

B-Assignment Telematics

TCP/IP modelling in OMNeT++

Author

Jeroen Idserda

Supervisors

Dr.ir. Geert Heijen
Dr.ir. Pieter-Tjerk de Boer

Date

July 6, 2004



University of Twente
The Netherlands

Table of Contents

1	Introduction	4
1.1	Introduction	4
1.2	Problem Description	4
1.3	Approach	4
1.4	Outline	4
2	OMNeT++	5
2.1	What is OMNeT++	5
2.1.1	History	5
2.1.2	NED	5
2.1.3	Module implementation in C++	7
2.1.4	Running a simulation	7
2.2	Queuing models in OMNeT++	8
2.2.1	Queuing models	8
2.2.2	Average response time	8
2.2.3	Theory	9
2.2.4	Job Distribution	9
3	TCP/IP	11
3.1	Introduction	11
3.2	TCP	11
3.2.1	Internals	11
3.2.2	Versions	12
3.3	Areas of interest	13
4	TCP/IP-implementation in OMNeT++	14
4.1	Existing implementation	14
4.2	Changes	15
4.2.1	Tcpdump output	15
4.2.2	Delayed Acknowledgements	16
4.2.3	Persist Timer	17
4.3	Further improvements	17
5	Analysis of TCP/IP in OMNeT++	18
5.1	TCP Friendly	18
5.1.1	TCP Friendly formula	18
5.1.2	Simulation Setup	18
5.1.3	Results	19
5.1.4	Implications	20
5.2	Asymmetric connection	20
5.2.1	TCP and asymmetric connections	20
5.2.2	Test results: Throughput	22
5.2.3	Test results: Congestion window	23
5.2.4	Test results: Miscellaneous	23
5.2.5	Implications	24
6	Conclusion	25
7	References	26

Abbreviation List

ACK	Acknowledgment
ADSL	Asymmetric Digital Subscriber Line
ARPAnet	Advanced Research Projects Agency Network
FCFS	First Come First Served
FIFO	First in First out
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISN	Initial Sequence Number
KB	Kilobyte
LCFS	Last Come First Served
MSS	Maximum Segment Size
NCP	Network Control Protocol
NED	NEtwork Description
OMNeT++	Objective Modular Network Testbed in C++
RFC	Request for Comments
RTT	Round Trip Time
TCB	Transmission Control Block
TCP	Transmission Control Protocol

1 Introduction

1.1 Introduction

OMNeT++ [1] is an open source simulation package. Development started in 1992, and since then different contributors have written several of models for it. Some of these models simulate simple queuing models, others simulate more realistic protocols such as TCP/IP. OMNeT++ is used by universities and companies like Lucent and Siemens.

One of these existing models is the IPSuite package. This package includes support for IPv4, TCP, UDP and QoS models. TCP is one of the most used protocols today because of the popularity of the Internet. It is thus useful to explore the behaviour of this protocol further, by making use of simulations. This simulation can be done in OMNeT++ , but is this TCP model really useful?

1.2 Problem Description

The first purpose of this assignment is to get insight in OMNeT++. After this, the capabilities and shortcomings of the TCP model should be shown, by experimenting with the model.

1.3 Approach

First, the OMNeT++ package will be studied. After that, some simple simulations will be run. The M/M/1 model will be studied and the simulation for this will be executed in OMNeT++. The outcomes from these tests will be compared with the published results and theoretical outcomes.

Then the more extended models will be looked into. In order to understand the TCP model, TCP/IP is studied first. Then several experiments will be done with this model: comparison with TCP Friendly formula and performance issues with asymmetric connections. All this will be described in this report.

1.4 Outline

The rest of this report is structured as follows: first, a description of the OMNeT++ package is given. This description includes the simulation of a simple queuing model, and a comparison with the simulation outcomes and the theoretic results. In chapter 3, a description of the TCP protocol is given, including an overview of the different features. In chapter 4, we look at the TCP implementation of the IPSuite package for OMNeT++. The changes made to this implementation are also listed there. This model is used in chapter 5, in which the outcomes from several simulations are described. In the last chapter, a conclusion is given.

2 OMNeT++

2.1 What is OMNeT++

2.1.1 History

OMNeT++ is a discrete event simulator in development since 1992. Its primary use is to simulate communication networks and other distributed systems. It runs on both Unix and Windows. Other similar programs are the widely used ns-2 [2], which has extensive support for TCP, and OPNET [3].

OMNeT++ started with a programming assignment at the Technical University of Budapest (Hungary), to which two students applied. One of them, András Varga, still is the maintainer of this open source simulation package. During the years several people contributed to OMNeT++, among which several students from the Technical University of Delft (The Netherlands) and Budapest. Milestones in the development are the first public release in 1997 and the added animation functionality in 1998, which made the package even more usable for education.

In 2000 several people at the University of Karlsruhe created the TCP model for OMNeT++.

When working on this report, version 2.3 of OMNeT++ came out. This version included several bug fixes and important changes and additions to the manual. The website also had a major update. A beta version of version 3.0 is now also available. All simulations in this report were done using version 2.3b2 of OMNeT++.

2.1.2 NED

A simulation in OMNeT++ is written in two different languages, NED and C++. These two languages separate the design of a topology and the implementation of the modules that exist in this topology.

The NED (NEtwork Description) language is used to describe the layout (topology) of the simulation.

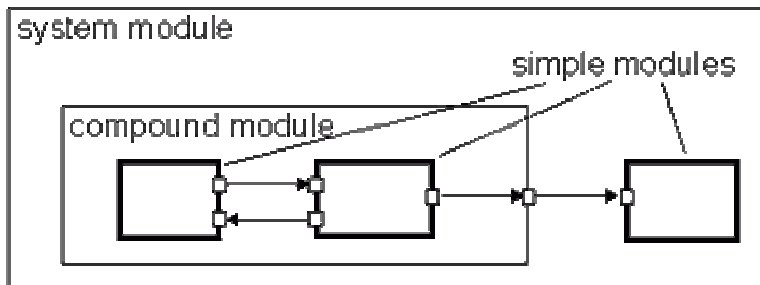


Figure 2-1: Visual network module layout

As you can see in Figure 2-1, each simple module described by the NED language can also be used in a large compound module, which is also described in the NED language. One simple module can for example be a TCP implementation, the other one an Ethernet implementation, and together they can form an Internet host (the compound module).

There are three distinctive elements that can be described in a NED file:

- Module definitions. In these blocks you can describe simple and compound modules and set parameters of these modules. Also, you can define gates that connect the modules.
- Channel definitions. Describe channels (links) that connect modules.
- Network definitions. To get the whole simulation running, you'll have to describe which module is the top level module for the network you wish to simulate. This top level module is an instance of the system module.

In Figure 2-2 the network specification in a .NED file is given for the simulation used in chapter 5.1. There are three simple modules in this simulation. One is of the type ‘ClientNode’, the other one is a ‘ServerNode’ and the last one, Switch, connects these first two. The modules have several parameters and are connected using channels ‘Downstream’ and ‘Upstream’. The switch has two inputs and two outputs, which connect to the Client- and ServerNode. Together they form the network ‘tcpclservnet’. The ‘display’ tags were removed from this example; they only describe the way the modules are placed in the visual representation of the simulation.

<pre> import "TCPClientNode", "TCPServerNode", "switch"; module AsymClientServer submodules: client1: TCPClientNode; parameters: // TCP parameters local_addr = (10 << 24) + 2, server_addr = (10 << 24) + 1, // network parameters numOfPorts = 1, routingFile = "client1.irt", numOfProcessors = 1; gatesizes: in[1], out[1]; server1: TCPServerNode; parameters: // TCP parameters local_addr = (10 << 24) + 1, // network parameters numOfPorts = 1, routingFile = "server1.irt", numOfProcessors = 1; gatesizes: in[1], out[1]; switch: Switch; gatesizes: in[2] out[2]; connections nocheck: server1.out[0] --> switch.in[0]; switch.out[0] --> server1.in[0]; client1.out[0] --> Upstream --> switch.in[1]; switch.out[1] --> Downstream --> client1.in[0]; endmodule channel Upstream datarate 1000000000; // 1 gigabit delay 0.03; endchannel channel Downstream datarate 1000000000; // 1 gigabit delay 0.03; endchannel network tcpclservnet : AsymClientServer endnetwork </pre>	<p>Import several other .NED files. The simple modules used are defined there.</p> <p>Simple Module name Define submodules</p> <p>Define module #1 Parameters of this module</p> <p>IP routing table</p> <p>Gates (input/output)</p> <p>Define module #2 Parameters of this module</p> <p>Gates (input/output)</p> <p>Define module #3 The switch has multiple in- and outputs.</p> <p>Define connections</p> <p>Channel definition, Upstream Define channel properties, throughput and delay</p> <p>Channel definition, Downstream Define channel properties, throughput and delay</p> <p>Define the network tcpclservnet, instance of module AsymClientServer</p>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 2-2: NED language example

2.1.3 Module implementation in C++

The implementation of the modules defined in the NED language is done in C++. You can use the OMNeT++ simulation library, with which you can simulate any kind of behaviour in your models. This library class includes mechanisms to schedule and cancel events, gather statistical data and produce data that can be inspected (plotted) using the graphical tool ‘Plove’, which is also included in the package.

2.1.4 Running a simulation

A simulation in OMNeT++ can be run in two different ways: visual and text-only. The visual simulations are shown in the following graphics. This way of running the simulation is particularly useful when first running the simulation, or to get acquainted with the protocols or networks the program simulates. It shows all the messages that are exchanged between the modules in an animation. Also, with larger simulations, you can look deeper into each module, to see what messages are exchanged internally.

Figure 2-3 shows the main window of the simulation. This is again the simulation used in chapter 5.1. This screen includes several controls to run the simulation. Simulations can be run step by step, normal (shows every message as an animation), fast (show animations, but faster) and express (which doesn’t shows any animation). It is also possible to run the simulation until some point. In this screen, you can also see all modules and their parameters. There is also a list of scheduled events.

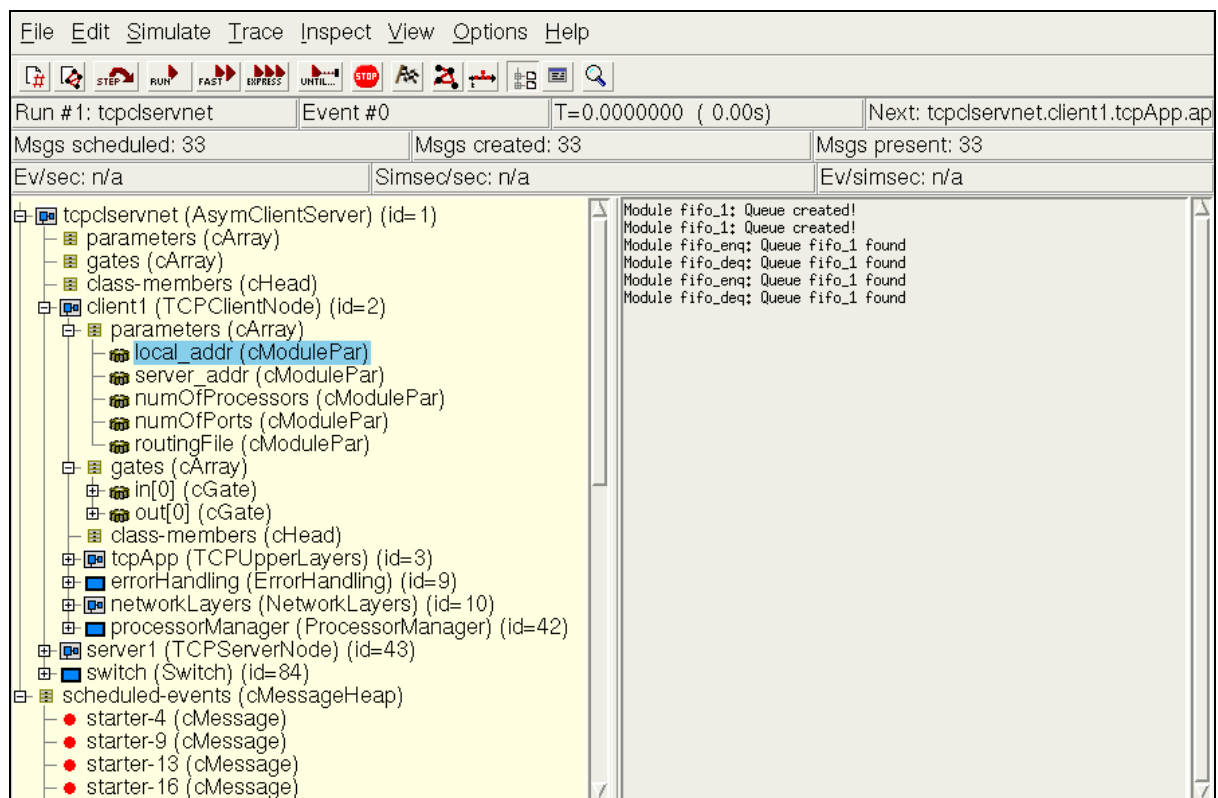


Figure 2-3: OMNeT++ main window

Figure 2-4 to Figure 2-6 show the actual network that is simulated. Figure 2-4 is the top view of the network, which has only three components (client1, switch and server1). Figure 2-5 show the internal modules of the client1 module. And again, in Figure 2-6, the internal modules of the ‘tcpApp’ are shown.

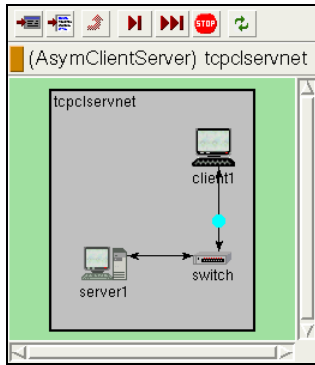


Figure 2-4: Top network view

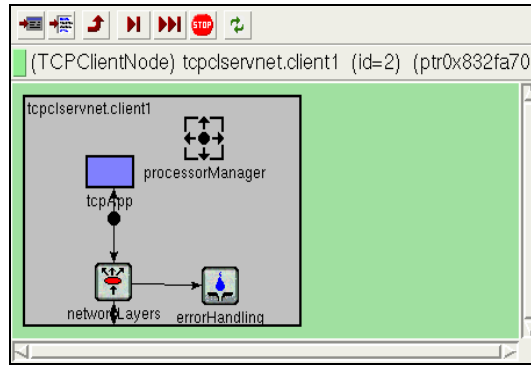


Figure 2-5: Internal view client1

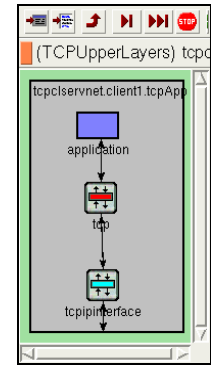


Figure 2-6: Internal view tcpApp

In Figure 2-4 you can see a message (the dot) travelling between client1 and the switch. This is the way all messages are displayed, as you can also see in the next figure, where it is again a dot.

2.2 Queuing models in OMNeT++

2.2.1 Queuing models

Queuing models are applicable in many different areas. Queues in supermarkets can be represented using such a model, but also queues in computer networks. In OMNeT++ there is a model of a particular kind of queue: the M/M/1 queue [4].

Different types of queues can be described using the Kendall notation [5]:

Queue: A/S/m /B /K /SD

In which A is the distribution of the inter arrival times (the time between the arrival of packet j and packet j+1 at the queue), B the distribution of service times (the time it takes to process 1 packet) and m the number of available servers to process incoming packets. The last three parameters represent the amount of buffers (B), the size of the available packet population (K) and the way the incoming packets are handled (SD, for example First Come, First Served (FCFS, also known as FIFO) or Last Come, First Served (LCFS)). If the last three parameters are omitted, it is assumed that: the number of buffers (B) is infinite, the number of packets in the population is infinite and the packets are handled in a first come first served manner.

Using this notation, the M/M/1 queue stands for a queue in which both the inter arrival and service times are exponentially distributed, and only one processor is available.

2.2.2 Average response time

By simulating an M/M/1 queue in OMNeT++ it is possible to determine the average response time a packet experiences when travelling from a generator to its endpoint (the sink). In this simulation it is possible to use the exponential distribution for simulating both service and inter arrival times. In the simulation, the inter arrival time is distributed with parameter $\lambda=1/3$ (1/3 packets arrive on average per time unit) and the service time is distributed with parameter $\mu=1/2$ (on average 1/2 packets per time unit are processed).

We are sending 10 000 packets per run. To get an accurate outcome, the simulation is run 50 times with a different seed value each time. The different seed values are 10 000 000 values apart and are calculated by the seedtool utility that is available with OMNeT++.

The size of the FIFO-queue in this simulation is unlimited. When creating this queue in OMNeT++, no size limit for this queue is given, so it is assumed that this queue is unlimited. A queue in OMNeT is just an object container, which contains the message objects that are generated.

The average response time, as acquired from the simulation, is equal to 6,0414262. It is possible now to calculate a confidence interval (CI) for this data. The number of collected response times is large enough so that the distribution of response times will approximately follow the normal distribution.

The 95% confidence interval, in which α is equal to 0.05 (5%), can be calculated as follows [5]:

$$[\bar{x} - z_{1-(\alpha/2)}s/\sqrt{n}, \bar{x} + z_{1-(\alpha/2)}s/\sqrt{n}]$$

In which \bar{x} is the average response time, s the standard deviation, n the number of simulation-runs (50) and $z_{1-(\alpha/2)}$ the probability according to the normal distribution. The 95% CI then is:

$$[6.0414262 - (1.960 * 0,26933564 6 / \sqrt{50}), 6.0414262 + (1.960 * 0,26933564 6 / \sqrt{50})] = [5,9667702, 6.1160822]$$

This means that there is a 95% probability that in a real queuing situation, with the same parameters, the average response time found would lie in this interval.

2.2.3 Theory

A lot of things are known about this M/M/1 queue. Because of this it is possible to calculate the average response time without running the simulation, when the process and inter arrival times are given.

The mean response time in an M/M/1 queue is given by [5]:

$$E[r] = (1/\mu)/(1-\rho)$$

In which $\rho = \lambda/\mu$ is the traffic intensity. In this equation, λ is equal to the number of packets that arrive per time unit and μ is equal to the amount of packets that are processed per time unit. When filling in the earlier used values for μ and λ , the calculated average response time is:

$$E[r] = (1/0.5)/(1 - ((1/3)/0.5)) = 2/(1/3) = 6$$

This value is in the confidence interval calculated from the simulation data. We can conclude from this that the simulation model is correct when looking at the mean response time.

2.2.4 Job Distribution

OMNeT++ can also produce a graphical presentation of data that is generated by the simulation. The next image shows the job distribution: the probability that at a given time there are n packets in the system. On the vertical axis the probability is shown for a given number of packets on the horizontal axis.

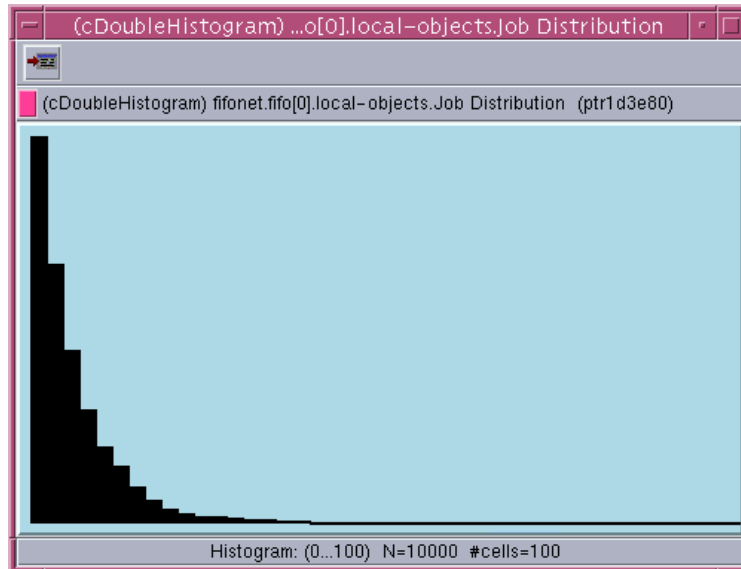


Figure 2-8: Job Distribution from simulation

The numerical data for this graph is also available in OMNeT. It is now possible to compare the results from the simulation with the theoretic probabilities.

The probability of n packets in the system is given by [5]:

$$P_n = (1 - \rho)\rho^n, n = 0, 1, \dots, \infty$$

This results in the following image:

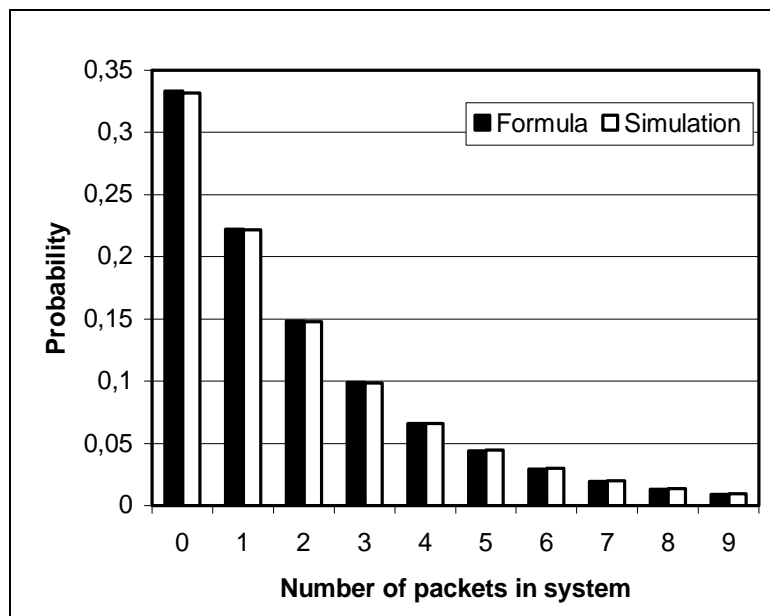


Figure 2-9: Job Distribution: simulation and theory compared

The results from the simulation were acquired by taking the average of the 50 runs. Although no confidence interval is given for the simulation results, it is clear that both bars are as good as equal. So, in this case, the simulation results comply with the theory.

3 TCP/IP

3.1 Introduction

The first outline for the Transfer Control Protocol (TCP) was given in 1974, by Bob Kahn and Vinton Cerf. They continued working on it during the 1970's, while the ARPAnet, a network developed in the United States intended to link research facilities, was starting to grow. This ARPAnet first used the Network Control Protocol (NCP) to operate the network, but in 1983 it was replaced by TCP in combination with the Internet Protocol (IP). The latter took care of the routing of messages in the network. This ARPAnet started to grow and also became available to home users, and soon the public part of this network was called the 'Internet'. The popularity of the Internet also contributed to the popularity of TCP/IP, which is one of the most used network protocols nowadays.

The TCP/IP protocol suite consists of four layers:

The first layer, the application layer, consists of applications that make use of the TCP/IP protocol stack, like FTP, telnet and web services (HTTP). The TCP protocol, which supports reliable data transport, is positioned in the transport layer. The UDP-protocol is also part of this layer. The network layer takes care of routing packets through the network. IP can be found in this layer, together with ICMP. The bottom Link layer consists of the physical network interface

Application
Transport
Network
Link

Figure 3-1: TCP/IP protocol suite layers

The focus of this report will be on TCP, located in the transport layer.

3.2 TCP

3.2.1 Internals

The Transmission Control Protocol supports reliable, connection-oriented data transfer between applications. It works on top of IP, which is an unreliable transport service. TCP uses a 20 bytes header (plus optional options) to manage connection setup, connection termination and data transfer. The layout of this header is shown in Figure 3-2. This header information is added to data that the application using TCP wants to send. But first, this data is divided into segments, each with its own header information. This data is then reassembled at the receiving side. All these segments are passed to the IP-layer, where another 20 bytes of header is added to the packet.

0	4	10	16	24	31
Source Port			Destination Port		
Sequence Number					
Acknowledgement Number					
HLEN	Reserved	Code Bits	Window		
Checksum			Urgent Pointer		
Options (if any)				Padding	
Data					

Figure 3-2: TCP header fields

TCP provides a reliable service. To realize this, several techniques are used to make sure that all data arrives in the end, and that TCP makes optimal use of the underlying network resources.

TCP makes use of the sliding window protocol. Each packet that is sent with TCP has a sequence number. The receiving side acknowledges every packet. Using the sliding window protocol, it is possible to send a variable number of packets before they are acknowledged.

TCP also uses advanced flow and congestion control. In order to use the link between two hosts to its full capacity, but not send more data than the network or the other host can handle, advanced flow and congestion control mechanisms are used. To do this, both sender and receiver have two 'windows': the receive and send window. The receive window is the amount of data the receiver can receive. This value becomes smaller when for example the buffer of the receiver is not emptied by the application yet. This value is sent in the 'Window' field of the TCP header. The sender keeps track of its own send window, called the 'congestion window'. The amount of data a sender can send is limited by the size of the receive window of the other host, and the amount of the data the network can handle (the congestion window).

The initial value of the congestion window is set to one segment. For every acknowledgement that is received, this window grows with one segment. So the sender starts with sending one segment. When an ACK is received for this segment, the congestion window grows to two segments. Two segments are sent then. When acknowledgements are received for these packets, the congestion window grows to four segments. This window grows thus exponential. This algorithm is called *slow start*.

When a packet is lost, indicated by an expiring retransmission timer or by receiving three duplicate acknowledgments, this congestion window is set to one segment (congestion avoidance) or half the congestion window plus three segments (fast recovery). This is further explained below.

3.2.2 Versions

The first version of TCP is described in RFC 793 [6] from 1981. In this specification, the basic protocol elements were specified, such as the sliding window algorithm, header format and a retransmission timer, which resends packets after no acknowledgements are received for a packet after some time. Shortly after this first specification several additions to TCP were published. The 'delayed acknowledgement' feature was introduced to limit the overhead of sending acknowledgments for every received packet [7]. When a receiver uses this technique, an acknowledgment does not have to be sent for every packet that is received. There are several circumstances in which an ACK has to be sent immediately [8]. In other cases, a timer is set and when it expires an ACK is sent nevertheless.

In the past 20 years, several other improvements have been made to the original specification:

A newer version, called TCP Tahoe, supports the *fast retransmit* [9] algorithm. This algorithm comes into play when the sending side receives four acknowledgments that acknowledge the same packet (three duplicate acknowledgments). When an acknowledgment is sent by the receiver, the 'acknowledgment number' field indicates the sequence number of the packet it wishes to receive next. If after that it receives a packet with a sequence number that is not the same as the sequence number asked for, it just sends the acknowledgment for this sequence number again. This indicates that somewhere a packet with a sequence number in between is delayed or lost. When a sender has received three duplicate acknowledgments for a packet with sequence number X, it just resends this packet. Also, it adjusts its value for the congestion window to one packet. When other ACK packets are received, this value grows exponentially to a certain threshold (half of the congestion window), and after that it again grows linearly (*congestion avoidance*).

After that TCP Reno was introduced. TCP Reno added *fast recovery* [10] to the *fast retransmit* algorithm. When three duplicate ACKs are received, the congestion window is cut in half, rather than setting it back to one packet. This value is saved in the variable *ssthresh*. For all other duplicate ACK packets that are received the congestion window grows exponential. When a new ACK is received (not duplicate), the congestion window is set back to the value of *ssthresh* and after that it grows linear.

TCP NewReno is the same as TCP Reno, but resends more than one packet after three duplicate ACKs are received.

TCP Sack is another additional feature that improves performance under heavy congestion with multiple packet losses. TCP Vegas has several other improvements, but only performs better in homogenous settings, and worse when working with for example TCP Reno, the most used TCP version nowadays.

3.3 Areas of interest

TCP was designed to support maximum throughput over many kinds of different networks. Many improvements that have been done over the past years were targeted at improving this throughput. With a simulator, it is quite easy to determine the throughput with different version of the TCP protocol stack. It is also possible to change other factors in the simulator, like line speed and delay. TCP should work in many different setups. One of those setups is with an asymmetric connection. In this setup, the downstream channel is much faster than the upstream channel. Using a simulator, it is possible to show how TCP reacts under these circumstances.

4 TCP/IP-implementation in OMNeT++

4.1 Existing implementation

The Internet Protocol Suite (IPSuite) [11] provides IPv4, TCP and UDP models for simulations in OMNeT++. It was first developed by several people at the University of Karlsruhe [12]. In the end of 2003, the author of OMNeT++, Andras Varga, took over the maintenance of the IPSuite package. He also created more documentation for the package, which makes it easier to use and make adjustments to the model. At the same time, a model for IPv6 was also created [13].

The whole IPSuite model is modular. All the components of a real life TCP setup are divided into a large number of different modules. Each module can have several parameters. In Figure 4-1, a simplified model of the complete TCP model is given. The complete model can be found in [14]. All these different modules are defined in the .ned files, as explained in paragraph 2.1.2. The complete TCP finite state machine is implemented by these modules.

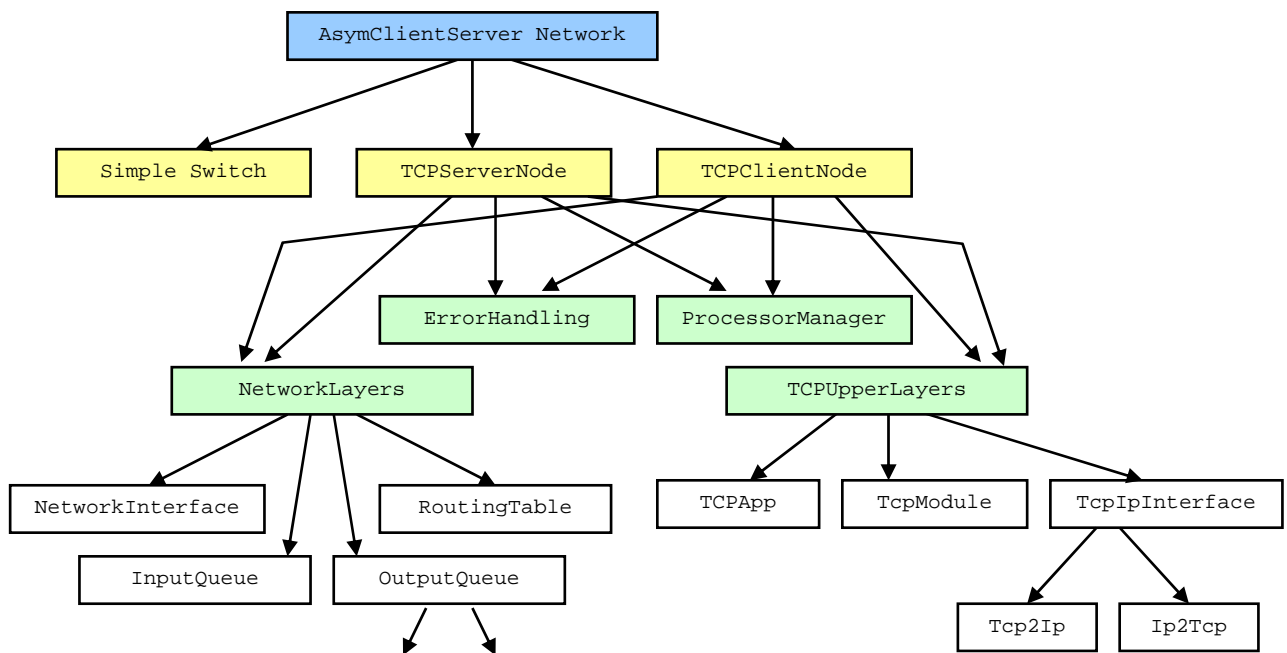


Figure 4-1: TCP model in OMNeT++

The top module is ‘AsymClientServer Network’. This module models a simple client-server network, in which the upload and download bandwidth can be set separately (asymmetric bandwidth). This network consists of a single client (TCPClientNode) and a server (TCPServerNode), which are more specific instances of the general IPNode module. The simple switch connects these two nodes. All this switch does is send packets to the other host, and drop packets on the downstream link (from server to client) with a given probability (module parameter).

The two TCP nodes both have four sub modules. The ErrorHandling module takes care of error notifications that come from other protocol modules. These messages are printed and then discarded. The ProcessorManager simulates the processor usage of a TCP node. Using this node it is possible to set a processor delay for every IP-packet that has to be processed. The NetworkLayers module models the network and lower layers. The NetworkInterface module simulates the actual network interfaces, and routing is done in the RoutingTable module. There are also two modules for in and output queueing. The OutputQueue module has more sub modules, but they are not shown here.

The real TCP stack can be found in the TCPUpperLayers module and its sub modules. The application that makes use of TCP is modelled in TCPApp. In TcpIpInterface and its sub modules the transfer of TCP packets to the IP layer (Tcp2Ip) and IP packets to the TCP layer (Ip2Tcp) takes place. The concrete implementation of the TCP protocol can be found in TcpModule. All the aspects of TCP, like retransmit timers and the sliding window algorithm, are all implemented in this single module.

All basic aspects of the first TCP version are implemented in this module. It is possible to turn several features on or off. These features are delayed acknowledgements, fast retransmit, fast recovery. With all these features turned on, the TCP implementation acts as a TCP Reno implementation. It is also possible to use the TCP NewReno implementation.

4.2 Changes

All changes made to the IPSuite were done in the TcpModule, where the TCP implementation is located. While trying to use the TCP model for several simulations, several faults in the existing TCP implementation showed up. Several features did not work as expected, and some aspects just were not implemented. In the rest of this chapter the changes to the TCP model will be shown.

4.2.1 Tcpdump output

The existing TCP module could produce a lot of debugging information, about transitions in the TCP FSM and the different values in the transmission control block (TCB), which is used to store connection information, but this information was not useful when trying to find errors in the TCP implementation. In [15], the output of the Unix program called ‘tcpdump’ is used to show how TCP works.

Since the TCP model in OMNeT++ is a full TCP implementation, it was also possible to make the model output its debug information in the same format as the ‘tcpdump’ program. An example of this is shown in Figure 4-2. In this example, the tcpdump output for a server is shown (address 167772161), to which a client (167772162) connects. The server then sends 4580 bytes of data and the client then terminates the connection.

The first field is the line number. Then the IP address in decimal notation of the sending host (with port number, 0 in this case) is shown, followed by the receiving host and port. Then, several flags in the TCP header are shown, if set. These are stored in the ‘code bit’ field in the TCP header in Figure 3-2. For some transfers, the initial sequence number for the packet is shown, together with the implied ending sequence number, depending on the number of data bytes in the packet. If the ACK flag is set, ‘ack’ together with the sequence number that is acknowledged is printed. After that, win <number> indicates the size of the receive window being advertised by the sender. Since the sender receives no data, this window stays the same all the time. Finally, on the first two lines, options of the connection that are negotiated during the connection setup are shown between <>.

```

1 167772162.0 > 167772161.0 S 0:0(0) win 65535 <mss 1460>
2 167772161.0 > 167772162.0 S 10695:10696(1) ack 1 win 65535 <mss 1460>
3 167772162.0 > 167772161.0 . 1:2(1) ack 10696 win 65535
4 167772161.0 > 167772162.0 . 10696:12156(1460) ack 1 win 65535
5 167772162.0 > 167772161.0 . ack 12156 win 65535
6 167772161.0 > 167772162.0 . 12156:13616(1460) ack 1 win 65535
7 167772161.0 > 167772162.0 . 13616:15076(1460) ack 1 win 65535
8 167772162.0 > 167772161.0 . ack 13616 win 65535
9 167772161.0 > 167772162.0 P 15076:15276(200) ack 1 win 65535
10 167772162.0 > 167772161.0 . ack 15076 win 65535
11 167772162.0 > 167772161.0 . ack 15276 win 65535
12 167772162.0 > 167772161.0 F 1:1(0) ack 15276 win 65535
13 167772161.0 > 167772162.0 . ack 2 win 65535
14 167772161.0 > 167772162.0 F 15276:15277(1) ack 2 win 65535
15 167772162.0 > 167772161.0 . 2:3(1) ack 15277 win 65535

```

Figure 4-2: Tcpdump output from simulation

In lines 1, 2 and 3, the connection setup is shown. The client sends a SYN segment to the server, with an initial sequence number (ISN), which is 0. The server responds to this with another SYN segments, with its own ISN, and includes an ACK in that segment for the received SYN. Finally, the client sends an ACK for the SYN segment the server sent. The connection is now established. In the next lines, the server sends data in packets of 1460 bytes. Delayed acknowledgments are turned off here, so an acknowledgement is sent for every packet. The last data is sent together with the P (push) flag turned on. The client then terminates the connection. This connection termination takes 4 packets (lines 12-15).

4.2.2 Delayed Acknowledgements

There were some problems with the TCP implementation when the delayed acknowledgement feature was turned on.

TCP does normally not send an ACK immediately to acknowledge received data. Instead, it makes use of delayed acknowledgements. This comes from the fact that TCP just waits for data to send to which the acknowledgement can be attached (the so called piggybacking). When this does not happen for some time, a timer goes off and the ACK is sent eventually. An ACK is sent immediately when an out of sequence packet (a packet that has a different sequence number then expected) is received. There is another mechanism that triggers the immediate sending of an ACK. When there are two outstanding (not acknowledged packets), TCP sends an ACK immediately. So, when two consecutive packets are received, an ACK is sent after that.

The timer that scheduled the transmission of the delayed ACK packets was implemented correctly, but the mechanism that sends an ACK when two not yet acknowledged consecutive packets were received was not present. I've added this to the TCP module, by using a counter that increments for every packet that is received for which no immediate ACK is sent. When this counter is equal to 2, an immediate ACK is sent. This counter is reset when the delayed ACK timer goes off, or an immediate ACK is sent for another reason.

There is another reason to send an immediate ACK. A TCP node sends duplicate ACK packets when a certain packet is lost. When this packet that is asked for is received eventually, an immediate ACK should be sent to get the data transfer going again. I've implemented this by registering the sequence number for which three duplicate ACKs are sent at the sender side. When this packet is eventually received, an immediate ACK is sent.

4.2.3 Persist Timer

The *persist timer* mechanism was not present in the TCP implementation. TCP normally performs flow control by letting the receiver specify the amount of data it is willing to receive (the window size). When this window is equal to 0, the sender stops sending, but eventually, the sending process should start again. If the receiver sends an ACK with a window size greater than zero, the sender should start sending again. But, if this ACK is lost, the receiver waits for data, but the sender still thinks the receiver's window is zero. This is a deadlock situation.

The persist timer solves this. This timer is set when the other node advertises a window size of 0. The timeout period is calculated using the normal TCP exponential backoff, which is about 1.5 seconds for a typical LAN for the first timeout. This value multiplied by 2 for the second timeout, by 4 for the third, etcetera, but is always bounded between 5 and 60 seconds. When this timer expires, 1 byte of data is sent. The receiver responds to this by sending an ACK, either with 0 or greater window size.

This additional timer is implemented the same way the other timers are implemented in the model (i.e. the retransmit timer). The exponential backoff mechanism isn't used yet; the timer has a fixed value.

4.3 Further improvements

Despite the changes described above, there are still some errors left in the TCP implementation. The results from the simulations in the next chapter will show this. It was not possible to find and fix all the errors during this assignment.

Also, the exponential backoff of the persist timer has to be implemented. Also, new variants of TCP like TCP Sack could be implemented.

5 Analysis of TCP/IP in OMNeT++

5.1 TCP Friendly

5.1.1 TCP Friendly formula

TCP uses the Congestion Avoidance algorithm to estimate the amount of bandwidth available between two Internet hosts. This algorithm was added to the first version of TCP [6]. This algorithm slowly increases the congestion window, whenever an acknowledgement is received from the other end of the connection. When a loss is suffered (this indicates a congestion), this window is decreased by half.

It is possible to derive an equation for the available bandwidth between two hosts when you know all the aspects of this algorithm [16].

$$Bandwidth = \frac{1.22 * MTU}{RTT * \sqrt{Loss}}$$

In which MTU is the packet size used (in bytes), RTT the round trip time in seconds and Loss is the loss rate on the downstream of the connection. Bandwidth is then given in bytes per second.

This formula has been verified by simulations for loss rates up to 5% [17]. At high loss rates the formula overestimates the bandwidth. In the next paragraph this formula will be held against results from the simulation.

5.1.2 Simulation Setup

The following simple network set up was used to verify the TCP friendly formula in the simulation:

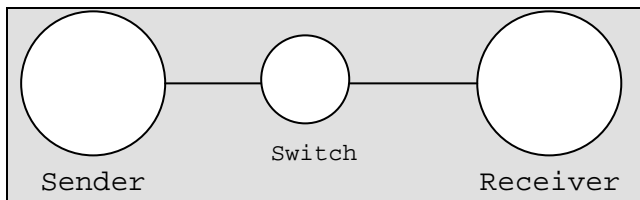


Figure 5-1: Layout simulated network

The sender sends an amount of data to the receiver. The RTT (round trip time) is fixed (0.06s = 60ms). The element between the sender and receiver drops packages randomly on the *downstream* link (from sender to receiver).

There are a lot of different TCP versions and each version has several distinct features which can be turned on or off in the simulation. In this simulation TCP Reno was used. This is the standard TCP protocol with several additional features. These features are slow start, congestion avoidance, fast retransmit (all included in TCP Tahoe) and fast recovery. The delayed acknowledgment feature was turned off.

For each amount of packet loss, ten simulations were run, each with a different random seed. The amount of packet loss was increased by 0.1% each run. The amount of data sent by the sender was equal in every simulation (1000000000 bits, approximately 120 megabytes). This should be enough to get reliable results. The throughput was calculated by dividing the amount of data sent by the time that passed between the opening of a TCP connection by the client and the closing of this connection. It also takes some time to set up and tear down the connection, and no data is sent then, but this is negligible.

The link bandwidth was set to 1 gigabit per second. This bandwidth is much larger than the theoretic maximum throughput, but this is just to be sure the available link bandwidth isn't the bottleneck. The receive window was set to maximum, 65535 bytes (64 KB). So in one round trip time period (60 ms), a maximum of 64 KB can be sent. This comes down to $64 * \frac{1000}{60} = 1066,67$ KB per second. This is the maximum throughput, as limited by the receive window, but other factors can be the bottleneck too.

5.1.3 Results

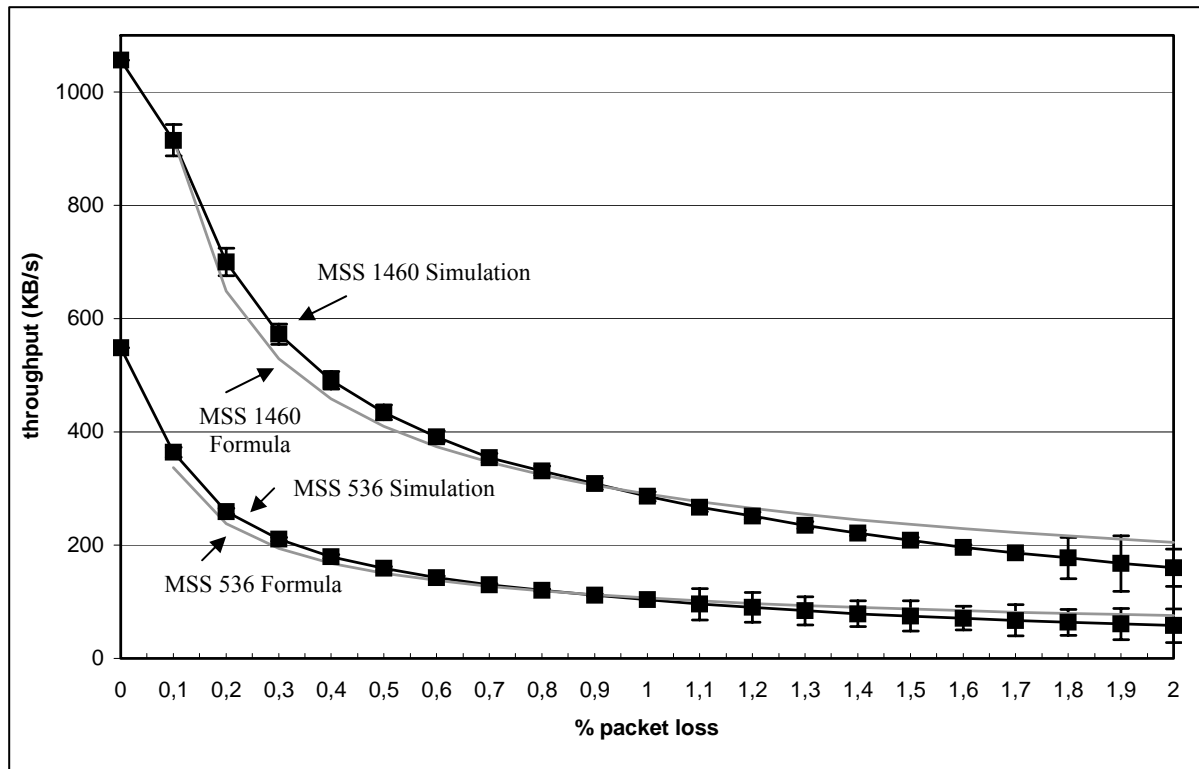


Figure 5-2: Throughput results from simulation

The black lines, marked with squares, indicate the measured throughput, as acquired from the simulation. Next to each black line there is a slim brighter line that indicate the expected throughput as calculated with the TCP-Friendly formula. The upper pair of lines displays the throughput with a large maximum segment size (MSS) with 1460 bytes per packet. With the lower pair of lines a small packet size of 536 bytes was used.

The vertical lines that accompany the simulation results indicate the 95% confidence interval. The calculation of this is explained in paragraph 2.2.2. The meaning of this interval is that there is a 95% probability that in a real situation on the Internet, with the same link parameters as were used here, the average throughput will be between the possible values indicated by the confidence interval as indicated in Figure 5-2. This interval is larger for the smaller packet size when the packet loss exceeds 1.1 percent. This is because the amount of samples (successful simulations) is less there. This expands the interval. The confidence interval is almost 0 when there is no packet loss. Since no packet loss occurs, the throughput found is the same in all simulation runs. This results in a very small (close to 0) confidence interval.

The simulation results differ from the formula when the loss rate is greater than 1,5 percent. This difference can probably be attributed to the not fully correct implementation of the error handling mechanism.

The first simulation outcomes showed a large gap between the formula and the simulation with low loss rates when using the large MSS. The cause of this seemed to be the limitation of the receiver window by the TCP implementation. The maximum window size was set to 64 KB, but there was a variable called `rcv_buf_usage_thresh` in the `tcpmodule` that somehow limited the maximum receive buffer usage at only 50%. Setting this threshold to 100% did not lead to the expected throughput, but doubling the maximum receive window size to 128 KB did. In normal TCP, it is not possible to advertise a window greater than 64 KB, since there are only 16 bits available for this in the header. It is however possible to use the window scale option [18], but this is not implemented in the OMNeT++ TCP model. Fortunately, a TCP header in the simulation is just a struct, in which the receive window variable is of the type `long`, which is 32 bits. So it is possible to use a receive window greater than 64 KB. From zero to 0.5% packet loss this larger receive window was used, with the larger packet size (1460 bytes).

5.1.4 Implications

The graph in paragraph 5.1.3 only shows packets loss from zero to two percent. The simulation was also run for packet loss up to five percent. The higher the packet loss, the smaller the amount of successful simulations was. Because of the not fully working TCP implementation, the simulation with higher packet loss rates just stopped (i.e. there weren't any packets sent/received) after a certain amount of time. Because of this, the results from the simulations that ran successful weren't very reliable, because of the small amount of data. It is presumable that only the simulation with the 'lower than average' packet loss percentage succeeded, so this would also contribute to the unreliability of the data.

The gap between the results and the formula was really large with high loss rates. The throughput from the formula was about two times as high as the throughput from the simulation. It is known that the formula overestimates the throughput for higher loss rates, but this should only be true for loss rates higher than 5 percent. So this is clearly an error caused by the incorrect implementation of TCP in OMNeT++. But, between zero and two percent packet loss, the simulation is working according to the formula.

5.2 Asymmetric connection

5.2.1 TCP and asymmetric connections

A lot of today's Internet connections that home users have are in one or more ways asymmetric. These connections can be asymmetric by available bandwidth, loss rate or access time. The most frequently used is the asymmetry by bandwidth. Examples of this are asymmetric digital subscriber line (ADSL) technology and cable modems. The download/upload ratio can be as large as 25:1. This means that the downstream bandwidth can be 25 times as large as the available upstream bandwidth.

The execution of TCP is affected by this asymmetry, for example during unidirectional data transfer. Especially when there are losses on the downstream link, the overall throughput suffers from this asymmetry [19]. TCP uses acknowledgments to establish a reliable connection. When these acknowledgments can't be sent fast enough, the sender will not be able to use the full bandwidth of the downstream link.

With asymmetric connections, you can define a *normalized bandwidth ratio* [20], k . This is the ratio between the down- and upstream bandwidth, divided by the packet size ratio used in both directions. So, let's calculate this ratio when the downstream bandwidth is 10Mbit/s and the upstream bandwidth is 28,8kbit/s. The packets on the downstream path are 1460 bytes and the acknowledgment-packets on the upstream-channel are 40 bytes (20 bytes IP-header, 20 bytes TCP-header and no data). We will use these values to simulate a connection in the following paragraphs.

$$k = \frac{\frac{10 * 10^6}{28,8 * 10^3}}{\frac{1460}{40}} = \frac{347 \frac{2}{9}}{36 \frac{1}{2}} \approx 9,5$$

Figure 5-3: Normalized bandwidth calculation

In Figure 5-3 the normalized bandwidth ratio is calculated. The proportion of the down and upstream bandwidth is divided by the proportion of the packet sizes. In our case, k equals 9,5. This means, that if the receiver sends more then one ACK every 9,5 data packets it has received, the upstream link will be full before the downstream channel is used to it is full capacity. If this continues for some time, the send buffer at the receiving side will get full (with ACK packets) and packets from this queue will be dropped. It will then take even more time before the downstream bandwidth is fully used, because the sender's congestion window is increased only each time an ACK is received.

5.2.2 Test results: Throughput

Several test with a asymmetric connection were done using the TCP-model in OMNeT++. Downstream bandwidth was set to 10Mbit/s and upstream bandwidth to 28.8kbit/s. The propagation delay for both channels was set to 0.03 seconds (30 ms). Pkackets were dropped on the downstream link. As in the previous simulation, 1000000000 bits (approximately 120 megabytes) were sent from sender to receiver via the downstream link. The receiving side did only send ACK-packets, and no other data besides that.

Two different packet sizes were used: 536 and 1460 bytes. There were four different test runs, two for each packet size, one with delayed acknowledgment turned off, and one with this feature turned on. TCP Reno was used for all test runs.

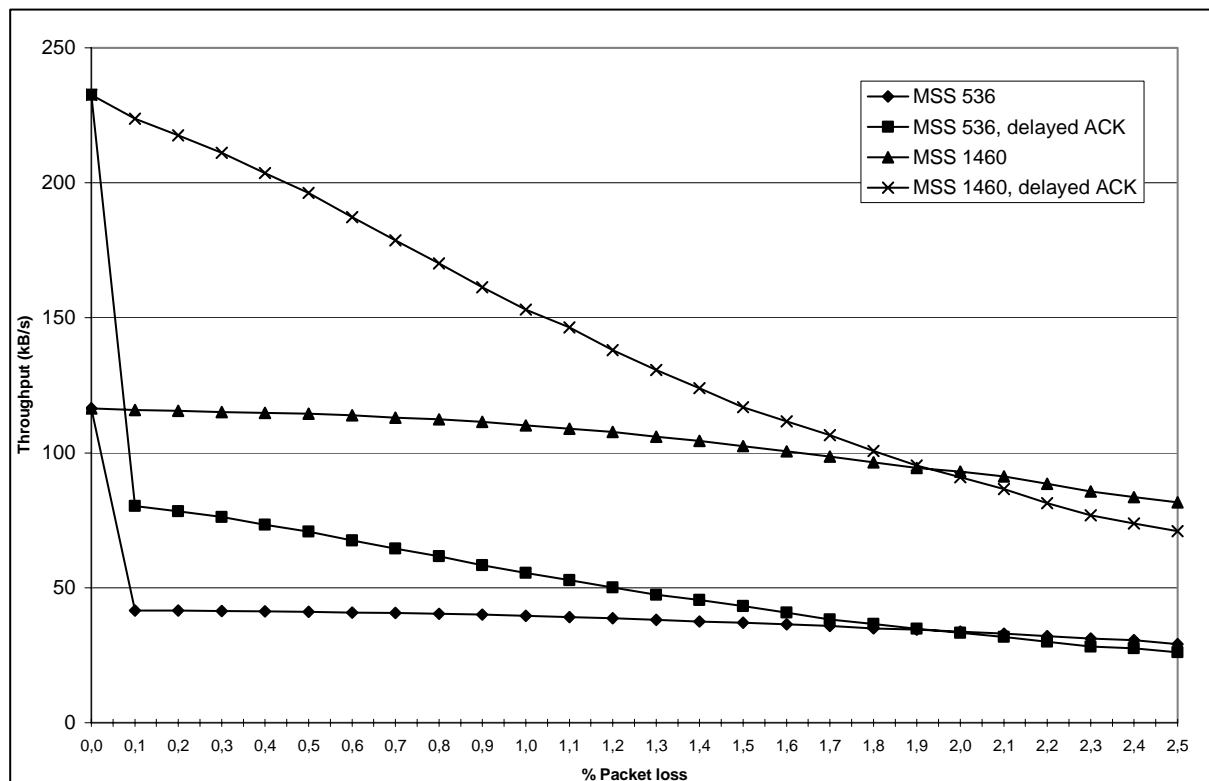


Figure 5-4: Throughput Asymmetric connection

There is a big difference in speed between the two packet sizes. Since the growth of the congestion window is limited, only a limited amount of packets can be sent. The larger the amount of data per packet is, the higher the throughput. The throughput is also larger when delayed acknowledgments are used, because only half the amount of ACK-packets is sent. The upstream channel is only half as much congested, so the congestion window can grow faster. But the most important reason for the lower throughput is the fact that when the maximum amount of packets (size of congestion window) is outstanding (not acknowledged yet), no other data can be sent until an acknowledgement arrives. So if the congestion window has grown to it is maximum, it is still no guarantee for a higher throughput.

5.2.3 Test results: Congestion window

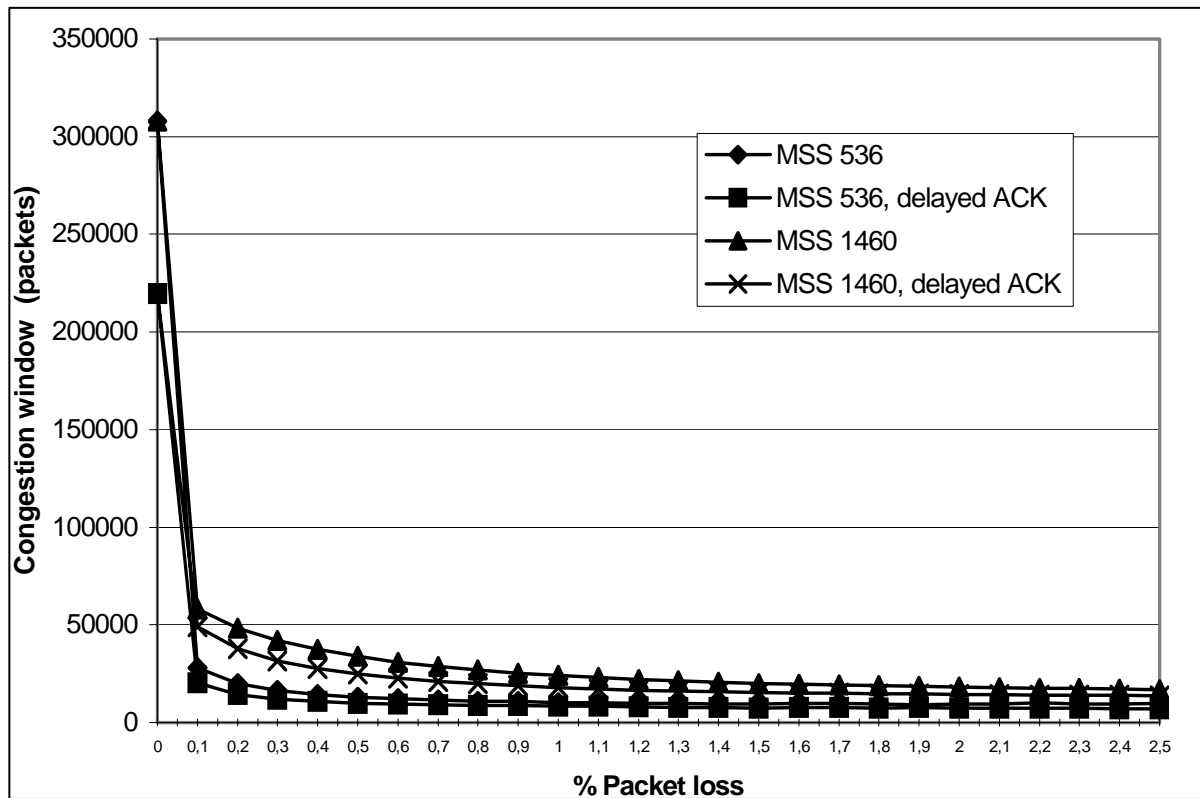


Figure 5-5: Average congestion window asymmetric connection

The graph above shows the average congestion window, as recorded at the sender. Recall, that the senders congestion window increases for every acknowledgement that is received. The window becomes smaller when a loss is suffered. So, when there is no packet loss at all, this window can grow unlimited. This is shown in the graph by the high peak at zero percent packet loss.

5.2.4 Test results: Miscellaneous

All the previous graphs were made by collecting data from the OMNeT++ simulation ‘by hand’, after the simulation was finished. It is also possible to let OMNeT++ record data during the simulation. This data is then stored in so called output vectors. In these vectors, a timestamp is stored together with data from that time. You can use multidimensional vectors, storing multiple data fields at the same moment.

These output vectors can be visualised by a tool called Plove, which is included in the OMNeT++ distribution. This tool uses Gnuplot [21] to plot the recorded data.

In Figure 5-5, the average congestion window was shown. It is also possible to show this window at certain point in time. The congestion window was recorded into an output vector, and visualized using Plove. The results of this can be seen in Figure 5-6.

6 Conclusion

The goal of this assignment was to get to know the OMNeT++ simulator environment, by experimenting with different models. This was done using the M/M/1 queue simulation first, and after that the IP Suite, which holds models for IP and TCP, was thoroughly investigated.

The TCP/IP model in OMNeT++ proved to be working, but not free of errors. Some features were missing, and some were not correctly implemented. All TCP behavior was coded into one single module (tcpmodule), which made it hard to understand and change this implementation. For future version, it might be possible to split TCP functionality into several modules.

Several improvements were made to the TCP implementation. These improvements are also adopted by the maintainers of the IPSuite package and are available in the latest version. Considering the current status of the model, and the ongoing development to the IP Suite package, it is valid to conclude that the IP Suite package is useful for simulation and exploring basic TCP/IP behavior.

OMNeT++ was the main subject of this assignment. Because two simulations were looked into, it was possible to get a good impression of the program. The separation of the description of the network layout and the actual implementation of the nodes of this network is a positive point of the program. The description of the network is done in the simple structured NED language and for the implementation of the nodes the C++ libraries provides all the flexibility and functionality needed. The graphical interface is very useful for showing the network layout and the exchange of messages in the network. For simulation development purposes however it is not often used.

So, is OMNeT++ a useful simulator? Yes, and maybe no. It has got a lot of useful functions and a lot of people are contributing to the open source packet. But not everyone is going to build his or her simulation from scratch. They want to use existing 100% working models to build their simulation on. There are already a number of models present for OMNeT++, for example the IPv6Suite and models for 802.11. If this list of models keeps growing, OMNeT++ can become even more successful.

7 References

1. Varga, A., OMNeT++, <http://www.omnetpp.org>
2. ns-2, <http://www.isi.edu/nsnam/ns/>
3. OPNET, <http://www.opnet.com/>
4. Foreest, N.v., Queues in OMNeT++, <http://whale.hit.bme.hu/cgi-bin/contrib.pl?dir=models&txt=Queues-2.0>
5. Jain, R., (1991). *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling*, New York: Wiley. xxvii, 685 p.
6. Institute, I.S. and U.o.S. California, (1981). RFC 793: Transmission Control Protocol.
7. Clark, D.D., (1982). RFC 813: Window and acknowledgement strategy in TCP. MIT Laboratory for Computer Science, Computer Systems and Communications Group.
8. Braden, R., (1989). RFC 1122: Requirements for Internet Hosts - Communication Layers. Internet Engineering Task Force.
9. Allman, M., V. Paxson, and W. Stevens, (1999). RFC 2581: TCP Congestion Control (Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms).
10. Jacobson, V., (1990). Modified TCP Congestion Avoidance Algorithm, in *end2end-interest mailing list*.
11. IPSuite: Simulation Models for IPv4 Networks, <http://ctieware.eng.monash.edu.au/twiki/bin/view/Simulation/IPv4Suite>
12. Kaage, U., V. Kahmann, and F. Jondral, (2001). An Omnet++ TCP Model.
13. Lai, J., et al., IPv6Suite, <http://ctieware.eng.monash.edu.au/twiki/bin/view/Simulation/IPv6Suite>
14. Varga, A. (2004). IPSuite documentation. 2004.
15. Stevens, W.R. and G.R. Wright, (1994). *TCP/IP illustrated*. Addison-Wesley professional computing series, Reading, Mass.: Addison-Wesley Pub. Co.
16. Mahdavi, J. and S. Floyd, (1997). TCP-Friendly Unicast Rate-Based Flow Control.
17. Floyd, S. and K. Fall, (1997). Router mechanisms to Support End-to-End Congestion Control.
18. Jacobson, V., R. Braden, and D. Borman, (1992). RFC 1323: TCP Extensions for High Performance.
19. Balakrishnan, H. and V.N. Padmanabhan (2001). How Network Assymetry Affects TCP, in *IEEE Communications Magazine*. 2001. p. 60-67.
20. Lakshman, T.V., U. Madhow, and B. Suter. Window-based Error Recovery and Flow Control with a Slow Acknowledgement Channel: A Study of TCP/IP Performance. in *IEEE INFOCOM*. April 1997. Kobe, Japan.
21. Gnuplot, <http://www.gnuplot.info/>