Universiteit van Amsterdam
System & Network Engineering

# Emulating network latency on high performance networks

*Authors:*
Berry Hoekstra
bhoekstra@os3.nl

Niels Monen
nmonen@os3.nl

*Supervisors:*
Cosmin Dumitru
c.dumitru@uva.nl

Ralph Koning
r.koning@uva.nl

February 6, 2011
Version 1.0 — Revision 127

*This page is left blank intentionally*

**Abstract**

Testing the behaviour of applications and protocols on various network setups can be a difficult task to realise. With the emergence of high-speed connectivity, new research is needed to evaluate the behaviour of (existing) applications and protocols. In this research, the netem software was evaluated as a tool to emulate network characteristics on 10 GigE and 40 GigE setups. Throughput was measured using kernels with different tick speeds. However, expected throughput could not be achieved when emulating characteristics on egress traffic. Using different kernel time resolutions does not mitigate this problem. We suspect that netem is not optimised for such a high throughput link and can not cope with the large amount of packets coming in, even though the network parameters are optimally configured. We advise to only use netem when a maximum speed of 4 to 5 Gigabit per second is expected.

# Contents

# 1 Introduction

Testing the behaviour of applications and protocols on various network setups can be a difficult task to realise. With the emergence of high-speed connectivity, new research is needed to evaluate the behaviour of (existing) applications and protocols. To recreate different network characteristics, a real-world environment can be used. However, this can bring along the burden of communicating with the many parties that are involved to create a testbed with similar, state-of-the-art equipment. Another approach is to recreate the setup in the lab using network resources such as cases containing kilometers of fiber interlinked using amplifiers. Although this approach will probably have the exact characteristics of the real-world environment, resources like these are not always available. Other solutions, like the emulation of network properties like delay and jitter, might provide a much easier approach. Expensive hardware solutions are available for this, like the proprietary solutions that Ixia provides [1].

## 1.1 Research

In this work we study the possibility to use a generic server as a "delay box" between nodes to emulate the different network properties of high-performance links using software. This approach enables the use of off-the-shelf hardware, while eliminating high costs and effort. By doing this research, we will answer the following question:

> *What are the characteristics of long distance high-speed links and to what extent can they be emulated with off-the-shelf hardware?*

The following sub-questions will help to answer the main research question.

- What solutions are available for this purpose?
- What is the effect of using different network parameters?
- Does it matter if a real-time or regular kernel is used?

## 1.2 Network delay

### 1.2.1 Definition of network delay

In a real-world environment, different factors may cause delay. In general, the delay on a computer network is called "network latency". Some define latency as the amount of time for a data packet to reach its destination [2]. In our case, we measure the network latency by the amount of time it takes for it to travel from the sender to its destination and back. This is called the Round Trip Time

(RTT) [3]. Delay is not only specified by the Round Trip Time, though. We identify the total delay of a link using three characteristics; RTT, jitter and jitter distribution. The jitter is the variation in the RTT, which is not always steady over time. The time when jitter occurs is the jitter distribution.

### 1.2.2 Causes of network delay

Multiple factors may cause a packet to be delayed on a network link, which will have effect on the data throughput between nodes that are connected through this link. The data throughput can be calculated with the following formula, wherein RWIN is the receive window in MB, and RTT the Round Trip Time:

$$Throughput \leq \frac{RWIN}{RTT}$$

Among other factors, the total amount of delay depends on the distance of a link, the state of intermediate routers and the amount of jitter. The capabilities and configuration of network parameters also have a role in the delay.

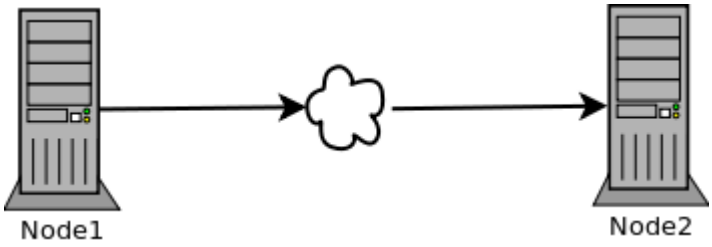



Figure 1: Link with multiple (unknown) factors.

If the variation in RTT is very high, the throughput of a link can't be considered reliable. The delay in packet delivery may differ for every period of time, depending on the situation on the link. For example, the jitter can be led back to the congestion state of the queue in an intermediate router or to connections that go down and have to converge again (using the BGP routing protocol).

The amount of jitter on a LAN isn't very high, though. This is because in most cases, LANs are not very large networks. A network like a LAN can be considered reliable if the jitter is in the order of ten percent of the average RTT [2].

The causes of delay are discussed below.

**Light limitations** In high-speed networks, connectivity is provided using optical connectors and fibers. Data transfer is possible by transmitting light across the fiber. It is known that the speed of light in a vacuum is limited to 299,792,458 meters per second [4], which will enable us to calculate the

theoretical minimum time of a signal to reach its destination. To give an example, a fiber with the literal distance from Amsterdam to San Diego has a length of 9028 kilometers [5]. To calculate the maximum theoretical speed, the following formula applies, where C is the speed of light:

$$C(m/s) = 299,792,458$$

$$Delay(s) = \frac{Distance(m)}{C(m/s)}$$

$$\frac{9028000}{299,792,458} = 0,030114s = 30,114ms$$

Among this theoretical limit, other factors like the fracturing of light has a negative effect on the strength of the light, but the calculation above shows the theoretical maximum of the speed of light in a vacuum. In optical connections, the light travels through glass, which makes the propagation medium air. This also further degrades the speed to a theoretical maximum speed of the light, and thus the signal.

**Fairness (QoS)** On a busy (shared) link, routers have to process all packets with a certain amount of fairness. Therefore, packet delay occurs on one traffic stream when packets of another packet stream are processed by the router. This approach will grant both streams an even amount of bandwidth. This is also called Quality of Service, or QoS [6].

**TCP configuration** A data stream follows a certain path towards its destination target. In a situation where TCP parameters are not optimally adapted on the sending and receiving node, the throughput is not optimal. Later in this report, we will discuss what parameters can be changed in order to achieve the optimal data throughput.

**Maximum Transmission Unit** The Maximum Transmission Unit (MTU), specifies the size in bytes of a packet that the Ethernet layer can send in one frame. When dealing with a fast data throughput, the MTU value can be optimised to achieve higher speeds [7].

**Transmission delay** is the time that is required for a network device to send all the bits of a packet onto the network.

**Queueing delay** is the amount of time a packet is spending in the packet queues of an intermediate routing device.

**Processing delay** is the amount of time a router needs to read and process a packets header before it decides what to do with it.

### 1.2.3 Measuring network delay

To emulate a real-world link in a controlled lab environment, we acquired delay statistics of the link to emulate. An easy method to measure the network delay is by using the "`ping`" tool [8]. Ping uses the Echo feature of the ICMP [9] protocol. An originating machine sends an ICMP Echo Request. The destination address will reply to this request with an ICMP Echo Reply message, which will reach the originator again. The amount of time between the ICMP Echo Request and the ICMP Echo Reply can be considered the RTT [3]. An example can be seen below:

Ping to info4u.os3.nl

```
root@box$ ping -c 2 info4u.os3.nl
PING info4u.os3.nl (145.100.96.70) 56(84) bytes of data.
64 bytes from info4u.os3.nl (145.100.96.70): icmp_seq=1 ttl=63 time=0.147 ms
64 bytes from info4u.os3.nl (145.100.96.70): icmp_seq=2 ttl=63 time=0.237 ms

--- info4u.os3.nl ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.147/0.192/0.237/0.045 ms
```

In the example above, the value after the "time" parameter is the RTT value, which is the amount of time it takes for packet to travel from the sender to the destination target and back. Although there are other tools available to measure network delay, we decided to use ping, as it is the most convenient way to measure the RTT to use together with netem.

## 1.3 Improving network throughput

As we discussed in Section 1.2.2 , many factors can influence the delay on a network. While most of them are uncontrollable, there are some parameters that can be tweaked to optimise the throughput of a network. We will discuss these parameters below.

**Increase the MTU** By increasing the size of the MTU, more bytes can be sent to the target destination in every Ethernet frame. This will reduce the amount of frames that have to be sent. By setting the Ethernet MTU to 9000 bytes throughout an entire network path, fragmentation of the frames won't occur. A MTU of 9000 bytes is commonly known as Jumbo Frames. We can assume Jumbo Frames along the complete path, as most National Research and Education Networks (NREN) currently support Jumbo Frames in their entire path [10].

**Bandwidth Delay Product** The maximum amount of bytes that can be on a link in transfer at any given time is called the Bandwidth Delay Product (BDP). This value is the product of the bandwidth in bits per second and the end-to-end delay of the connection in seconds. The size of the BDP can be used as the size of the TCP window size. By optimally configuring the TCP window size, one can achieve the maximum throughput of the line. For example, the following formula applies for a 1 Gigabit Ethernet connection on a link with 154 milliseconds delay. Bandwidth (B) is in bits per second and delay (D) is in seconds.

$$BDP = B \times D$$

$$1000000000 \times 0.100 = 100000000 bits$$

$$\frac{100000000}{8} = 12500000 Bytes \approx 12.50 MB$$

**TCP window size** In the example above, the maximum amount of in-flight data at any given time is about 12.50 MB. This is much larger than the 64 KB that was originally configured in the TCP stack. This problem is solved with TCP window scaling [11]. TCP window scaling is an extension to TCP for high performance links. It allows the TCP protocol to send data when it had not yet received TCP acknowledgements of previously sent data. The sender has to keep sending data during the time that is equal to the RTT of the link to achieve the maximum throughput. The receiver has to advertise a window size that is sufficient enough.

To enable a larger TCP window size in Linux, TCP window scaling has to be enabled in `/proc/sys/net/ipv4/tcp_window_scaling`. On Linux, TCP window scaling is enabled by default from kernel 2.6.8 (August 2004) [12], so in most cases it won't be necessary to enable it manually.

If the maximum window size would still have been limited to 64 KB, the maximum BDP would be only 0.5% of the maximum throughput (5 Megabit per second). To get optimal throughput on the 1 Gigabit link with 100 milliseconds delay, a window size equal, or higher than the BDP would be needed.

**Buffers** On a high performance link, throughput can be very high. Nodes on both ends need to process the large amount of incoming data while still receiving even more incoming traffic. To keep up with the ongoing data stream, a buffer is needed that is large enough to temporarily store the ongoing data stream. Increasing buffer size will use more memory in the system, so it is critical that enough memory is present to prevent swapping and decreasing performance.

| Description | Location |
|---|---|
| Default size of the TCP receive window | `/proc/sys/net/core/rmem_default` |
| Maximum size of the TCP receive window | `/proc/sys/net/core/rmem_max` |
| Default size of the TCP transmit window | `/proc/sys/net/core/wmem_default` |
| Maximum size of the TCP transmit window | `/proc/sys/net/core/wmem_max` |

Table 1: File locations containing the socket buffer parameters.

On a Linux system, the receiving and sending socket are meant for this. The socket buffers can be configured in the following parameters:

The maximum size of the buffer per socket is specified in `/proc/sys/net/core/optmem_max`. Apart from the socket buffers, the TCP protocol also has some dedicated buffers. These buffers are used for the TCP autotuning feature, which is enabled in `/proc/sys/net/ipv4/tcp_moderate_rcvbuf`. Machines with a Linux distribution with kernel 2.6.17 or later installed have autotuning enabled by default. This enables the TCP window scaling feature as discussed in Section 1.3.

The TCP buffers that are used for the autotuning feature can be configured in the following parameters:

| Description | Location |
|---|---|
| Maximum size of TCP buffer space | `/proc/sys/net/ipv4/tcp_mem` |
| Minimum, default and maximum receive window size | `/proc/sys/net/ipv4/tcp_rmem` |
| Minimum, default and maximum transmit window size | `/proc/sys/net/ipv4/tcp_wmem` |

Table 2: File locations containing the TCP buffer parameters.

**Packet queue length** Every node has a queue different from the buffer in which the packets are "stored" temporarily before they will be sent on the line. This is the transfer queue length, which is specified as "`txqueuelen`" on Linux. The `txqueuelen` specifies how many packets the TCP packet queue can contain and can be set using the `ifconfig` tool: `ifconfig <NIC> txqueuelen 10000`.

The receiving packet queue length is specified in `/proc/sys/net/core/netdev_max_backlog`.

**Disable unnecessary TCP parameters** On dedicated links and controlled lab environments, the throughput can be optimised by disabling some TCP parameters.

Overhead can be reduced by disabling the TCP Segment Acknowledgements (SACK), which will normally reduce the overhead that is generated by the default TCP Acknowledgements by only acknowledging complete segments of TCP packets [13], which allows for more packet loss per frame. For testing purposes, to reach optimal throughput, it can be disabled in `/proc/sys/net/ipv4/tcp_sack`.

Another TCP parameter that can be disabled is the timestamp feature, which makes the TCP headers 12 bytes smaller. This option is controlled in the `/proc/sys/net/ipv4/tcp_timestamps` file.

It must be noted that this is not desired on (shared) Internet links, as uncontrollable events might occur.

# 2 Related research

In this section, existing studies are reviewed and existing tools are discussed.

## 2.1 Literature review

Many papers that discuss high performance networking already exist.

A study by Yee-Ting Li et al. evaluates the TCP protocol by comparing different TCP congestion algorithms [14]. The links we try to emulate use the H-TCP congestion algorithm. H-TCP originates from the Hamilton Institute, and is used on high speed networks with high latency, also known as Long Fat Networks (LFN). This paper concludes that H-TCP stands out in fairness and is also backwards compatible to low links that have a low BDP [15].

Previous OS3 students have researched how to achieve optimal throughput through two end-to-end connected nodes over 10 Gigabit Ethernet network interfaces [16] , which was useful in our research as some of the network parameters were already pointed out.

Other studies specifically focus on the best tool to use for network emulation [17]. This study points out that some of the research is done using netem as an emulation tool, while some use NIST Net. We have seen other researchers use DummyNet [18] , which will be discussed in short later in this paper.

The research of Stephen Hemminger [19] [20] shows that netem can be used to emulate certain paths, but is unable to completely emulate the Internet, or emulate large parts of it. The study shows that this is caused by the many factors that are involved in real-world networks, such as asymmetrical routes and constantly changing congestion states, which is very hard to emulate. Although the study only focuses on 1 GigE links, it discusses the netem workings in great detail, which we found useful during our study.

While Hemminger used 1 Gigabit Ethernet connections to evaluate netem, Wu et al. have done research on connections of 10 Gigabit Ethernet [21]. Research was done on the usability of network emulation and measurement tools, but also on the optimal network parameters to use on 10 Gigabit Ethernet connections.

None of these provide insight into the emulation on high performance links. However, we can use the results of those studies in this research for the purpose of comparing test results.

## 2.2 Existing tools

The research discussed above is done by using the following tools:

### 2.2.1 Network emulation tools

There are multiple solutions available for simulating or emulating the characteristics of a network. The simulating of network characteristics consists of building a model of the environment that needs to be simulated. This model has to contain similar components of the network to be simulated. So in this case where we want to simulate a long distance high-speed link, we would have to create the delay, by adding factors to the network that would generate identical characteristics.

With emulation, software is used that can exactly reproduce the behaviour of the emulated network in such a way that it appears to be the same link. We will focus on emulating the network characteristics, because this approach is more feasible for testing the behaviour of applications, without any additional costs to actually build the underlying model of the network or link. This can be done using different network emulation tools. Tools such as netem [22], NIST Net [23] and DummyNet [24] can set parameters like the delay, packet loss and jitter on a link. They are all designed very similarly, but there are some differences.

**netem** is a kernel module that is included in the kernel by default since version 2.6.7, when it was still named "delay". It was renamed to "netem" since kernel version 2.6.8 and since 2.4.28 in the 2.4 branch. netem is an enhancement of the Quality of Service (QoS) and Differentiated services (diffserv) features that are available in the Linux kernel. These services are used to add the properties to the egress traffic flow.

Initially, the netem functionality had to be compiled into the Linux kernel, but is compiled by default on modern kernels. For example, a default installation of a CentOS 5.5 [25] distribution has a kernel with built-in netem functionality.

In addition to netem, there are other tools available that provide similar functionality.

**NIST Net** is very similar to netem. NIST Net is available in a loadable kernel module. After examining the netem source code we see that there are functions in netem that originate from the NIST Net source code. However, netem does emulation on egress traffic, while NIST Net emulates on the ingress traffic.

**DummyNet** was originally developed for FreeBSD. However, there is a Linux version available.

**web100** is a Kernel Instrument Set (SET). It is an advanced management interface for TCP, so statistics on all significant protocol events can be captured. Besides this it can also change characteristics of an TCP transmission through an API. [26]

**EmuLab** is a network testbed, which is maintained by the "Flux Group, University of Utah", on which you can test several network characteristics through a web-interface for free. [27]

NIST Net and netem are kernel modules. In this research the focus lies on netem, because netem widely used and located in the kernel by default. It also uses parts of the NIST Net and DummyNet code.

### 2.2.2 Network measurement tools

**Iperf** was developed by NLAND/DAST for measuring the maximum TCP and UDP bandwidth performance [28]. Iperf works with a client-server setup and generates a traffic stream from the client to the server. After testing is done a report is sent back to the client. Reports can be shown per given time interval and can be saved to CSV format. This feature helps to easily create plots of the network throughput. An example research of Iperf [29] shows how useful Iperf can be in the testing of network throughput in a web100 emulated setup.

**netperf** is a benchmark tool to measure the performance of many different types of networking [30]. The same client-server method as Iperf uses is used. However, we found it not as reliable as Iperf, as the reported throughput seemed strange.

**nuttcp** is also a network performance benchmark tool. It is based on nttcp, which was an enhancement of ttcp.[31]. The reporting capabilities are not as advanced as they are in Iperf. This might be because Iperf spends more CPU cycles than nuttcp to show the right throughput. This was also pointed out in the research by Wu et al. [21].

We found that Iperf was the appropriate tool to use as a traffic generator. The choice was mainly based on the (reliable) reporting capabilities that Iperf provides.

# 3 Methodology

As discussed in Section 2.2.1 and 2.2.2 , we chose netem to emulate network properties and Iperf to generate network traffic. Due to the time limitations of this project (4 to 5 weeks), we postponed the extensive testing of the other tools as future research.

## 3.1 International link

In the introduction in Section 1 , we discussed how to obtain network characteristics. We were provided with access to two nodes on a international lightpath link. One node was located at the University of Amsterdam, the second node was located in San Diego. The nodes were interconnected using an optical lightpath connection that was supplied by SURFnet, the Dutch NREN [32]. 10 GigE optical network interfaces supplied the network connectivity.

### 3.1.1 Obtaining the characteristics

With access to both nodes on the link we could obtain the characteristics of the network. Depending on the amount of captured RTT data, we can consider the results more reliable, because of varying network conditions like congestion on the link. The following is a traceroute of the node in Amsterdam to the node in San Diego:

Traceroute from Amsterdam to San Diego over lightpath.

```
[rp32@amsterdam ~]$ mtr 67.58.46.160
1. 67.58.61.233                                  104.8
2. 3703calit2.calit2.optiputer.net                185.1
3. 1201e1200-12016509.calit2.optiputer.net        190.8
4. 67.58.46.160                                  186.0
```

As the traceroute shows, there are only 3 intermediate hops to the end node.

To measure the RTT, `ping` is used for a time span of 24 hours.

Ping from Amsterdam to San Diego over lightpath.

```
[rp32@amsterdam ~]$ ping 67.58.46.160
PING 67.58.46.160 (67.58.46.160) 56(84) bytes of data.
64 bytes from 67.58.46.160: icmp_seq=1 ttl=61 time=184 ms
64 bytes from 67.58.46.160: icmp_seq=2 ttl=61 time=184 ms
...
64 bytes from 67.58.46.160: icmp_seq=4478 ttl=61 time=184 ms
64 bytes from 67.58.46.160: icmp_seq=4479 ttl=61 time=184 ms

--- 67.58.46.160 ping statistics ---
70016 packets transmitted, 70015 received, 0% packet loss, time 70018061ms
rtt min/avg/max/mdev = 184.871/184.914/185.014/0.100 ms
```

As discussed in the introduction in Section 1 , the complete characteristics are the RTT, the jitter, and the jitter distribution are needed to fully emulate the characteristics. With netem, this can be done by using custom distribution tables which are located in `/usr/lib/tc/` on a 32-bit installation or in `/usr/lib64/tc/` on a 64-bit installation. The distribution table can be derived from the collected ping data. This is done with the following command:

Extracting the RTT data from the ping data.

```
[rp32@amsterdam ~]$ cat pingdata.txt | grep icmp_seq | \
cut -d'=' -f4 | cut -d' ' -f1 > rttdata.txt
```

Next, we generate the actual distribution table that is compatible with netem. This is done using the "`maketable`" tool from the `iproute2` package. The custom distribution table is copied into the appropriate directory to use with netem.

Extracting the distribution table from the RTT data.

```
[rp32@amsterdam ~]$ ./maketable rttdata.txt > distdata.dist
[rp32@amsterdam ~]$ cp distdata.dist /usr/lib64/tc/
```

The "stats" utility, also part of the iproute2 package, can be used to view information on the average RTT, the variation of the RTT, and the distribution of the RTT overtime. This information can be viewed by doing the following:

Extracting characteristics using stats.

```
[rp3@amsterdam ~]$ ./stats sdiego.rtt
mu =      184.000071
sigma =     0.008450
rho =      -0.000071
```

### 3.1.2 Testing the throughput

Now the characteristics of the link were known, we measured the throughput of the lightpath using both the UDP and the TCP protocol. On both nodes, nuttcp and Iperf was installed. We used Iperf to do the measuring of the throughput.

**UDP** As UDP is a connectionless protocol, we measured the throughput of a packet stream in a best effort test. Using Iperf, we gained a network throughput of approximately 5 Gigabit per second. In the following example, the node in San Diego acted as the server, while the node in Amsterdam was acting as the traffic generator for sending UDP packets.

UDP throughput test.

```
[uva@sandiego ~]$ iperf -s -u -l63K -w <window size>
[rp32@amsterdam ~]$ iperf -c <server IP> -u -l63K -w <window size>
```

**TCP** As TCP is a connection oriented protocol, sessions must be maintained. The packets that are required to do this can cause a lot of overhead because of the acknowledging of packets. Unfortunately, no root access was allowed on the box, which limited us to optimise the network parameters to achieve the maximum throughput. In Table 3, an overview of TCP parameters on both machines can be seen.

| Parameter | Value Amsterdam | Value San Diego |
|---|---|---|
| /proc/sys/net/core/rmem_default | 65536 | 129024 |
| /proc/sys/net/core/rmem_max | 167108864 | 16777216 |
| /proc/sys/net/core/wmem_default | 65536 | 129024 |
| /proc/sys/net/core/wmem_max | 167108864 | 16777216 |
| /proc/sys/net/ipv4/tcp_mem | 88080384 88080384 88080384 | 196608 262144 393216 |
| /proc/sys/net/ipv4/tcp_rmem | 4096 33554432 63554432 | 4096 87380 16777216 |
| /proc/sys/net/ipv4/tcp_wmem | 4096 33554432 63554432 | 4096 16384 16777216 |
| /proc/sys/net/core/optmem_max | 524288 | 20480 |
| /proc/sys/net/ipv4/tcp_moderate_rcvbuf | 1 | 1 |
| /proc/sys/net/core/netdev_max_backlog | 1000 | 1000 |
| /proc/sys/net/ipv4/tcp_sack | 1 | 1 |
| /proc/sys/net/ipv4/tcp_timestamps | 1 | 1 |
| /proc/sys/net/ipv4/tcp_congestion_control | reno | bic |
| txqueuelen | 1000 | 1000 |
| MTU | 9000 | 9000 |

Table 3: TCP parameters of Amsterdam and San Diego nodes.

In the following example, the node in San Diego acted as the server, while the node in Amsterdam was acting as the traffic generator for sending TCP packets.

TCP throughput test.

```
[uva@sandiego ~]$ iperf -s -w <window size>
[rp32@amsterdam ~]$ iperf -c <server IP> -w <window size>
```

This test only showed us a throughput of around 1 Gigabit per second. This is possibly due to the TCP parameters that are not optimally configured. The sending node in Amsterdam has a maximum sending TCP window size of approximately 160 MB, while the receiving side in San Diego only has a receiving TCP window of 16 MB. Iperf can use the double amount of this, to guarantee higher throughput. We calculated the maximum throughput of the link to be around 1400 Megabit per second, which should be limited to the receiving side in this case. We expect the TCP autotuning algorithm and TCP overhead to cause the average throughput to be around 1 Gigabit per second.

The limitation in UDP throughput might be caused due to hardware limitations on an intermediate lightpath switch, which divides the total amount of capacity of the switch over multiple optical ports. This problem also limits the maximum TCP throughput if optimal network parameters were applied.

### 3.1.3 Lab setup

The lab setup consisted of three machines with identical specifications. The initial setup was a setup of two generic end nodes and a machine that emulated the characteristics by adding properties to the egress traffic. In this setup, the two end nodes were connected to each other through the delay box that was configured as a bridge (daisy chaining). The details on the configuration of all three machines can be found in Appendix B. More information on the network interface cards (NICs) is specified in Appendix C.

We took the approach to perform tests on interfaces of 10 Gigabit Ethernet and 40 Gigabit Ethernet to emulate the characteristics of the international link to San Diego.

Figure 2 is a visualisation of the first lab setup. The setup consisted of three nodes equipped with dual port 10 GigE PCI-Express network interface cards. Fiber optic cables provided the connectivity between the nodes. Specifications on the used NICs can be found in Appendix C.

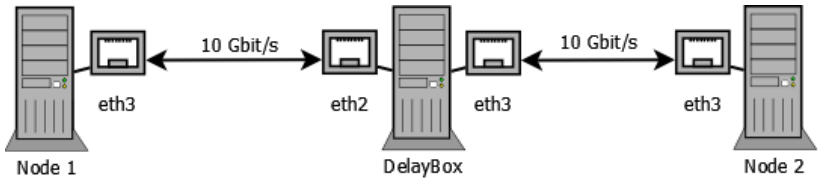



Figure 2: 10GigE Lab setup visualisation.

Figure 3 shows the second setup, which is are two nodes in a 2U enclosure. Because of the lack of 40 GigE network interfaces, we had to connect the setup back-to-back. This limited us to applying the emulation on the same node that sent the traffic to the receiver node.

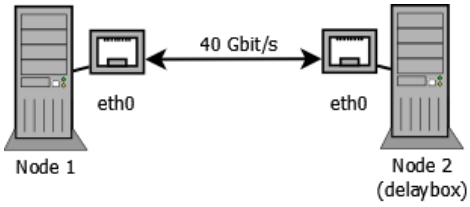



Figure 3: 40GigE Lab setup visualisation.

## 3.2 Test procedures

The properties of the example destination is then emulated on the lab setup using 10 GigE and 40 GigE links.

We emulated those characteristics on the lab setup on 10 GigE interfaces, and investigate if netem can reliably emulate the characteristics. The influence of the resolution of the kernel is also studied as on some high-speed links very low delay times are present, which will result in very fine-grained differences in the RTTs. Because netem greatly depends on the resolution of the Linux kernel [19] to emulate, delay emulation can only be done on high resolution kernels (>1000HZ) to emulate delays smaller than 1ms.

At last, we moved to a lab setup with 40GigE connectivity to see if netem is suitable on very high-speed links by using the same variation of settings.

### 3.2.1 Emulating network characteristics

As said in section 2.2.1, we choose netem to emulate the characteristics of the real-world link. Netem is depending on the resolution of the kernel to apply fine-grained emulation on outgoing packets [22].

The resolution of the kernel determines when a process is allowed to run. If a kernel is running at a resolution of 1000 Hz (1000 ticks per second), a process is allowed a time slice of 1 millisecond to run. This is also known as a kernel tick rate of 1 millisecond.

When processing very high speed traffic and applying emulation at the same time, the kernel needs to run at a very high tick rate. The next step after 1000 Hz is the Real-Time kernel. This kernel has a guaranteed system response time [33], which means it will achieve the lowest possible latency at any cost.

Another kernel we tested is the Tickless kernel. This kernel has been developed to save energy when idle, but also ticks "on demand", which we thought to be interesting to look at.

As discussed in Section 2.2.1, netem only provides the possibility to emulate network properties on egress traffic. This can be done by using the "`tc`" Traffic Control command, which is also part of the `iproute2` package.

By giving the following parameters, characteristics that were previously acquired are emulated.

Adding delay with tc.

```
[root@box ~]$ tc qdisc add dev <NIC> root netem \
delay <delay in ms>ms
```

Adding delay and jitter with tc.

```
[root@box ~]$ tc qdisc add dev <NIC> root netem \
delay <ms delay>ms <ms jitter>ms
```

Adding delay jitter and distribution with tc.

```
[root@box ~]$ tc qdisc add dev <NIC> root netem \
delay <ms delay>ms <ms jitter>ms distribution <table>
```

In this case, we emulate a delay of 184.000071 milliseconds, in combination with 0.008450 milliseconds of jitter. The jitter is varying over time, this is specified in the distribution table that was calculated using the `maketable` tool. In this case, the distribution table is saved in `/usr/lib64/tc/distdata.dist`.

### 3.2.2 Generating traffic

At this stage, the emulated characteristics are applied to all egress traffic on the specified interface. To measure the network speed, we need to send data over the path. These data streams have to be generated to move data from one node to the other node via the delaybox.

On the receiving node, Iperf is started in server mode, while on the sending node, Iperf connects to the server and can start sending data. Throughput reports are written to CSV format every two seconds, so the resulting traffic could be analysed.

## 3.3 Test plan

Before we start the actual testing, we made sure that the connectivity from both nodes to the delay box were working correctly. This was done by testing the maximum throughput back-to-back. We started Iperf as a server on the delaybox, and tested the links with Iperf as clients on the nodes. As Wu et al. already stated in its paper [21] , a default MTU of 1500 bytes is inadequate for 10 Gigabit/s links. After testing the throughput with both values, we came to

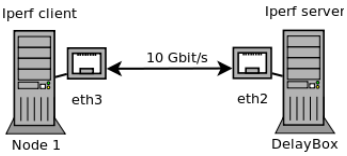



Figure 4: Test the link between Node1 and Delaybox.

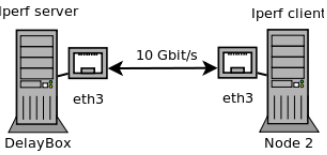



Figure 5: Test the link between Node2 and Delaybox.

the same conclusion and used a MTU of 9000 bytes for the full path for all the following tests.

After the initial tests to confirm the links are working as expected, we tested if the delaybox could bridge the amount of traffic. This we did with every kernel to confirm that not any other part of the kernel is delaying or slowing down the throughput.

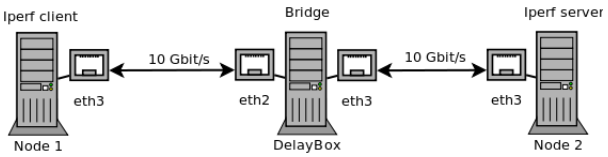



Figure 6: Test the bridging capabilities of the Delaybox.

As discussed in Section 3.2.1, we will be using the following kernels to test which one can achieve the best emulation:

- 100HZ
- 1000HZ
- Tickless
- Real Time

All these kernels will be tested using optimal TCP kernel parameters, which we configured with the values shown in Table 4.

| Parameter | Value |
|---|---|
| /proc/sys/net/core/rmem_default | 524288000 |
| /proc/sys/net/core/rmem_max | 524288000 |
| /proc/sys/net/core/wmem_default | 524288000 |
| /proc/sys/net/core/wmem_max | 524288000 |
| /proc/sys/net/ipv4/tcp_mem | 524288000 524288000 524288000 |
| /proc/sys/net/ipv4/tcp_rmem | 4096 104857600 524288000 |
| /proc/sys/net/ipv4/tcp_wmem | 4096 104857600 524288000 |
| /proc/sys/net/core/optmem_max | 524288000 |
| /proc/sys/net/core/netdev_max_backlog | 250000 |
| /proc/sys/net/ipv4/tcp_sack | 0 |
| /proc/sys/net/ipv4/tcp_timestamps | 0 |
| /proc/sys/net/ipv4/tcp_congestion_control | htcp |
| txqueuelen | 10000 |
| MTU | 9000 |

Table 4: TCP parameters of the lab setup.

Per kernel we tested the throughput with different characteristics applied. These tests are:

- Baseline of local link (no delay)
- added delay
- added delay + jitter
- added delay + jitter + distribution

We also tested what the impact was of using different TCP window sizes. The BDP of a link with almost no delay (local link) is different from the BDP of a link with high delays. For the tests we used the window sizes shown in table 5.

| 10 GigE | 40 GigE |
|---|---|
| 16KB | 16KB |
| 512KB | 512KB |
| 50MB | 50MB |
| 100MB | 100MB |
| 200MB | 500MB |
| 233MB (optimal) | 950MB (optimal) |
| 500MB | 1000MB |

Table 5: TCP window sizes.

Per window size we tested the throughput for 5 minutes, and plotted these tests into graphs, which can be seen in Section 4. To get these detailed results, we used the CSV output functionality of Iperf.

# 4 Measurements

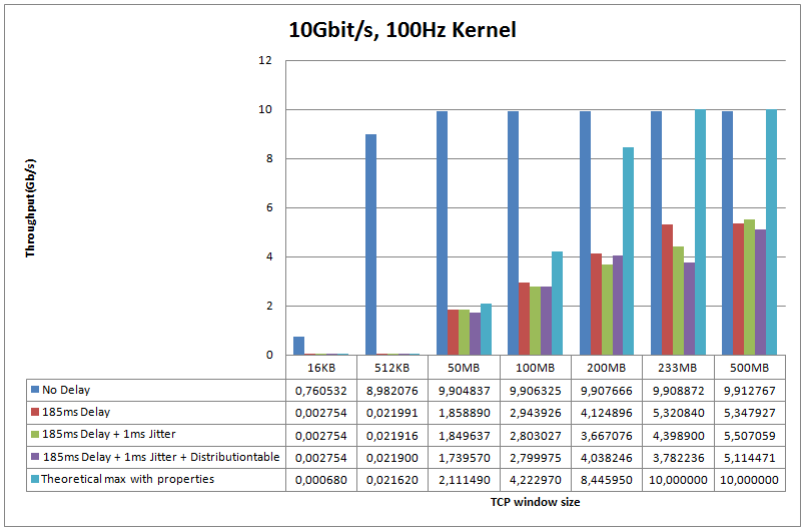
In this Section, we will discuss the results of the tests that are described in the test plan in Section 3.3. We will not discuss the results on the 1GigE setup, as there are many other researches done on this speed [19],[34]. The different kernels and window sizes that are used will be discussed for both 10 GigE and 40 GigE connections.

## 4.1 10 Gigabit Ethernet

This paragraph shows the results of the tests done on the 10 GigE setup.

### 4.1.1 100 Hz kernel

Figure 7 shows the test results while running a kernel with a 100 Hz resolution on the delaybox.




| | 16KB | 512KB | 50MB | 100MB | 200MB | 233MB | 500MB |
|---|---|---|---|---|---|---|---|
| No Delay | 0,760532 | 8,982076 | 9,904837 | 9,906325 | 9,907666 | 9,908872 | 9,912767 |
| 185ms Delay | 0,002754 | 0,021991 | 1,858890 | 2,943926 | 4,124896 | 5,320840 | 5,347927 |
| 185ms Delay + 1ms Jitter | 0,002754 | 0,021916 | 1,849637 | 2,803027 | 3,667076 | 4,398900 | 5,507059 |
| 185ms Delay + 1ms Jitter + Distributiontable | 0,002754 | 0,021900 | 1,739570 | 2,799975 | 4,038246 | 3,782236 | 5,114471 |
| Theoretical max with properties | 0,000680 | 0,021620 | 2,111490 | 4,222970 | 8,445950 | 10,000000 | 10,000000 |

Figure 7: 10GigE 100Hz Kernel.

The optimal TCP window size that we calculated is about 650 KB. When we configured this window size on both sending and receiving nodes, with no delay applied on the delaybox, we measured a total throughput of 9.9 Gigabit per second, which can be considered a fully utilised connection.

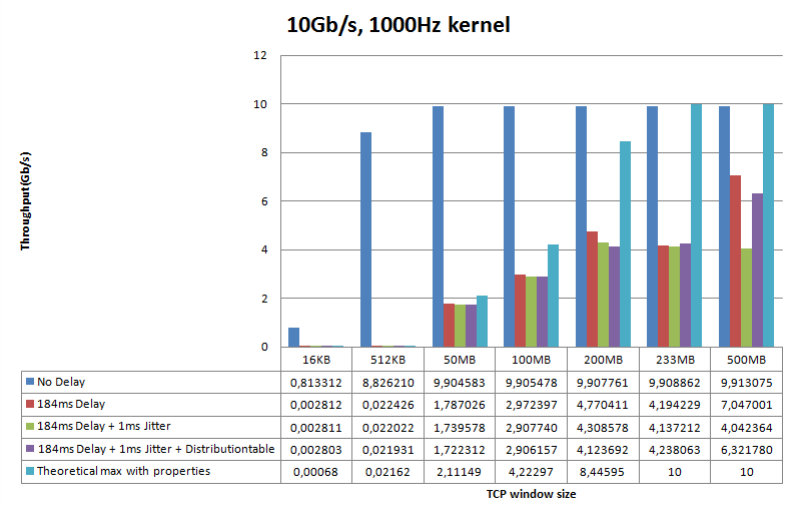
However, when only delay is applied on the traffic, the expected throughput of the TCP stream is not achieved. With the optimal TCP window size configured, which is 233MB for a 184ms delay, our measurements only reached 5.3 Gigabit

per second. This is only 53.54% of the expected throughput. The cause of this drop can be traced back to netem. We suspect netem isn't multi-threaded as we only see 1 core that is dedicated to the kernel and is fully utilised when the emulation is being applied. This can be led back to the low results.

When adding the jitter and distribution table in addition to the delay, results are similar as before. In addition to the single threaded netem kernel module, another downside of using a 100 Hz kernel is that it seems unable to maintain the proper amount of delay on the link. When the characteristics are emulated, we noticed a difference of 10ms in the RTT. Instead of a fairly stable RTT of 184 milliseconds, we have seen spikes to 194 milliseconds. This basically reflects as a link with a high amount of jitter, which makes it an "unreliable" link.

### 4.1.2 1000 Hz kernel

Figure 8 shows the test results using a 1000 Hz kernel on the delaybox. The measurements show different results as the tests done using a 100 Hz kernel , with the exception of test using a TCP window size of 500 MB. At this time, we can't explain why or how this happened. We expect it to be around the same throughput of the optimal window size, like the delay with jitter result.




| | 16KB | 512KB | 50MB | 100MB | 200MB | 233MB | 500MB |
|---|---|---|---|---|---|---|---|
| No Delay | 0,813312 | 8,826210 | 9,904583 | 9,905478 | 9,907761 | 9,908862 | 9,913075 |
| 184ms Delay | 0,002812 | 0,022426 | 1,787026 | 2,972397 | 4,770411 | 4,194229 | 7,047001 |
| 184ms Delay + 1ms Jitter | 0,002811 | 0,022022 | 1,739578 | 2,907740 | 4,308578 | 4,137212 | 4,042364 |
| 184ms Delay + 1ms Jitter + Distributiontable | 0,002803 | 0,021931 | 1,722312 | 2,906157 | 4,123692 | 4,238063 | 6,321780 |
| Theoretical max with properties | 0,00068 | 0,02162 | 2,11149 | 4,22297 | 8,44595 | 10 | 10 |

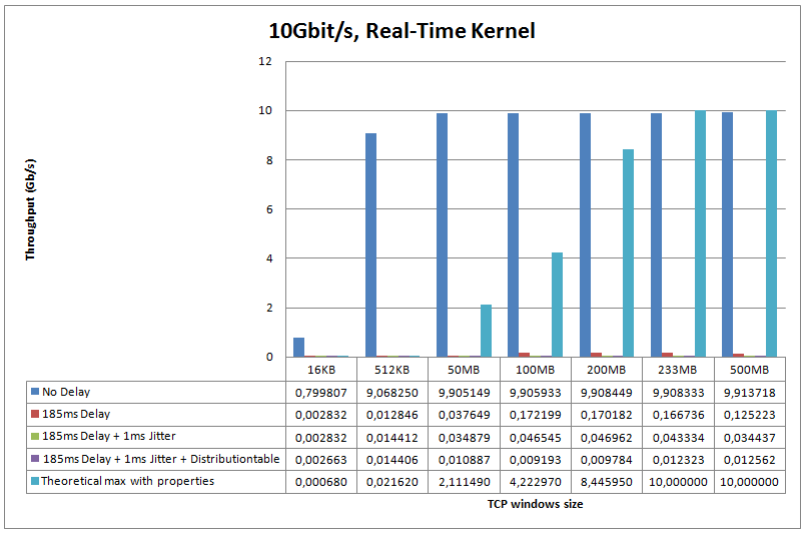
Figure 8: 10GigE 1000Hz Kernel.

When focusing on the test with the optimal TCP window size of 233 MB, we see that when using a 1000 Hz kernel, the different tests show more steady results than when a 100 Hz kernel is used. This can be led back to the resolution of the kernel, which can apply delay more carefully due to it's faster tick rate. The test with delay, jitter and jitter distribution applied seem similar though, but

the tests done using 1000 Hz look more steady. This does not mean that emulation is completely successful. Only around 40% of the expected throughput is measured.

### 4.1.3 Real-Time kernel

In figure 9, the measurements of the Real-Time kernel are shown. The graph shows that netem can not emulate properties on a 10 GigE link using this (default) Real-Time kernel. When there is no delay applied, the throughput is as expected. However, when only adding delay and using an optimal window size, throughput drops to only 166 Megabits per second. When adding the jitter and distribution table, throughput drops to around 12 Megabits per second. This can be explained by the huge amount of interrupts the CPU has to handle to be real-time. The consequence of this is that the kernel can't process the packets fast enough.

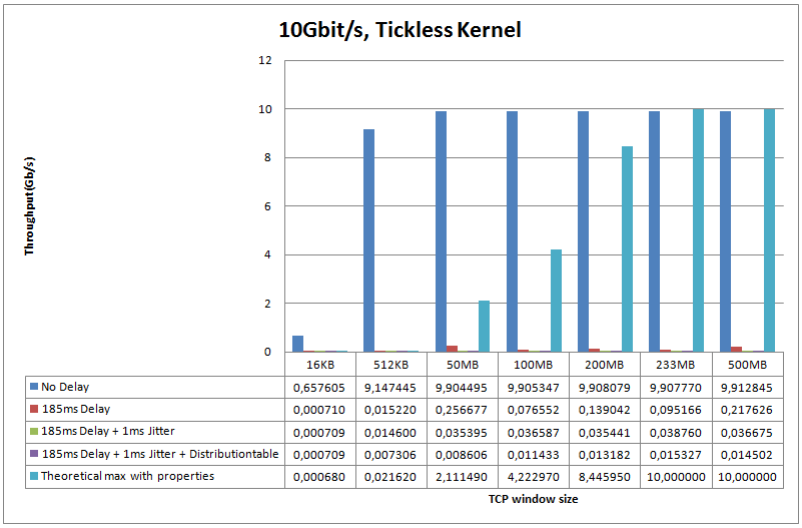



| | 16KB | 512KB | 50MB | 100MB | 200MB | 233MB | 500MB |
|---|---|---|---|---|---|---|---|
| No Delay | 0,799807 | 9,068250 | 9,905149 | 9,905933 | 9,908449 | 9,908333 | 9,913718 |
| 185ms Delay | 0,002832 | 0,012846 | 0,037649 | 0,172199 | 0,170182 | 0,166736 | 0,125223 |
| 185ms Delay + 1ms Jitter | 0,002832 | 0,014412 | 0,034879 | 0,046545 | 0,046962 | 0,043334 | 0,034437 |
| 185ms Delay + 1ms Jitter + Distributiontable | 0,002663 | 0,014406 | 0,010887 | 0,009193 | 0,009784 | 0,012323 | 0,012562 |
| Theoretical max with properties | 0,000680 | 0,021620 | 2,111490 | 4,222970 | 8,445950 | 10,000000 | 10,000000 |

Figure 9: 10GigE Real-Time Kernel.

### 4.1.4 Tickless kernel

Figure 10 are the measurements from the Tickless kernel. We expected it to perform about the same as the 1000 Hz kernel, because it was configured with a maximum resolution of 1000 Hz. This figure doesn't come close to the 1000 Hz measurements, and we suspect netem can't get ticks. This results in only a maximum throughput of 95Mb/s with only delay applied, and a maximum throughput of 15Mb/s when the jitter and distribution are applied.




| | 16KB | 512KB | 50MB | 100MB | 200MB | 233MB | 500MB |
|---|---|---|---|---|---|---|---|
| No Delay | 0,657605 | 9,147445 | 9,904495 | 9,905347 | 9,908079 | 9,907770 | 9,912845 |
| 185ms Delay | 0,000710 | 0,015220 | 0,256677 | 0,076552 | 0,139042 | 0,095166 | 0,217626 |
| 185ms Delay + 1ms Jitter | 0,000709 | 0,014600 | 0,035395 | 0,036587 | 0,035441 | 0,038760 | 0,036675 |
| 185ms Delay + 1ms Jitter + Distributiontable | 0,000709 | 0,007306 | 0,008606 | 0,011433 | 0,013182 | 0,015327 | 0,014502 |
| Theoretical max with properties | 0,000680 | 0,021620 | 2,111490 | 4,222970 | 8,445950 | 10,000000 | 10,000000 |

Figure 10: 10GigE Tickless Kernel.

### 4.1.5 Kernel comparison

In Figure 11, the results of the optimal window size tests are plotted for each kernel. These plots are each averages of two tests done. The plot clearly shows that the 1000 Hz kernel is the best choice when trying to add emulation on a 10 GigE link. The spikes in the 100 Hz plot can be explained by the H-TCP congestion control algorithm. This is because H-TCP initially increases the window size very fast to achieve best performance and ties its window reduction to the estimated buffers of the network and the time since the last congestion event [14] [35].

## 4.2 40 Gigabit Ethernet

This paragraph shows the results of our tests on the 40 GigE setup. After the tests done on 10GigE, we didn't expect the throughput to be very high. Also because this setup didn't consist of two nodes and a separate delaybox, the traffic generator also had to act as a delaybox(see section 3.1.3). Also, we expected the full theoretical limit of 40 Gigabit per second could not be reached because of the "limited" speed of the PCI Express bus [36].

The 100 Hz results can be seen in figure 12. We expected to see around 20 Gigabit per second, which is achieved without any emulation applied. When only applying delay, we see a throughput of 2 Gigabit per second, which is what netem can emulate in this case. When jitter and distribution is added, the throughput reach beyond 360 Megabit per second.

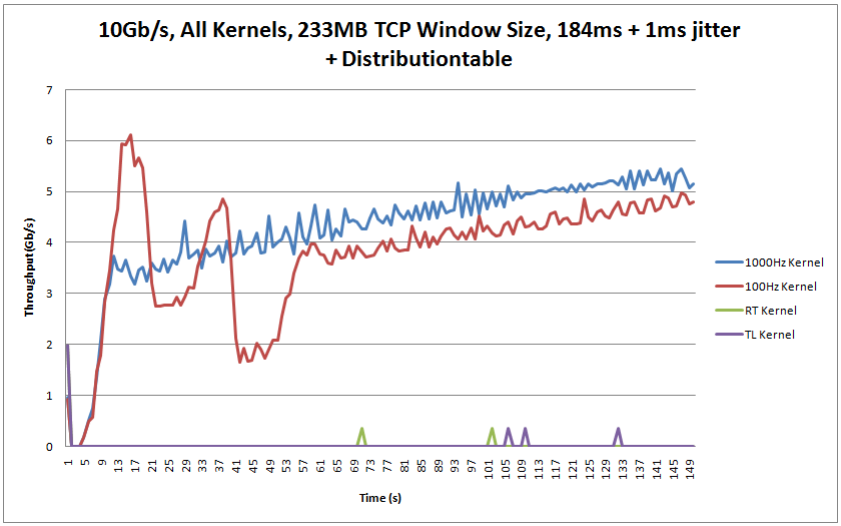
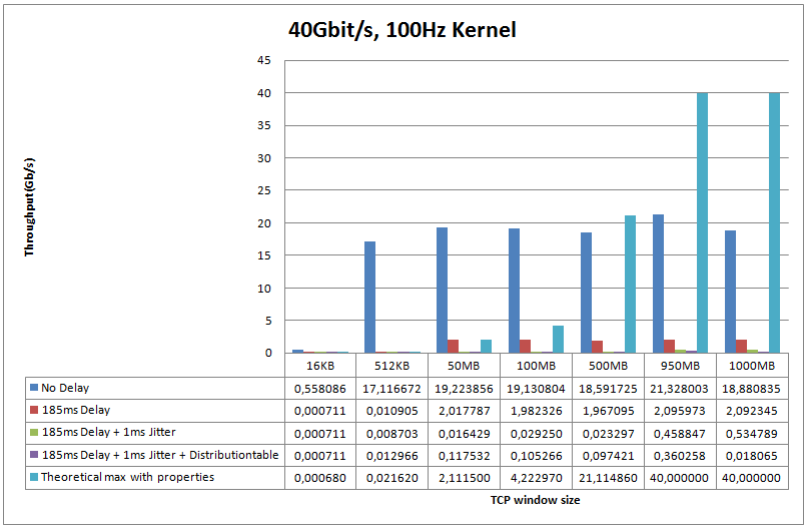



Figure 11: 10GigE all kernels.

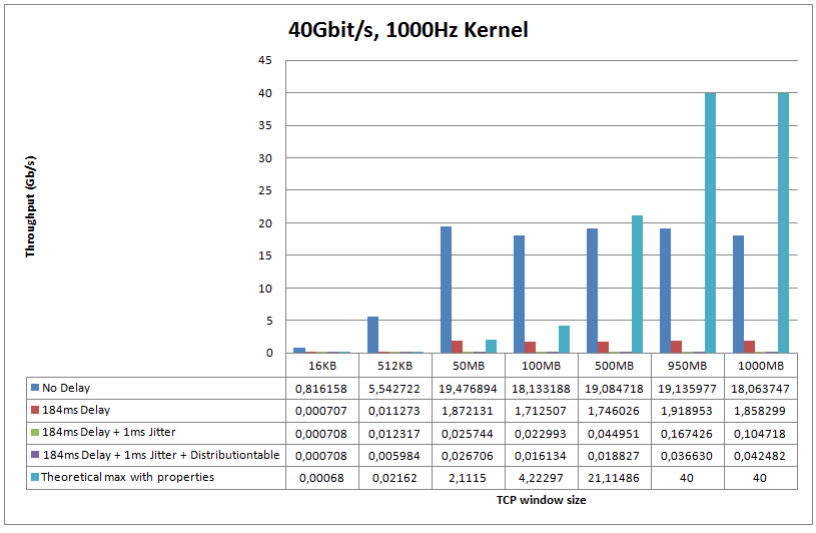
The 1000 Hz, Real-Time and Tickless kernel achieve the same results, as can be seen in figures 13,14 and 15. We suspect the huge amount of packets netem has to process is just too much. Also because netem is working on the sending node, we suspect the CPU is too busy sending packets so it is unable to reach the 4 Gigabit throughput which is reached on the 10 GigE setup. We suspect the faster CPUs in these 40 GigE nodes to be the reason why the results on the Real-Time and Tickless can still be 2Gb/s with only delay added.

Figure 16 shows the throughput of all the kernels with the optimal TCP window size set. It is reasonable to say the performance isn't good, as discussed above.




| | 16KB | 512KB | 50MB | 100MB | 500MB | 950MB | 1000MB |
|---|---|---|---|---|---|---|---|
| No Delay | 0,558086 | 17,116672 | 19,223856 | 19,130804 | 18,591725 | 21,328003 | 18,880835 |
| 185ms Delay | 0,000711 | 0,010905 | 2,017787 | 1,982326 | 1,967095 | 2,095973 | 2,092345 |
| 185ms Delay + 1ms Jitter | 0,000711 | 0,008703 | 0,016429 | 0,029250 | 0,023297 | 0,458847 | 0,534789 |
| 185ms Delay + 1ms Jitter + Distributiontable | 0,000711 | 0,012966 | 0,117532 | 0,105266 | 0,097421 | 0,360258 | 0,018065 |
| Theoretical max with properties | 0,000680 | 0,021620 | 2,111500 | 4,222970 | 21,114860 | 40,000000 | 40,000000 |

Figure 12: 40GigE 100Hz Kernel.




| | 16KB | 512KB | 50MB | 100MB | 500MB | 950MB | 1000MB |
|---|---|---|---|---|---|---|---|
| No Delay | 0,816158 | 5,542722 | 19,476894 | 18,133188 | 19,084718 | 19,135977 | 18,063747 |
| 184ms Delay | 0,000707 | 0,011273 | 1,872131 | 1,712507 | 1,746026 | 1,918953 | 1,858299 |
| 184ms Delay + 1ms Jitter | 0,000708 | 0,012317 | 0,025744 | 0,022993 | 0,044951 | 0,167426 | 0,104718 |
| 184ms Delay + 1ms Jitter + Distributiontable | 0,000708 | 0,005984 | 0,026706 | 0,016134 | 0,018827 | 0,036630 | 0,042482 |
| Theoretical max with properties | 0,00068 | 0,02162 | 2,1115 | 4,22297 | 21,11486 | 40 | 40 |

Figure 13: 40GigE 1000Hz Kernel.

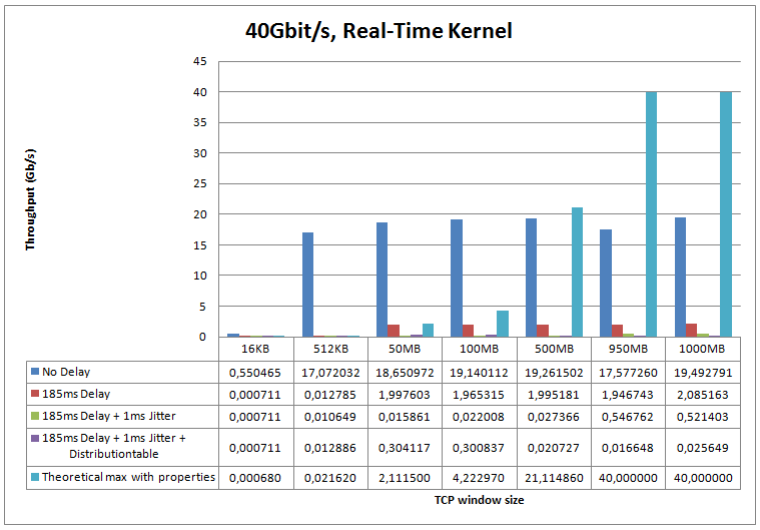



Figure 14: 40GigE Real-Time Kernel.

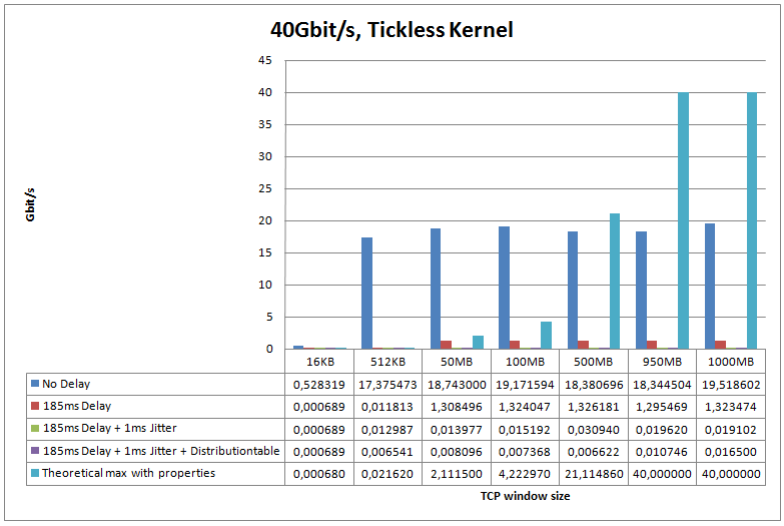



Figure 15: 40GigE Tickless Kernel.

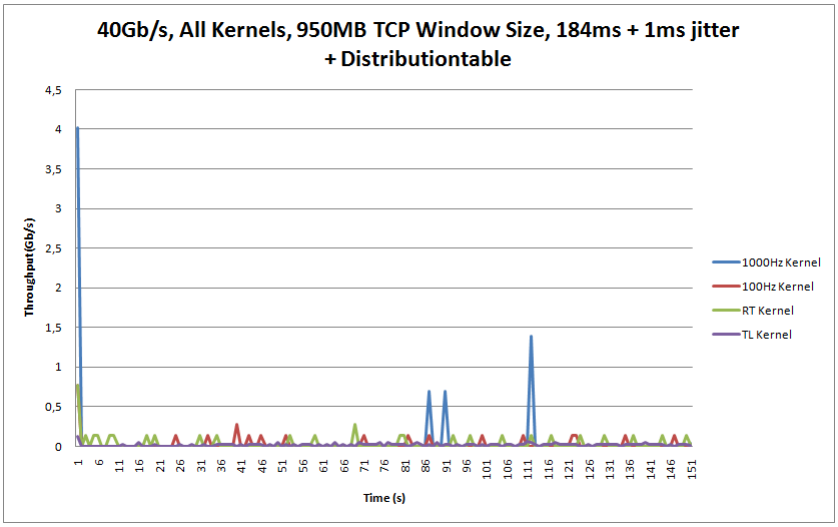



Figure 16: 40GigE all kernels.

# 5 Conclusion and recommendations

In this section we state our conclusions of the results that were discussed in the previous sections.

Even though this research was not about achieving maximum throughput on certain network situations, we have gained much knowledge about the different parameters that can be tweaked to achieve optimal network performance. The lab setup was first tweaked to achieve maximum throughput when no delay was applied. Studying the throughput while emulating network characteristics with netem show some interesting results when this delay is applied.

It was already proved that netem can be used reliably on 1 GigE links and lower. However, on interfaces with 10 GigE and 40 GigE optical connectivity, expected throughput cannot be achieved when emulating characteristics on egress traffic. Using different kernel resolutions does not mitigate this problem. Even using a (default) Real-Time kernel does not mitigate this problem. Too many CPU cycles are needed to process the network interrupts caused by the large amount of incoming traffic, which causes the drop in throughput. On the 40 GigE setup, a huge drop in performance is measured on all the different kernels.

We suspect that netem is not optimised for such a high throughput link and can not cope with the large amount of packets coming in, even though the network parameters are optimally configured. If you have a maximum link speed of 4 to 5 Gigabit per second, we advise to only use netem on a delaybox running a kernel at a resolution of 1000 Hz. The 100 Hz kernel causes more delay than initially configured, which causes unreliable jitter on the link.

# 6 Future work

Due to time limitationsit was not feasible to study every aspect extensively. They are discussed in this Section.

**Interrupt Coalescence** More tweaking can be done to ultimately tweak all possible parameters. During this research, we only tweaked the parameters that seemed the most obvious to us. One additional parameter we came to is by tweaking the Interrupt Coalescence of the links. This feature is available in some network drivers, and can mitigate the fact that an interrupt is generated for every frame that is received on the network card. Interrupt Coalescence will "buffer" the incoming frames to generate only one interrupt for multiple frames [37]. Because buffering is needed, additional delay can occur. It might be interesting to see what effects this can cause to the throughput.

The ideal configuration can also be a high burden to achieve in the real world. However, we think it is interesting to see what different parameters mean to the behaviour of a network. Although it is not always possible to configure a NIC with a different Interrupt Coalescence value (driver limitations), it might be interesting to include it in tests.

**Tweaking Real-Time kernel** The Real-Time kernel used the measurements was the default kernel. Kernel parameters can be tweaked to allocate more CPU cycles to the network card, which can optimise the processing of interrupts caused by the network card.

**Tweaking international link** Performing tweaking on the international link might provide better insight into the real-world link. To prevent overhead in the communication to the administrators in San Diego, root access needs to be acquired on both machines.

**40 GigE re-test** The measurements done on the 40 GigE setup were done using only two 40 GigE network interfaces. We think it is interesting to see the results when 40 GigE can be considered "production ready", when there are three dual-port cards, or 4 single-port cards available. This way, a bridged setup can be realised, and tests can be re-done.

**Emulation tool comparison** This research only provides insight when using netem. It might be interesting to compare different network emulation tools using the 10 GigE and 40 GigE setup. Once again, due to time limitations, it was only feasible to do measurements on the throughput while using netem.

# A DelayBox setup script

To easily recreate the delaybox we setup in the lab setup 3.1.3, we created a simple bash script that sets the optimal TCP kernel parameters at boot time.

DelayBox setup script

```
ifconfig eth2 down
ifconfig eth3 down
ifconfig eth2 0.0.0.0
ifconfig eth3 0.0.0.0
brctl addbr br0
brctl addif br0 eth2
brctl addif br0 eth3
ifconfig eth2 promisc
ifconfig eth3 promisc
ifconfig br0 up
ifconfig eth2 up
ifconfig eth3 up
ifconfig eth2 mtu 9000
ifconfig eth3 mtu 9000
ifconfig br0 mtu 9000
ifconfig eth2 txqueuelen 10000
ifconfig eth3 txqueuelen 10000
ifconfig br0 txqueuelen 10000

sysctl -w net.ipv4.tcp_timestamps=0
sysctl -w net.ipv4.tcp_sack=0
sysctl -w net.core.netdev_max_backlog=250000
sysctl -w net.core.rmem_max=524288000
sysctl -w net.core.wmem_max=524288000
sysctl -w net.core.rmem_default=524288000
sysctl -w net.core.wmem_default=524288000
sysctl -w net.core.optmem_max=524288000
sysctl -w net.ipv4.tcp_mem="524288000 524288000 524288000"
sysctl -w net.ipv4.tcp_rmem="4096 104857600 524288000"
sysctl -w net.ipv4.tcp_wmem="4096 104857600 524288000"
sysctl -w net.ipv4.tcp_congestion_control="htcp"
```

# B Server hardware

In this section, we will describe the specifications of the used machines.

- Node 1

**Brand** Dell

**Model** PowerEdge R210

**CPU** Intel(R) Xeon(R) CPU L3426 @ 1.87GHz

**Memory** 8GB

**NIC** Embedded Broadcom 5716 (x2)

**Default kernel** 2.6.18-194.32.1.el5 #1 SMP Wed Jan 5 17:52:25 EST 2011 x86_64 x86_64 x86_64 GNU/Linux

- Node 2

  **Brand** Dell

  **Model** PowerEdge R210

  **CPU** Intel(R) Xeon(R) CPU L3426 @ 1.87GHz

  **Memory** 8GB

  **NIC** Embedded Broadcom 5716 (x2)

  **Default kernel** 2.6.18-194.32.1.el5 #1 SMP Wed Jan 5 17:52:25 EST 2011 x86_64 x86_64 x86_64 GNU/Linux

- Delay Box

  **Brand** Dell

  **Model** PowerEdge R210

  **CPU** Intel(R) Xeon(R) CPU L3426 @ 1.87GHz

  **Memory** 8GB

  **NIC** Embedded Broadcom 5716 (x2)

  **Default kernel** 2.6.18-194.32.1.el5 #1 SMP Wed Jan 5 17:52:25 EST 2011 x86_64 x86_64 x86_64 GNU/Linux

  **RT kernel** 2.6.33.7.2-rt30 #10 SMP PREEMPT RT Mon Jan 31 11:10:14 CET 2011 x86_64 x86_64 x86_64 GNU/Linux

  **TL kernel** 2.6.33.7.2 #10 SMP Mon Jan 31 11:10:14 CET 2011 x86_64 x86_64 x86_64 GNU/Linux

- Delay Box and Node 2 - 40GigE

  **Brand** Supermicro

  **Model** X8DTT-H

  **CPU** Intel(R) Xeon(R) CPU E5620 @ 2.40GHz

  **Memory** 24GB

  **NIC** Embedded Intel Corporation 82574L (2x)

  **Default kernel** 2.6.18-194.11.4.el5 #1 SMP Tue Sep 21 05:04:09 EDT 2010 x86_64 x86_64 x86_64 GNU/Linux

**RT kernel** 2.6.33.7.2-rt30 #10 SMP PREEMPT RT Mon Jan 31 11:10:14 CET 2011 x86_64 x86_64 x86_64 GNU/Linux

**TL kernel** 2.6.33.7.2 #10 SMP Mon Jan 31 11:10:14 CET 2011 x86_64 x86_64 x86_64 GNU/Linux

# C Network Interface Cards

In this section the network hardware is specified.

- 1GigE

  **Brand** Broadcom Corporation

  **Model** NetXtreme II BCM5716

  **Revision** Revision 20

  **Standard** Gigabit Ethernet

- 10GigE
  - End nodes

    **Brand** Mellanox Technologies
    **Model** MT26448 (ConnectX EN 10GigE PCIe 5.0GT/s ()
    **Revision** Revision b0
    **Standard** 10 Gigabit Ethernet
  - Delay Box

    **Brand** Chelsio Communications
    **Model** T320 10GbE Dual Port Adapter
    **Revision** N/A
    **Standard** 10 Gigabit Ethernet

- 40GigE

  **Brand** Mellanox Technologies

  **Model** MT26478

  **Revision** Revision b0

  **Standard** 40 Gigabit Ethernet

# References

[1] Ixia. Leader in converged ip testing. http://ixiacom.com/, January 2011. [Online; Consulted on January 3, 2011].

[2] Measuring network delay. Website, http://etd.adm.unipi.it/theses/available/etd-06242004-163624/unrestricted/4MeasuringNetworkDelay.doc. [Online; consulted on January 21st, 2011].

[3] Wikipedia: Round-trip delay time. Website, http://en.wikipedia.org/wiki/Round-trip_delay_time. [Online; consulted on January 13th, 2011].

[4] Wikipedia: Speed of light. Website, http://en.wikipedia.org/wiki/Speed_of_light. [Online; consulted on January 27th, 2011].

[5] Wolframalpha: Distance from amsterdam to san diego. Website, http://www.wolframalpha.com/input/?i=distance+from+amsterdam+to+san+diego. [Online; consulted on January 27th, 2011].

[6] G. Huston. Next Steps for the IP QoS Architecture. RFC 2990 (Informational), November 2000.

[7] Wikipedia: Maximum transmission unit. Website, http://en.wikipedia.org/wiki/Maximum_transmission_unit. [Online; consulted on January 20th, 2011].

[8] Ping man page. Shipped with most Linux distributions, http://linux.die.net/man/8/ping.

[9] J. Postel. Internet control message protocol. RFC 792, September 1981.

[10] Wikipedia: Jumbo frames. Website, http://en.wikipedia.org/wiki/Jumbo_frames. [Online; consulted on January 19th, 2011].

[11] V. Jacobson, R. Braden, and D. Borman. Tcp extensions for high performance, 1992.

[12] Wikipedia: Tcp window scale option. Website, http://en.wikipedia.org/wiki/TCP_window_scale_option. [Online; consulted on January 27th, 2011].

[13] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996.

[14] Yee-Ting Li Douglas Leith Robert N. Shorten. Experimental evaluation of tcp protocols for high-speed networks. http://www.hamilton.ie/net/eval/ToNfinal.pdf.

[15] Hamilton h-tcp presentation. Website, http://www.hamilton.ie/net/htcp-ietf63.pdf. [Online; consulted on January 7th, 2011].

[16] Alexandru Giurgiu Jeroen Vanderauwera. Communication channel performance measurement, August 2010.

[17] Esma Yildirim, Ibrahim H. Suslu, and Tevfik Kosar. Which network measurement tool is right for you? a multidimensional comparison study.

[18] Grenville Armitage and Lawrence Stewart. Some thoughts on emulating jitter for user experience trials. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, NetGames '04, pages 157–160, New York, NY, USA, 2004. ACM.

[19] Stephen Hemminger. Network emulation with netem. `http://devresources.linuxfoundation.org/shemminger/netem/LCA2005_paper.pdf`, April 2005. [Open Source Development Lab research].

[20] Stephen Hemminger. Netem - emulating real networks in the lab. `http://devresources.linuxfoundation.org/shemminger/LCA2005_netem.pdf`, April 2005.

[21] Yixin Wu, Suman Kumar, and Seung-Jong Park. Measurement and performance issues of transport protocols over 10gbps high-speed optical networks. *Computer Networks*, 54(3):475 – 488, 2010.

[22] netem homepage. `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`. [Online; consulted on January 5th, 2011].

[23] Nist net home page. Website, `http://snad.ncsl.nist.gov/nistnet/`. [Online; consulted on January 12th, 2011].

[24] Dummynet home page. Website, `http://info.iet.unipi.it/~luigi/ip_dummynet/`. [Online; consulted on January 12th, 2011].

[25] Centos, the community enterprise operating system. Website, `http://www.centos.org`. [Online; consulted on January 28th, 2011].

[26] Matt Mathis John Heffner Raghu Reddy. Web100: Extended tcp instrumentation. `http://www.web100.org/docs/mathis03web100.pdf`.

[27] Emulab testbed. Website, `http://www.emulab.net`. [Online; consulted on January 13th, 2011].

[28] Iperf website. Website, `http://sourceforge.net/projects/iperf/`. [Online; consulted on Februari 3th, 2011].

[29] Ajay Tirumala, Les Cottrell, and Tom Dunigan. Measuring end-to-end bandwidth with iperf using web100. In *Web100, Proc. of Passive and Active Measurement Workshop*, page 2003, 2003.

[30] Netperf website. Website, `http://www.netperf.org/netperf/`. [Online; consulted on Februari 3th, 2011].

[31] nuttcp website. Website, http://nuttcp.org. [Online; consulted on Februari 3th, 2011].

[32] Surfnet - dutch nren. Website, http://www.surfnet.nl. [Online; consulted on February 4th, 2011].

[33] Rt preemt howto. https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO.

[34] P. Vicat-Blanc Primet R. Takano Y. Kodama T. Kudoh O. Gluck C. Otal. Large scale gigabit emulated testbed for grid transport evaluation. http://www.ens-lyon.fr/LIP/RESO/Software/EWAN.sov/primet_pascale.pdf. [National Institute of Advanced Industrial Science and Technology (AIST). AXE, Inc, Japan].

[35] Long Le Injong Rhee Sangtae Ha, Yusung Kim and Lisong Xu. A step toward realistic evaluation of high-speed tcp protocols. Website, http://www4.ncsu.edu/~rhee/export/bitcp/asteppaper.htm. [Online; consulted on February 4th, 2011].

[36] Server performance tuning, pci express. Website, https://noc.sara.nl/wiki/Server_Performance_Tuning#PCI_Express. [Online; consulted on February 4th, 2011].

[37] Interrupt coalescence. http://kb.pert.geant.net/PERTKB/InterruptCoalescence.