Mice and Elephants

Ioannis Giannoulatos `<ioannis.giannoulatoss@os3.nl>`

Master *System and Network Engineering*
University of Amsterdam, The Netherlands

March 21, 2014

Abstract

Next generation networks with large bandwidth and long delay, also known as Long Fat Network (LFN) pose a major challenge to Transmission Control Protocol (TCP) performance. Specifically, sustaining the link utilisation of these networks at a high level is quite challenging, especially when large (elephant) flows are intermixed with small (mice) flows. A small amount of packet loss can cause a huge performance loss (1 out of 22,000 lost packets causes 80x reduction in data transfer, when the Round Trip Time (RTT) is more than about 10 ms). Under supervision of Ronald van der Pol and Marijke Kaat from SURFnet, this research investigates the several configuration parameters of the TCP network stack of the Linux kernel. Moreover, it will be investigated how these configuration choices in correlation with tc(8) traffic shaping in Linux Operating System (OS) can provide better overall throughput of the link.

Acknowledgements

I would like to strongly thank my Supervisors Ronald Van der Pol and Marijke Kaat for their valuable guidance and assistance that they have provided in order for this project to be completed. Furthermore, I would like to express my gratitude to the University of Amsterdam and particularly the System and Network Engineering master that provided me the opportunity to complete this research.

Ioannis Giannoulatos

Contents

Chapter 1

Introduction

Next generation networks with large Bandwidth-Delay Product (BDP) pose a major challenge to TCP performance. Moreover, due to the requirement to carry a wide variety of network services and traffic types, sustaining high link utilisation and throughput can be quite challenging especially when large(elephant) flows are intermixed with small(mice) flows. To alleviate this problem a lot of research has been carried out [2] [3] [4] [5].

A small amount of packet loss can cause a huge performance loss (1 out of 22,000 lost packets causes 80% reduction in data transfer) [6]. The most important reason for high packet loss rate is the behaviour of TCP's congestion control algorithm. The main problem is trying to maximise the throughput of the shared link until packet drop occurs. When multiple flows traverse the shared link all of them are competing for the available bandwidth. Furthermore, not optimising the configuration parameters of the TCP network stack on the end-hosts can result in losing packets due to the numerous -but limited- buffers of the kernel.

To alleviate the problem of TCP connections experiencing alarming high loss rates, even after having undergone a number of enhancements (Hamilton TCP (HTCP), Cubic), this research project evaluates the usage of queue management mechanisms that already exist in the Linux kernel of the end-hosts.

This research paper is structured as follows. In Chapter 2 the theoretical background needed in determining which factors affect the throughput of the link is analysed.Furthermore, the effect that packet loss has on throughput is evaluated and more insight is provided on the Traffic Control (TC) mechanisms that exist in the Linux Kernel. In Chapter 3 the experiments are divided into three sections that further investigate the subjects mentioned. Lastly, in Chapter 4 the conclusions are listed and further research is mentioned.

1.1 Research Question

For this Research project, I worked on answering the following research questions:

How can we achieve constant throughput and high utilisation of the link, while intermixing small and large TCP flows?

This research question leads to the following sub-questions:

- What changes should be made in the configuration of the Linux TCP network stack to achieve the highest throughput?
- What effect does packet loss have on throughput?
- How can already existing TC techniques be used in order to provide a better throughput on the link with less packet loss?

1.2 Approach

During this research project the main objective was to achieve high utilisation and overall throughput of the provided link between two end-hosts.Therefore the approach could be divided into 4 subcategories:

- Configuration of the end-hosts so that we could achieve the highest overall throughput. In order to achieve the highest possible utilisation and throughput throughout our experiments configuration of several parameters on the end-hosts is used.
- Explore the effect of packet loss on throughput in order to verify the theory and evaluate how packet loss affects throughput in LFN.
- Use existing Traffic Control mechanisms in order to create a single constant TCP flow of limited throughput.

- Evaluate the possible advantages of using Traffic Control mechanisms for more than one flow.

However, my research was limited by the following assumptions :

- Complete knowledge of the flows that exist in the network. If the flows that traverse the networks are not known then TC policies cannot be created effectively.
- Focusing on networks with large BDP a.k.a LFNs.
- Usage of already embedded Traffic Control mechanisms in the Linux kernel.

Chapter 2

Background

Our aim in this chapter is to provide the theoretical background needed to create an understanding of how TCP flows should behave in LFNs and therefore be able to create a baseline for our observations during the experimentation.

Section 2.1 provides an overview of the TCP protocol. Because in these research TCP flows are created and evaluated, giving a short introduction of several key aspects of TCP is essential. In LFNs as stated in the Introduction (1) a small amount of packet loss can drop the throughput of a link significantly.. Therefore, in sections 2.1.1 and 2.2 the mechanism that is included in the TCP protocol is described and how packet loss could affect the throughput of a link theoretically. Lastly, this research also evaluates how already existing Traffic Control techniques are performing in LFN and ,as a result, a detailed analysis is provided in section 2.3.

2.1 TCP

TCP/Internet Protocol (IP) is the most important networking protocol suite in the world. It is the basis for the Internet and the language spoken by the vast majority of the world's networked computers. As depicted in 2.1, it is divided in four layers. Furthermore, it can be noticed that many protocols reside in each layer. The TCP protocol resides in the transport layer.

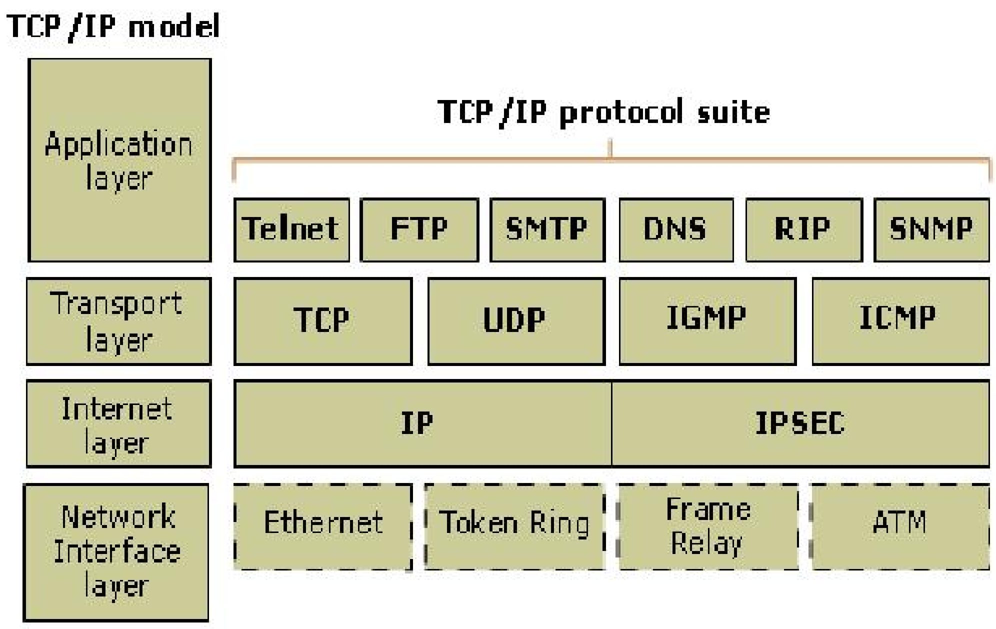


Figure 2.1: TCP/IP network stack and protocols that reside in each layer

TCP is a connection-oriented, reliable transport protocol for TCP/IP applications. It guarantees reliable delivery, and therefore, TCP sometimes incurs relative long delays while waiting for out of order messages or retransmissions of lost messages. A technique known as positive acknowledgement with retransmission is what enables TCP to create a reliable stream of data between two end-hosts. This technique requires the receiver to respond with an acknowledgement message as it receives data. The key features that set TCP apart from User Datagram Protocol (UDP) which is another transport protocol are:

- Ordered data transfer. TCP verifies by using timestamps and sequence numbers that packets are received in the right order.
- Retransmission of lost packets.
- Error-free data transfer. TCP verifies that data are delivered without errors.
- Flow Control. In a connection between two hosts, one host tells the other the number of bytes it is willing to receive from the former one;

this is its receive window, which becomes the other's send window. Managing the size of these windows is what is called Flow Control in the TCP protocol.

- Congestion control.(section 2.1.1)

2.1.1 Congestion Control

Congestion control is the mechanism adopted by TCP in order to achieve high performance and avoid congestion collapse. Congestion collapse is a condition where little or no useful communication is happening due to congestion. This mechanism is used in order to keep the data flow that enters the network below a certain rate that would cause congestion and therefore, packet loss. Furthermore, it is used in order to achieve fairness when multiple TCP flows exist in the network. Modern TCP implementations contain four intertwined algorithms [7], [8]:

- Slow-Start. This is the initial phase during a TCP data connection. In this phase the sender's window is increasing rapidly until a congestion is detected on the link.
- Congestion avoidance. This allows the device to drop the rate at which segments are sent quickly when congestion occurs.
- Fast retransmit. This is an enhancement to TCP which reduces the time a sender waits before retransmitting a lost segment.
- Fast recovery. In the Fast Recovery algorithm, during Congestion Avoidance mode, when packets (detected through 3 duplicate Acknowledgments) are not received, the congestion window size is reduced to the slow-start threshold, rather than the smaller initial value.

Congestion control has gone through extended research in order to achieve higher link utilisation and faster convergence when multiple TCP flows traverse the link. Several implementations of TCP exist that perform better than the initial Tahoe implementation [9] . Such implementations are newReno, htcp [10] and Cubic [11].

2.2 TCP Throughput Testing

A practical methodology for measuring end-to-end TCP throughput in a managed IP network is described in RFC6349 [12]. According to the methodology proposed by Constantine, et al. in RFC 6349 [12] in order to evaluate the TCP throughput of one or more TCP flows, many variables are involved. This research, as the mentioned RFC, focuses on these:

- Bottleneck Bandwidth : The lowest bandwidth along the complete path.
- Round-Trip Time(RTT) : The elapsed time between the clocking of the first bit of a TCP segment sent and the receipt of the last bit of the corresponding TCP Acknowledgement.
- Send and Receive Socket Buffers : The Socket Buffers must be large enough to fill the BDP. BDP refers to the product of a data link's capacity (in bits per second) and its end-to-end delay (in seconds).
- Path Minimum TCP Receive and Send window (RWND and SWND): As mentioned in section 2.1,TCP is connection oriented, and at the transmitting side, it uses a congestion window (TCP CWND). At the receiving end, TCP uses a receive window (TCP RWND) to inform the transmitting end on how many bytes it is capable of accepting at a given time.
- Path Maximum Transmission Unit(MTU) : MTU is the Maximum Transmission Unit which is the largest packet (layer 3, including the layer 3 header) that can be transmitted.
- Packet loss

In the experimental setup, bottleneck bandwidth and RTT are static. Furthermore, Send and Receive socket buffers and path's MTU are going to be configured so that the maximum throughput can be achieved on the link. This is essential in order to measure the effect of packet loss while checking how the sender's congestion window changes.

A formula was provided by Mathis et al. at [13] which predicts how throughput is affected by packet loss theoretically. This provides a baseline for this research's experiments. The formula is :

$$Rate <= (MSS/RTT) * (1/\sqrt{p}) \tag{2.1}$$

In this equation MSS stands for Maximum Segment Size(bits), RTT is the Round Trip Time(s) and p is the percentage of packet loss that exist in the network. This equation will be used during our experiments to see whether our environment is responding in accordance to it.

2.3 Traffic Control [1]

Traffic Control (TC) is the process of managing, prioritising, controlling or reducing the network traffic. In this research,the effect on throughput of one or more TCP flows is evaluated when TC is used in a Long Fat Pipe (LFP). Linux embeds a module that provides this functionality. The module resides between the IP Layer and the network interface driver as depicted in figure 2.2.

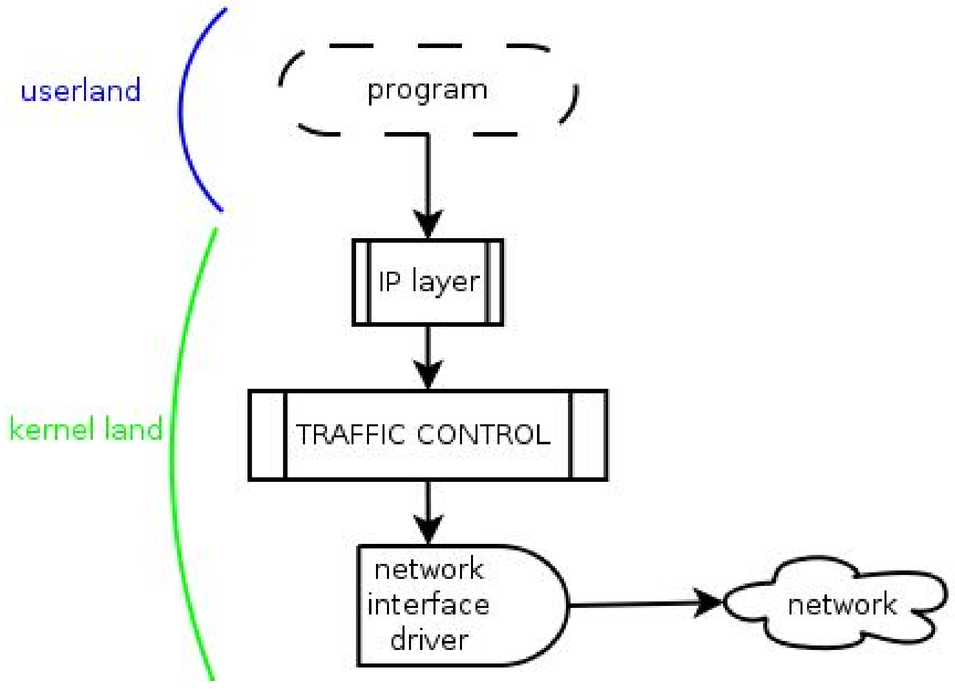


Figure 2.2: Traffic Control

The TC module is used to control the dequeueing of the packets from the sender's buffer that are transmitted on the network. At the core, it is composed of queueing disciplines or "qdisc", that represent the scheduling policies applied to a queue. Each interface on the end host can have multiple queues that can be managed with TC in order to change several characteristics of the transmission of data. Such characteristics are the delay and the rate that data are transmitted.

Queueing disciplines are divided into two categories, the Classless and the Classful. Classless queueing disciplines are those that, by and large accept data and only reschedule, delay or drop it. Examples of these are :

- *p_fifo* : This queue is the kernel's default and as the name indicates it is a First In First Out queue.
- *Token Bucket Filter (TBF)* : A "qdisc" that only passes packets arriving at a rate which is not exceeding some administratively set rate, but with the possibility to allow short bursts in excess of this rate.
- *Stochastic Fairness Queueing (SFQ)* : Round Robin implementation with FIFO queues.

Classful queueing disciplines contain multiple classes. These classes are internal to the "qdisc" and may contain a "qdisc" as well. This creates a tree policy which can be used in order to shape the outgoing traffic of a host. Filters are used in order to divide the outgoing traffic and assign it to different qdiscs and therefore, to different scheduling/policing/shaping ruleset. Traffic shaping is the retiming (delaying) of packets (or frames) until they meet specified bandwidth and or burstiness limits. The Classful queueing disciplines are :

- *PRIO* : Doesn't actually shape, it only subdivides traffic based on how the filters were configured. Therefore, it is used if the user wants to give priority to a certain kind of traffic without using the Type of Service (TOS) flags.
- *Class Based Queueing (CBQ)* : It is a queuing discipline for the network scheduler that allows traffic to share bandwidth, after being grouped by classes. The classes can be based upon a variety of parameters, such as priority, interface, or originating program. It can shape and prioritize chosen traffic flows according to the configuration.
- *Hierarchical Token Bucket (HTB)* : Has the same functionality as CBQ. The major difference between CBQ and HTB is that in CBQ when

you allocate certain bandwidth to class, it cannot use more than the allocated bandwidth even if more bandwidth is available. However, in HTB you can specify how much extra bandwidth (if available) can be used by a class. In this research, this functionality is important in order to achieve higher utilization of the link. When multiple flows exist in the link and one flow ends, the other flow should be able to use the remaining bandwidth of the link.

In the experiments of this research HTB is chosen as the queueing discipline. This is due to the fact that shaping of the traffic is needed (so we cannot use any classless or PRIO "qdisc") and because of the advantages stated in [14]. Traffic shaping is needed because the bandwidth is shared between multiple flows and we should be able to divide the link in lower-bandwidth parts which in turn could be assigned to the TCP flows. Some of the advantages of HTB that are also mentioned in [14] are :

- HTB does use TBF as estimator instead of EWMA(Exponentially Weighted Moving Average) of idle time (which is used in CBQ). As result one need to set only rate and burst (max/minburst ..). Estimators are used in order to verify if the class is overlimit meaning that it surpasses the bandwidth assigned to it.
- TBF is more precise in estimating rate. For proof see [14]
- "tc class show" shows actual byte and packet rate measured over 10 sec. This in our experiments would be very usefull in order to verify that qdisc configurations are working as they were supposed to.
- TBF doesn't need precise end of send time as ewma-idle needs. As a result, HTB works robustly on all network devices (including loopback and wireless) where CBQ gives weird and wrong rates.

Furthermore, the HTB3 is now part of the official kernel sources (from 2.4.20-pre1 and 2.5.31 onwards) and according to [15] HTB is not slower than CBQ, HTB. Because HTB is more precise, can work robustly on all network devices and is not slower than CBQ, this research uses it for the experimentation procedure. An example of the HTB queueing discipline is depicted in figure 2.3.

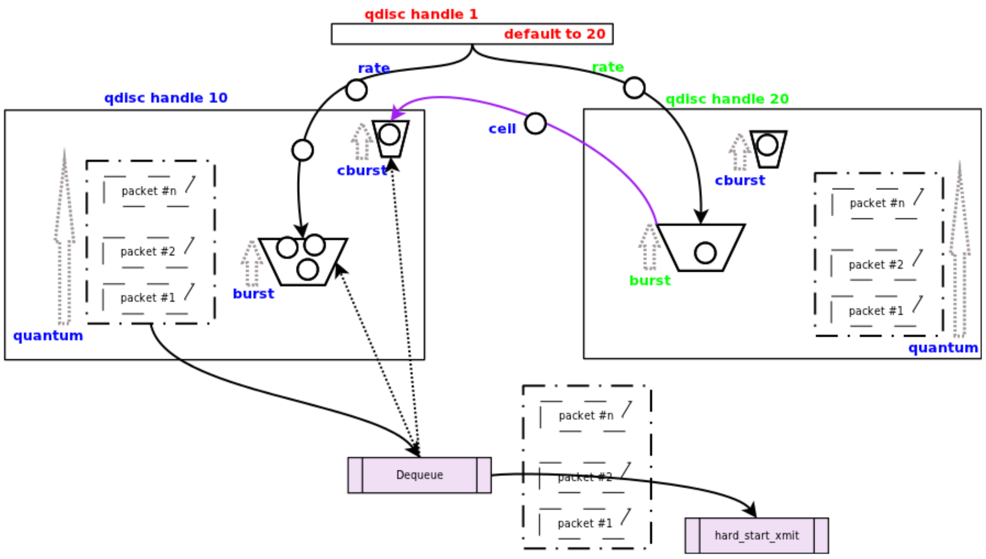


Figure 2.3: HTB example [16]

In figure 2.3 the following terminology is used:

- *burst* : The size of the token bucket.
- *rate* : The speed that tokens are generated and as a result, the packets are deqeueued.
- *quantum* : The maximal amount of bytes that a leaf in the tree policy will try to send at once.
- *ceil* : Maximum rate that a class can send traffic by borrowing rate from the parent class.
- *cburst* : Maximum burst that traffic can be sent.

In the example [16] above, we have a root "qdisc handle 1" and two leaves "qdisc handle 10" and "qdisc handle 20". The root will apply filters to decide where a packet should be directed and by default packets are sent to leaf #20 (default to 20).The leaf #10 has a rate value of 200kbits/s, a ceil value of of 400kbibts/s (which means it can borrow 200kbits/s more that its rate) and a priority (prio) of 1.The leaf #20 has a rate value of 200kbits/s, a ceil value of 200kbbits/s (which means it cannot borrow anything, rate == ceil) and a priority of 2.

In this example [16], the bucket for leaf #10 does not contain enough tokens to let the first packets pass, so it will try to borrow some from its neighbor leaf #20. The quantum value of leaf #10 is set to the MTU (1500 bytes), which means the maximal amount of data that leaf #10 will try to send is 1500 bytes. If packet #1 is 1400 bytes large, and the bucket in leaf #10 has enough tokens for 1000 bytes, then the leaf will try to borrow the remaining 400 bytes from its neighbor leaf #20.

Figure 2.3 shows that HTB support shaping of the bandwidth and that a class can use extra bandwidth through borrowing from other classes if it is available. Further theoretical,configuration details and examples can be found in [1], [16], [17].

Chapter 3

Experiments and analysis

This chapter provides an overview of the topology and tools used(section 3.1). Moreover, the configuration of the end hosts and in particular the configuration of the TCP/IP network stack is depicted in detail in section 3.3. These configurations were needed because using the default values for send and receive buffers, congestion avoidance algorithm and the network card would not provide a constant high utilization of the link. Furthermore, having small send and receive buffers could lead to having higher packet loss than expected and could affect our experimentation process.

To assess whether the relation between throughput and packet loss on our experimental environment follows the formula shown in section 2.2 we execute an experiment with virtual packet loss added and measure the throughput(section 3.4). This is needed in order to verify and evaluate how packet loss affects throughput on LFNs and in our particular experimental case supporting the initial observation that small percentage of packet loss can lead to a high loss in throughput.

In order to evaluate how Linux TC is functioning and to check how TC and TCP are cooperating we create a single flow with limited bandwidth. This is depicted in section 3.5. In order to achieve high utilization of the link when multiple flows exist the bandwidth should be divided between them. However, high utilization could not be achieved if by using TC the TCP flows are not able to take over the whole bandwidth part assigned to them or if they are not able to sustain a constant high throughput. In order to verify that we create one flow, limit the bandwidth that is assigned to it and then verify if the throughput is stable and if the flow reaches the bandwidth we assigned to it.

This research's purpose is to evaluate the usage of Linux embedded TC when more than one TCP flow exists on the same LFP compared to TCP's native way of sharing the bandwidth [18]. Therefore, multiple experiments are run with two TCP flows competing for the LFP. This is depicted in section 3.6. In order to get a more complete view various configurations are used(with limited overall bandwidth, full overall link bandwidth, with priorities/queueing, without priorities/queueing).

3.1 Topology

The topology used for the experiments is depicted in figure 3.1. When traffic is sent from the sender it goes through the left switch and the router, then to Chicago,where it is looped and then to the receiver. This adds up delay so that a LFP is simulated. The RTT is 187 ms and throughout the network Jumbo frames are used (8948 Maximum Segment Size (MSS)). It can be noticed that the bottleneck of our testbed is between the end-hosts and the switches.

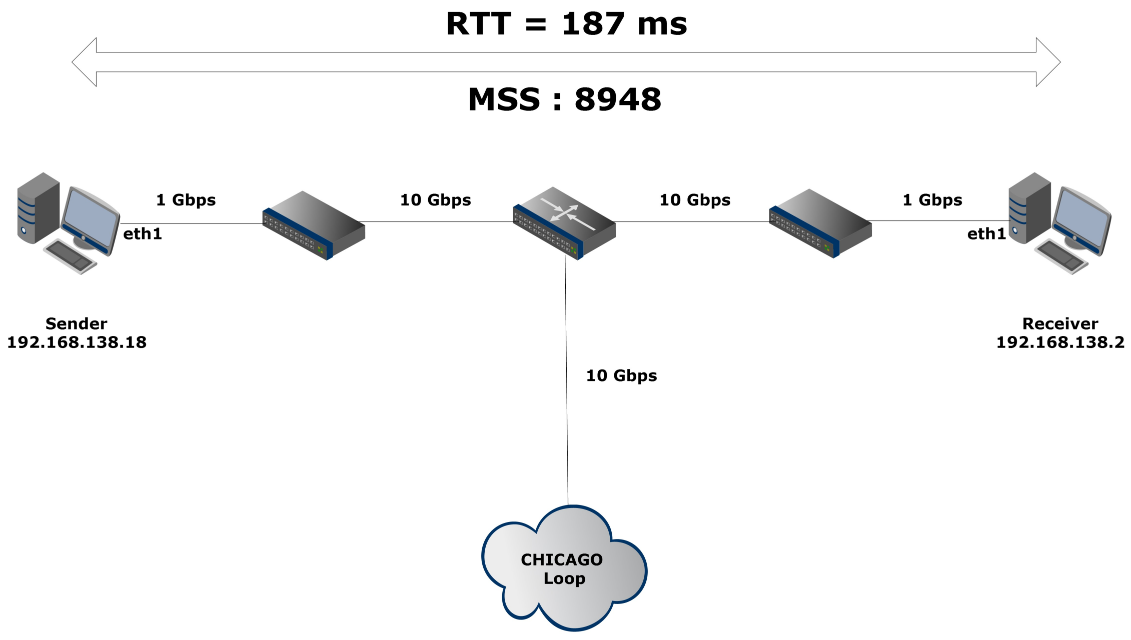


Figure 3.1: Experimental Setup, network topology

3.2 Tools

In order to perform the experiments that will lead to determining whether TC is able to be used instead of only counting on TCP's native way of sharing the bandwidth several tools were used. They should provide a way of creating the TCP flows needed for the experiments, analyse the packets that pass through the LFP and also, measure the Send and Receive congestion windows in order to verify,understand and therefore analyse our hypothesis and results.

The tools used for the experiments are :

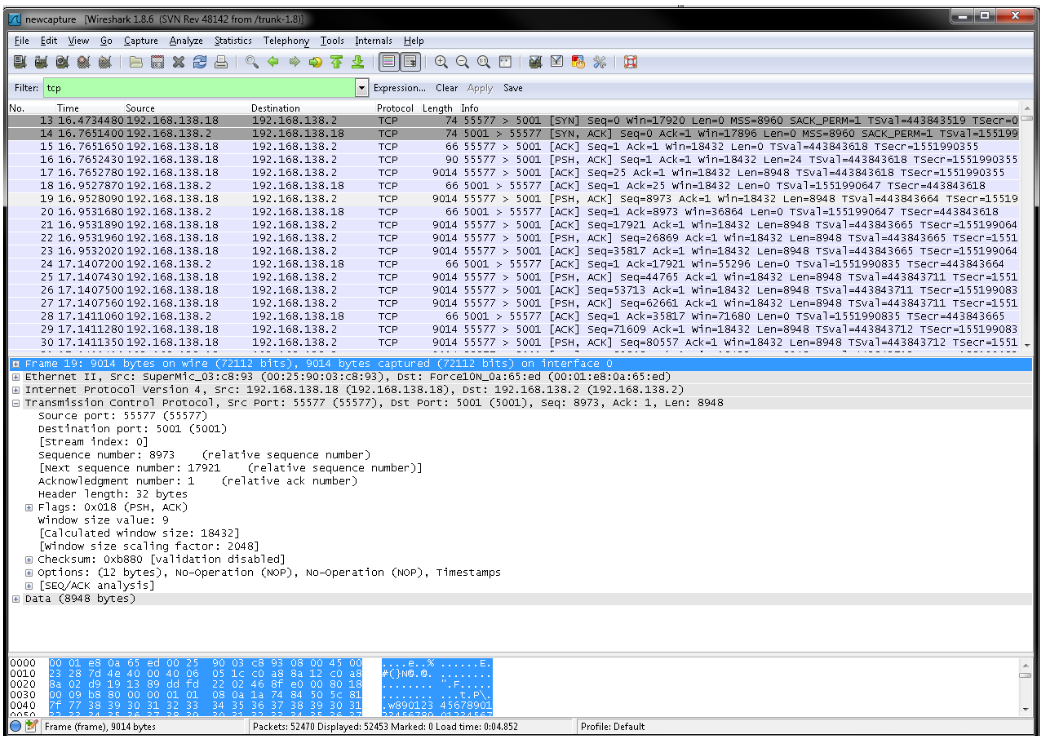
- Iperf[1].
- Tshark/Wireshark[2].

[1]http://code.google.com/p/iperf/
[2]http://www.wireshark.org/

- Traffic Control(TC) (see section 2.3).
- netem[3].
- Tcp probe[4].

Iperf is a commonly used network tool that can create TCP and UDP data streams and measure the throughput of a network that is carrying them. In our testbed these streams are sent from the sender to the receiver. Each iperf client is using one thread and it reports back the bandwidth every 1 second.

Wireshark and its command line edition tshark are free and open-source packet analysers. In our experiments it is used to measure packet loss and create the graphs of the throughput of each flow. An example of a wireshark capture is shown in figure 3.2.


newcapture [Wireshark 1.8.6 (SVN Rev 48142 from /trunk-1.8)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: tcp
Expression... Clear Apply Save

```
No.   Time       Source          Destination     Protocol Length Info
   13 16.4734480 192.168.138.18  192.168.138.2   TCP        74 55577 > 5001 [SYN] Seq=0 Win=17920 Len=0 MSS=8960 SACK_PERM=1 TSval=443843519 TSecr=0
   14 16.7651400 192.168.138.2   192.168.138.18  TCP        74 5001 > 55577 [SYN, ACK] Seq=0 Ack=1 Win=17896 Len=0 MSS=8960 SACK_PERM=1 TSval=155199
   15 16.7651650 192.168.138.18  192.168.138.2   TCP        66 55577 > 5001 [ACK] Seq=1 Ack=1 Win=18432 Len=0 TSval=443843618 TSecr=1551990355
   16 16.7652430 192.168.138.18  192.168.138.2   TCP        90 55577 > 5001 [PSH, ACK] Seq=1 Ack=1 Win=18432 Len=24 TSval=443843618 TSecr=1551990355
   17 16.7652780 192.168.138.18  192.168.138.2   TCP      9014 55577 > 5001 [ACK] Seq=25 Ack=1 Win=18432 Len=8948 TSval=443843618 TSecr=1551990355
   18 16.9527870 192.168.138.2   192.168.138.18  TCP        66 5001 > 55577 [ACK] Seq=1 Ack=25 Win=18432 Len=0 TSval=1551990647 TSecr=443843618
   19 16.9528090 192.168.138.18  192.168.138.2   TCP      9014 55577 > 5001 [PSH, ACK] Seq=8973 Ack=1 Win=18432 Len=8948 TSval=443843664 TSecr=15519
   20 16.9531680 192.168.138.2   192.168.138.18  TCP        66 5001 > 55577 [ACK] Seq=1 Ack=8973 Win=36864 Len=0 TSval=1551990647 TSecr=443843618
   21 16.9531890 192.168.138.18  192.168.138.2   TCP      9014 55577 > 5001 [ACK] Seq=17921 Ack=1 Win=18432 Len=8948 TSval=443843665 TSecr=155199064
   22 16.9531960 192.168.138.18  192.168.138.2   TCP      9014 55577 > 5001 [PSH, ACK] Seq=26869 Ack=1 Win=18432 Len=8948 TSval=443843665 TSecr=1551
   23 16.9532020 192.168.138.18  192.168.138.2   TCP      9014 55577 > 5001 [ACK] Seq=35817 Ack=1 Win=18432 Len=8948 TSval=443843665 TSecr=155199064
   24 17.1407200 192.168.138.2   192.168.138.18  TCP        66 5001 > 55577 [ACK] Seq=1 Ack=17921 Win=55296 Len=0 TSval=1551990835 TSecr=443843664
   25 17.1407430 192.168.138.18  192.168.138.2   TCP      9014 55577 > 5001 [PSH, ACK] Seq=44765 Ack=1 Win=18432 Len=8948 TSval=443843711 TSecr=1551
   26 17.1407500 192.168.138.18  192.168.138.2   TCP      9014 55577 > 5001 [ACK] Seq=53713 Ack=1 Win=18432 Len=8948 TSval=443843711 TSecr=155199083
   27 17.1407560 192.168.138.18  192.168.138.2   TCP      9014 55577 > 5001 [PSH, ACK] Seq=62661 Ack=1 Win=18432 Len=8948 TSval=443843711 TSecr=1551
   28 17.1411060 192.168.138.2   192.168.138.18  TCP        66 5001 > 55577 [ACK] Seq=1 Ack=35817 Win=71680 Len=0 TSval=1551990835 TSecr=443843665
   29 17.1411280 192.168.138.18  192.168.138.2   TCP      9014 55577 > 5001 [ACK] Seq=71609 Ack=1 Win=18432 Len=8948 TSval=443843712 TSecr=155199083
   30 17.1411350 192.168.138.18  192.168.138.2   TCP      9014 55577 > 5001 [PSH, ACK] Seq=80557 Ack=1 Win=18432 Len=8948 TSval=443843712 TSecr=1551

⊞ Frame 19: 9014 bytes on wire (72112 bits), 9014 bytes captured (72112 bits) on interface 0
⊞ Ethernet II, Src: SuperMic_03:c8:93 (00:25:90:03:c8:93), Dst: Force10N_0a:65:ed (00:01:e8:0a:65:ed)
⊞ Internet Protocol Version 4, Src: 192.168.138.18 (192.168.138.18), Dst: 192.168.138.2 (192.168.138.2)
⊟ Transmission Control Protocol, Src Port: 55577 (55577), Dst Port: 5001 (5001), Seq: 8973, Ack: 1, Len: 8948
    Source port: 55577 (55577)
    Destination port: 5001 (5001)
    [Stream index: 0]
    Sequence number: 8973    (relative sequence number)
    [Next sequence number: 17921    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    Header length: 32 bytes
  ⊞ Flags: 0x018 (PSH, ACK)
    Window size value: 9
    [Calculated window size: 18432]
    [Window size scaling factor: 2048]
  ⊞ Checksum: 0xb880 [validation disabled]
  ⊞ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  ⊞ [SEQ/ACK analysis]
⊞ Data (8948 bytes)

0000  00 01 e8 0a 65 ed 00 25  90 03 c8 93 08 00 45 00   ....e..% ......E.
0010  23 28 7d 4e 40 00 40 06  05 1c c0 a8 8a 12 c0 a8   #(}N@.@. ........
0020  8a 02 d9 19 13 89 dd fd  22 02 46 8f e0 00 80 18   ........ ".F.....
0030  00 09 b8 80 00 00 01 01  08 0a 1a 74 84 50 5c 81   ........ ...t.P\.
0040  7f 77 38 39 30 31 32 33  34 35 36 37 38 39 30 31   .w890123 45678901

Frame (frame), 9014 bytes | Packets: 52470 Displayed: 52453 Marked: 0 Load time: 0:04.852 | Profile: Default
```

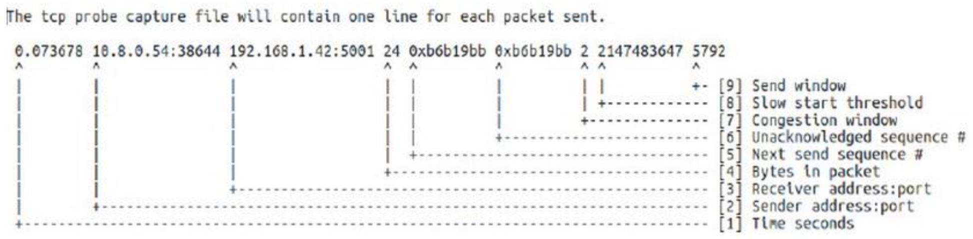
Figure 3.2: Wireshark capture

[3]http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

[4]http://www.linuxfoundation.org/collaborate/workgroups/networking/tcpprobe

Netem provides Network Emulation functionality for testing protocols by emulating the properties of wide area networks. The current version emulates variable delay, loss, duplication and re-ordering. In our experiment it is used in order to add virtual packet loss to the link. This is needed in order to verify that in LFPs the relations of packet loss and throughput follows the Mathis' formula.

Lastly, TCP probe was used. TCP probe is a module that records the state of a TCP connection in response to incoming packets. It works by inserting a hook into the tcp_recv processing path using kprobe so that the size of the congestion window and sequence number can be captured. As mentioned, by capturing and analysing the sender's congestion window, we could be able to notice whether by using TC higher or lower packet loss was noticed. An example output is depicted in figure 3.3.



```
The tcp probe capture file will contain one line for each packet sent.

0.073678 10.8.0.54:38644 192.168.1.42:5001 24 0xb6b19bb 0xb6b19bb 2 2147483647 5792
^        ^               ^                 ^  ^         ^         ^ ^          ^
|        |               |                 |  |         |         | |          +- [9] Send window
|        |               |                 |  |         |         | +------------ [8] Slow start threshold
|        |               |                 |  |         |         +-------------- [7] Congestion window
|        |               |                 |  |         +------------------------ [6] Unacknowledged sequence #
|        |               |                 |  +---------------------------------- [5] Next send sequence #
|        |               |                 +------------------------------------- [4] Bytes in packet
|        |               +------------------------------------------------------- [3] Receiver address:port
|        +----------------------------------------------------------------------- [2] Sender address:port
+-------------------------------------------------------------------------------- [1] Time seconds
```

Figure 3.3: TCP Probe example

3.3 Network Tuning Step-by-Step

In this section, the steps needed to tune the end hosts are described in order to achieve optimal throughput for use with 1 GigE network interfaces. The default network configuration does not allow us to benefit from the full network pipe because the system TCP/IP stack's buffer sizes (receive and send windows) are too small by default in each OS presently on the market (MS Vista is an exception) and because the MTU size by default is 1500 bytes. Using an MTU of 9000 bytes (Jumbo Frames) improves network performance by decreasing overhead.

There has been extensive research on this field which was used as a guideline for the values used [19] [20]. The steps are:

- **Customisation of the memory settings buffers for TCP/IP**:

```
### sets min/default/max TCP read buffer, default 4096 87380 174760
net.ipv4.tcp_rmem = 4096 87380 67108864
### sets min/default/max TCP write buffer, default 4096 87380 174760
net.ipv4.tcp_wmem = 4096 65536 67108864
### Since jumbo frames are enabled
net.ipv4.tcp_mtu_probing = 1
```

- **Core Memory Tuning**

```
### Increase the length of the processor input queue,
### default 30000
net.core.netdev_max_backlog = 250000
### Maximum receive socket buffer size, default 131071
net.core.rmem_max = 67108864
### Maximum send socket buffer size, default 131071
net.core.wmem_max = 67108864
```

- **TCP Congestion Avoidance Algorithm**

```
### The suggested congestion control algorithm is htcp ,
### default cubic
net.ipv4.tcp_congestion_control = htcp
```

- **Network Card Interface Configuration**

```
### Disable segmentation offloading on the network interface,
### default on.
### The kernel is performing the segmentation.
ethtool -k ethN tso off
### Change the sending queue size, default 1000
/sbin/ifconfig ethN txqueuelen 10000
```

- **Optional customisations**

```
### (Optional) Turn SACK support off, default 1
net.ipv4.tcp_sack = 0
### (Optional) Turn TCP timestamp support off, default 1,
### reduces CPU usage
net.ipv4.tcp_timestamps = 0
```

In our working environment the customisations that are mentioned as optional were not enabled. Turning Selective Acknowledgements off could reduce slightly the host processing load but on the other hand, reduce the maximum throughput which would therefore reduce the utilization of the link. Furthermore, since the total bandwidth of the link was 1Gbps disabling the TCP timestamps was not needed because the receiver can handle the load of the traffic. However, we reckon that for 10Gbps links it would be optimal as mentioned also in [20].

3.4 Effect of packet loss on network performance

In this experiment virtual packet loss is added in order to evaluate if the relation between packet loss and throughput is according to the formula shown in section 2.2. A single TCP flow was created and as the virtual packet loss was increasing the corresponding throughput was measured using both iperf reports and tshark/wireshark.

In our working environment the effect of packet loss on network performance is depicted in figure 3.4.

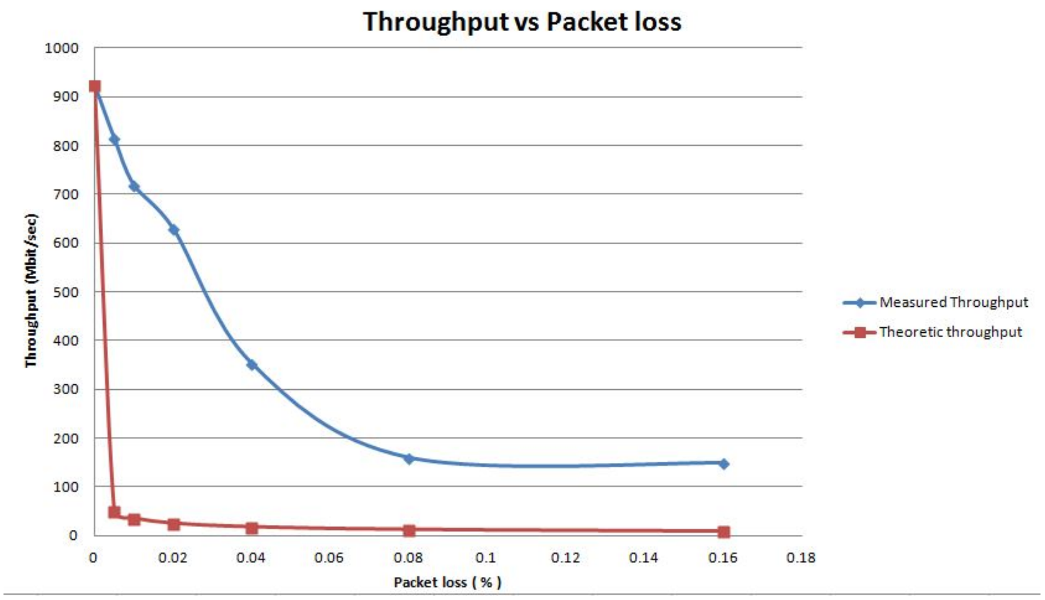


Figure 3.4: Throughput vs Packet Loss

The virtual packet loss is added on both the receiver and sender end-hosts by using netem, which adds Network Emulation functionality to TC. Netem is controlled through TC. The added packet loss is specified in the TC command in percent. It causes that percent of packets to be randomly dropped. The blue line in figure 3.4 is the measured throughput while the red is the theoretical throughput according to the Mathis et. al. formula mentioned in section 2.2. The RTT of the link used is 187ms which was used in order to calculate the theoretical throughput. Even though this experiment was run 3 times the results were almost identical and as a result one plot is given.

From figure 3.4 can be noticed that throughout the experiment the measured throughput in our experiment is higher that the theoretical throughput according to Mathis' formula. Furthermore, the rate with which throughput is decreasing is substantially less in our experimental setup compared to the theoretical throughput.

The Netem and TC parameters used are :

```
### add an ingress qdisc:
 tc qdisc add dev eth1 ingress
 tc filter add dev eth1 parent ffff: protocol ip u32
    match u32 0 0 flowid 1:1 action mirred egress
    redirect dev ifb1

tc qdisc add dev eth1 root netem loss <loss>
tc qdisc add dev ifb1 root netem loss <loss>
```

3.5 Creating a constant flow

In order to evaluate how Linux TC is functioning and to check how TC and TCP are cooperating we create a single flow with limited bandwidth. High utilization could not be achieved if by using TC the TCP flows are not able to take over the whole bandwidth part assigned to them or if they are not able to sustain a constant high throughput. In order to verify that we create one flow, limit the bandwidth that is assigned to it and then verify if the throughput is stable and if the can reach the bandwidth assigned to it.

During this research a constant 200Mbps flow was created by using Linux TC. The throughput of the flow is depicted in figure 3.5.

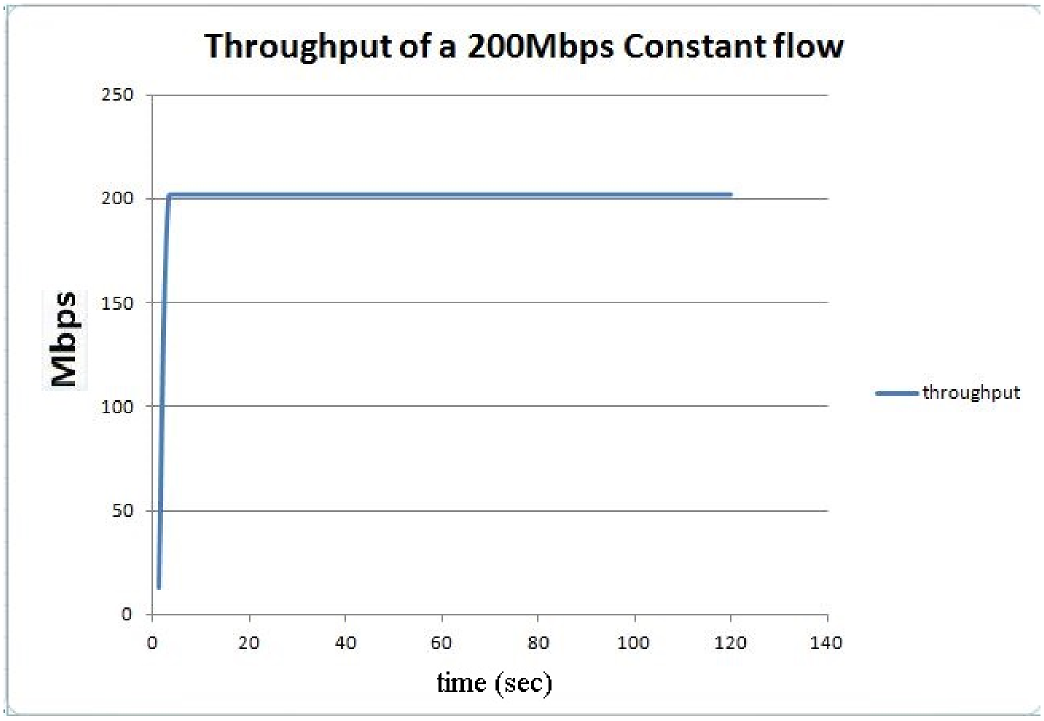


Figure 3.5: Constant rate flow 200Mbps

In order to create the constant TCP flow, both an ingress queueing discipline and an egress needs to be created using TC. Moreover, two classes need to be created that limit the bandwidth available and that are children of the respective queueing disciplines. The configuration needed follows:

```
### add an ingress qdisc:
tc qdisc add dev eth1 ingress
tc filter add dev eth1 parent ffff: protocol ip u32
   match u32 0 0 flowid 1:1 action mirred egress
   redirect dev ifb1

### add an HTB qdisc on the egress interface:
tc qdisc add dev eth1 root handle 1: htb default 1
tc class add dev eth1 parent 1: classid 1:1 htb
   rate 200Mbit ceil 200Mbit cburst 0 burst 0

### add an HTB qdisc on the ingress interface:
```

```
tc qdisc add dev ifb1 root handle 1: htb default 1
tc class add dev ifb1 parent 1: classid 1:1 htb
   rate 200Mbit ceil 200Mbit cburst 0 burst 0
```

Furthermore, in order to verify if the flow is constant, we make sure that the congestion window is stable using TCP Probe. This is depicted in figure 3.6.

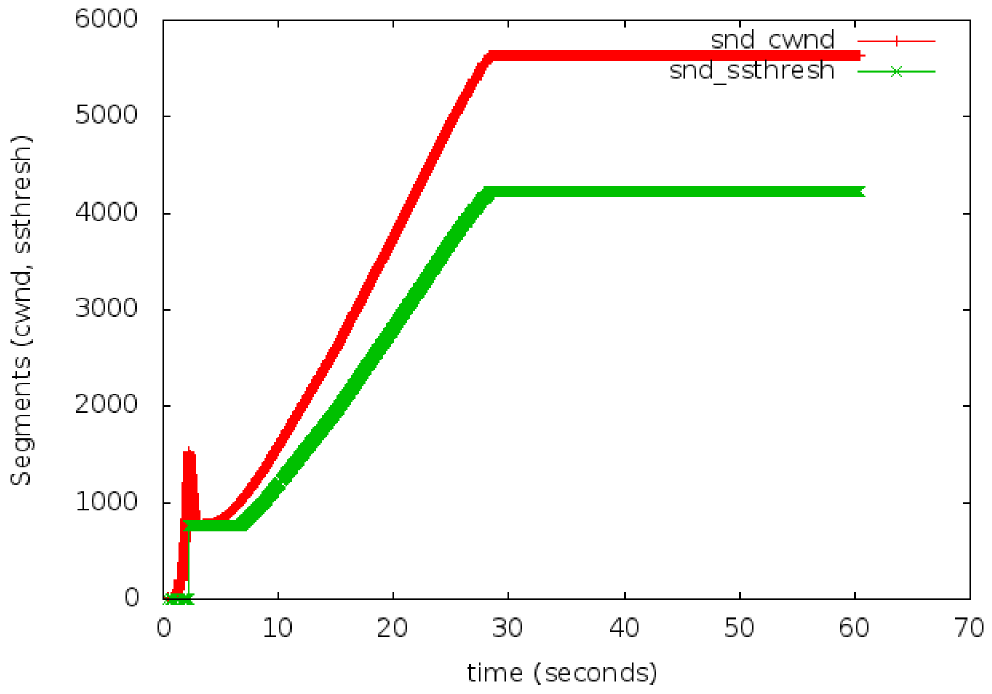


Figure 3.6: TCP Probe output for constant flow 200Mbps

Figure 3.6 depicts the congestion window(cwnd) and the Slow-Start threshold (ssthresh). It shows that the throughput should be stabilised around the 30th second. However, according to our report from iperf(figure 3.5) and wireshark(figure 3.7) the flow reaches the highest throughput at around the 2nd second. This leads us to assume that tcp probe is not reporting as it should assuming that the packet captures we received from wireshark are correct (and this is assumed throughout this research cause they appear to be the most trustworthy).

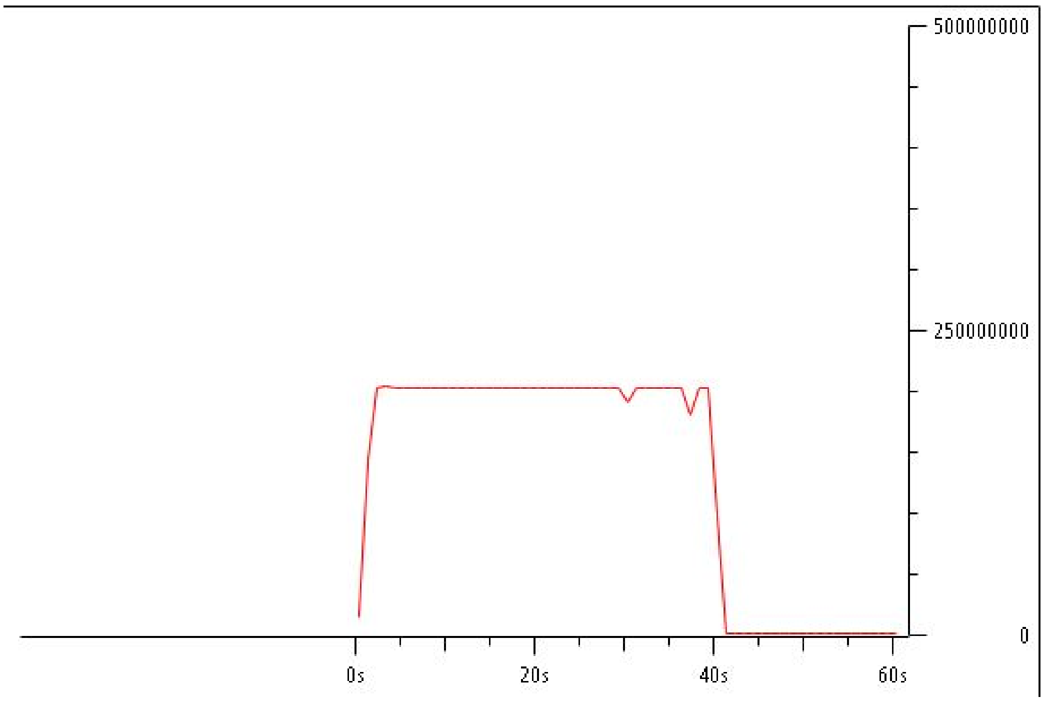


Figure 3.7: Throughput(bits/sec) vs Time(sec) - Constant flow

3.6 More than one flow intermixing using Traffic Control

This section demonstrates the performance of two flows intermixing in the same link. In the experiments one flow (flow 1 for simplicity reasons) exists throughout the experiment(0-120 sec) and another flow(flow 2) tries to enter the network at the twentieth(20) second of the experiments. Flow 2 lasts for 30 seconds. The following sections exhibit the variations in throughput, the congestion window,the Slow-Start threshold and how many packets of flow 2 manage to be transferred. Furthermore, the amount of packets transferred from flow 2 is indicated. In the wireshark reports for the amount of packets transferred, the Displayed packets are the packets of flow 2 that managed to reach the receiver. Therefore, we can perform a rounded evaluation of the performance of TCP and TC in each scenario. In all the experiments the iperf parameters remain the same(the window size is 32Mbyte). This is enough to achieve the highest throughput because according to the theoretical background 24510.5 KByte window is enough to achieve 1Gbps throughput.

3.6.1 Full link,no traffic control, 2 Flows competing for the bandwidth

In figure 3.8 the overall throughput of the link is depicted(black line) as well as the throughput of flow 1(green line) and flow 2(red line). The overall throughput appears to be unstable and this results in lower throughput and utilisation of the link.

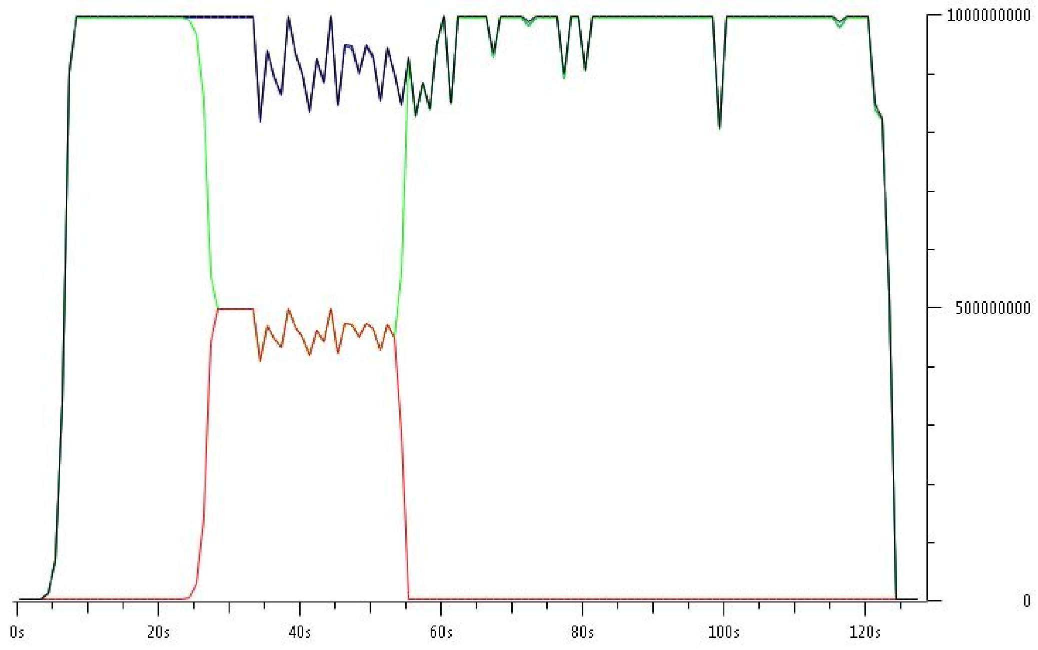


Figure 3.8: Throughput(bits/sec) vs Time(sec)

In figures 3.9 and 3.10 the congestion windows and Slow-Start thresholds are depicted. Even though, TCP probe reports them to be stable we can see in the previous figure (3.8) that the throughput is not stable and therefore, we can assume that TCP probe is not reporting the values for the congestion window and Slow-Start threshold correctly.

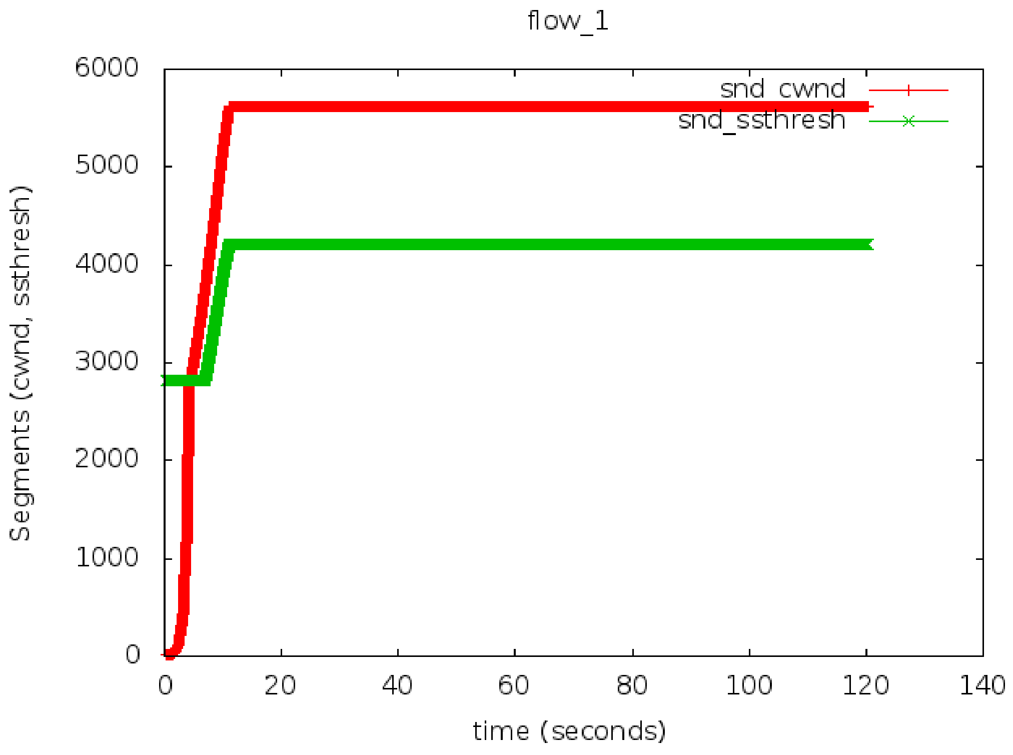


Figure 3.9: Congestion Window and Slow-Start threshold for flow 1

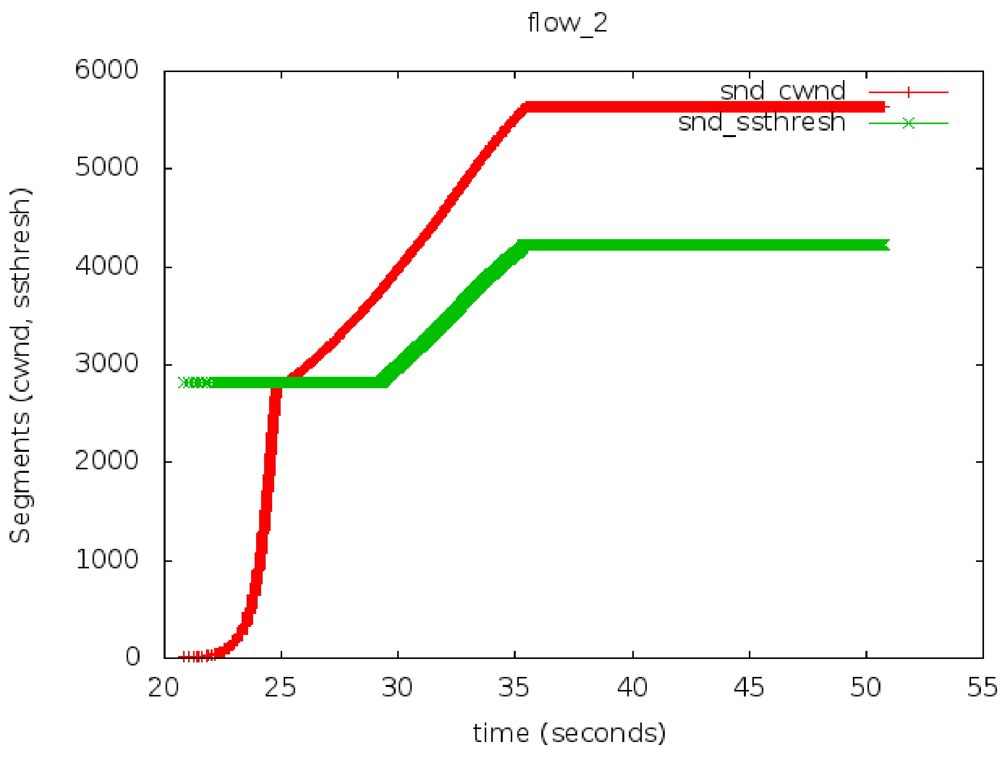
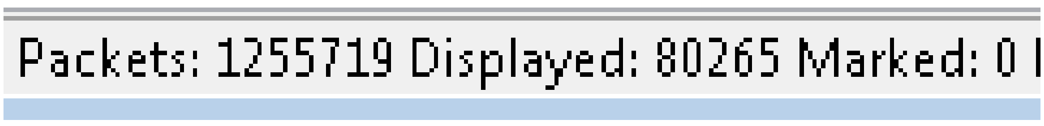


Figure 3.10: Congestion Window and Slow-Start threshold for flow 2

Figure 3.11 indicates the amount of packets transferred when 2 flows are competing for the link. This creates a baseline with which we will compare the following experiments and draw our conclusions.



Packets: 1255719 Displayed: 80265 Marked: 0 I

Figure 3.11: Number of packets second flow - wireshark report

3.6.2 Full link,divided in half,with priorities borrowing

In this experiment, the overall bandwidth of the link is divided between the two flows by using Traffic Control. With priorities and borrowing means that flow 1 is able to achieve throughput up to 1Gbps(full link throughput) as long as another flow with higher priority does not exist in the link. If another flow (flow 2) exists on the link then it can achieve throughput up to 500 Mbps. The overall throughput of the link(black line) and the throughput of flow 1(green line) and flow 2(red line) are depicted in figure 3.12.

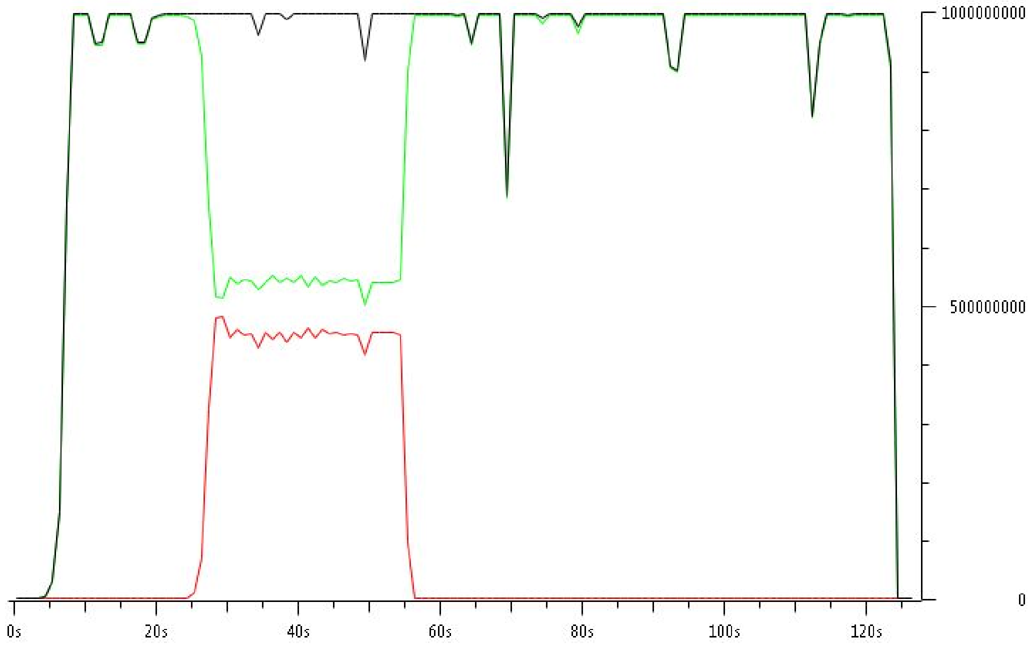


Figure 3.12: Throughput(bits/sec) vs Time(sec)

In figures 3.13 and 3.14 the congestion window and the Slow-Start threshold are depicted for each flow. It can be noticed that around the 20th second for flow 1, even though the throughput decreases due to the fact that flow 2 started transmitting both the congestion window and the Slow-Start thresholds are stable. This is contradicting which leads us to assume that tcp probe is erroneous.

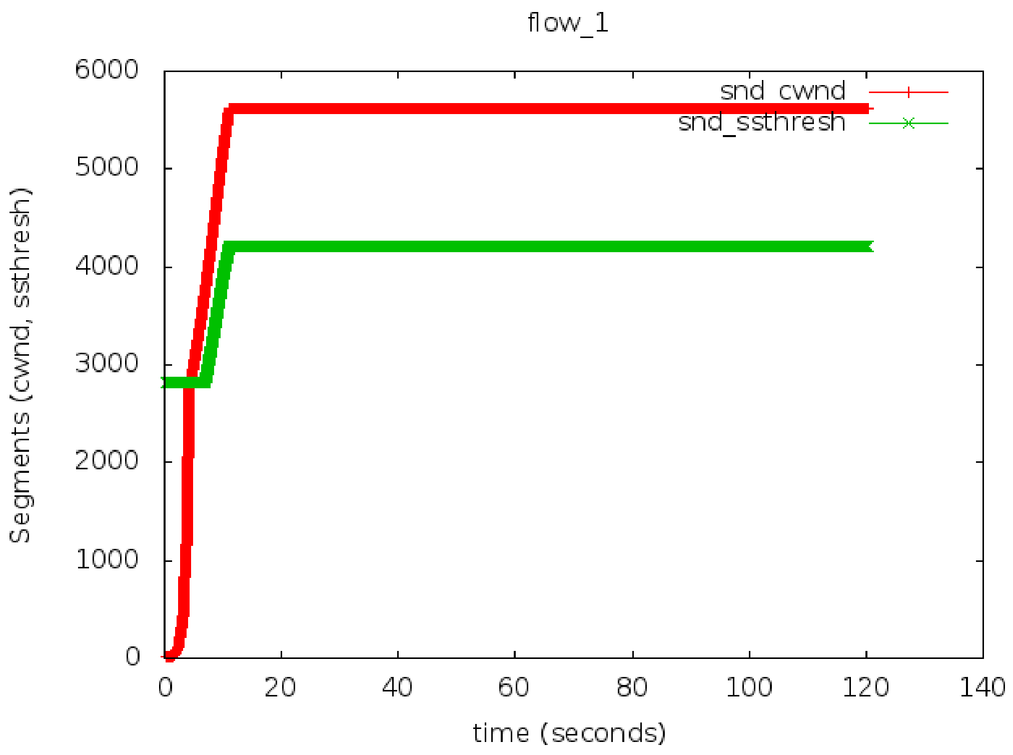


Figure 3.13: Congestion Window and Slow-Start threshold for flow 1

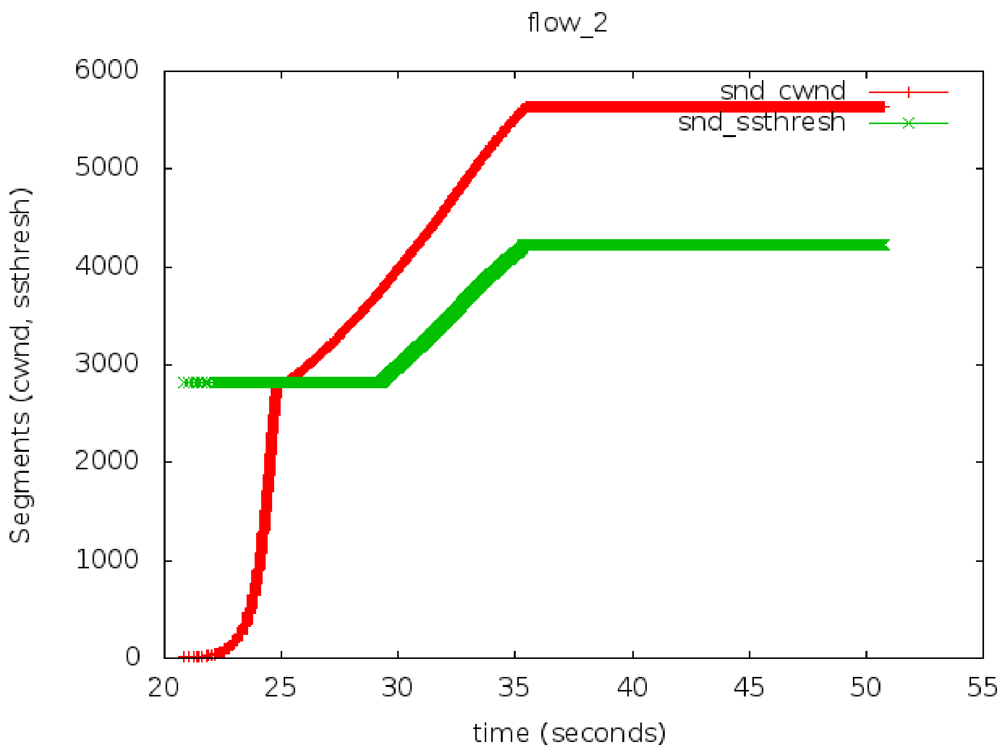


Figure 3.14: Congestion Window and Slow-Start threshold for flow 2

In figure 3.15 the overall number of packets that were transmitted are displayed followed by the number of packets that were transmitted for flow 2.

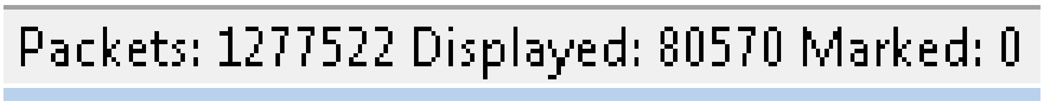

Packets: 1277522 Displayed: 80570 Marked: 0

Figure 3.15: Number of packets second flow - wireshark report

3.6.3 Full link,divided 800-200,with priorities borrowing

In this experiment the link is divided in an uneven way. Flow 1 can achieve up to 1Gbps throughput while flow 2 can achieve up to 200Mbps. Figure

3.16 indicates the overall throughput and the throughput for each flow.

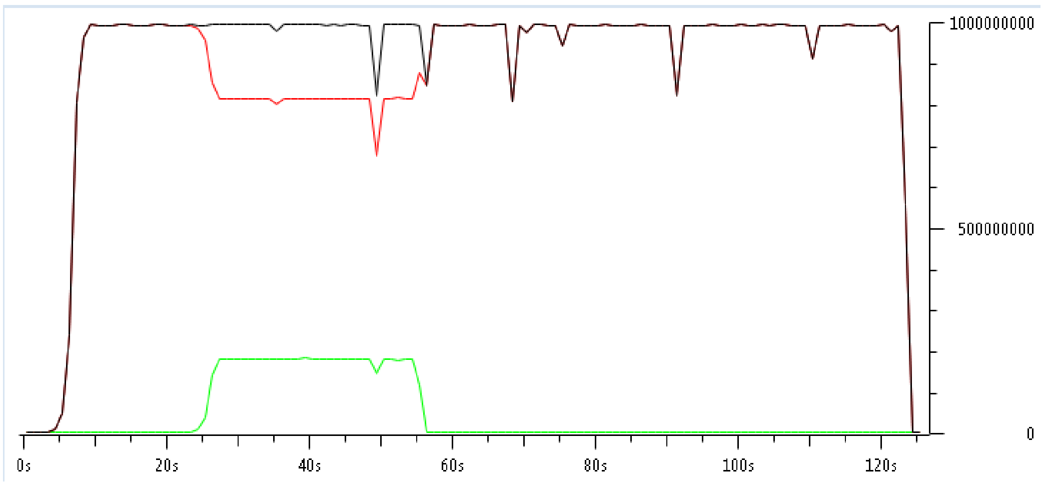


Figure 3.16: Throughput(bits/sec) vs Time(sec)

From figures 3.17 and 3.18 in comparison to 3.16, it can be deducted again that as in the previous section congestion window for flow 1 is stable instead of decreasing around the 20th second. This leads us to disregard these figures from our conclusions because they appear to be erroneous.

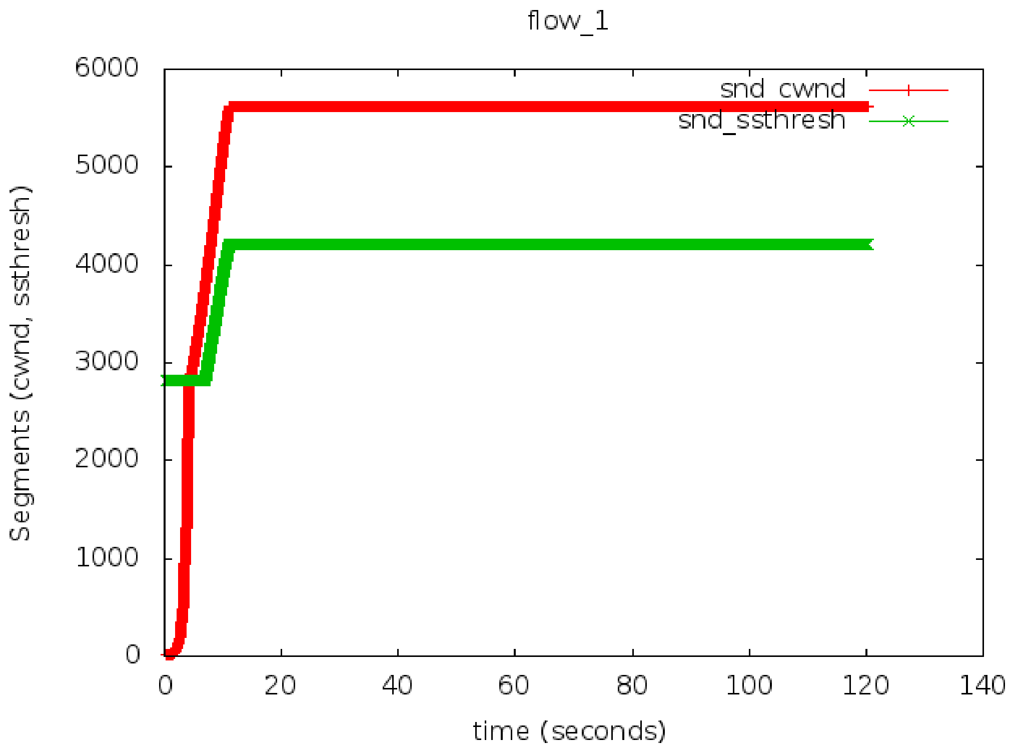


Figure 3.17: Congestion Window and Slow-Start threshold for flow 1

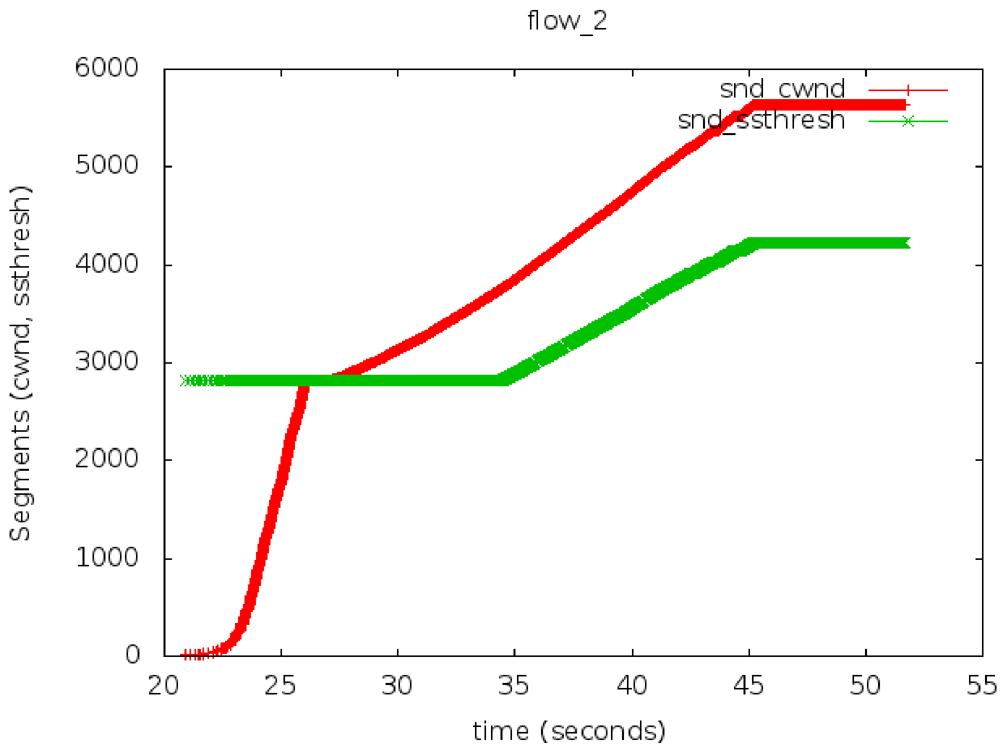
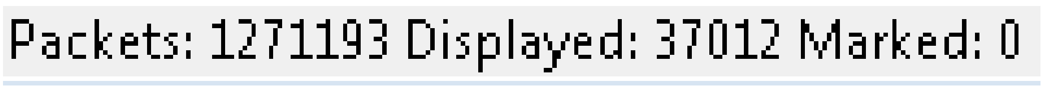


Figure 3.18: Congestion Window and Slow-Start threshold for flow 2

In figure 3.19 the overall number of packets that were transmitted are displayed followed by the number of packets that were transmitted for flow 2.



Packets: 1271193 Displayed: 37012 Marked: 0

Figure 3.19: Number of packets second flow - wireshark report

3.6.4 Full link,divided 800-200,without priorities borrowing

In this experiment, the overall bandwidth is divided statically between the two flows. Flow 1 can achieve up to 800Mbps throughput without taking into account if another flow exists on the link. On the other hand, Flow 2 can achieve up to 200Mbps.

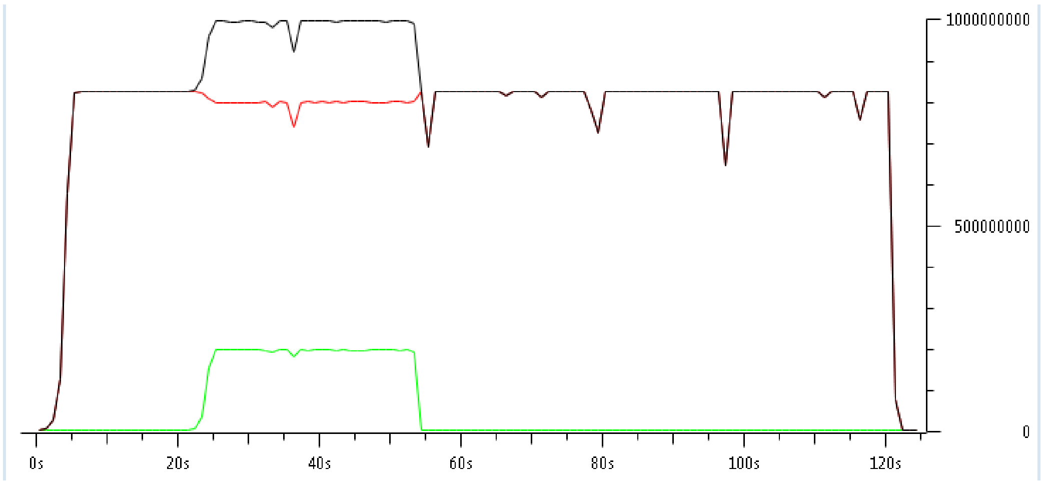


Figure 3.20: Throughput(bits/sec) vs Time(sec)

In figures 3.21 and 3.22 the congestion window and the Slow-Start thresholds can be noticed for each flow. As far as figure 3.21 is concerned, the congestion window appears to increase until the 20th second. However, it can seen in the wireshark graph that flow 1 has reached a stable throughput in the first 5 seconds and then stabilised. Furthermore, figure 3.22 indicate that the congestion window was increasing from the 20th second to approximately the 46th second. However, it appears to be erroneous in comparison to the throughput reported for flow 2 in the figure 3.20.

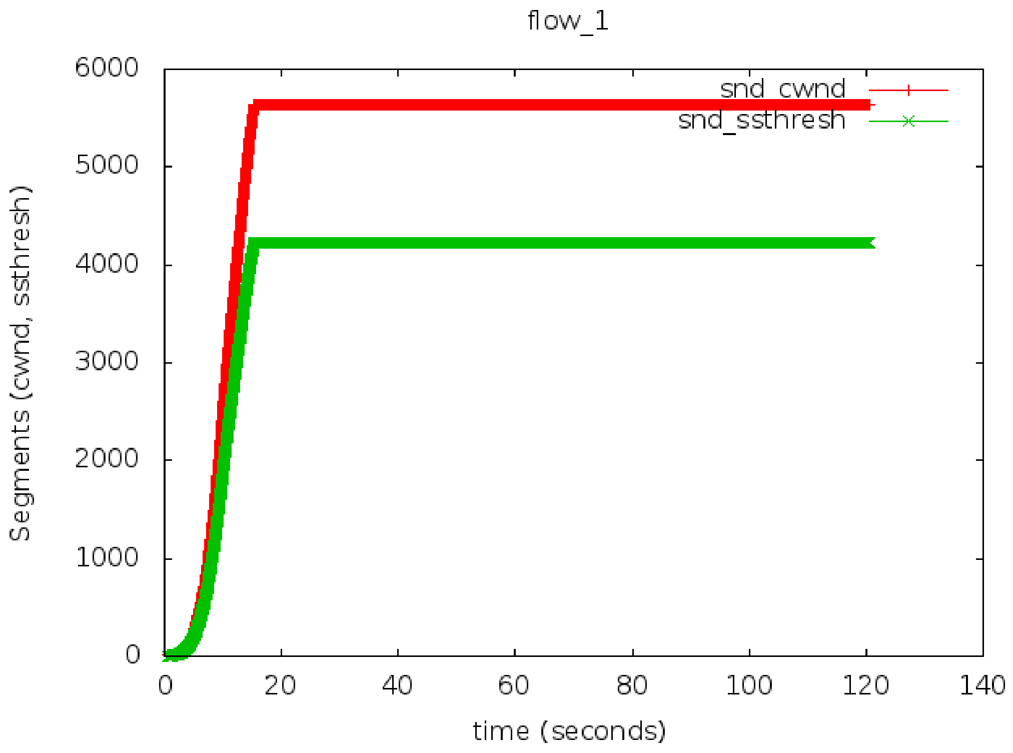


Figure 3.21: Congestion Window and Slow-Start threshold for flow 1

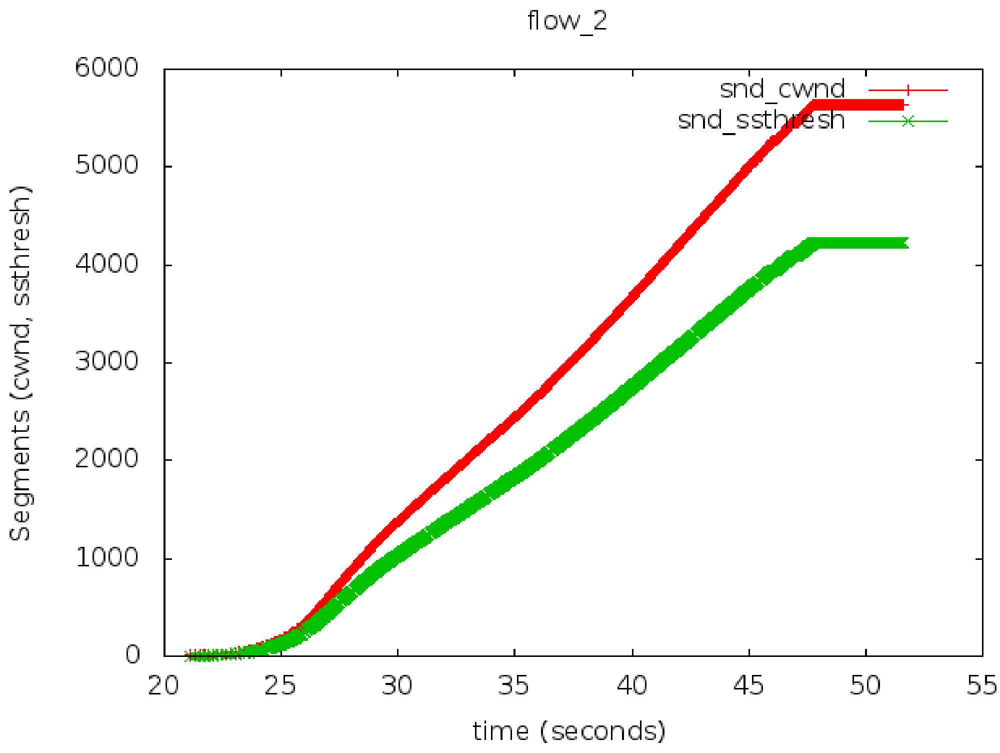


Figure 3.22: Congestion Window and Slow-Start threshold for flow 2

In figure 3.23 the overall number of packets transferred is indicated as well as the number of packets transferred for flow 2.

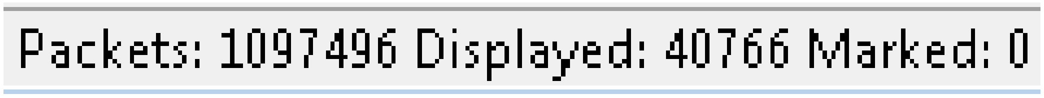

Packets: 1097496 Displayed: 40766 Marked: 0

Figure 3.23: Number of packets second flow - wireshark report

3.6.5 Link limited to 400Mbps,no further traffic control,2 flows competeting

In order to fully evaluate how TCP performs in correlation with the Traffic Control that is embedded in the Linux Kernel we limit the overall throughput of the link to 400Mbps and we leave the two flows to compete for the bandwidth. This creates a baseline and provides further information whether TC provides a stable way of creating limited throughput flows with TC. Figure 3.24 depicts the overall throughput of the link (black line), the throughput achieved by flow 1 (green line) and by flow 2(red line).

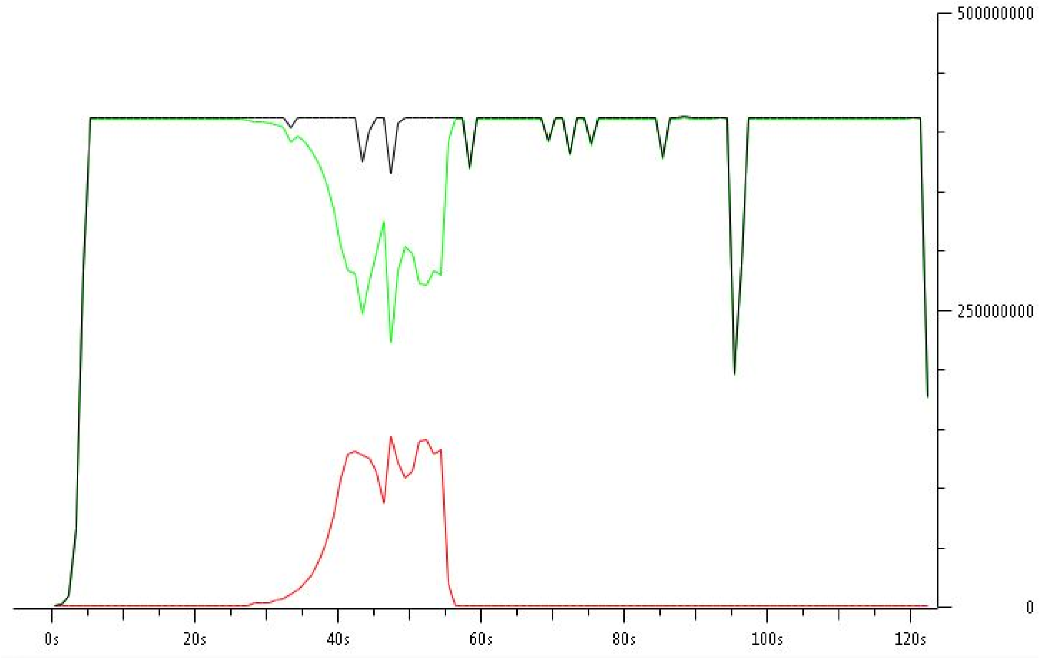


Figure 3.24: Throughput(bits/sec) vs Time(sec)

Figure 3.25 depicts the congestion window and Slow-Start threshold for flow 1. Even though the congestion window appears to drop during the 40th second it should also drop during the 20th second when the bandwidth assigned to this flow was being limited by TC due to the fact that flow 2 was starting to transmit packets.

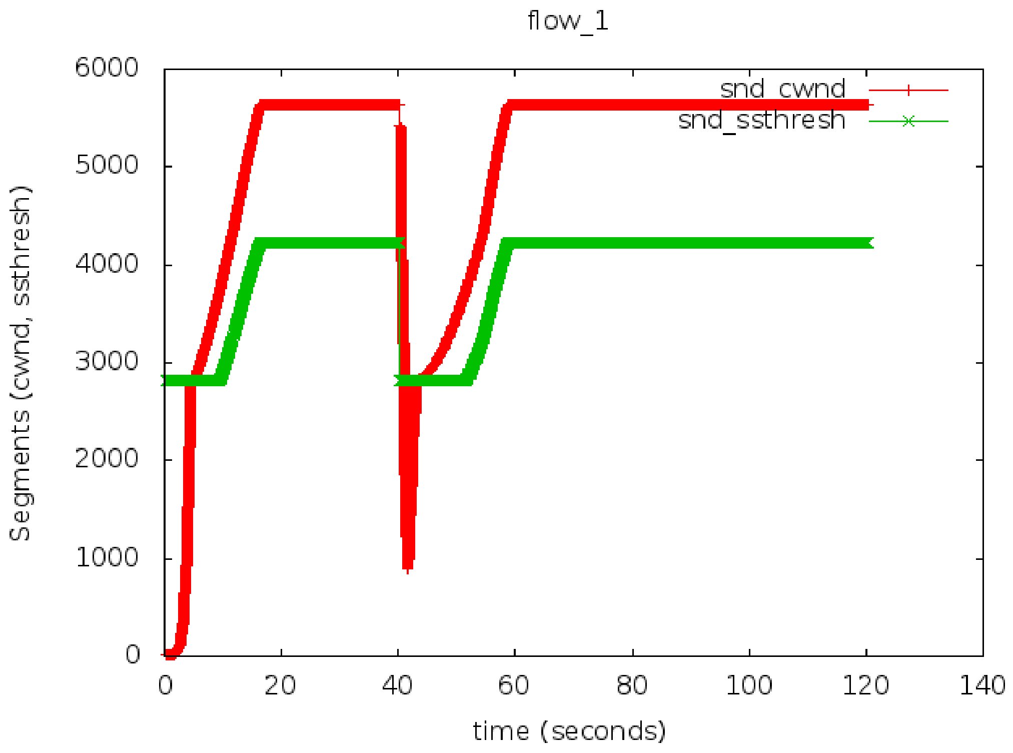


Figure 3.25: Congestion Window and Slow-Start threshold for flow 1

Figure 3.26 is again erroneous because even though there was a decrease in throughput after the 45th second as shown in the wireshark graph, the congestion window kept increasing.

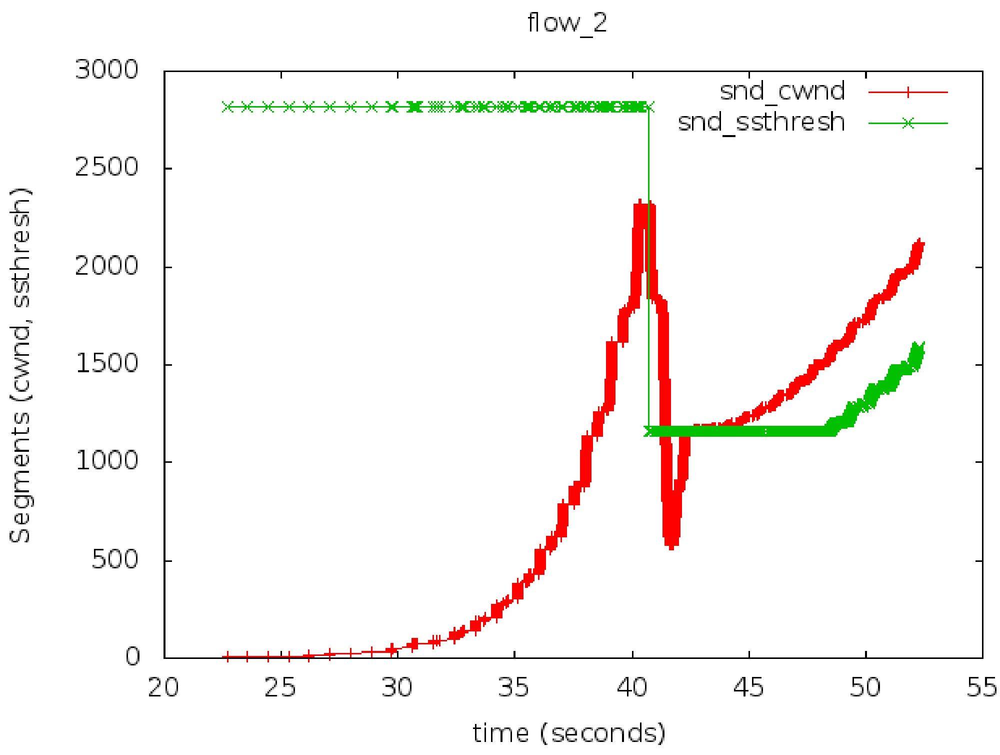


Figure 3.26: Congestion Window and Slow-Start threshold for flow 2

In figure 3.27 the overall number of packets that were transmitted are displayed followed by the number of packets that were transmitted for flow 2.

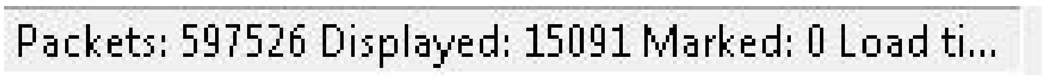

Packets: 597526 Displayed: 15091 Marked: 0 Load ti...

Figure 3.27: Number of packets second flow - wireshark report

3.6.6 Link limited to 400Mbps,divided in half,with priorities borrowing

Figure 3.28 depicts the overall throughput and the throughput for each flow respectively. Furthermore, figures 3.29 3.30 3.31 depict the congestion window, the Slow-Start threshold and the number of packets from flow 2 that are received from the receiver.

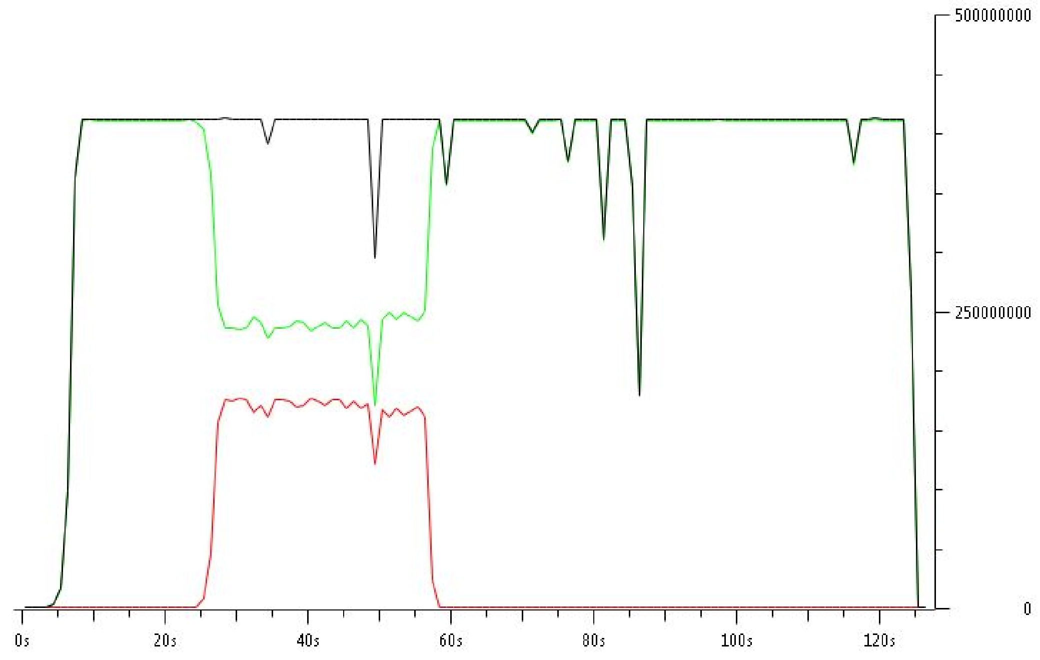


Figure 3.28: Throughput(bits/sec) vs Time(sec)

In figures 3.29 and 3.30 the congestion window and the Slow-Start thresholds are depicted for each flow. It can be noticed that around the 20th second for flow 1, even though the throughput decreases due to the fact that flow 2 started transmitting both the congestion window and the Slow-Start thresholds are stable. This is contradicting which leads us to assume that tcp probe is erroneous.

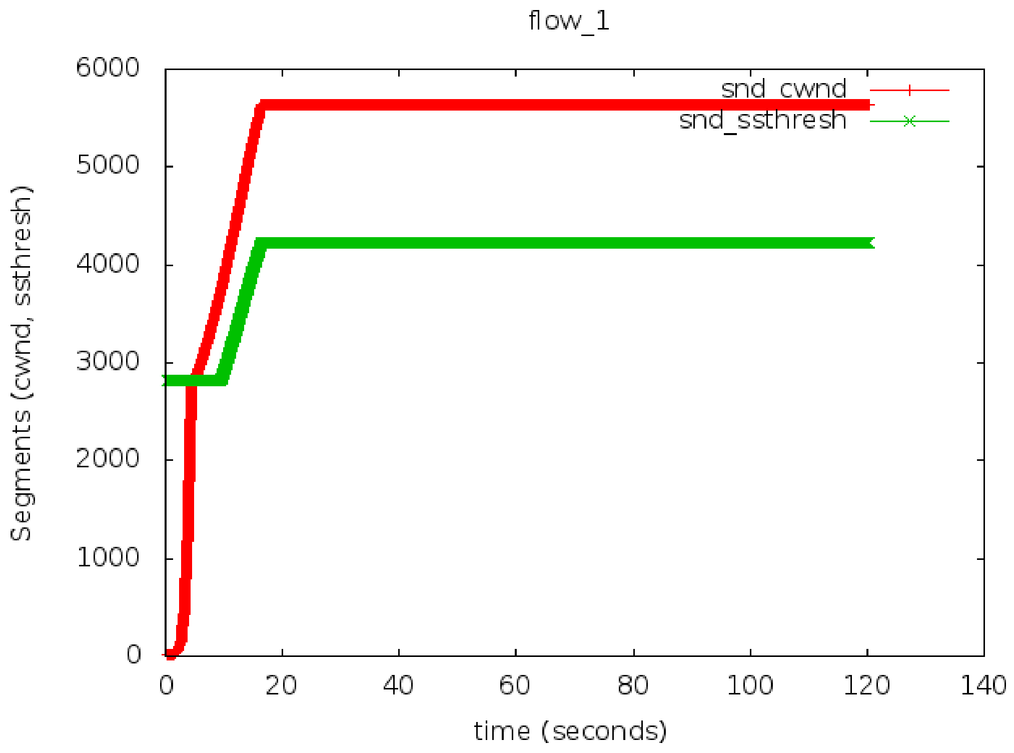


Figure 3.29: Congestion Window and Slow-Start threshold for flow 1

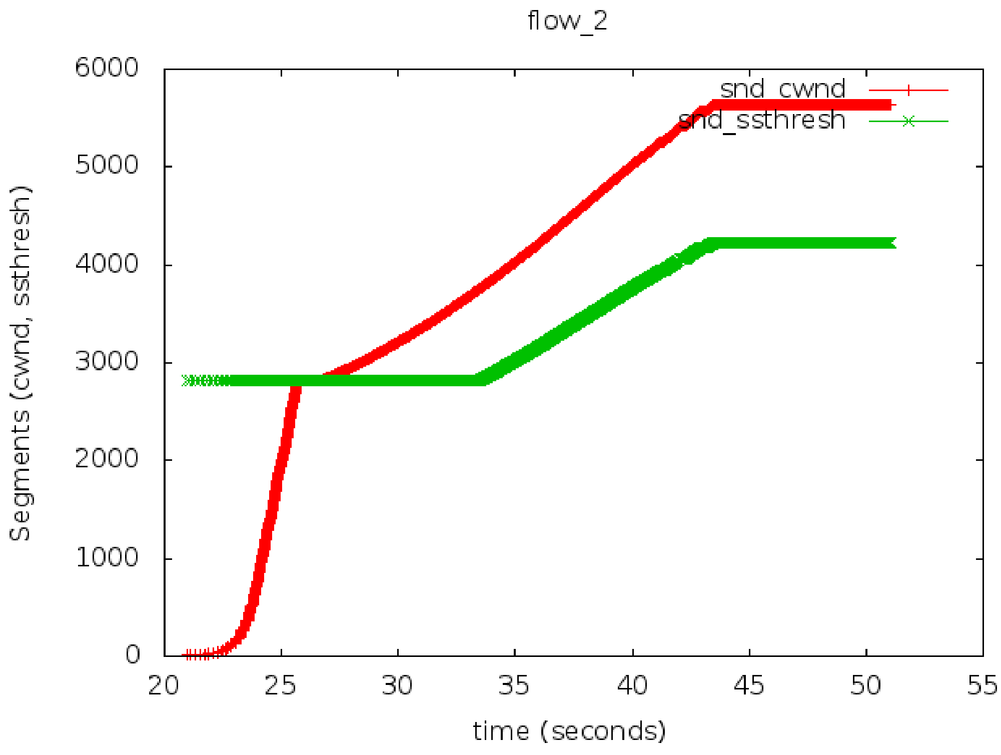
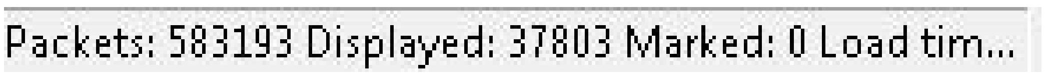


Figure 3.30: Congestion Window and Slow-Start threshold for flow 2

In the following figure the overall amount of packets transferred during this experiment followed by the number of packets transferred for the second flow.



Packets: 583193 Displayed: 37803 Marked: 0 Load tim...

Figure 3.31: Number of packets second flow - wireshark report

3.6.7 Link limited to 400Mbps,divided 300-100,with priorities borrowing

The following figures depict how the two flows behave when the bandwidth is divided in an uneven way using TC. As with the previous experiments figures 3.33 and 3.34 which were created by using tcp probe appear to be erroneous.

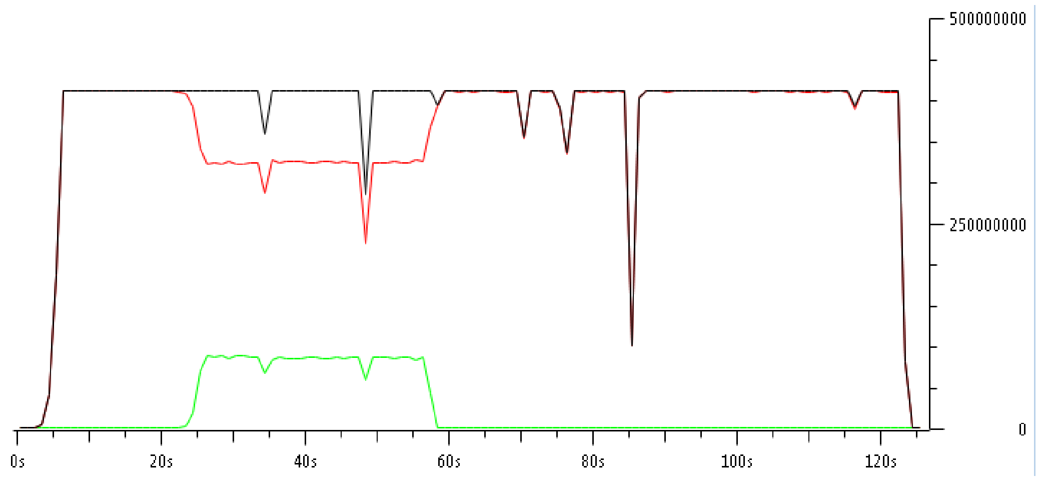


Figure 3.32: Throughput(bits/sec) vs Time(sec)

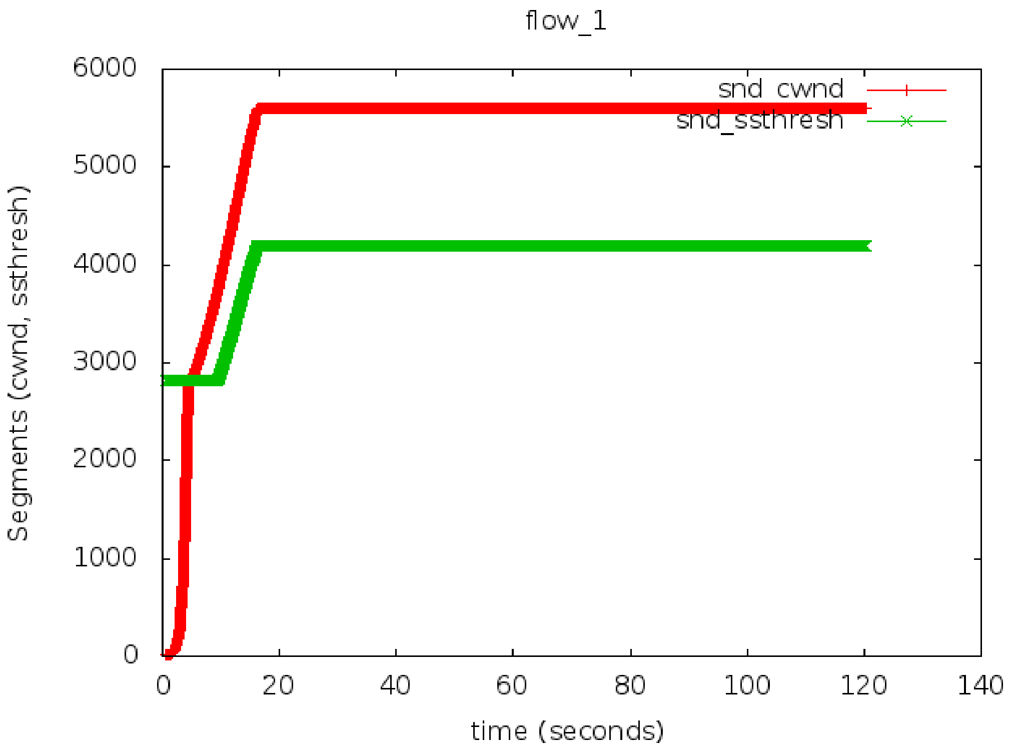


Figure 3.33: Congestion Window and Slow-Start threshold for flow 1

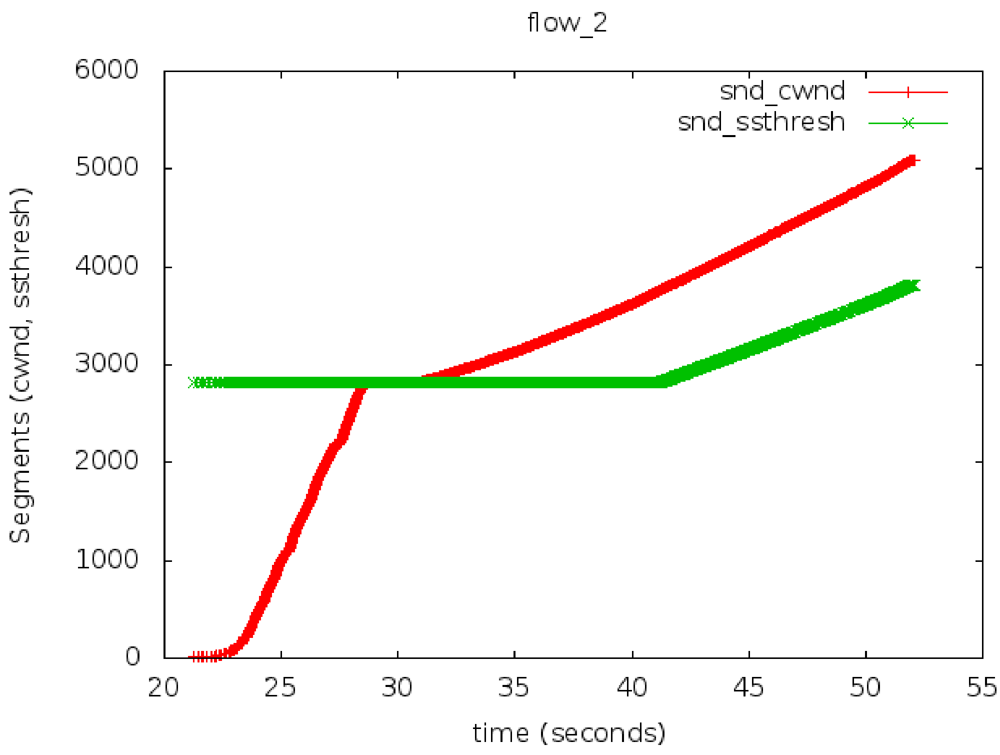
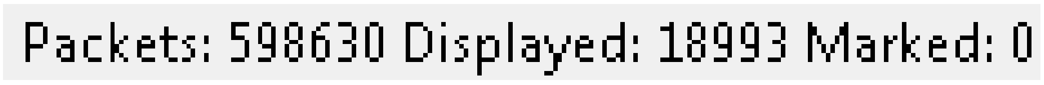


Figure 3.34: Congestion Window and Slow-Start threshold for flow 2

In figure 3.35 the overall number of packets transferred is indicated as well as the number of packets transferred for flow 2.



Packets: 598630 Displayed: 18993 Marked: 0

Figure 3.35: Number of packets second flow - wireshark report

3.7 Overview of results

Through these experiments, it can be deducted that it is possible to create a constant flow with limited bandwidth through TC which has high utilization of the part of the link assigned to it and constant throughput (section 3.5). Moreover, intermixing flows using TC can be more stable and achieve higher overall throughput for the link(section section 3.6). In the graphs that were plotted by using the wireshark packet captures (3.8, 3.12, 3.16, 3.20, 3.24, 3.28, 3.32) it can be seen that when TCP is responsible for sharing/dividing the bandwidth of the link the overall throughput is less than using TC and pre-assigning the bandwidth of the link to the flows . However, TC increases the complexity because of the configuration needed in order to divide traffic into classes while TC works natively. Hence, it can be used effectively in limited situations, such as in networks where the number of the TCP flows is known beforehand. An example of a network which behaves like that is a database network.

Furthermore, from table 3.1 can be deducted that the overall number of packets that are transferred is more when TC is used to divide the bandwidth for both flows, when the full link bandwidth is used and when bandwidth is divided equally between the flows. This is compared to the experiment without TC and can be seen in the first column of the table. Figures 3.19 and 3.23 indicate that, as expected, if static bandwidth division is used the total number of packets transferred is going to be less. This is expected because in the experiment without static division the first flow is allowed to borrow the bandwidth of the full link while with static division it can use up to the assigned bandwidth(800Mbps). However, an increase in the number of packets transferred for the second flow is indicated. This is probably due to the fact that when priorities/borrowing is enabled there is a delay caused by the need to reduce the available bandwidth for flow 1. As a result, flow 2 is able to reach the highest throughput later than when static division is used(figure 3.20 and figure 3.16).

Lastly, TCP probe reports for all the experiments appear to be erroneous. Even though according to the wireshark reports packet losses happen the window size doesn't change as it should (drop and then rise again). Due to the limited time of this research why that has happened was not determined.

Wireshark results			
Full link bandwidth utilized			
No traffic Control			
Overall Packets	Flow 2 Packets	Flow 1 Packets(%)	Flow 2 Packets(%)
1255719	80265	93,6	6,3
With TC, Divided in half, with priorities and borrowing			
1277522	80570	93,6	6,3
Divided 800-200 with priorities-borrowing			
1271193	37012	97,1	2,9
Divided 800-200 without priorities-borrowing			
1097496	40766	96,3	3,7
Link bandwidth limited to 400Mbps			
Divided in half with priorities-borrowing			
583193	37803	93,5	6,5
Divided 300-100 with priorities-borrowing			
598630	18993	96,9	3,1

Table 3.1: Overview of packets sent during the experiments

Chapter 4

Conclusions

Through this research we tried to answer the following research questions:

How can we achieve constant throughput and high utilisation of the link, while intermixing small and large TCP flows?

During this research project Linux TC was evaluated in comparison with the native bandwidth division for multiple flows from the TCP protocol. It can be deducted from the results that in specific occasions such as when the full link is used and we have 2 flows on the link, TC can provide a more stable division of the bandwidth, sustain high throughput(3.7) and high utilization of the link.

However, using TC demands complicated configuration and complete knowledge of the flows that exist in the network which does not scale well for much more than two flows. This limits the potential usage and counters some of the advantages of using TC instead of the native bandwidth division algorithm from TCP protocol.

What changes should be made in the configuration of the Linux TCP network stack to achieve the highest throughput?

In order to achieve the highest possible throughput we have to change the memory settings buffer for the TCP/IP network stack, change the length of the processor's input queue and configure the Network Card Interface to allow the kernel to perform the segmentation and to alter the sending queue size. These are depicted in section 3.3 with more details.

What effect does packet loss have on throughput?

From figure 3.4 in section 3.4 can be noticed that throughout the experiment the measured throughput in our experiment is higher that the theoretical throughput according to Mathis' formula. Furthermore, the rate with which throughput is decreasing is substantially less in our experimental setup compared to the theoretical throughput.

How can already existing traffic shaping techniques be used in order to provide a better throughput on the link with less packet loss?

From section 3.6 we can deduct that :

- By using TC better utilisation of the link is achieved when full link is used.
- There is a small deviation on the bandwidth allocation when using TC.
- The throughput of the intermixed flows is more stable when TC was used, instead of leaving only the mechanisms of TCP take care of the division of the bandwidth.

4.1 Discussion

This research project's purpose was to evaluate if queue management mechanisms such as the Linux embedded TC could be used in order to lower the number of packet lost during a connection between two hosts. This is done through assigning pre-defined bandwidth to different flows, measuring the number of packets transferred and drawing our conclusions. However, this configuration demands complete knowledge of the traffic that goes through the network and familiarisation with the already existing mechanisms that TCP offers in order to be able to do a comparison.

Furthermore, we tried to configure the network so that we could achieve as high utilization as possible. This was done according to various references and was based on our experimental setup. With these configuration we could achieve high stable throughput with or without TC in use. It is reasonable, however, that these configuration choices were for our experimental setup and should change in case end hosts' hardware or link is different than ours.

The throughput versus packet loss was evaluated in this research. It was noticed that throughput did not decrease as rapidly as Mathis' formula would suggest and that the overall throughput was always higher. However, this could be because Mathis' formula was created with an older version of TCP implementation in mind. However, we can see the effect of the various enhancements from older versions of TCP to HTCP that was used in our experimental setup and which is suggested by our references as the optimal one for LFNs.

Moreover, in all the experiments tcp probe was used in order to get the congestion window. In all the cases, it appeared to not be working as it should. There are various cases where even though packet captures show that throughput is not stable. In these cases, we were expecting that a packet loss occured and therefore the congestion window should drop. However, this is not the case and we cannot provide an answer why it happens.

Despite the limitations of TC, it can be a powerful tool in certain occasions such as a Database network where all the flows are controlled and scheduled. It would provide the same , if not better overall throughput than the native TCP bandwidth division mechanism and would add the flexibility to manually assign the desired bandwidth to specific flows or even to give them priority.

4.2 Further research

Using TC is a very interesting way of distributing the overall bandwidth to multiple flows. However, it is open to much more research in order to completely evaluate if it is a viable solution in order to avoid packet loss or variation in the overall throughput achieved especially when more than two flows are intermixed in a link. Further research could be performed by:

- Creating an Openflow monitored testbed and the forwarding rules needed in order utilise the links fully
- Intermixing more than two flows
- Changing the TCP implementation instead of only altering parameters in order for it to cooperate better with the Traffic Control policies or even avoid them altogether.

Bibliography

[1] TC tutorial. Available at http://chsoft.biz/lartc/tc_tutorial.html.

[2] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. Available at http://web.univ-pau.fr/~cpham/TCP/xu_INFOCOM_2004.pdf.

[3] Cheng Jin, David Wei, Steven Low. FAST TCP: Motivation, Architecture, Algorithms, and Performance. Available at http://web.univ-pau.fr/~cpham/TCP/FAST-infocom2004.pdf.

[4] Tom Kelly. Scalable TCP: Improving performances in high speed wide area networks. Available at http://web.univ-pau.fr/~cpham/TCP/extended.pdf.

[5] Michael Welzl. Scalable Performance Signalling and Congestion Avoidance (CADPC/PTP).

[6] Les. Cottrell. Throughput versus loss, February 2000.

[7] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, April 1999. Available at https://tools.ietf.org/html/rfc2581.

[8] The TCP/IP guide. Available at http://www.tcpipguide.com/free/index.htm.

[9] Information Sciences Institute. Transmission Control Protocol. RFC 793, Jon Postel, September 1981. Available at http://www.ietf.org/rfc/rfc793.txt.

[10] D. Leith and R. Shorten. H-TCP: TCP for high-speed and long-distance networks. pages 1–16. Available at http://www.hamilton.ie/net/htcp3.pdf.

[11] Injong Rhee and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. pages 1–6. Available at http://www4.ncsu.edu/~rhee/export/bitcp/cubic-paper.pdf.

[12] B. Constantine, G. Forget, Bell Canada, R. Geib, and R. Schrage. Framework for TCP Throughput Testing. RFC 6349, B. Constantine, August 2011. Available at https://tools.ietf.org/html/rfc6349#page-17.

[13] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. *Computer Communication Review*, 27:67–82, July 1997.

[14] Martin Devera. HTB Analysis. Available at http://luxik.cdi.cz/~devik/qos/htb/old/htbmeas1.htm.

[15] Martin Devera. HTB 3 performance compared. Available at http://luxik.cdi.cz/~devik/qos/htb/htb3perf/cbqhtb3perf.htm.

[16] Journey to the Center of the Linux Kernel: Traffic Control, Shaping and QoS. Available at http://wiki.linuxwall.info/doku.php/en:ressources:dossiers:networking:traffic_control#journey_to_the_center_of_the_linux_kerneltraffic_control_shaping_and_qos.

[17] Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul Shroeder, and Jasper Spaans. Linux Advanced Routing '&' Traffic Control HOWTO. Available at http://tldp.org/HOWTO/Adv-Routing-HOWTO/.

[18] Robert Morris. TCP Behavior with Many Flows. Available at http://web.univ-pau.fr/~cpham/TCP/TCPmanyFlows.pdf.

[19] Linux Tuning: Expert Guide. Available at http://fasterdata.es.net/host-tuning/linux/expert/.

[20] Emilio Valente. Capturing 10G versus 1G Traffic Using Correct Settings! 2009.

List of Acronyms

BDP Bandwidth-Delay Product
CBQ Class Based Queueing
LFN Long Fat Network
HTB Hierarchical Token Bucket
HTCP Hamilton TCP
IP Internet Protocol
LFP Long Fat Pipe
OS Operating System
SFQ Stochastic Fairness Queueing
RTT Round Trip Time
TBF Token Bucket Filter
TC Traffic Control
TCP Transmission Control Protocol
TOS Type of Service
MSS Maximum Segment Size
UDP User Datagram Protocol

List of Figures