

3. Signalling route set (RS)

   The signalling route set will be a child of the signalling point, but it will not refer to any other MO.

4. Signalling link set (LS)

   The signalling link set will be a child of the signalling point and will use a reference to the signalling route set which connects the same two adjacent signalling points as the signalling link set does.

5. Signalling link (L)

   The signalling link will be a child of the signalling link set and requires a reference to the hardware connecting MOs.

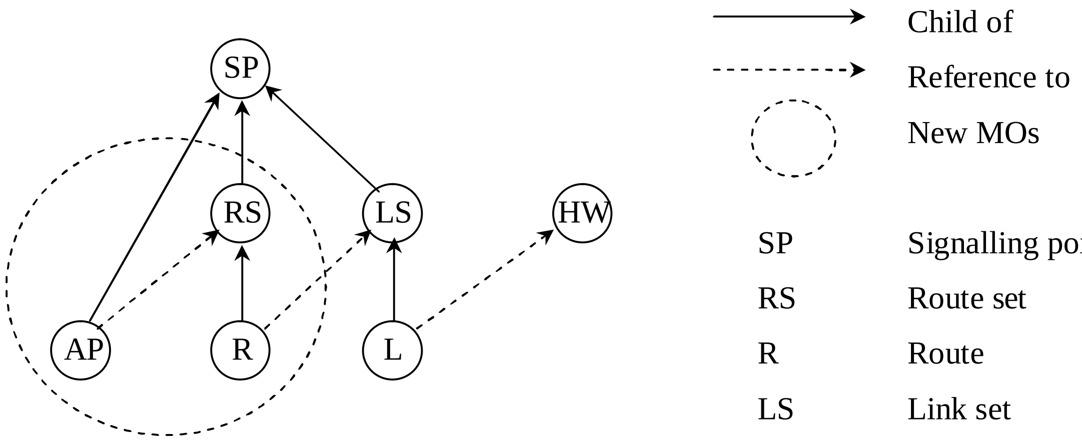
6. Access point for LOCO (AP)

The access point will be a child of the signalling point and requires a reference to the signalling route set.

7. Signalling route (R)

   The signalling route will be a child of the signalling route set and requires a reference to the signalling link set which it will use.

To add routes to a non-adjacent signalling point, the procedure is slightly different. In order to have the route work there have to be routes between every signalling point on the path to the destination signalling point.



SP	Signalling point
RS	Route set
R	Route
LS	Link set
L	Link
HW	Hardware

Figure 3-15 Configuration of a non-adjacently connected SP.

In order to add routes to a non-adjacent signalling point, a new set of MOs must be created (those circled). The order of creation for these MOs is the same as for the MOs created for adjacent signalling points. All references outside the circle refere to already existing MOs. The attribute “destPointCode” on the route set will be set to the non-adjacent destination signalling point. The route has an attribute named “linkSetM3uId” which is a reference and will be set to the link set that goes to the first signalling point in the desired path to the destination signalling point. This means that it is possible to define the beginning of the path for the route, but not the whole path. The link set will only have a reference to the route set that is configured between the same adjacent signalling points and so, the route set for non-adjacent signalling points will not be referred to by any link set.

In order to add a route to an adjacent signalling point, a new set of all MOs except for the signalling point must be created in the same way as with the first set of MOs.

The LOCO access point will be used by LOCO to send simulated traffic over the network, see section 3.6.2 (Network traffic simulation with LOCO) for more information.
When deleting MOs the order of the statements needs to be reversed, i.e. the MO created last is the one that must be deleted first. This is because references or parent-references that point to a non-existing MO are not permitted. If there is an error in the MO-script, e.g. trying to delete an MO with children or trying to create an MO with an already used name, the MO-Script will stop and the user has to clean up manually, using the MO-Shell or by modifying the script, in order to be able to rerun the script.

3.6 Running tests

3.6.1 Introduction

When the network configuration is completed, it is time to test the network with traffic. There is a tool called LOCO [section 3.6.2] which may be used for simulation of network traffic. The goal of this experiment is to test a fail-over for a broadband connection.

3.6.2 Network traffic simulation with LOCO

Sending traffic through a network is done with assistance from the access points. There are different types of access points to use depending on which kind of traffic that is to be sent. In this experiment, LOCO will be used to simulate traffic. This implies that the access point is of type LOCO. The access point is connected to the route set destined for the same signalling point as the access point [section 3.5.3]. The access point can only be connected to one route set and thereby only have one starting signalling point and one destination signalling point. In this experiment, there are access points at both SP 2-280 and SP 2-190. This is because traffic must be sent in both directions or LOCO will otherwise wait forever for incoming traffic.

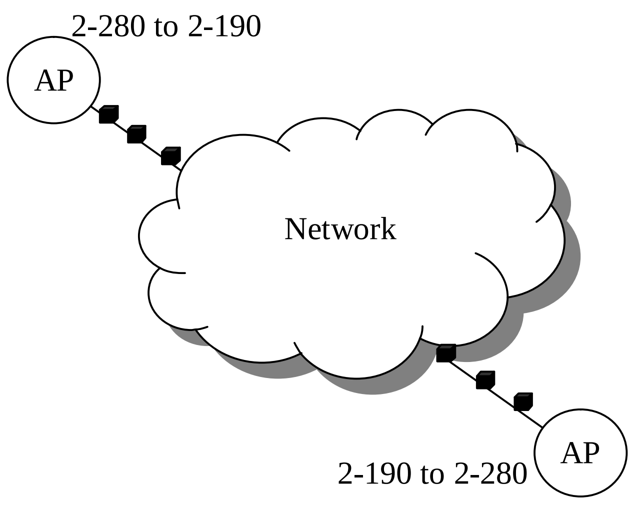


Figure 3-16 Sending traffic using LOCO

LOCO requires that traffic is sent in both directions, and for this experiment, two simulations will be run. The simulations are independent in respect of the settings, e.g. packet length, packet transfer intensity, number of packets to send etc., but the simulations must have the same ID to tell LOCO that they belong to each other. It is possible to have one access point set to loop the incoming traffic back to its origination, but this is only used for measuring the roundtrip time and this mode will not be used.
When starting LOCO there are a number of commands and options which require to be specified. Startmtp is the LOCO command used for running broadband traffic.

Startmtp argument	**Description**
Mp	Main Processor, defines which board that will do the processing
Froid	Facade Resource Object Identifier, ID of the originating access point. Unique within the node.
Pint	Packet intensity, defines how many packets per second that will be sent. Use –cont for an infinite number of packets.
Plen	Packet length, defines the length of each packet
Ni	Network indicator, defines the type of the network. National networks have 2 as indicator.
Simno	Simulation number, the identifier for the simulation

Table 3 Command arguments for Startmtp (in LOCO)

All LOCO commands are run in OSE, either directly using the prompt, or in a telnet session to board 10. Commands entered in the MO-Shell which MO-Shell does not recognize will be interpreted as OSE commands.
A typical LOCO launch would look like:

Loco startmtp -mp 001200 -froid <fro> -pint 6 -n 100 -plen 150 -pv 2 -ni 2 -simno 1

The fro ID is fetched from within the MO-shell, at the originating node, using the command:

fro <PID or string of originating access point>

which will show the fro ID for the specified MO.
Terminating running simulations is best performed by running

Loco stopallsim

When the simulation has finished or aborted, statistics about the simulation are printed on screen. The statistics include the number of packets sent and received; kilobytes of data sent and received and the number of packets received containing CRC errors and packet loss.

3.6.3 CETP (Common Expect Test Platform)

The CETP consists of scripts written in TCL. The CETP provides easy-to-use commands to connect to a CPP node and run commands on it, both in OSE, on board 10, and in an MO-Shell. The CETP is very suitable for testing applications where the test is based on output from the applications. This is because the CETP provides functions to specify what is and what is not expected as output. Usually, there are test suites (series of test cases) containing several test cases that test a component or a specific event. There is one test script for each test case. The test script defines what is to be performed in the test case. In this experiment, one test suite containing only one test case is used. The test script used in this experiment was provided and was suited to fit this project. The test case in this experiment requires that the CPP nodes are running and interconnected. The test script logs in to both CPP nodes, executes a LOCO command on each of them, i.e. initiates the transfers and checks for a confirmation from LOCO that the transfers have started. While sending traffic, the test script tells the user to eject the board to which SP 2-290 is connected. This will trigger the fail-over procedure. When all traffic is sent, the script evaluates the output from LOCO.
The CETP has a small configuration file that specifies for example where to find the CPP nodes and which ports to use. This file has to be specified when initiating the CETP. Further,

the CETP needs the test script, the name of the standardisation that will be used and the name of the test group.
Starting CETP is performed by, in a shell, running:

```
CETP `pwd`/test_suite.exp -config `pwd`/CETP_configuration.cfg
                                        -bb_ituitu_mtp3 -tg1.1.2
```

The first parameter is the test script and the second is the configuration file. The penultimate parameter is the standardisation, where bb, itu and mtp3 mean broadband, the European telephony standard and the use of mtp3 traffic respectively. The last parameter is the name of the test group.

3.6.4 Test case

The test was performed using three different test cases with different parameter setups. The parameters changed were number of packets to send, packet size and packet generation intensity. Traffic is sent from SP 2-280 via SP 2-290 to Sp 2-190 and in the reverse direction from SP 2-190 to SP 2-280, as shown in Figure 3-17. The test is to verify that the transfers continue after the failure of SP 2-290. Figure 3-17 shows the test that was run:

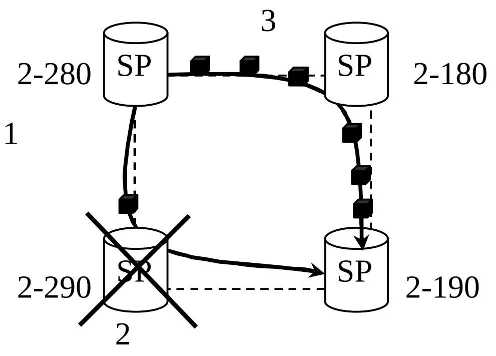


Figure 3-17 Rerouting when sending traffic with LOCO

The following scenario describes the test of a fail-over:

1. Traffic is sent from SP 2-280 to SP 2-190 via SP 2-290. SP 2-280 looks in its route set for a route to SP 2-190. The route set for SP 2-280 to SP 2-190 contains two routes; one route via SP 2-290 and another route via SP 2-180. The route via SP 2-290 has the highest priority and therefore becomes the chosen carrier for the traffic.

2. SP 2-280 and SP 2-290 exist in node A and SP 2-180 and SP 2-190 exist in node B [Figure 3-6]. Every SP exists on a different board and boards can be ejected manually from the CPP emulator [section 3.3.9]. When manually ejecting board 7 on Node A the connection between SP 2-280 and SP 2-290 is interrupted so the traffic has to find another route.

3. SP 2-280 looks in its route set again for a route to SP 2-190 and finds another route that goes via SP 2-180 and continues the sending of the traffic over the new route.

The expected result from this test is that the traffic finds the route via SP 2-180 when the route via SP 2-290 fails in the same way as it does on a physical (hardware) node.

LOCO requires that traffic be sent in both directions. If the transmission starts simultaneously at both sides, in principle there should be no problem to send packets in both directions. However, there were problems synchronising the end of the transfers. One transfer stopped before the other one, which then waited forever for incoming traffic and thus never finished. The solution was to set one side to send traffic continuously as this has no effect for the fail-over testing. The continuously ongoing transfer was terminated immediately after the fixed-size transfer finished in order to record the output.

3.7 Summary

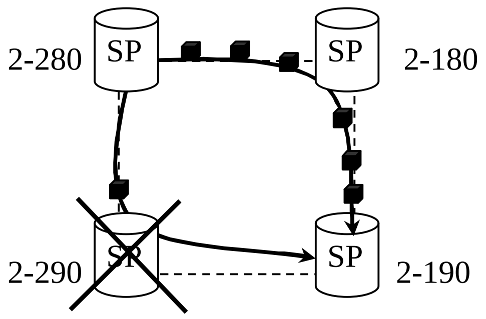


Figure 3-18 Fail-over rerouting

The main purpose of chapter 3 was to describe the approach of testing a fail-over for broadband traffic in the CPP emulator. A fail-over works well in a physical CPP node but has never been tested on an emulated CPP node. If a fail-over also works in an emulated node, more tests of this kind can be performed in future projects using an emulator instead of running tests on a physical (hardware) CPP node.

Preparations and configurations must be performed to test the fail-over. Firstly two emulated nodes must be connected to each other. In this case, two nodes running on the same host were used. When the nodes were configured and connected to each other the logical network had to be created using MO-scripts. Finally, LOCO was used to simulate traffic over the network. The fail-over was tested using test-scripts, written in TCL and executed by the CETP, and by manually ejecting board 7 in one of the emulators to disable SP 2-290.
Figure 3-18 describes the scenario for a fail-over. The network was configured as in Figure 3-18. Traffic is sent from SP 2-280 via SP 2-290 to SP 2-190. When ejecting board 7 in the CPP emulator SP 2-290 is disabled and the traffic has to find another route. The traffic should find the path from SP 2-280 via SP 2-180 to SP 2-190 if the fail-over works as it should.

4 Results and evaluation

4.1 Introduction

The results from the fail-over are presented using three graphs containing results from three different test cases. There were six measurements for each test case. In the results presented in this dissertation, the values for the x and y-axes are confidential and therefore not shown.

4.2 Results

The experiment was performed with six different measurements for three different test cases. The test cases used different parameters to test the fail-over procedure. When sending traffic with LOCO there are three parameters that may be varied. To be able to conclude if the fail-over worked or not, the effect of the different parameters on performance was studied.
The available parameters are the number of packets to send, packet size and packet intensity. For each test case, two of the parameters were set to a constant value and the third varied throughout the measurements.
The first test case [section 4.2.1] used a different number of packets to send, the second test case [section 4.2.2] used different packet sizes and the third test case [section 4.2.3] used different packet intensities. When the board is ejected, a number of packets are lost as shown in the graphs.
The results were determined using the output from LOCO [section 3.6.2]. The direct measurements comprise the total number of packets sent and the total number of packets received. The number of packets lost were calculated for node A by calculating the difference between the number of packets sent from node B and the number of packets received at node A.

4.2.1 Test 1 (Number of packets)

Test 1 was performed with six different measurements. The packet intensity and the size of the packets were set to constant values. The values for the packet size are represented with values in the interval from number of packets n to 6*n.

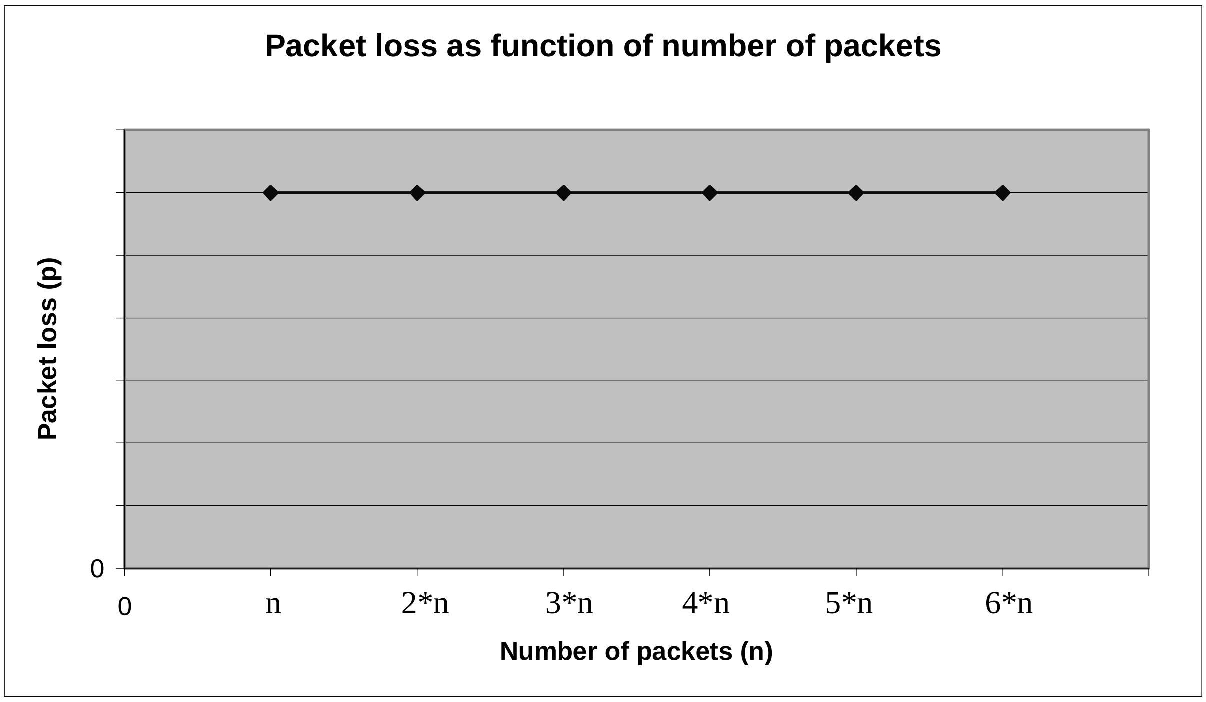


Figure 4-1 Packet loss as function of number of packets

Figure 4-1 shows that the packet loss is constant when the packet intensity and packet size have a fixed value and the total number of sent packets is varied.

4.2.2 Test 2 (Packet size)

Test 2 was performed with six different measurements. The packet intensity and the number of packets sent were set to constant values. The values for the packet size is represented with values in the interval from size (a non-zero value) to 36*size.

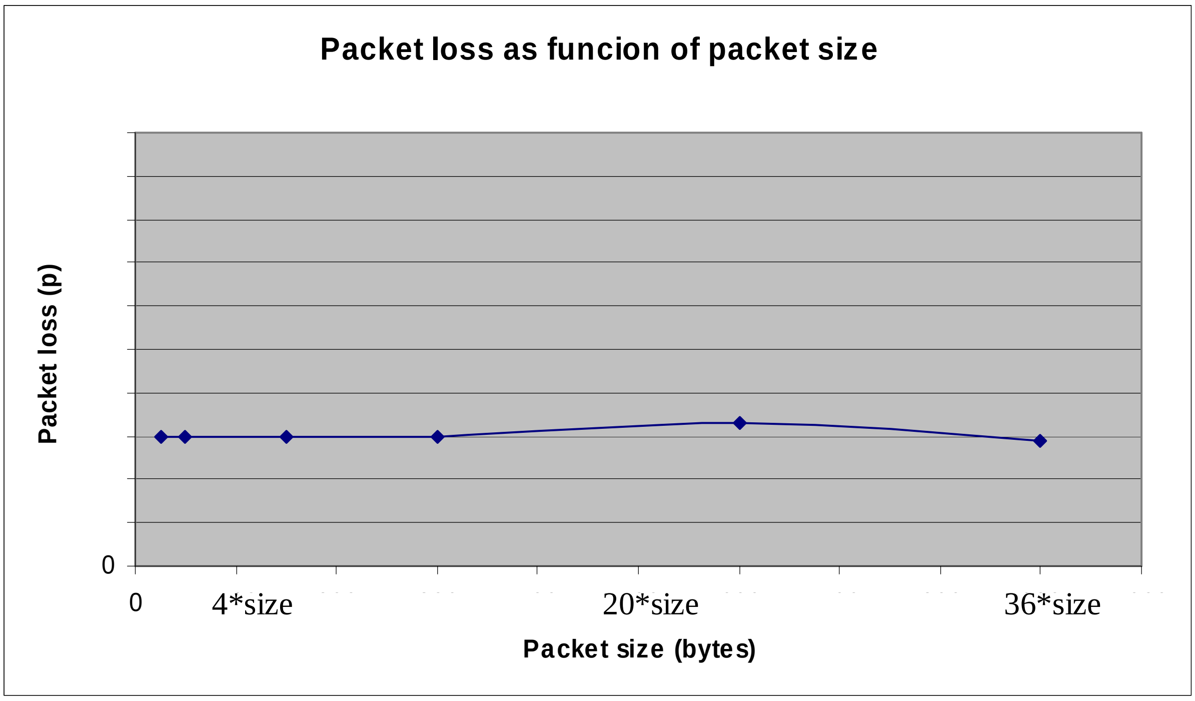


Figure 4-2 Packet loss as function of packet size

Figure 4-2 indicates that there are a constant number of packets lost, independent of the packet size. There are values that differ from the mode value (most common value) but this difference is so small that it does not affect the result of the fail-over. The results from measurements with the same parameters may differ from each other by a small value (ε).

4.2.3 Test 3 (Packet intensity)

Test 3 was performed with six different packet intensity values. The packet size and the number of packets sent were set to constant values. The values for the packet intensity are represented by values in the interval i to 6*i.

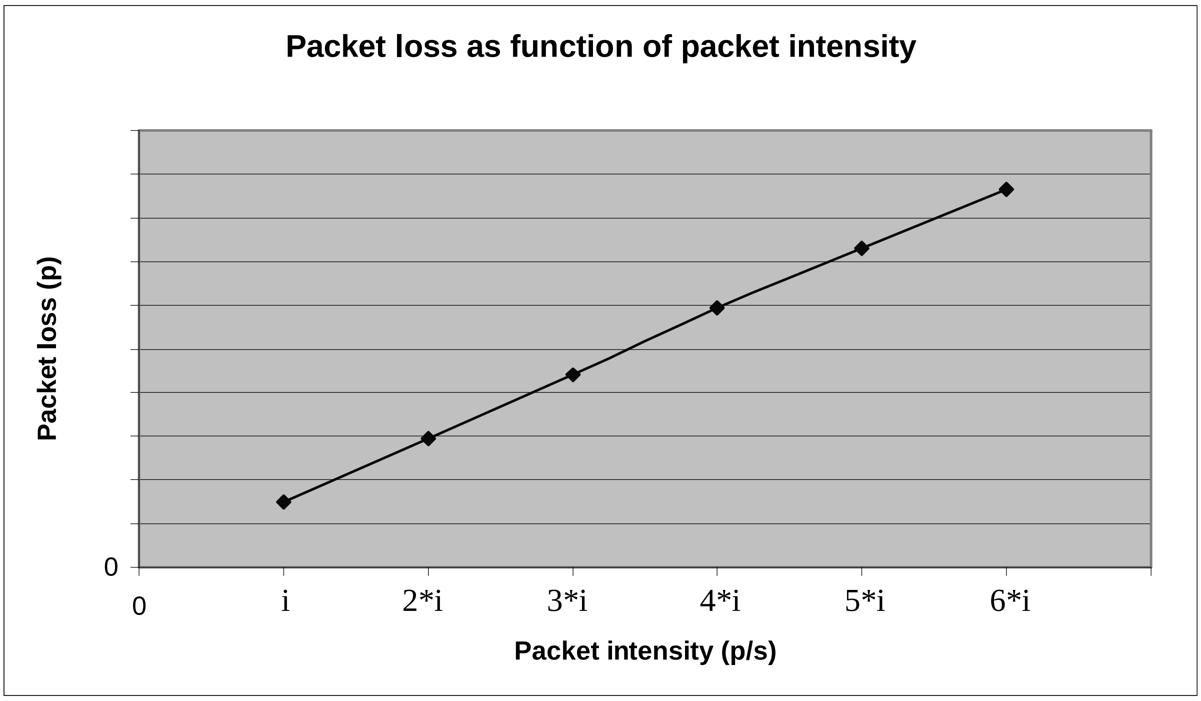


Figure 4-3 Packet loss as a function of packet transfer intensity

In contrast to the other test cases, the packet intensity affects the packet loss. The graph (Figure 4-3) is linear and the packet loss increases as the packet intensity increases.

4.3 Evaluation

The meaning with an emulator is that it should work exactly as the real environment where the actual software runs. The expectation from the fail-over test was that it should work in the emulated CPP node [section 2.4.7], as it works in a physical CPP node [section 2.4.6]. The result was that the fail-over works as expected.

For this test case, a successful fail-over implies that

The transfer of packets from point A to point B should only lose packets within a certain interval of time when the fail-over occurs.

The graphs for the packet size [Figure 4-2] and number of packets sent [Figure 4-1] does not indicate any affect of the fail-over. The value for the packet loss has a constant value even though the number of packets sent or the packet size differs. For the packet sizes, there is a small difference between the number of lost packets and the mode value for two of the measurements. This difference from the mode value is small despite the large changes of packet size. This indicates that the number of packets lost when the packet size differs can be interpreted as a constant value. The packet loss is constant independent of the packet size and the number of packets sent. The only parameter that affects the packet loss is the packet intensity, because the packet loss increases when the packet intensity increases.

The following conclusion was drawn:
Figure 4-3 shows a certain packet loss for a certain packet generation intensity and when the intensity increases the packet loss increases as well. The graph is linear which means that there is a constant factor, which determines the packet loss.
The packet intensity times a constant factor equals the packet loss.

Packet loss may be described as a function ***PL(x),***
where x is the Packet intensity (packets/second)
PL(x) = cx *(where c is the constant factor)*
The units for x are given as the number of packets per second and
the units for PL(x) are given as the number of packets
therefore the units for c are seconds.

The constant factor, c, represents a time interval within which packets are lost. After the tests were run, a time interval of 1.5 seconds was calculated. This value itself does not indicate that the fail-over works but the fact that it is a constant value for all the measurements indicates a successfully performed fail-over. A successfully performed fail-over implies that packet losses should only occur within a certain time interval from the ejection of the board until the traffic has been rerouted. When sending traffic from point A to point B and a fail-over occurred, there was a certain interval of time when the packets were lost. Therefore it can be concluded that the fail-over is working as expected as in a real physical CPP node.

4.4 Summary

The primary purpose with an emulator is that it should work as the real environment. The fail-over was tested with three different test cases and the results matched expectations. The expectation was that since a fail-over works in a physical CPP node, the fail-over should work in the emulated CPP node. The tests confirmed this. The result was presented as graphs of the results from the test cases.

5 Conclusions and evaluation

5.1 Why test the CPP emulator

There could potentially be bugs when working with an emulator. If there is an unknown bug in the hardware system this implies that the implementation of the emulator does not include this bug and therefore not represent the real environment. It could be difficult to implement an emulator that acts as a hardware system because the hardware system could be very complex. There is a need for testing the emulator properly to ensure that testing in the emulator works just as well as in the real CPP environment.

5.2 Defining a successfully performed test case

When evaluating the result from the test, conclusions are made about:

- The results from the measurements [section 4.2] show that packets were lost for a certain interval of time. This is a requirement for a successfully performed fail-over. There should not be any packet losses except when the fail-over occurs. If it can be shown that there are no losses outwit the fail-over interval, the test has succeeded. The main task when analysing the data from the experiment was to look for a pattern. In this case, it was easy because in the measurements for the packet intensity, packets were lost within a certain time interval. The results in section 4.2, indicate a successfully performed fail-over.
- The graph in Figure 4-3 is linear which indicates that packet loss depends on the packet intensity multiplied by a constant factor (see section 4.3 Evaluation). There was a certain amount of time allowed for a fail-over and this fail-over time could easily be calculated with this formula:

$$\frac{\sum_{i=1}^{n} \frac{packetloss_i}{int\,ensity_i}}{n}$$

- A fail-over in the setup for this experiment takes about 1.5 seconds. When analysing results it is clear that packet loss only occurred for 1.5 seconds and number of packets that were lost is equal to the formula:

  *Packet loss = 1.5 * intensity*

As previously mentioned a successfully performed test case could be defined:

> *The transfer of packets from point A to point B should only lose packets within a certain interval of time when the fail-over occurs.*

A condition was established to define when a test case was successful:

$$\left(packetloss \leq 1.5 * int\,ensity + 2\right) \Rightarrow Pass$$

The addition with “2” in the condition is for the measurements with very low intensity. The packet loss can differ slightly from 1.5 * intensity even though a correct fail-over has taken place. For the cases with higher intensity there is no need for this margin, but is good to make the acceptance for the test cases as general as possible.

5.3 Details for the fail-over

This section is a more in-depth discussion about what happens when a fail-over occurs and why packets are lost.

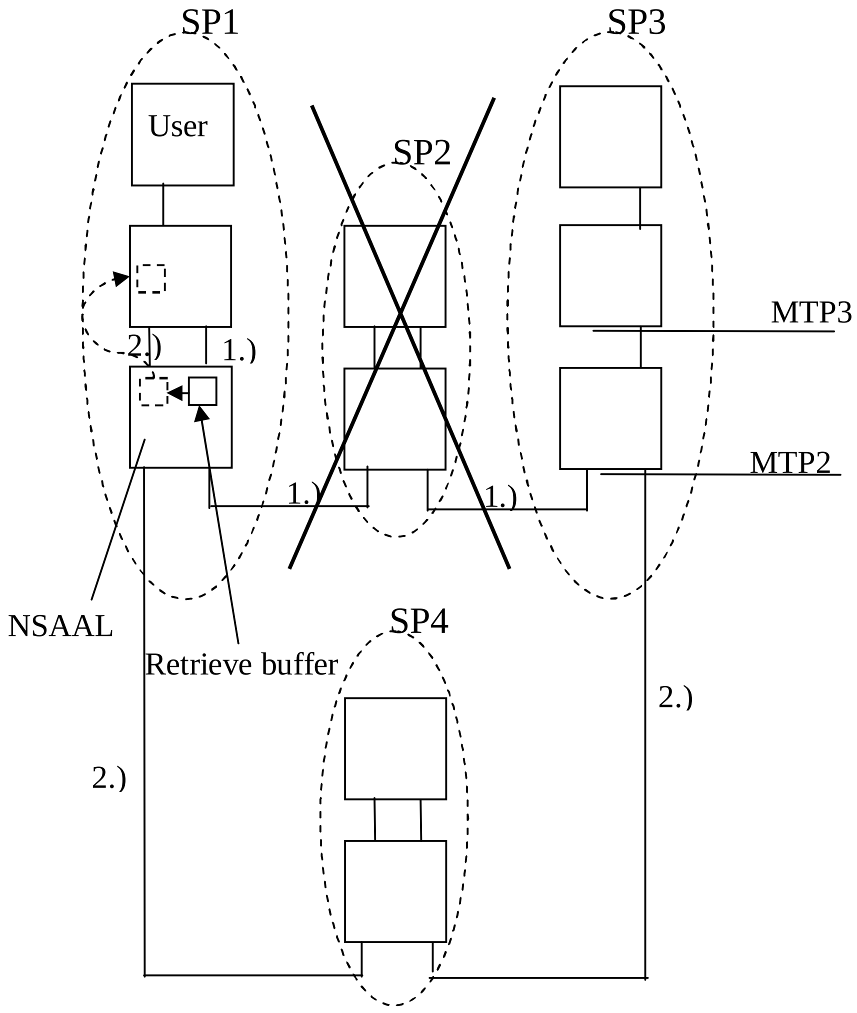


Figure 5-1 Fail-over details

Figure 5-1 shows what happens when a user (in this case LOCO) sends traffic from SP1 to SP3 via SP2 and a fail-over occurs. LOCO has an access point [section 3.5.3] connected to SP1 in the MTP3 layer and the traffic is passed to MTP2 [Table 1]. Every packet is buffered in the retrieve buffer. The packets are sent to SP2 but are not removed from the buffer until an ACK (packet to acknowledge that the sent packet is received at its destination) for each packet is received from SP2. It is the NSAAL termination point [section 3.5.3] that controls whether each packet gets an acknowledgement.

There is a timer for the NSAAL termination point and when all ACKs are lost until the timer has expired the switching of the routes is triggered. The actual switching of the routes does not take more than a few milliseconds. In this experiment, packets are lost during about 1.5 seconds due to the effect of the timer.

When the fail-over occurs, it is only possible to lose packets that are received and ACKED by SP2 but not yet transferred to SP3. When the board for SP2 is ejected, the packets that are on that board are lost. The packets have already been ACKED at SP1, so they will not be kept in the retrieve buffer for SP1. The packets that are transmitted from SP1 after the ejection of the board are buffered in the retrieve buffer on SP1 until a new route is set via SP4.
There are two possible scenarios when configuring an SP. The SP could be configured as an STP (Signalling Transfer Point), which means that the alternative route with priority two has the same sort of carrier as route one. In that case, the buffered packets are sent back up to the layer 3 MTP3 for retransmission. In this experiment, each carrier is configured for broadband traffic, so this should be the solution, but the 1.5 seconds that the fail-over takes is not convincing. This configuration does not lose any packets except the packets that were on the board. The subsequent packets are buffered and retransmitted.
The other scenario is that the SP is configured as a SGW (Signalling GateWay), which means that the route having priority two has another type of carrier than route one, for example, route one could send broadband traffic and route two could send IP traffic. In that case, the retrieve buffer is filled for 1.5 seconds and is not retransmitted. When the retrieve buffer is not retransmitted the packets are lost for the 1.5 seconds before a new route is set up.

In the beginning of the experiment, it was not known which configuration was used. First after the result of the experiment, it was possible to make a conclusion about the configuration. The configuration that was used for this experiment was the SGW signalling point configuration, indicated by the packet loss of 1.5 seconds. The important result from this experiment is that packets were lost for a constant time of 1.5 seconds in every test that was performed and which scenario which was configured does not matter. The packets were lost for only a certain time interval (as discussed above) and this is enough to conclude that the fail-over works correctly.

5.4 Project Evaluation

5.4.1 Experiment performance

The work of this experiment adopted a sequential approach. Thus to be able to perform task 2, task 1 has to be performed first. To send traffic over the network there has to be a network to send traffic on. The network must therefore be created before any experiments can be run. The configuration of the network took much more time than estimated. It would have been easier if it were possible to work more in parallel, but since this was not possible, the only approach was to solve the current problem to be able to proceed.

5.4.2 The Environment

The principal task was to get the configuration of the network up and running. This task took up much of the available time for the project. The remaining tasks were dependent on having a functioning environment. The emulator is currently under development, which means that the emulator was not guaranteed to be in a working state all the time. The CETP [section 2.4.3] was also under development and different versions had to be incorporated in the project at different points in time, requiring extra testing. Despite these issues, the emulator is a good tool and definitely an alternative to performing all of the testing on a physical CPP node [section 2.4.6].

5.4.3 Evaluation

This work has been educational in many aspects. It was interesting to learn about the testing approach used in industry and to see how a particular telecom system worked in detail. It was also a good experience to learn a complex system from the beginning without having any a priori knowledge about the tools interacting within the system.
The learning phase in this project took some considerable time. The remaining time was used to configure the environment and system, write MO-scripts, edit the provided CETP script and finally perform the tests.
As the tools used in this project were internal, not much of the knowledge gained from university courses taken previously could be used, but instead many new experiences were

gained. Having previously studied simulation and emulation in an introductory course, it was interesting to see how an emulator is used in an industrial situation.
The project consisted of two tasks. The first task was to test the fail-over procedure. This task consisted of three major parts; configure the emulator, configure the network and finally run the tests. The test was performed with different parameters and it was concluded that packet size and number of packets did not affect the performance of this fail-over. The focus was on the packet intensity. After analysing the results from the tests, it was concluded that the fail-over worked according to the definition of a successful fail-over. The second task was to automate the ejection of the board to trigger the fail-over in the tests. This task was started but not finished due to lack of time. The second task was however a minor part of the project. The fact that the emulator and network was successfully configured, the tests were successfully performed and the outcome showed that the fail-over procedure works, led to the conclusion that the project was a success.
Finally, we were pleased with the approach we took, and with the result of the project. In retrospect, it is difficult to envisage taking a different approach to the project. Such projects by their nature involve a steep learning curve and the importance of fully understanding the components of any system became clear by the time we had completed the project. We would like to thank TietoEnator for making this work possible.

5.4.4 Future work

This experiment includes broadband connections, which means that the experiment has shown that a fail-over in the emulator works for a broadband connection. Future work consists of performing the same setup and test case for other connections, such as HSL (High Speed Link) or IP (Internet Protocol).
To perform the fail-over test, a board has to be ejected manually. To automate this requirement, enabling the ejection of the board in a script, is also another task for future work. This would mean that test cases could be performed without supervision. Some time was allocated to this task, but without success.

References

[1] Virtutech AB [www] <http://www.virtutech.com>(08 02 20)

[2] Tcl Developer Xchange [www] <http://www.tcl.tk>(08 02 20)

[3] Enea [www] <http://www.enea.com>(08 02 20)

[4] Clear Case [www] <http://en.wikipedia.org/wiki/Clearcase> (08 02 20)

[5] GNU Operating System [www] <http://www.gnu.org> (08 02 21)

[6] GNU GPL [www] <http://www.gnu.org/licenses/old-licenses/gpl-2.0.html> (08 02 21)

[7] The Open Group [www] <http://www.unix.org> (08 02 21)

[8] What is Linux [www] <http://www.linux.org/info/> (08 02 22)

[9] CPP Ericsson Connectivity Packet Platform [www] <http://www.artes.uu.se/industry/031111/5CPP-Artes.ppt#3> (08 06 05)

Internal references:

[10] Spångberg, Kjell. CPP emulator User Guide R12C: 2007-12-07

[11] Ying, Jie. LOCO User Guide: 2007-11-22

[12] Magnusson Finn, Smith David. MoShell 6.1d User Guide: 2005-09-29

[13] Einarsson Magnus, Heubeck Peter.ClearCase Introduktion: 2002-10-15

[14] Svensson Liza.CPP Signaling Overview: 2005-06-02

[15] CPP MOM [www] <https://extinfo.uab.ericsson.se/cpp_mom/> (08 04 14)

[16] The Expect Home Page [www] <http://expect.nist.gov> (08 04 22)