Introducing TCP in a 3G load generator

by

Henrik Lönndahl

LIU-IDA/LITH-EX-G--08/004--SE

2008-10-12
Final Thesis

Introducing TCP in a 3G load generator

by

Henrik Lönnåldahl

LIU-IDA/LITH-EX-G--08/004--SE

2008-10-12

Handledare: Staffan Wiklund
Examinator: Simin Nadjm-Tehrani
Abstract

In this thesis we investigate, implement and evaluate a solution for introducing the Transmission Control Protocol (TCP) into the software of a load generator. The load generator is a simulator used for simulating end-user generated activities in the Universal Mobile Telecommunication System (UMTS) network. The purpose of simulating traffic on the network is in this case to verify the functionality and robustness of the Radio Network Controller (RNC) node within the UMTS network.

TCP is a protocol that provides reliable data transfer over unreliable underlying network protocols. It is used as the main transport protocol of the Internet, thus it is also used in the UMTS network in order to provide connectivity for user equipment, such as 3G mobile phones, to services over the Internet. For the load generator to be able to produce more realistic traffic scenarios it is desirable to give it the ability to use TCP.

This thesis presents a solution of the problem where an open-source implementation of the TCP functionality was chosen, ported to the running platform of the load generator and then tested in a simulated test environment. The choice of the open source implementation of TCP was made by performing an investigation of available options. In the investigation an open source TCP/IP stack called lwIP was chosen. lwIP was then ported to the running platform of the load generator by wrapping and modify the source code. The tests of the ported TCP implementation were made in a simulated test environment with focus on testing basic TCP functionality. The tests showed that the TCP implementation produced provided the basic functionality that was asked for.
# Table of Content

## 1 Introduction

1.1 The Problem ........................................................................... 1  
1.1.1 Description .......................................................................1  
1.1.2 Requirement specification ................................................1  
1.1.3 Milestones ........................................................................2  
1.1.4 Problem Approach............................................................2

1.2 Thesis outline............................................................................ 2

## 2 UMTS

2.1 General .............................................................................. 4  
2.2 Architecture......................................................................... 4  
2.2.1 User Equipment................................................................4  
2.2.2 Core Network .................................................................5  
2.2.3 UTRAN...........................................................................5

## 3 The Load Generator

3.1 General .............................................................................. 7  
3.2 Scope of Use ....................................................................... 7  
3.3 Architecture.......................................................................... 7  
3.3.1 Simulator Application.......................................................8  
3.4 Running Environment............................................................9  
3.4.1 Connectivity Packet Platform .........................................9  
3.4.2 Operating System Embedded .........................................9  
3.5 Simulated Test Environment .................................................10  
3.5.1 Test suites ......................................................................10  
3.5.2 Test Implementation Work .............................................10

## 4 The Transmission Control Protocol

4.1 Overview.............................................................................. 11  
4.1.1 The TCP Segment..........................................................11  
4.2 Protocol Operation...............................................................12  
4.2.1 Connection Establishment...............................................12  
4.2.2 Data Transfer .................................................................13  
4.2.3 Connection Termination ..................................................14

## 5 Simulation of TCP Load

5.1 Protocol Stack Investigation ..................................................15
List of Figures

Figure 1 UMTS architecture..................................................................................4
Figure 2. Simplified Core Network architecture.................................................5
Figure 3. UTRAN architecture..............................................................................6
Figure 4. Load generator scope of use.................................................................7
Figure 5. Load generator architecture outline....................................................8
Figure 6. Simulator application design outline....................................................8
Figure 7. OSE signalling example........................................................................9
Figure 8. Test in simulated environment.............................................................10
Figure 9. TCP segment structure.........................................................................11
Figure 10. TCP three-way handshaking..............................................................12
Figure 11. TCP connection tear-down.................................................................14
Figure 12. External perspective of the TCP/IP process.......................................18
Figure 13. Internal data flow of the TCP/IP process...........................................20
Figure 14. Simplified UML diagram....................................................................21
Figure 15. Creation of a communication session using the Proxy....................24
Figure 16. Test outline..........................................................................................25
Figure 17. Simplified diagram of the test classes implemented...........................26
Figure 18. Simulation workflow example............................................................27

List of Tables

Table 1. test results..............................................................................................30
1 Introduction

This report is the documentation of a final thesis performed at Ericsson AB in Linköping. The thesis is a partial fulfilment of a degree in computer engineering (15 ECTS points). The main aim of the final thesis was to develop a portable implementation of the Transmission Control Protocol (TCP) suited to be integrated into a load generator used for testing the Radio Network Controller (RNC) node within the Universal Mobile Telecommunications System (UMTS) network. The load generator can be described as a simulator that can simulate traffic and behaviour of other nodes of the UMTS network in order to verify the functionality and robustness of the RNC node.

In this chapter the content of the final thesis is outlined. The chapter includes objectives, problem description, problem approach and requirement specification.

1.1 The Problem

1.1.1 Description

The load generator is using the User Datagram Protocol (UDP) for transfer of payload to external systems. In general when talking about networking, the UDP protocol is rarely used for other purposes than streaming video or other areas where the importance of every packet getting to the receiver in the correct order not is critical. Instead, the TCP protocol is far more often used. This is among other things because of its sequence control and higher reliability. Thus, to make the load generator produce more realistic traffic it was desirable to make it be able to send its payload via the TCP protocol.

The problem of this final thesis was defined as to develop an implementation of the TCP protocol that could be used in the load generator to simulate traffic to and from different nodes of the UMTS network. The problem was divided into three parts.

- A pre-study in which studies of UMTS and the load generator was included.
- The actual implementation of the solution as well as documentation of it.
- An evaluation of the TCP implementation produced.

1.1.2 Requirement specification

Here the requirements of the final thesis are presented.

Requirements on the TCP implementation

T1: The implementation should be able to handle up to 1000 simultaneous TCP sessions.

T2: The implementation should have some kind of flow control for data transferred from other processes.

T3: The implementation should have trace- and debugging functions.

T4: TCP shall be implemented as one single process, with one thread executing.

T5: The CPU load caused by the implementation shall be comparable to other existing TCP implementations.

Other requirements

T6: A technical documentation of the TCP implementation should be produced.
1.1.3 Milestones
To solve the problem of this final thesis project three milestones were defined.

The first milestone was reached when an acceptable solution proposal of implementing TCP in the load generator was found that met the requirements in the requirement specification.

The second milestone was reached when the solution was implemented. A technical documentation of the implementation shall also be produced.

The third milestone was reached when an evaluation of the TCP implementation was done.

1.1.4 Problem Approach
In order to find a solution to the problem and fulfil the goals the problem was approached as follows:

Studies
The following aspects needed to be studied in order to get an understanding of what was to be done:

• Studies of the basic architecture and terminology of the UMTS network in order to get an overview of the usage area of the load generator.
• Studies of the load generator. This included studies of the functionality and abilities of the load generator as well as the running environment of it.
• Studies of the development environment that is used when developing and testing software to the load generator.

Solution proposal
The following high level solution proposal was used when approaching the problem:

• Find existing open source code of a TCP implementation that fulfils as many of our requirements as possible.
• Investigate how the open source implementation can be modified in order to fit in to the load generators application layer.
• Implement the solution.
• Evaluate the implementation.

Evaluation
The evaluation of the implementation was carried out by evaluating if the implementation meets the requirements specified in the requirements specification. Part of the evaluation made use of a test framework that is available in the development environment of the load generator. Different basic tests showed that the TCP implementations basic functionality was satisfactory. Further tests on the performance aspect (T5) remained as future work.

1.2 Thesis outline
UMTS. This chapter presents the basics of the UMTS network regarding architecture and terminology in a top-down manner.
The Load Generator. A brief presentation of the load generator including scope of use, design, functionality and running environment.

The Transmission Control Protocol. This chapter introduces TCP. We describe the principles and behaviour of the protocol together with a brief explanation of the TCP packet structure.

Simulation of TCP load. This chapter describes what has been done in this final thesis. The chapter starts with a presentation of the investigation of protocol stacks that was carried out. The second part describes the TCP/IP process that was implemented based on the protocol stack chosen in the investigation.

Evaluation. Chapter 6 describes the evaluation of the TCP/IP process described in the previous chapter. Description of the method of testing, test cases and discussion of test validity is included.

Conclusion. Chapter 7 gives the reader a summary of what has been done in this final thesis project. We also discuss how well the requirements of the thesis were fulfilled, as well as personal experiences that were gained during the work.
2 UMTS

In this chapter the basics of UMTS is presented in a top down manner. This chapter first gives a general view of UMTS and a presentation of the basic UMTS architecture [1, 2, 3, 4]. The aim of this chapter is to give the reader a rough idea of where in the UMTS network the load generator has its working area.

2.1 General

Universal Mobile Telecommunication System (UMTS) is one of the third generation mobile technologies built from the 2G Global System for Mobile (GSM) network architecture. UMTS uses Wideband Code Division Multiple Access (WCDMA) technology in its air interface. This technique is faster and supports more users than the 2G air interface technology Time Division Multiple Access (TDMA) which instead of code division uses time slots.

UMTS together with the High-Speed Downlink Packet Access protocol (HSDPA) allows end users to transfer data with a maximum speed up to 14.4 Mbit/s, which can be compared to the fastest 2G technique Enhanced Data Rates for GSM Evolution (EDGE) maximum data transfer rate of up to 2Mbit/s. The faster data transfer rates of UMTS enables new features to the end user like for example live TV and video calls.

UMTS is standardized by the 3rd Generation Partnership Project (3GPP) which is a collaboration between associations working with telecommunications. 3GPP:s main purpose is to make the 3rd generation mobile system globally applicable.

2.2 Architecture

The architecture of the UMTS network can be divided into three main parts. The User Equipment, the UMTS Terrestrial Radio Access Network (UTRAN) and the Core Network (CN).

Figure 1 UMTS architecture.

2.2.1 User Equipment

The user equipment (UE) term in the UMTS network represents the wireless equipment operated by an end user of the UMTS network. A common type of UE is a 3G phone, but it could also be a card in a computer or a component in a handheld computer.

The UE connects via an air interface called Uu to the RBS node in the UTRAN.
2.2.2 Core Network
The main task for the core network (CN) is to provide routing, switching and transit for user traffic. The architecture of the CN is much based on the architecture of the GSM network together with GPRS functionality.

The CN has two domains. One that deals with circuit-switched data like speech, and one that deals with packet-switched data, for example streaming video.

Components of the circuit-switched division are for example the Mobile services Switching Centre (MSC) and Visitor location register (VLR). The MSC provides the interface between the radio systems and the fixed networks like for example the Public Switched Telephone Network (PSTN). The MSC performs all necessary tasks for handling the circuit switched services provided in the UMTS network. The Visitor Location Register (VLR) is used by the MSC to retrieve information about for example roaming UE:s. The Gateway MSC (GMSC) is the MSC that provides the actual routing to the PSTN network.

Components of the packet-switched division are the Serving GPRS Support Node (SGSN) and Gateway GPRS Support Node (GGSN). The SGSN node stores subscription information and location information for packet switched services. The GGSN node stores subscription and routing information for subscribers.

There are also components that are shared between the both divisions.

![Core Network Diagram](image.png)

Figure 2. Simplified Core Network architecture.

2.2.3 UTRAN
The part of the UMTS network that provides connectivity between the UE and the core network is called the UMTS Terrestrial Radio Access Network (UTRAN). The UTRAN consists of a set of Radio Network Subsystems (RNS) which in its turn consists of a set of Radio Base Stations (RBS) and one Radio Network Controller (RNC) interconnected. The RNC node provides control functionality over one or several RBS nodes and is connected to the RBS nodes via an interface called Iub. The RNC is connected to the CN (Core Network) via two interfaces, Iu-CS (Circuit switched) and
Iu-PS (Packet switched). Also, RNC nodes in the same network are connected with an interface called Iur.

The RBS has a set of geographical areas called cells, often three, which provide connectivity to the UTRAN for UE:s located in any of these cells.

![Figure 3. UTRAN architecture.](image)

The functionality of the RNC node in the UTRAN is under continuous development. For example improvements of RNC features are made, as well as development of new ones. In order to verify the behaviour and capacity of the RNC the load generator is used to simulate work load on the network. This will be further described in the next chapter.
3 The Load Generator

In this chapter the basics of the load generator and its running environment is presented. The chapter is derived from Ericsson Internal documents [5, 6, 7].

3.1 General

The load generator can be described as a simulator used for verification of the RNC node in a UMTS network. It generates traffic by simulating end user activities at different nodes in the UMTS network.

In short, the load generator consists of a C/C++ software application running over a platform called Connectivity Packet Platform (CPP). The development of the load generator started in 2002 and was from the beginning an extension of another tool for testing.

3.2 Scope of Use

The load generator can be connected to the Iub, Iu-PS and Iu-CS interfaces, which makes it possible to simulate components of the Core Network (CN), the User Equipment (UE) and the RBS node. It is also possible to use the load generator together with a real CN.

![Load generator scope of use diagram](image)

Figure 4. Load generator scope of use.

The load generator can simulate both Circuit switched (CS) and Packet Switched (PS) payload and signalling both from the simulated UE:s and the simulated Core Network.

3.3 Architecture

The architecture of the load generator at a high level consists of a control interface, the simulator application itself. Under the simulator application different high-level protocols operate, and under that the CPP platform operates including low-level protocols and the hardware.
3.3.1 Simulator Application

The structure of the simulator application consists of three main component types, namely The Behavior Generators (BG), the Signal Generators (SG) and the User Data Generators (UDG). The application also includes a control interface and implementations of several high-level protocols such as for example UDP. Each of the components that the load generator simulates has its own type of BGs and SGs.

A Behaviour generator generates events that are caused by user activities on the network. The events will differ depending on the type of BG.

A Signal Generator generates the signalling procedures for a simulated node. For example the signals sent by the MSC node to connect two simulated UE:s in a call.

A User Data Generator (UDG) generates the actual packets, speech and dataframes that are to be sent out on the network. The content of the packets and dataframes depends on the behaviour settings and the type of UDG.
3.4 Running Environment

3.4.1 Connectivity Packet Platform
The running environment of the simulator application is the Connectivity Packet Platform (CPP). CPP is a scalable platform that consists of both hardware and software. The software includes among other things a real-time control system, media processing and a packet transport system. The software of interest in this thesis is the control system which consists of a distribution of Operating System Embedded (OSE).

The hardware of the CPP consists of a multi-processor environment with processors based on the Power-PC architecture.

3.4.2 Operating System Embedded
Operating System Embedded (OSE) is part of the CPP platforms control system, thus it is also the operating system that the load generator runs over. It is a scalable real time operating system (RTOS) optimized for high-availability and distributed systems.

OSE fundamental parts that are interesting in this thesis are its processes and interprocess communication. The communication between processes can be done in several ways in OSE, but the standard way is to use the OSE signalling system. By using pre-defined or user-defined signals together with the unique id of a process the send() system-call can be used by the processes to communicate with other processes. Each process has a single queue where the signals intended for the specific process are queued up. The receiving process has the ability to choose which type of signal it wants to fetch from its queue at the moment.

![OSE signalling example](image)

*Figure 7. OSE signalling example.*

The fact that user-defined signals are available makes it possible to use signals to transport data between processes.

3.5 Simulated Test Environment
In order to conveniently test software developed for integration into the load generator the load generator has its own test framework. The test framework allows the programmer to test basic functionality of the software without having to set up the tests directly into the target environment.
The test framework consists of a C/C++ library that provides functionality for simulating the OSE environment in a UNIX environment and also functionality for creating test suites. The programmer can with the framework simulate the OSE operating system together with the OSE processes that are needed by simply compiling the code into one single executable file.

![Executable file diagram](image)

*Figure 8. Test in simulated environment.*

### 3.5.1 Test suites
The test suite of the simulation is what drives the test forward. A test suite consists of one or several test cases in which one or several test steps is executed. For every test step the result is verified. If a test step produces an unexpected result it fails, and thus the whole test case fails. If on the other hand the test step is a success, the next one is executed until the whole test suite is done.

### 3.5.2 Test Implementation Work
When using the test framework there is some implementation work that needs to be done by the programmer:

- Behaviour of the processes that we wish to simulate. The test framework simulates the interfaces of the processes and simulates OSE with its inter-process communication, but not the actions of the processes.
- Test suites along with test cases and test steps are all defined by the programmer.
4 The Transmission Control Protocol

This chapter introduces the Transmission Control Protocol (TCP) in a brief manner [8]. The aim is to give an understanding of the basic functionality and usage of the TCP.

4.1 Overview

TCP is a protocol that resides in the OSI transport layer. It provides a reliable, connection-oriented way to transport data. TCP is used in Internet's transport-layer and is therefore one of the cornerstones in today’s Internet.

TCP is optimized to deliver accurate data and not so much to deliver it fast. Sometimes TCP can cause large delivery delays, but on the other hand it is reliable in the manner of every packet getting to the destination and in the correct order.

TCP operates next to the OSI application layer, only concerning itself about the two end-systems sending data to each other and not so much how the data is routed between them. The lower levels of data transport are left for other protocols to take care of. On the Internet for example, TCP is operating close to applications such as web browsers and web servers, while the Internet Protocol (IP) is the underlying layer knowing where to route the data.

4.1.1 The TCP Segment

The payload data to be transferred when using TCP is divided into TCP segments. A TCP segment consists of a header field including protocol specific information followed by the actual payload. There is a Maximum Segment Size (MSS) that limits the size of the payload data field. The MSS is negotiated in each TCP connection.

As seen above the structure of a TCP segment consists of a number of fields storing different information.

Figure 9. TCP segment structure.
In the beginning of the segment TCP port information is stored. Different TCP ports are used by different upper-layer applications, thus, the port number tells which upper-layer application the segment is destined to and which application that sent it.

Next, the segment sequence number is stored and under that the acknowledgement number. This information is used for providing a reliable data transfer discussed in section 4.2.2.

Next the header length, segment flags and the receive windows is stored. The segment flags tell the receiver system what kind of segment it is. If for example the segment is an acknowledgment of received data the ACK flag is set. Besides the ACK flag the most important flags is the FIN flag, which is set when the connection is terminated and the SYN flag which is set when a connection is established. The receive window is used by TCP to implement flow control, which is further discussed in section 4.2.2.

The rest of the segment contains the Internet checksum which is used to check that the packet is uncorrupted, a urgent data pointer that points at data in the segment that is urgent, some options regarding TCP and the actual data field of the segment.

4.2 Protocol Operation

4.2.1 Connection Establishment
TCP is called connection-oriented because before one application can send data via TCP to another, the two applications first have to establish a connection. This is done with so called handshaking. The handshaking consists of the end systems sending a few initial TCP segments between each other to establish the connection.

The handshake is initiated by side A sending a empty segment with the SYN-flag set. This segment is the initial connection initiative and is called the SYN segment. Next, the side B replies to side A with a SYN-ACK segment telling side A that we have received the SYN segment and that side B agrees to start a connection. Finally the side A sends an ACK segment back to side B telling it that it has received the SYN-ACK segment and that the connection now is established. After these three steps, A and B can transport data between each other.

![TCP three-way handshaking](image)

Figure 10. TCP three-way handshaking.
4.2.2 Data Transfer
The data transfer of TCP provides a reliable way to transfer data over an unreliable underlying protocol like for example IP. TCP ensures that data sent from system A to system B is uncorrupted, without gaps, without any duplicate data and that it arrives in the right sequence. On top of that, TCP also implements functionality of flow- and congestion control.

Sequence Numbers and Acknowledgements
The sequence control of TCP is implemented by letting each TCP segment include a sequence number. The sequence number reflects the stream of bytes transmitted, not the stream of segments. The sequence number of the TCP segment is thus the number of the first byte transmitted in the segment.

For every segment the sequence number has to be incremented. With this sequence number the receiver can monitor if a segment arrives in sequence or not. If a segment arrives out of sequence there are two choices. Either the receiver can drop the segment, or it can keep it until the preceding segments arrive.

When data arrives the receiver of the TCP segment sends an ACK segment as a reply to the sender telling the sender that it has received all data preceding the sequence number. If it for example sends an ACK of sequence number 7 and the last acknowledged sequence was 1, it means that it has received byte number 1-6. If the sender receives an ACK for sequence number 3, and its last sent byte had sequence number 6, it knows that bytes number 3, 4 and 5 has been lost on the way to the receiver. In this case the sender will retransmit the missing bytes.

TCP Checksum
In order to ensure that packets sent via TCP are not corrupted every segment includes a 16-bit checksum. In short, this is a number that summarizes the data and the header of the segment. The checksum is computed and filled in before the packet is sent and when the packet is received the checksum is computed again. If the result is the same as the checksum stored in the segment, the segment is not corrupted.

Flow Control
In a TCP connection both end systems have a receive buffer. Data that has arrived uncorrupted and in sequence is placed in this buffer waiting for the application that used the TCP connection to take care of the data. This may not happen right away, and the buffer could start to grow. To avoid overflowing the buffer and the receiving application with data TCP implements flow control.

Flow control of TCP is implemented by having the sender to maintain a variable called receive window. In short, this variable keeps the amount of free buffer space in the receivers receive buffer. The receiver will in each segment sent to the sender specify how much data it is willing to buffer. The sender is then forbidden to send more data than specified in the last information update about the receive window at the receiving system.

Congestion Control
In order to keep the data flow at a level where the link between the two end systems does not risk a collapse, TCP implements a congestion control mechanism. The congestion control of TCP uses four algorithms that use different parameters to figure out the condition of the link that is used. Among other things, the volume of
acknowledgements, or lack of acknowledgements for sent data is used to regulate the data sending speed.

4.2.3 Connection Termination

A termination of a TCP connection is initiated when end-system A sends a segment with the FIN flag set to system B. System B replies with an ACK segment. The next step is for system B to send a FIN segment, and system A to send an ACK segment.

As seen in the figure, the A system goes into a time-wait state after it has received the FIN segment from the B system. The reason for this is to be able to resend the final ACK in case it gets lost on the way to system B.

There are variations of the connection tear down handshaking. For example it can be done with a three-way handshake instead. In this case system A initiates the tear-down by sending a FIN segment. The B system then replies with a combined FIN and ACK segment. Then system A replies by sending the last ACK segment.

Figure 11. TCP connection tear-down.
5 Simulation of TCP Load

In this chapter the actual result of this final thesis is presented. The first section describes the investigation of possible open source stacks which was the first part of the implementation phase. The resulting software implementation is presented in section two.

5.1 Protocol Stack Investigation

This thesis proposes that an open source protocol stack is to be ported into the running environment of the load generator in order to provide the basic TCP functionality.

In order to find such a protocol stack an investigation of available open source protocol stacks was made.

5.1.1 Method

The investigation of protocol stacks was performed by first searching the Internet in order to find different alternatives. Then the alternatives were compared with the considerations presented in chapter 5.1.2 in mind.

5.1.2 Considerations

In the investigation of protocol stacks the following aspects were taken into consideration in order to find the best alternative:

- **Portability.** The protocol stack should be portable to arbitrary underlying operating system in an easy and flexible way.

- **Documentation.** The protocol stack should be well documented both in the manner of usage and implementation. Also, the source code itself should be well commented.

- **Usability.** The protocol stack should have an interface that focuses on simplicity and allows integration to other software in an easy way. The stack should also have debugging functions.

- **Functionality.** It is positive if the protocol stack implements more functionality than TCP for future needs. This could for example be support for other high-level protocols that could be used to simulate end-user activities.

- **Implementation.** The protocol stack must be implemented in the C/C++ programming languages in order to fit the surrounding environment.

5.1.3 Protocol Stacks

This section includes a brief presentation of the protocol stacks considered in the investigation.
LwIP

LwIP is a small independent implementation of the TCP over IP protocol suite that has been developed by Adam Dunkels at the Swedish Institute of Computer Science (SICS). LwIP focuses on using as little resources as possible while still having full scale TCP support.

The LwIP stack includes support for a number of protocols and functions other than TCP and IP. For example it implements the UDP and PPP protocols and has support for collecting statistics. When compiling the source code you have the possibility to exclude the modules you do not need in order to reduce code size.

The LwIP stack is designed to be ported to arbitrary platforms. This is done by implementing the operating system emulation layer that is a part of the stack source code. This layer works as an interface between LwIP and the underlying operating system. There is a document provided with LwIP that describes what needs to be implemented in the emulation layer.

The LwIP stack is documented in the manner of both usage and implementation. There is a separate design document, a readme file and an Internet wiki page that contains useful information for developers.

uIP

The uIP is another stack written by Adam Dunkels at SICS. This TCP/IP stack is mainly aimed at providing connectivity to embedded microcontrollers and is therefore much focused on low RAM usage. The uIP stack provides besides the TCP and IP protocols also for example UDP and ARP.

The uIP stack is designed to be ported to any system, and the implementation of this is similar to the LwIP stack.

The uIP stack is well documented with information about how to port and use it.

Linux Protocol stack

Linux is an open-source operating system that includes its own TCP/IP stack. An alternative would be to lift this stack out of Linux and use it in our implementation.

The Linux TCP/IP stack is widely used and has been developed for several years and is documented in literature available to buy.

5.1.4 Investigation Result

The investigation of open source protocol stacks resulted in choosing the LwIP stack for our implementation. LwIP seemed to be a good choice when considering most of the aspects that were important, while the other options covered less of the considerations.

The most important reasons to why LwIP was chosen were the following advantages:

- The stack is designed to be portable to any platform in an easy way and includes documentation on how to do it.
- LwIP is still actively supported by a development group. The development group is developing new features to the LwIP stack, improves the existing source code and is also available for answering questions through a mailing list.
- Well documented with technical documentation regarding both usage and implementation.
• Designed for conserving memory while still providing full scale TCP/IP support.
• Implements more than TCP functionality. For example PPP and ARP.
• It is possible to tune code size by only compiling the modules you really need.

An issue discovered regarding the lwIP protocol stack was that the TCP and IP functionality is not completely separated. This makes it hard to use only the TCP functionality of the stack. This however is acceptable; we might as well let lwIP take care of the IP functionality as well.

When considering the uIP stack it seemed to be a good choice regarding portability and documentation. On the other hand it did not seem to be as good as lwIP regarding usability and additional functionality. The uIP stack is very much focused on conserving resources, while this was not that important in our case.

When considering the Linux protocol stack the biggest issue was to find documentation about it other than expensive books. Since it was hard to find information about the Linux protocol stack it was hard to judge if it met the considerations of the investigation.

5.2 The TCP/IP Process

This chapter presents the result of the work that has been done in the implementation part of this final thesis.

5.2.1 Overview

The implementation phase of this final thesis resulted in an OSE process with one single thread running. The process is providing the functionality of the TCP and IP protocols for a payload generating process.

The TCP/IP process interfaces the application layer of the OSI model in one end of its interface and the data link layer of the OSI model in the other. In the application layer several instances of a payload generating OSE process will reside, which will be able to both send and receive payload data via the TCP/IP process. Beside the pure payload data, control signals are sent between the payload generating process and the TCP/IP process in order to provide flow control. The data link layer is represented by an OSE process that can use arbitrary technology for transporting data. The only requirement on the layer 2 process is that it should provide pure TCP/IP packets to the TCP/IP process. Figure 12 describes the organization and data flow among the processes.
5.2.2 Functionality Outline
The functionality of the TCP/IP process is here presented in a brief manner.

Session Management
The payload generating process can request the TCP/IP process to create a communication session. The data transfer between the payload generating process and the TCP/IP process is always done through a session. The session contains all the necessary information needed by the TCP/IP process, like for example TCP, IP and flow control related information.

Buffering management
When the TCP/IP process receives data, it has the ability of buffering it in both ends of its external interface. Payload data sent from the payload generating process is, if the internal lwIP send buffers is full, buffered in a FIFO outbound buffer where it stays until there is room for the payload in the lwIP buffer. There is one outbound buffer per session in the TCP/IP process.

When a TCP/IP packet is sent from the layer 2 process to the TCP/IP process and there is no room in the lwIP internal memory for incoming data, the whole packet is buffered in a FIFO inbound buffer waiting for later processing. There is one buffer for all incoming packets from the layer 2 process.

The buffering functionality is implemented by letting two instances of a buffer handler object take care of inbound and outbound buffering separately.
Flow Control

In the TCP/IP process there are two flow control functionalities implemented monitoring the data flow between the payload generating process and the TCP/IP process. The first function ensures that the payload data arrives to the TCP/IP process in the correct order. The second function monitors the outbound buffer. If the buffer reaches a given level of fullness, the sending payload generator instance will receive a warning signal. This allows the payload generating process to regulate its data sending speed.

Flow control between the TCP/IP process and the layer 2 process is managed by TCP. If for example the inbound buffer gets full, the incoming data is dropped by the TCP/IP process. When the data is dropped, no ACK packet is sent to the remote host, which will result in a retransmission of the data from the remote host. This is further described in section 4.2.2.

Debugging

The TCP/IP process has trace functions for debugging included. For debugging the TCP and IP functionality the lwIP source code includes a set of trace macros. The peripheral source code of the TCP/IP process uses a set of trace macros that are included in the load generator software.

The macros provide trace functionality by simply printing out information about what is happening at a certain point in the execution of the process. The trace macros belong to different trace groups, which are used in different situations. For example, one group of trace macros are used to trace when execution enters functions, while another is used to trace when an OSE signal is sent. When debugging the process, the programmer has the ability to enable or disable printouts of different trace groups in order to extract different kinds of information.

5.2.3 Design Outline

The TCP/IP process has been implemented in the C and C++ programming languages and the size of the TCP/IP process source code is about 6000 lines of code if the lwIP source code is excluded. The lwIP source code itself has over 20 000 lines of code, but only a small part of it is used for the TCP and IP functionality. The design of the process focuses on performance rather than simplicity. For example copying of data is avoided as much as possible. It is desired that the TCP/IP process operates as fast as possible so that it will not slow the payload generating process down.

The process uses the OSE signalling system in order to interact with other processes, both in the manner of signalling and data transport. This makes it compatible to arbitrary OSE processes.

The core functionality of the process, namely the TCP and IP functionality, is provided by the lwIP stack source code. To provide the functionality and behaviour that is desired, but not provided by lwIP, a number of peripheral classes are included in the process. These classes provide and encapsulate for example OSE signalling, buffering and flow control functionality. The classes are designed so that each of them is dedicated to solve one or a few different problems.

The maximum number of communication sessions that the TCP/IP process can handle is set at compile time. Thus, in theory the process can handle an arbitrary number of sessions.
5.2.4 Data Flow
Internally in the TCP/IP process, the payload data flows through a maximum of four different modules. When the payload generating process is sending data, the TCP/IP process needs to associate it with a communication session. After the session handling the data can either go straight to the TCP/IP processing, or to an outbound buffer in case there is no room in lwIP internal buffers. After this, the payload is encapsulated in a TCP/IP packet and is transported in an OSE signal to the underlying layer 2 process.

When the layer 2 process delivers a TCP/IP packet to the TCP/IP process, it is either forwarded straight to the TCP/IP processing or buffered if there is no memory in the lwIP internal buffers. After the TCP/IP processing the payload data is forwarded to the session handling where it needs to be associated with a session, and then it is sent in an OSE signal to the payload process.

![Figure 13. Internal data flow of the TCP/IP process.](image)

5.2.5 Class Description
In addition to the lwIP source code ten classes have been implemented as a part of the process. These classes provide the functionality needed for making use of the lwIP TCP/IP functionality in the desired manner. In this section the classes are briefly presented and described. The classes in grey are the classes implemented in this thesis.
The TcpProxy class is the API used by other processes when communicating with the TCP/IP process. The proxy class encapsulates the OSE signalling in order to make usage of the TCP/IP process easier. The proxy includes methods for sending all signals that the TCP/IP process can receive.

SignalDispatcher
The SignalDispatcher is responsible to initially receive and dispatch signals to the signal receivers of the process. The two receivers of the TCP/IP process are the TcpTimerHandler and the TcpSignalReceiver. This class is an existing class used in other places in the load generator source code, and was not implemented as a part of this thesis.

TcpSignalReceiver
This class is responsible for receiving all incoming signals, except timer signals, and perform appropriate actions depending on the type of the incoming signal. The process has only one instance of this class when running.
TcpBufferHandler
The buffer handler is a class for handling the buffers within the TCP/IP process. It can handle an arbitrary number of buffers. In the TCP/IP process there are two instances of this class, one for handling outbound buffers and one for handling the inbound buffer.

Pdu
The Pdu class is a container class used by the buffer handler class to store payload data in the buffers.

TcpSessionHandler
This class is responsible for handling the communication sessions of the TCP/IP process. The process has only one instance of this class when running.

TcpSession
The TcpSession class is a container class used for storing all necessary information needed to keep track of the TCP/IP communication. There is one instance of this class for every communication session.

TcpStackHandler
The Stack handling class serves as an interface to the lwIP stack for the other classes of the TCP/IP process. The process has only one instance of this class when running.

TcpSignalSender
This class handles all outgoing signalling of the TCP/IP process. The process has only one instance of this class when running.

TcpTimerHandler
This class handles the timers of the TCP/IP implementation. It receives timer signals from the OSE timer server and calls the TcpTimer object that the signal was destined to. The process has only one instance of this class when running.

TcpTimer
This class implements the timer functionality of the TCP/IP process. Two timers are used in order to implement the TCP functionality.

5.2.6 Application Programming Interface
The Application Programming Interface (API) towards the TCP/IP process is implemented with OSE signals. However, for making the signalling to the TCP/IP process easier a Proxy has been implemented. The Proxy is a class that provides a simple function-call based interface towards the TCP/IP process that hides the underlying OSE signalling. Whenever a process wants to send signals to the TCP/IP process it can instantiate the Proxy and use the appropriate function instead of creating and sending the OSE signal manually.
When receiving signals from the TCP/IP process it is on the other hand up to the using process to take care of the signals. Below, the API of the TCP/IP process is explained from a using process point of view.

**Session Management**
The payload generating process can create a communication session between itself and the TCP/IP process via the Proxy class by calling the create session function. The Proxy will then send a session creation request which will be processed by the TCP/IP process. A reply will be sent from the TCP/IP process which will confirm or reject the creation. A confirmation includes a session reference number which will be used by the payload generating process when it wants to send data via the session. A rejection will include an error code of what went wrong with the creation.

**Data Transfer**
Data can be transferred to and from the TCP/IP process either from the payload generating process or from the layer 2 process. Both using process can send data to the TCP/IP process via the Proxy class with two different function calls. The payload generating process however must always transfer data together with a session reference number, or else the data will be rejected.

**Error Handling**
If an error occurs in the TCP/IP process that concerns the payload generator process, it will receive an OSE signal with information about what is wrong. The information about errors is represented as a set of error codes. If for example the payload generator tries to send data via a session that does not exist, the payload process will receive a signal from the TCP/IP process rejecting the data transfer, including an error code meaning that the session was not found.

**Flow Control**
The flow control described in section 5.2.2 is implemented in the API by using OSE signals containing an error code.

The functionality of sequence control is implemented by including a sequence number in the data signals that is sent from the payload process to the TCP/IP process. The TCP/IP process stores the last received sequence number in the communication session. If the incoming sequence number in the data signal not is the last received number plus one, something is wrong. If so, the TCP/IP process sends a rejection signal including an error code telling the payload generating process that a sequence error has occurred.

When monitoring the buffers fullness, the TCP/IP process keeps track of the buffer sizes and the level of fullness of which it should warn the payload process that the buffer is nearly full. When the warning level is reached, a confirmation signal is sent to the payload process telling it how many bytes that is left in the buffer.
5.2.7 Documentation
A document [9] with focus on usage and behaviour of the TCP/IP process has been produced. The document also presents a brief overview of the design as well as options that are available in the TCP/IP process. Due to the level of detail in the matter of usage and options of the TCP/IP process, the documentation is an internal document of Ericsson AB and cannot be presented as a part of this report.
6 Evaluation

This chapter presents the evaluation of the TCP/IP process produced. The chapter starts by describing how the test environment was created. Then we present what tests were performed and the test results. We also discuss the validity of the tests.

6.1 TCP/IP Functionality Test Method

6.1.1 Outline

In order to test the TCP/IP process basic functionality the test framework described in section 3.5 was used. With the framework a payload generating process and a layer 2 process was simulated. The idea of the method was simply to establish communication between two instances of the payload generating process via the TCP/IP process.

Each of the payload instances creates one or several communication sessions in the TCP/IP process. Then one of them sends data via the TCP/IP process destined for an IP address owned by the other payload instance. The TCP/IP process performs the processing and sends the packet to the simulated layer 2 process. As shown in figure 12 the simulated layer 2 process receives packets from the TCP/IP process and sends it right back instead of transporting it out on a network. This method simulates the surroundings of the TCP/IP process and makes it possible to test its basic functionality.

6.1.2 Design

This test setup required the programming of the behaviour of the two simulated processes as well as defining the test suite that drives the test forward. The programming needed to implement the tests resulted in about 6000 lines of code.

In order to implement the behaviour of the simulated processes two interface classes were implemented. Classes for representing a payload instance and a session has also been created. These classes are used to store information used by the payload generating
A test suite with four test cases was also implemented. In each test case a number of test steps and events are included.

**Test Classes**

- **Sim. Layer 2 Process**
  - Implements the actions of the simulated layer 2 process.
  - L2If

- **Sim. Payload Process**
  - Implements the actions of the simulated payload generating process.
  - PayloadIf
  - PayloadInstance
  - Session

- **TestSuite**
  - TestCase
  - TestStep
  - TestEvent

*Figure 17. Simplified diagram of the test classes implemented.*

In short, the test case starts by executing its first test step. The test step performs an action which generates work in the TCP/IP process. For example, it could generate a request from the payload generator process to send data to the TCP/IP process. The interface classes of the simulated processes receive signals from the TCP/IP process as a result of the test step action, and generate different test events depending on the type of signal it receives. These events are sent back to the test step class which depending on the type of the event performs a new action. It could for example send a reply signal or advance to the next test step. This cycle is then repeated until a test case fails or if the whole test suite finishes.
6.2 TCP/IP Process Functionality Test Suite

As a part of the evaluation of the TCP/IP process tests were performed in order to verify its basic functionality. When testing the TCP/IP process a number of test cases were chosen in order to create a test suite. The purpose of the test suite was to focus on testing the basic functionality rather than testing special cases. The basic tests are important in order to be able to execute more sophisticated test cases in a later test phase.

The tests in the test suite are based on the test outline described in section 6.1.1. The following test cases were included in the test suite:

6.2.1 Small Data Transfer Test

Purpose: To check that the TCP/IP process can establish a connection between two IP addresses, send small amounts of data and terminate the connection.

Test steps:
1. Process initiation.
2. Creation of a listening session.
3. Creation of a connecting session.
4. Send 2, 20 and 375 bytes of data between the sessions.
5. Terminate sessions.

Expected results: Initiation, creation of sessions should be successful. The data sent should reach the receiver. Termination of sessions should be successful.

Pass Criteria: The result of the execution of the test case should be in accordance with the expected result.
6.2.2 Large Data Transfer Test

Purpose: To check that the TCP/IP process can establish a connection between two IP addresses, send 1 Mb of data and terminate the connection.

Test steps:
1. Process initiation.
2. Creation of a listening session.
3. Creation of a connecting session.
4. Send 1 Mb of data between the sessions.
5. Terminate sessions.

Expected results: Initiation, creation of sessions should be successful. The data sent should reach the receiver. Termination of sessions should be successful.

Pass Criteria: The result of the execution of the test case should be in accordance with the expected result.

6.2.3 Small Data Transfer between Multiple Sessions

Purpose: To check that the TCP/IP process can establish connections between 100 IP addresses, send small amounts of data between them and terminate the connection.

Test steps:
1. Process initiation.
2. Creation of 500 listening sessions.
3. Creation of 500 connecting sessions.
4. Send 2, 20 and 375 byte of data between the sessions.
5. Terminate sessions.

Expected results: Initiation, creation of sessions should be successful. The data sent should reach the receivers. Termination of sessions should be successful.

Pass Criteria: The result of the execution of the test case should be in accordance with the expected result.

6.2.4 Large Data Transfer between Multiple Sessions

Purpose: To check that the TCP/IP process can establish connections between 100 IP addresses, send 1 Mb of data between them and terminate the connection.

Test steps:
1. Process initiation.
2. Creation of 500 listening sessions.
3. Creation of 500 connecting sessions.
4. Send 1 Mb of data between the sessions.
5. Terminate sessions.

Expected results: Initiation, creation of sessions should be successful. The data sent should reach the receivers. Termination of sessions should be successful.
Pass Criteria: The result of the execution of the test case should be in accordance with the expected result.

A test not included in the basic test suite was also done in order to verify the T5 requirement of this thesis:

### 6.2.5 Basic CPU Load Test in the Simulated Environment

**Purpose:** To measure the load of the CPU while executing the basic test suite.

**Test steps:**
1. Execution of the basic test suite.
2. Usage of the standard UNIX command “top” while test suite is running in order to measure CPU load.

**Expected result:** The CPU load caused by the test suite process should be displayed by the top command.

**Pass Criteria:** The CPU load should be successfully measured.

### 6.3 Test Validity

The validity of the tests executed in the simulated environment reflects execution in the target environment, but with a few differences:

- The hardware of the target environment is different from the hardware that the simulation was executed on. Since the processor architectures are different the interpretation to machine code could differ.
- Different compiler versions are used depending on compiling to the target environment or the UNIX environment.
- The tests executed in the test suite are without timer functionality. In the test framework there are timer functionality available, however time in the execution has to be stepped forward by calling a function in the test code. Since time is not naturally stepped forward there is no meaning to use it, since the timing probably not will reflect the real case.
- The basic CPU load test (described in section 6.2.5) does not only measure the CPU load caused by the execution of the TCP/IP process, but also the execution of the test framework, since they are compiled into one executable file. Thus, the CPU load test will not give us the exact CPU load caused by the TCP/IP process, but a CPU load that we can be sure is higher than the CPU load caused by the TCP/IP process itself.
6.4 Test results

The result of the test suite is presented below.

<table>
<thead>
<tr>
<th>Test</th>
<th>Final Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Data Transfer</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>Large Data Transfer</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>Small Data Transfer between</td>
<td>Passed</td>
<td>In order to be able to create multiple sessions, memory adjustments in</td>
</tr>
<tr>
<td>Multiple Sessions</td>
<td></td>
<td>the lwIP source code had to be done</td>
</tr>
<tr>
<td>Large Data Transfer between</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>Multiple Sessions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic CPU load test in the</td>
<td>~ 0,5 % - 10</td>
<td>Executed on a server with a Solaris operating system. The server had</td>
</tr>
<tr>
<td>Simulated Environment</td>
<td>% of the CPU</td>
<td>4 UltraSparc 1,6 GHz CPU:s</td>
</tr>
<tr>
<td></td>
<td>capacity</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. test results.

All the test cases included in the test suite did after some debugging in the TCP/IP process pass.

Besides executing the test suite the TCP/IP process source code has been executed together with a tool for detecting memory leaks. This test resulted in no memory leaks found.
7 Conclusion

The result of this final thesis resulted in the following work:

- A pre-study of the UMTS and the basics of the load generator.
- The implementation of a single threaded OSE process which provides the functionality of the TCP and IP protocols for an arbitrary OSE process.
- The implementation of a test environment for the TCP/IP process using a test framework included in the software of the load generator.
- The implementation and execution of a basic test suite in the test environment implemented.

7.1 Requirements fulfilled

The requirements presented in this final thesis (see section 1.1.2) are fulfilled with one consideration.

The T5 requirement cannot be said to be fulfilled for sure because there was not enough time to verify this. To ensure that the CPU load of the TCP/IP process was comparable to other TCP implementations a test in the target environment needed to be done, as well as an investigation of other TCP implementations CPU loads. On the other hand, the basic CPU load test (described in section 6.2.5) gave us a hint of the CPU load caused by the TCP/IP process.

7.2 Future work

There are tasks that would be interesting to do regarding development and testing of the TCP/IP process, but had to be left behind because of lack of time.

The TCP/IP process needs to be further tested. It would be interesting to do more basic tests in the simulated environment. Negative tests could be included in which unexpected and faulty actions are performed. The purpose of such tests is to see how the error handling of the TCP/IP process works. The next step after the negative tests would be to do tests in the target environment of the load generator. Benchmark tests could be done where the performance of the TCP/IP process is measured, as well as robustness tests where we basically put a high workload on the TCP/IP process and find out how much it can take.

More future work is in the aspect of logging and collection of statistics. The lwIP source code has support for collection of protocol specific statistics. This could be interesting to collect, log and analyze.

7.3 Personal Experiences

My personal experience of working with this project is here presented.

The pre-study that was carried out in the beginning was of great importance since it was the foundation of the later working phases. It could however been done even more thorough in order to get more understanding of the problem to be solved.

The investigation of available open source protocol stacks resulted in the lwIP stack, which was not a surprise. The lwIP stack is a well known, well working implementation which is widely used by others for similar purposes. Thus, the investigation did not generate much work.
The implementation phase was the part of the final thesis which was the most time consuming. This had several reasons. If I had done a deeper pre-study and a deeper implementation plan I think that it would have taken less time. The biggest reason however was my lack of experience in programming such large applications in large, complicated environments.

The test phase of the final thesis is the part of the final thesis that contains the most future work since tests in the target environment are yet to be done. The test phase generated a lot more work than I planned from the beginning and was perhaps the most complicated task. However, I think the test phase gave me a lot of knowledge that can be useful later on.

The project was from the beginning carried out according to the time plan. The implementation work was finished on time, but the writing of this report was not. This was mainly because the test phase generated so much more work than I expected, and I prioritized the implementation and testing before writing the report.

The result of this final thesis is an achievement I am proud of. The task and its requirements are overall satisfactorily fulfilled and I have learned a whole lot that I will make use of later on.
8 References


[5] ”3Gsim user’s guide”, Ericsson Internal Document


## Terminology

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3GPP</td>
<td>3rd Generation Partnership Project</td>
</tr>
<tr>
<td>CN</td>
<td>Core Network</td>
</tr>
<tr>
<td>CPP</td>
<td>Connectivity Packet Platform</td>
</tr>
<tr>
<td>CS</td>
<td>Circuit Switched</td>
</tr>
<tr>
<td>FIFO</td>
<td>First in, first out</td>
</tr>
<tr>
<td>GGSN</td>
<td>Gateway GPRS Support Node</td>
</tr>
<tr>
<td>GMSC</td>
<td>Gateway Mobile services Switching Centre</td>
</tr>
<tr>
<td>GPRS</td>
<td>General Packet Radio Services</td>
</tr>
<tr>
<td>HSDPA</td>
<td>High-Speed Downlink Packet Access protocol</td>
</tr>
<tr>
<td>Iub</td>
<td>Interface between RBS and RNC</td>
</tr>
<tr>
<td>Iu-CS</td>
<td>Circuit switched interface between RNC and Core network</td>
</tr>
<tr>
<td>Iu-PS</td>
<td>Packet switched interface between RNC and Core Network</td>
</tr>
<tr>
<td>Iur</td>
<td>Interface between connected RNC nodes</td>
</tr>
<tr>
<td>MSC</td>
<td>Mobile services Switching Centre</td>
</tr>
<tr>
<td>NBAP</td>
<td>Node B Application Part</td>
</tr>
<tr>
<td>OSE</td>
<td>Operating System Embedded</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>PDU</td>
<td>Protocol Data Unit</td>
</tr>
<tr>
<td>PS</td>
<td>Packet Switched</td>
</tr>
<tr>
<td>PSTN</td>
<td>Public Switched Telephone Network</td>
</tr>
<tr>
<td>RANAP</td>
<td>Radio Access Network Application Part</td>
</tr>
<tr>
<td>RBS</td>
<td>Radio Base Station</td>
</tr>
<tr>
<td>RNC</td>
<td>Radio Network Controller</td>
</tr>
<tr>
<td>RNS</td>
<td>Radio Network Subsystem</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating System</td>
</tr>
<tr>
<td>SGSN</td>
<td>GPRS Support Node</td>
</tr>
<tr>
<td>TDMA</td>
<td>Time Division Multiple Access</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>UE</td>
<td>User Equipment</td>
</tr>
<tr>
<td>UMTS</td>
<td>Universal Mobile Telecommunication System</td>
</tr>
<tr>
<td>UTRAN</td>
<td>UMTS Terrestrial Radio Access Network</td>
</tr>
<tr>
<td>Uu</td>
<td>The air-interface between the UE and the RBS</td>
</tr>
<tr>
<td>VLR</td>
<td>Visitor Location Register</td>
</tr>
<tr>
<td>WCDMA</td>
<td>Wideband Code Division Multiple Access</td>
</tr>
</tbody>
</table>
Introducing TCP in a 3G load generator

Henrik Lönndahl

In this thesis we investigate, implement and evaluate a solution for introducing the Transmission Control Protocol (TCP) into the software of a load generator. The load generator is a simulator used for simulating end-user generated activities in the Universal Mobile Telecommunication System (UMTS) network. The purpose of simulating traffic on the network is in this case to verify the functionality and robustness of the Radio Network Controller (RNC) node within the UMTS network.

TCP is a protocol that provides reliable data transfer over unreliable underlying network protocols. It is used as the main transport protocol of the Internet, thus it is also used in the UMTS network in order to provide connectivity for user equipment, such as 3G mobile phones, to services over the Internet. For the load generator to be able to produce more realistic traffic scenarios it is desirable to give it the ability to use TCP.

This thesis presents a solution of the problem where an open-source implementation of the TCP functionality was chosen, ported to the running platform of the load generator and then tested in a simulated test environment. The choice of the open source implementation of TCP was made by performing an investigation of available options. In the investigation an open source TCP/IP stack called lwIP was chosen. lwIP was then ported to the running platform of the load generator by wrapping and modify the source code. The tests of the ported TCP implementation were made in a simulated test environment with focus on testing basic TCP functionality. The tests showed that the TCP implementation produced provided the basic functionality that was asked for.