

RP2 project: Communication Channel Performance Measurement

University of Amsterdam
Master of Science in System and Network Engineering

Class of 2009-2010

Alexandru Giurgiu (alex.giurgiu@os3.nl)
Jeroen Vanderauwera (jeroen.vanderauwera@os3.nl)

August 22, 2010

Abstract

Bottlenecks in high performance networking are difficult to pinpoint. In this report we try to identify which software and hardware parameters are the cause for these bottlenecks. The focus on this study is on the end points and not the link itself. It is shown that CPU load, memory swapping, bus speeds, MTUs and buffer sizes are parameters that cause the most of the performance issues. Further, we developed a performance measurement tool that gathers data from both end points, combines this information and gives a clear overview of the measured values for the different parameters. This way, warnings can be given to the end user.

Contents

| | | |
|----------|--|------------|
| 1 | Introduction | 4 |
| 1.1 | Research question | 4 |
| 2 | The layered view | 5 |
| 3 | Parameters that influence network performance | 8 |
| 3.1 | Hardware parameters | 9 |
| 3.1.1 | Network interface | 9 |
| 3.1.2 | PCI Express bus | 9 |
| 3.1.3 | CPU | 10 |
| 3.1.4 | Memory | 13 |
| 3.2 | Software parameters | 14 |
| 3.2.1 | Path MTU | 14 |
| 3.2.2 | TCP window and large window extensions | 17 |
| 3.2.3 | TCP Buffer size(window size) | 18 |
| 3.2.4 | UDP buffer size | 19 |
| 3.2.5 | Flow control | 19 |
| 3.2.6 | TCP Selective Acknowledgements Option | 20 |
| 4 | Creating a diagnostic tool | 21 |
| 4.1 | Integrated tools | 21 |
| 4.2 | Program structure | 21 |
| 4.3 | Client/server communication | 22 |
| 4.4 | Output of the tool | 23 |
| 4.5 | Challenges | 23 |
| 5 | Recommendations for optimal network performance | 25 |
| 6 | Conclusions | 26 |
| 7 | Future research | 28 |
| 8 | Acknowledgements | 29 |
| A | Tool source code | A-1 |
| A.1 | main.rb | A-1 |
| A.2 | compare.rb | A-8 |
| A.3 | netcom.rb | A-13 |
| A.4 | logging.rb | A-15 |

1 Introduction

Network performance tuning seems currently more an art than a science. The network performance between two end hosts can be influenced and limited by a sheer number of parameters. This includes the capacities and bandwidth usage of individual links in the network, the memory buffer of routers and switches along the way, as well as the memory size, CPU power, bus speed, and hardware configuration of end hosts. Furthermore, runtime parameters such as number of parallel streams, protocols on the network and TCP window size greatly affect the achieved performance. Multiple tools exist today to measure the overall obtainable performance. However, these tools often report different results and hardly any tool exists to determine the current and optimal values of individual system parameters, which makes it hard to determine the cause of reduced network performance.

1.1 Research question

Our research will lay more emphasis on the end points of the connection, rather than the network which is located between those end points. We want to investigate if it is possible to classify and analyze how some parameters influence network performance. Secondly we want to develop a tool that somehow combines the results of measurements done with other existing tools, and reports them in a clear and centralized way. This tool should be able to automatically gather and analyze information and as a result help pinpointing the bottleneck, which will be located somewhere in the layered model that can be seen in section 2. Therefore our research questions are:

- Is it possible to identify and classify the parameters which affect network performance?
- Is it possible to develop a tool that monitors the parameters and can pinpoint the cause of the reduced network performance?

2 The layered view

Figure 1 represents the TCP/IP model while figure 2 represents the hardware components that have direct influence on the network performance. On each layer are parameters that can introduce bottlenecks.

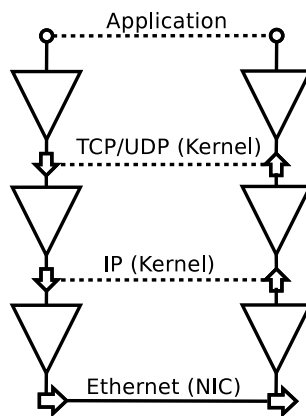


Figure 1: Layered view of the TCP/IP protocol stack

On the hardware aspect, bottlenecks can be introduced by CPU usage, bus speed, PCIe slot used, memory consumption and Network Interface Card (NIC) or its firmware. More about this, can be read in the sections below.

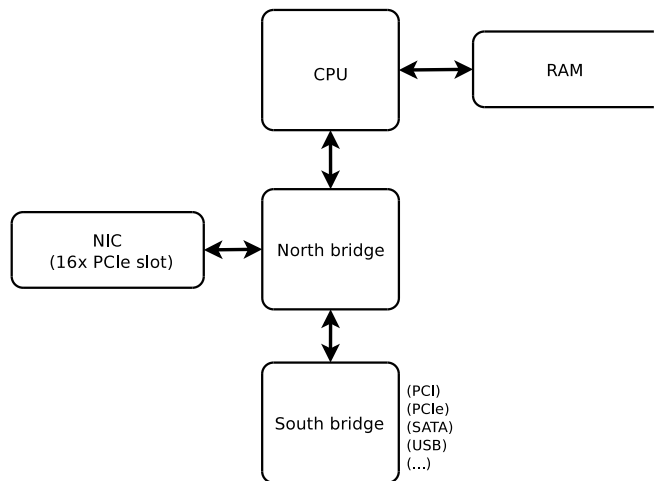


Figure 2: Hardware architecture of the Intel 5520 [1] and 5500 [2] chipset

The Ethernet layer At the Ethernet layer communication is done using frames, which can have a payload size ranging from 46 bytes to 9000

bytes when using Ethernet Jumbo frames. When using frames with a bigger MTU the overall efficiency of the protocol becomes higher, if there are no errors. In the real world, when there are a lot of transmission errors on a link and TCP is used, the corrupted frames will need to be retransmitted, which means that protocol overhead is influenced by the frame size and RTT. If UDP is used, this will result in packet loss. In our case, the machines are connected back to back with a short cable making it a reliable link. This means that a higher MTU should lead to better results.

IP layer Going up one layer, at the IP level, packets are used to send and receive data. The maximum size of a packet can be 64 kbytes, from which the header accounts for 20 bytes (more than 20 bytes in case the option field is used, extended maximum by another 40 bytes). Depending on the MTU at the Ethernet layer, one IP packet can be fragmented and sent over the wire in multiple Ethernet frames. Similarly to Ethernet, using a bigger IP packet size leads to a lower protocol overhead as there will be less headers to transmit for the same amount of data. If the header checksum mismatches the whole packet gets discarded resulting in packet retransmission for TCP and packet loss for UDP.

TCP TCP communicates using segments that have a 20 byte header and a total maximum size equal to the payload of the IP datagram. TCP achieves reliability by acknowledging all packets and retransmitting lost or damaged packets. The TCP window size determines how many bytes the receiver is willing to receive at this moment. If it is bigger, acknowledgments will have to be sent less often or vice versa if it is smaller. Protocol overhead is influenced by segment size and RTT.

UDP UDP is a connectionless protocol that does not offer any reliability. Applications that use UDP have to take care of reliability on their own if it is needed. In cases like VoIP or video streaming it is not required because old data becomes irrelevant after a short time. UDP has less overhead because it has a smaller header and does not use any acknowledgments, buffering, retransmission, etc. Theoretically it has a higher performance on very reliable links.

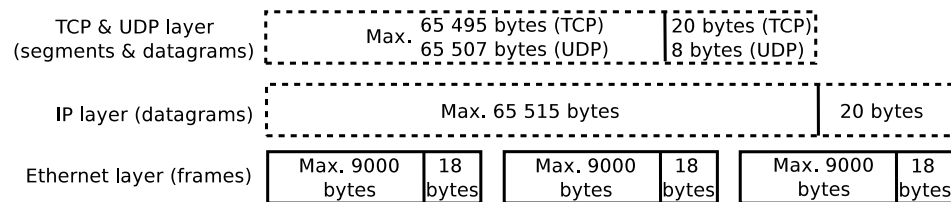


Figure 3: Layered view of packet sizes and constraints

As can be seen in figure 3, depending on the size of the Ethernet frames and the IP packets, the latter can be fragmented and send on the wire

using multiple frames. It should be noted that in case the IP protocol uses the *option* field in the header, the total payload amount will decrease by length of the option field. On every layer, tools are used to measure and determine these bottlenecks by testing different parameters. These tools are listed in the section below.

3 Parameters that influence network performance

In the following sections we explain the measurements we did to find out the influence of hardware and software parameters on network performance. Herefore a test environment is needed. The configuration can be seen in figure 4.

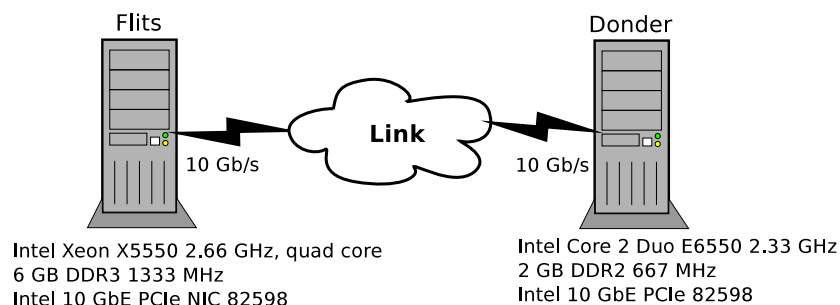


Figure 4: Test environment

Both machines are connected directly to each other over a 10 Gb/s copper link. We used the following tools to measure and determine the bottlenecks in our test environment:

Iperf[3] is a network testing tool used for measuring the TCP and UDP throughput of a network path. Iperf has a wide range of options and allows to test a large part of the software parameters that influence network performance. We preferred iperf because it implements shaping, which makes it usable for UDP. Additionally the implementation of the server is relatively simple and the output of the client and server are independent.

Ethtool[5] is a tool for displaying and changing ethernet card settings. The tool is used to modify the flow control and full duplex settings, and to monitor network interface statistics (buffers, dropped packets, ...)

Netstat[6] is used to display UDP and TCP protocol statistics as present in the kernel.

`/proc/net/dev` lists network statistics directly as recorded in the NIC driver. The number of Ethernet packets and total amount of bytes transferred can be read from here.

lookbusy[7] is an application which can generate predictable synthetic load on a Linux system. During the tests it is used to generate different amounts of CPU loads to see how it influences the network performance.

stress[8] is an application that is used to fill up the system memory.

`sysctl`[9] is a tool for examining and dynamically changing parameters in the Linux kernel. It is particularly used for changing TCP related settings.

3.1 Hardware parameters

When transmitting a large amount of data, several hardware parameters in the system can cause performance issues. In this section we will describe which possible bottlenecks can be identified as a source of the performance problems. We performed some measurements as well, mostly on the 2 different Internet Protocols: UDP and TCP.

3.1.1 Network interface

The network interface is the most straightforward bottleneck on a computer system. As a network is only as fast as the slowest link, one network interface can decrease the performance of the whole network connection. For example, network traffic which is transmitted at a data rate of 10 Gb/s, cannot be received at the same rate if the receiver has a 1 Gb/s network interface. For our tests we used a 10 Gb/s interface with a 10 Gb/s link. This is the maximum physical data rate possible in our experiments.

3.1.2 PCI Express bus

PCI Express slots transmit and receive data from the network interfaces to the CPU or conversely over so called lanes. Depending on the PCI Express slot version, the bus speed varies from 2 Gb/s to 8 Gb/s per lane. A PCI Express bus can have 1 up to 16 lanes. An overview can be found in table 1. If the system has a configuration with a bus speed of 8

| | 4 lanes | 8 lanes | 16 lanes |
|-----------------|-----------|---------|----------|
| PCIe 1.0 | 8 Gb/s | 16 Gb/s | 32 Gb/s |
| PCIe 2.0 | 16 Gb/s | 32 Gb/s | 64 Gb/s |
| PCIe 3.0 | 31.5 Gb/s | 63 Gb/s | 126 Gb/s |

Table 1: PCI Express bus speeds

Gb/s and traffic is received at 10 Gb/s on the network interface, the PCIe bus speed can be identified as a bottleneck. The systems we used each have two x16 generation 2.0 PCI express slots in which the 10 Gb NICs are inserted. The bandwidth provided by the x16 PCI express slots is 64 Gb/s, so in this case it will not be a bottleneck. Of course, PCI express has a protocol overhead because of the electrical encoding used but this has been taken into account in the above table.

However we did tested the network performance if one of the NICs was inserted in a PCIe 2.0 x1 slot. The network performance decreased dramatically to 1.5 and 1.7 Gb/s.

3.1.3 CPU

The fact that a CPU has multiple cores and threads can influence network speeds but these are not the only factors that effect performance, one other important factor being CPU frequency. As can be seen from our measurements in figure 5, UDP is just a bit faster than TCP. It pushes the data to the receiver without checking or acknowledging if the packets have arrived. On the other hand, TCP waits for ACKs before sending the next amount of data. This is also noticeable in figures 6 and 7, where UDP gets overall better performance than TCP. From figure 5 one can see that the difference between TCP and UDP performance is almost unnoticeable, leading to the conclusion that when CPU power is available the effect on TCP and UDP performance is minimal.

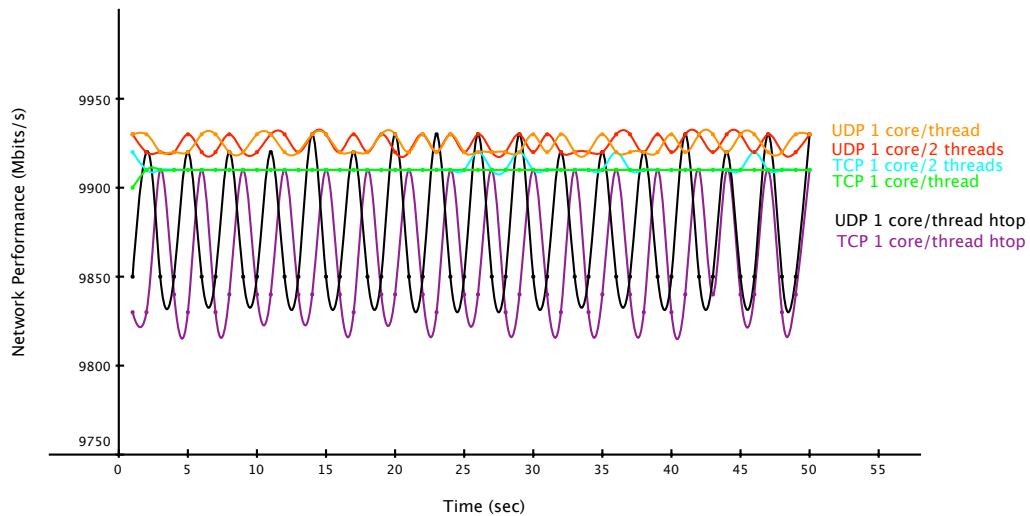


Figure 5: The effect of different amount of threads on network performance

In figures 6 and 7, the impact of CPU load on network performance is shown. We made the tests by inflicting synthetic load on each of the CPU cores and threads, using the *lookbusy* tool. Consistent with 20% CPU load, UDP network performance is not influenced at all, while TCP takes a considerable performance hit. It is clear that TCP is more dependent on the CPU than UDP is. When running htop for monitoring CPU load on the cores, one can notice that even a little amount of CPU load can influence the network performance as the CPU needs to switch between processing network traffic and htop. More about CPU load based tests, can be read below.

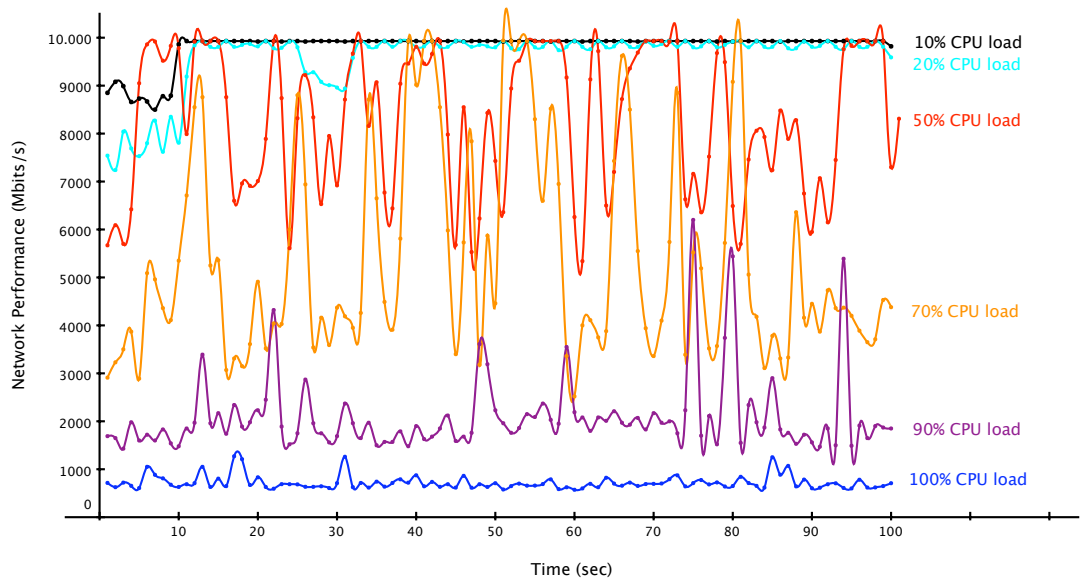


Figure 6: The effect of CPU load on UDP network performance

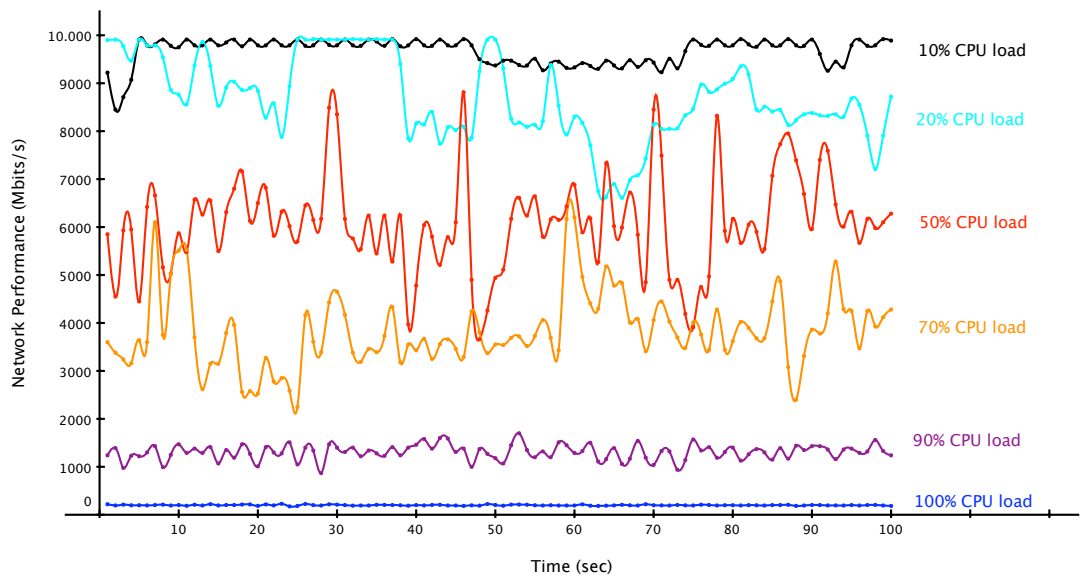


Figure 7: The effect of CPU load on TCP network performance

As can be seen in figure 6, the UDP performance is severely influenced by CPU load. At 10% and 20% CPU load, UDP performance is not affected, having the same maximum throughput. At 50% load, performance degrades considerably, from an average of 9.9 Gb/s to 7.3 Gb/s, with a high level of variation. Throughput drops to 5.4 Gb/s when the CPU is 70% loaded and the variation is even more pronounced. At 90% load UDP performance drops by a factor of about 5, while at 100% load the performance drops to 700 Mb/s. One can notice that performance is smoother at low and high CPU loads, while at intermediate loads the fluctuations are much higher.

TCP suffers even more from increased CPU loads. Even at 20% load the average performance drops to 8.7 Gb/s while at 50% the average is 6 Gb/s. Similarly to UDP, throughput is smoother at low and high CPU loads and jagged at intermediate loads.

The next performance test was performed on the receiver. As can be seen in figure 8 it appears that receiving data does not effect the CPU¹ that hard, as higher data rates are achieved with the same CPU load.

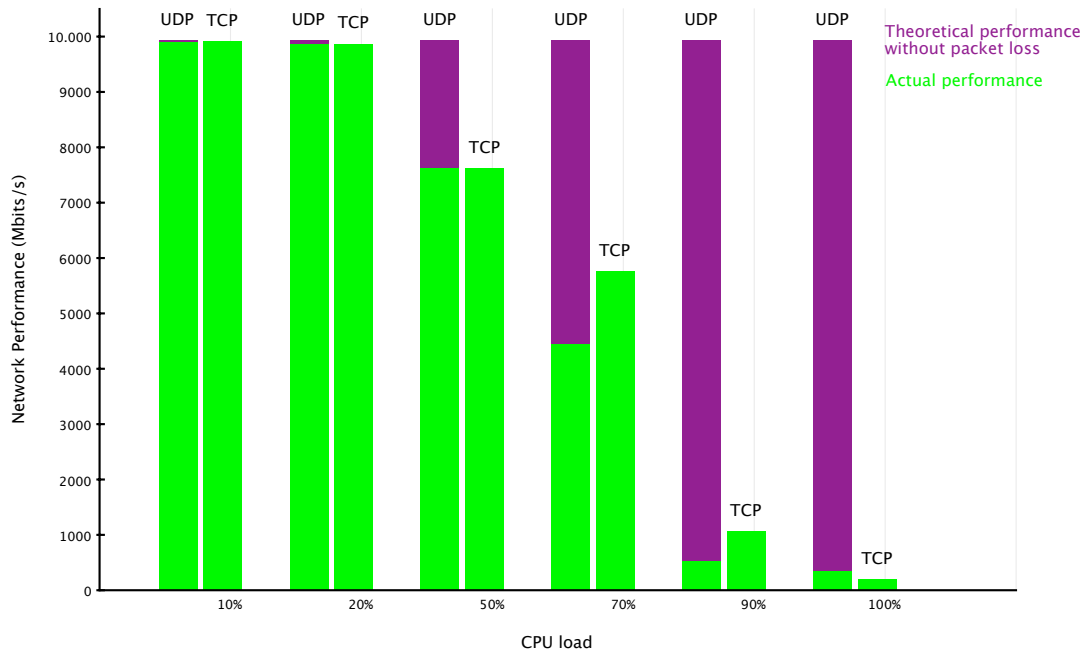


Figure 8: The effect of CPU load on network performance at the receiver

¹Note that in this case the CPU of the receiver has less cores and operates at a lower frequency than the transmitter.

3.1.4 Memory

Theoretically all data which is fed to the CPU, resides in memory first. This is why the memory can be a possible bottleneck. We tested if full memory had any effect on the network performance of both TCP and UDP. This seems not the case. Though if the memory is filled up, swapping occurs. In the graph 9 below one can determine that when the memory is filled up and starts swapping, the UDP network performance goes down, while TCP network performance fluctuates very strong. Because of the I/O interrupts, the CPU has to write content from the RAM to disk. If no more memory is needed and the RAM and the swap space is full, the memory stops swapping and network performance returns to its normal values. The measurements in the graph have been taken in two separate test rounds.

As can be seen the network performance at the receiving host is not that influenced as at the transmitting host. This is because iperf needs to generate the data it wants to send in memory. At the receiving host, the iperf server only counts the bytes and discards the data immediately. Another

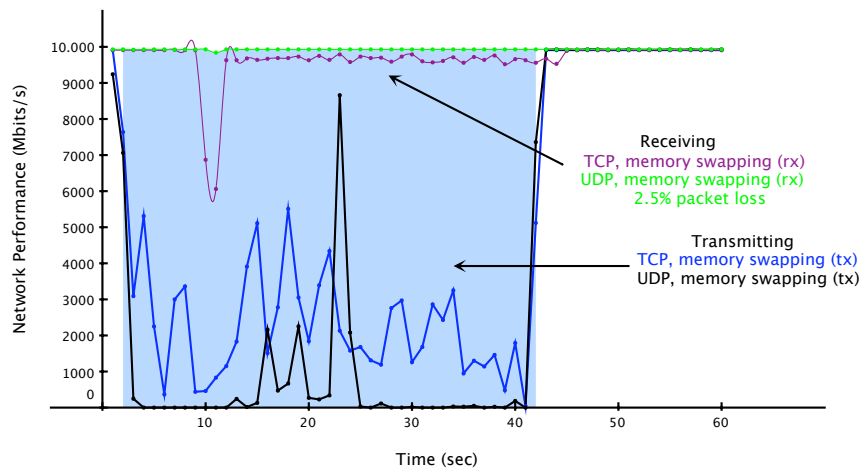


Figure 9: The effect of swapping on network performance

approach includes filling up the memory without going over the maximum capacity so swapping would not occur. This did not influence the network performance in any way. As of this, we can conclude that not the filled up memory is a bottleneck but the swapping, which introduces a lot of I/O interrupts, is. Though, a sufficient amount of RAM is still needed in order to provide the buffers with enough memory.

3.2 Software parameters

In this section we will list which parameters can be configured in order to obtain the best network performance. We will explain as well when and why these parameters could be a bottleneck.

3.2.1 Path MTU

The Maximum Transmission Unit can be modified in the following layers of the OSI model:

Data link layer In this layer it is possible to change the ethernet MTU to 1500 or even 9000 bytes if jumbo packets are used.

Network layer In this layer one can maximally use 64kbytes as MTU.

During our tests we used 3 different values for the Ethernet MTU size and 7 different values for the IP MTU size.

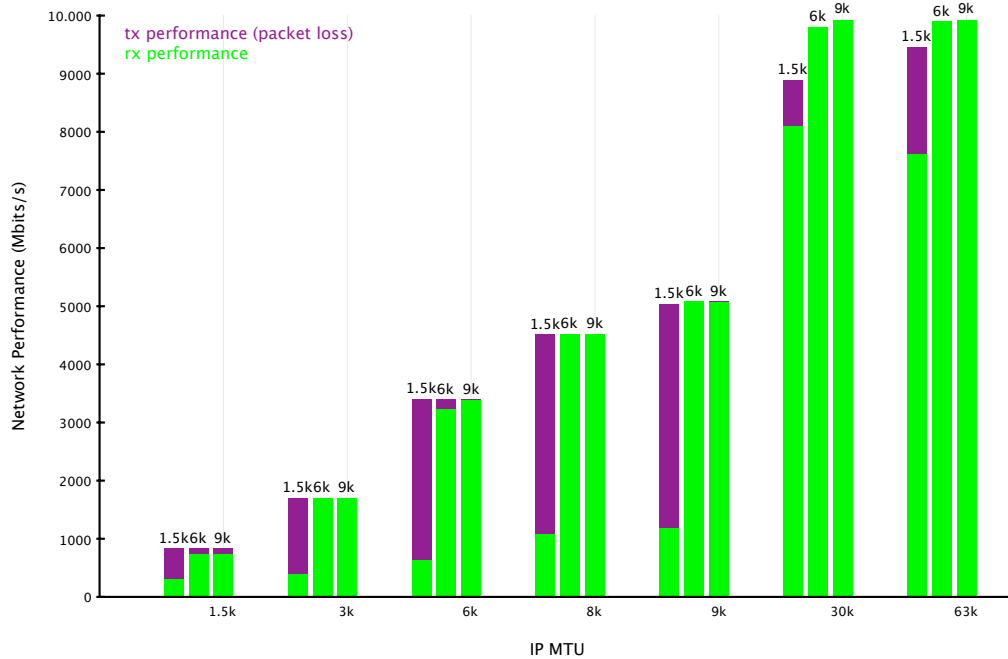


Figure 10: Ethernet and IP MTU influence on UDP performance

There are two observations to be made about UDP performance when it comes to MTU. As can be seen in figure 10, Ethernet MTU affects packet loss, while IP MTU affects throughput. We obtained maximum performance by using a 9000 bytes Ethernet MTU and a 63 kbytes² IP MTU. Comparing the results from UDP and TCP found in figures 10 and 12 we can see some strange discrepancies. We would expect that TCP performance would be on par with the actual UDP performance (green bars), but instead the values are similar to the ones that don't take into account the UDP loss (purple bars).

We reran the test with a Myri-10G NIC[10], manufactured by Myricom, which uses optical fiber technology. As can be seen in figure 11, in general the packet loss is lower. Note that the same trend can be seen in this figure as the figure above; high packet loss at a Ethernet MTU size of 1500 bytes and an increase of packet loss if the IP MTU size is 63 kbytes.

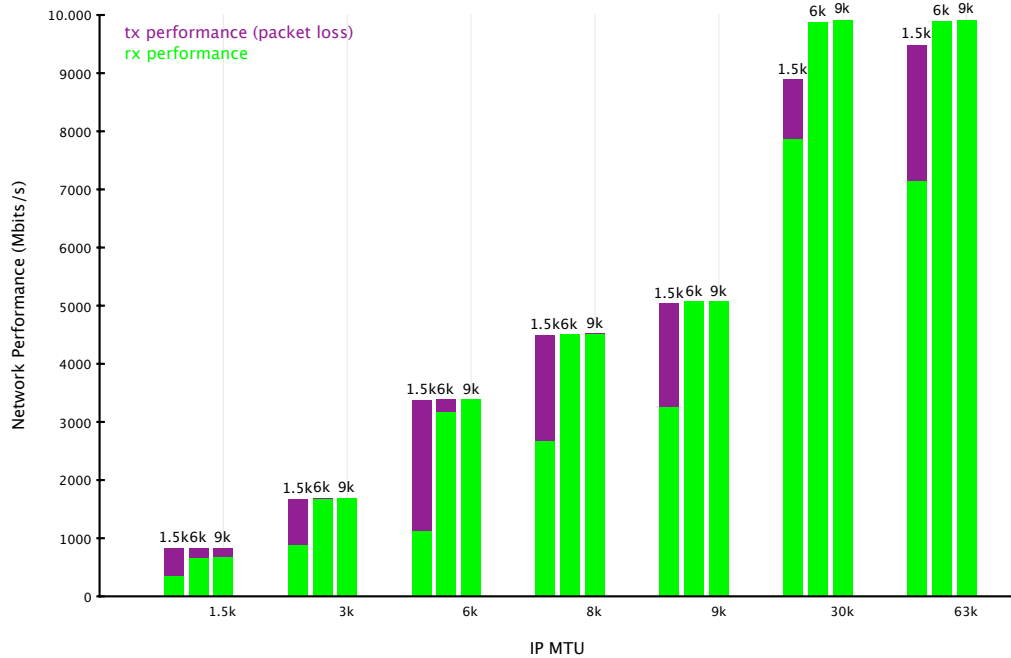


Figure 11: Ethernet and IP MTU influence, using fiber, on UDP performance

²63 kbytes is used because 64 kbytes gave malformed packet errors. This is because the MTU size, set in iperf, is the size of the payload. Hence the total size of the IP packet would be the 64 kbytes + 20 bytes, which would be too big.

Table 2 represents the packet loss of the chart above in numbers. What was less noticeable in the chart, but more clear in the table is when an IP MTU equal or more than 30 kbytes is used, the packet loss is higher when using the Myrinet 10-G NIC.

| IP MTU | Ethernet MTU | Myrinet 10-G | Intel 10GbE |
|-------------------------|-------------------|--------------|-------------|
| 1.5 kbytes | 1.5 kbytes | 58% | 67% |
| | 6 kbytes | 20% | 19% |
| | 9 kbytes | 19% | 18% |
| 3 kbytes | 1.5 kbytes | 47% | 77% |
| | 6 kbytes | 0.15% | 0.03% |
| | 9 kbytes | 0.2% | 0.01% |
| 6 kbytes | 1.5 kbytes | 67% | 81% |
| | 6 kbytes | 6.6% | 4.8% |
| | 9 kbytes | 0% | 0.16% |
| IP MTU 8 kbytes | 1.5 kbytes | 41% | 76% |
| | 6 kbytes | 0.09% | 0.23% |
| | 9 kbytes | 0.1% | 0.17% |
| IP MTU 9 kbytes | 1.5 kbytes | 35% | 77% |
| | 6 kbytes | 0.66% | 0.01% |
| | 9 kbytes | 0.16% | 0.37% |
| IP MTU 30 kbytes | 1.5 kbytes | 12% | 8.9% |
| | 6 kbytes | 0% | 0% |
| | 9 kbytes | 0% | 0% |
| IP MTU 63 kbytes | 1.5 kbytes | 25% | 19% |
| | 6 kbytes | 0% | 0% |
| | 9 kbytes | 0% | 0% |

Table 2: Myrinet 10-G vs Intel 10GbE packet loss

On TCP the Ethernet MTU does not have a very big impact on performance. On the other hand IP MTU influences performance quite a lot. The default IP MTU that most applications use is 8 kbytes, and that yields an average performance of 5 Gb/s. When the IP MTU is increased to 63 kbytes the performance goes up to 8 Gb/s.

Although headers consume proportionally less bandwidth when using bigger IP datagrams, the performance advantage does not correlate with the numbers we have from our tests on the 30k and 63k IP MTU sizes. There is no difference between the speed at this sizes, leading us to believe that

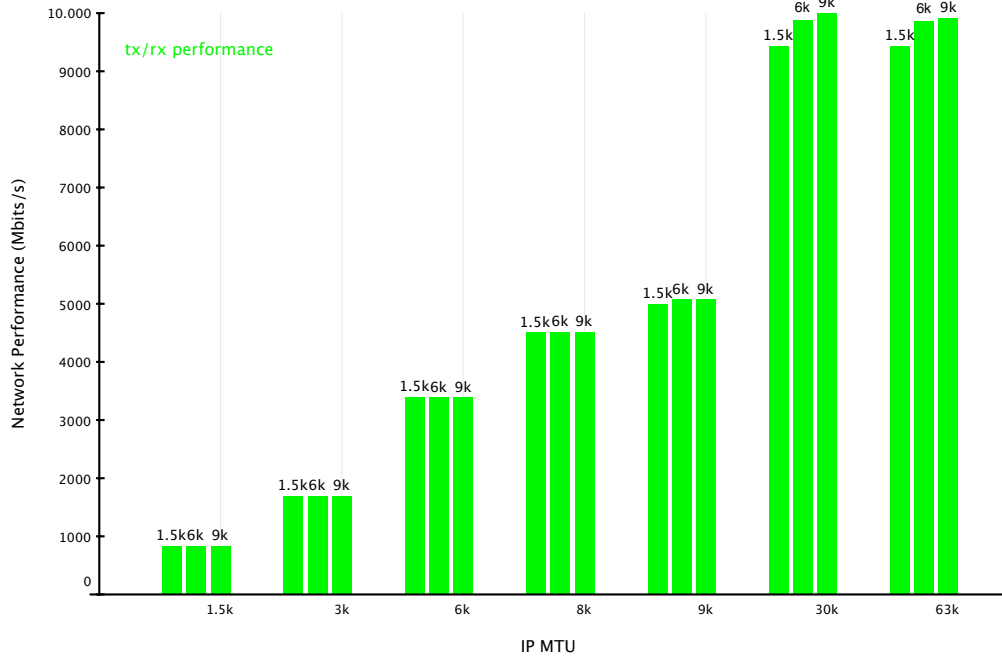


Figure 12: Ethernet and IP MTU influence on TCP performance

as IP MTU size increases, header count matters less and less.

The rerun of the TCP test on the Myri-10G NIC, did not give any anomalies and had approximately the same outcome.

3.2.2 TCP window and large window extensions

The RFC “TCP Extensions for High Performance” [12] presents a set of TCP extensions to improve performance over large bandwidth delay product paths and to provide reliable operation over very high-speed paths. One of the most important extensions is the TCP window scaling.

In most of the operating systems the TCP window is too small by default. This gives performance issues because the transmitter needs to wait for the acknowledgments before it can send any TCP packets again. The transfer speed can be calculated by the following formula:

$$Throughput \leq \frac{RWIN}{RTT} \quad (1)$$

RWIN = Receiving Window, RTT = Round Trip Time (= latency * 2)

The overhead can be calculated as follows:

$$Overhead = \frac{RWIN}{2^{tcp_adv_win_scale}} \quad (2)$$

`tcp_adv_win_scale` is 2 by default. For the default parameters in Linux for the receiving window (`tcp_rmem`) this gives the following result:

$$87380 - \frac{87380}{2^2} = 65536$$

If a transatlantic link would be used [13] with a round trip time of 150ms the following calculation would be made:

$\frac{65536}{0.150} = 436906\text{bytes/s} \approx 400\text{kbytes/s}$ As can be seen, this speed is very slow for these days. This is why the default Linux TCP window parameters need to be changed. There is a clear difference in network speed if a different TCP window size is set.

| Window size | Network performance |
|---------------------|---------------------|
| 32k | 1.14 Gb/s |
| 128k | 3.84 Gb/s |
| 512k | 9.47 Gb/s |
| 1M | 9.91 Gb/s |
| 8M | 9.92 Gb/s |
| 128M | 9.92 Gb/s |
| 195M (Kernel limit) | 9.93 Gb/s |

Table 3: TCP windows size(back-to-back on copper wire, RTT \approx 0)

3.2.3 TCP Buffer size(window size)

As one can see in the table `table:tcpwsiz` the TCP buffer size has a big influence on network performance. The main question here is why is performance so low when using the 32k TCP window size? We calculated the window size required to obtain full (10 gbit) performance on our test setup, with the two servers connected back to back using copper wire. The calculation were made using an average RTT of 5 microseconds(0.000005 seconds) and the result was that a 6.4k theoretical buffer size was enough to obtain full performance. So our 32k buffer size should be more than enough for the purpose.

$$10 \text{ gbits} = 10485760 \text{ kbits}$$

$$32 \text{ kbytes} = 256 \text{ kbits}$$

$$\frac{10485760}{256} = 40960 \text{ (required interrupts/second)}$$

We then calculated the amount of kernel interrupts required to process data at 10 gbit/sec speeds when using 32 kbyte buffer size, and the result was 40960 interrupts. We believe that this is the limiting factor in our test scenario because the resulted number of interrupts is way to big for the operating system to handle.

For TCP in general, there are two different types of buffers which should be configured at the least:

Maximum TCP buffer space This is the maximum amount of buffer

space available for all TCP connections. Most of the time this parameter (`net.ipv4.tcp_rmem`) is configured by default with a value which was adequate when transmitting small amount of data at low speeds. For high performance networks, this value is too small.

Socket buffer size This is the size of the buffer per TCP connection and is in fact the same as the TCP window size.

3.2.4 UDP buffer size

Like the TCP window, the UDP buffer size is too small by default on all OSs. We did some measurements involving configuring different UDP buffer sizes. In table 4, one can clearly see that a buffer of 512 kbytes is sufficient for back to back communication or if $RTT = 0$.

| UDP buffer size | Network performance | Packet loss |
|-----------------|---------------------|-------------|
| 128 kbytes | 4.13 Gb/s | 44% |
| 512 kbytes | 9.93 Gb/s | 0% |
| 2 Mbytes | 9.93 Gb/s | 0% |
| 8 Mbytes | 9.93 Gb/s | 0% |
| 128 Mbytes | 9.75 Gb/s | 3.2% |

Table 4: UDP buffer size

As can be seen at the first row of the table, there is a lot of packet loss. The reason for this is because the buffer size is too small to store all traffic at this high data rate and the kernel starts discarding the UDP packets. In fact this value (which is the `net.core.rmem_max`) is used as a default in the Linux kernel, which was sufficient years ago. However, as can be seen from our measurements, this is not optimal for high performance networking.

3.2.5 Flow control

Flow control should be enabled. When turned off, the receiver could be overrun with data if the sender has more CPU power and sends too much data too fast. As the receiver is saturated from the beginning of the transmission, the network performance will decrease. If flow control is enabled, the receiver will let the transmitter know if its buffer is full and the latter will transmit the data at a lower rate. In our own tests flow control did not make a difference because both cards were running optimally with all settings configured for full performance. We did witness the effects of flow control in situations where bottlenecks occurred. One situation was when we used a lower speed PCIe slot. With flow control off UDP had massive packet loss because the receiving card was not able to send the data fast enough to the higher layers. With flow control on, the UDP was slowed down to the point that the slower card could keep up with the stream.

3.2.6 TCP Selective Acknowledgements Option

TCP Selective Acknowledgements Option or SACK[14] is a mechanism that enables to optimize the retransmission of lost TCP packets if packets did not arrive at their destination. The receiver sends back SACK packets to the sender to inform him what packets did arrive. The sender can then retransmit only the missing data segments. An aggressive sender can retransmit the dropped packets immediately while they already could have been received successfully.

4 Creating a diagnostic tool

As a second goal for our project we investigated if it is possible and feasible to build a tool that can pinpoint the network performance bottleneck on a host. Taking into account the sheer number of factors that can influence the network performance and the complexity of the relations between them it is a challenge to gather all the required data in a central point, and based on that give a result. If indeed it is possible to build such a tool, perhaps one of the most important questions would be how reliable it can be?

Ideally we would use a tracing framework like DTrace (Solaris) or SystemTap (Linux) to read all data directly from the kernel without making use of external tools, thus having a unified and clean solution. Unfortunately we could not make use of SystemTap because it has a steep learning curve (needs deep knowledge of the Linux kernel), so in our short time frame it would not be feasible to build a working tool. Instead we integrated several Linux tools that report network and hardware statistics.

4.1 Integrated tools

In order to reach our goals and to gather data about static and dynamic parameters, we used the following tools:

ethtool From this tool we gathered data about flow control and the PCIe slot number.

iperf We used this to perform capacity measurements about network throughput and packet loss for TCP and UDP.

/proc/sys/net/ The files which can be found here were used to examine the TCP and UDP buffer sizes. They were used for reading TCP SACK settings.

lspci From the output of this command we could obtain the slot speed of the PCIe slot that contained the NIC.

netstat Was used to read the network counters in the NIC.

/proc/net/dev Was used to read the network counters in the kernel.

Based on the output of these tools we parsed the results in our program, where we gathered the essential parameters on the server.

4.2 Program structure

The program was written in the Ruby programming language. The main reason why we chose it, was because it is easy to prototype a tool in a short amount of time. The program contains of four classes.

Info In this class the local information about the different parameters is gathered.

Client This class is used when the tool is started in client mode.

Server This class is used when the tool is started in server mode.

Compare This class is executed only on the server after the data has been received from the client. It compares the data collected on the server with the data collected on the client and displays a side by side report about the tests.

Log Is used to give information to the end user while the programs are used and are communicating.

4.3 Client/server communication

In figure 13 the work flow of the tool is represented.

1. The user first has to start the server which will start gathering data locally and starts listening for incoming connections.
2. When the client is started, it will gather this data as well and make a connection to the server. At this point the client will send a command to the server to start the iperf servers: one for TCP and one for UDP.
3. Once the servers are started a message will be sent to the client to start the iperf clients. This will initiate the network measurements tests.
4. If the client is done testing after a specific amount of time, it will send a "kill" server command to the server.
5. Both systems will count their network statistics and the client will send all its information to the server. The latter will display this in a clear and comprehensive way to the end user.

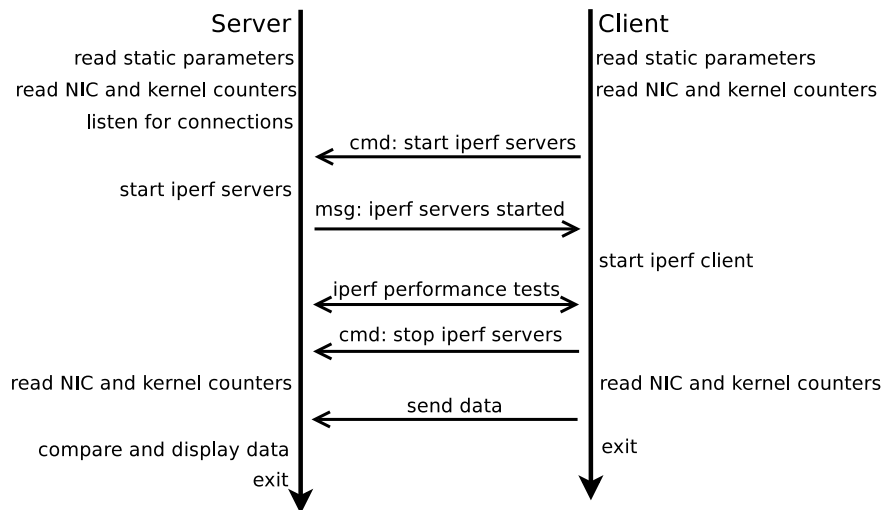


Figure 13: Communication between client and server

Communication is done using an XML/RPC interface over an HTTP connection. The Ruby library that we used is very high-level, which makes

it trivial to establish a connection and send information over this.

4.4 Output of the tool

As a result, the output is shown in a clear overview of both end points. An example can be found in figure 4.4. Output, marked in red, are warnings of possible settings that decreases network performance.

| -----General NIC information----- | *Server(flits)* | *Client(donder)* |
|---|-------------------------------|-------------------------------|
| Ip address: | 10.0.3.1 | 10.0.3.2 |
| Link speed: | 10000Mb/s | 10000Mb/s |
| Link up [Yes/No]: | yes | yes |
| Duplex [Full/Half]: | Full | Full |
| Flow control [On/Off]: | autoneg: on rx: on tx: on | autoneg: on rx: on tx: on |
| Ethernet MTU: | 9000 | 9000 |
| PCIe slot: | 2.5GT/s x8 | 2.5GT/s x4 |
| -----TCP settings----- | | |
| TCP window size [Initial Default Maximum]: | 4096 524287 102400000 | 4096 524287 102400000 |
| TCP buffer size [Initial Default Maximum]: | 572352 763136 1144704 | 191904 255872 383808 |
| TCP sending buffer [Initial Default Maximum]: | 4096 524287 102400000 | 4096 524287 102400000 |
| TCP SACK [On/Off]: | Off | Off |
| -----UDP settings----- | | |
| UDP buffer size [Initial Default Maximum]: | 572352 763136 1144704 | 191904 255872 383808 |
| UDP min. receiving buffer: | 4096 | 4096 |
| UDP min. sending buffer: | 4096 | 4096 |
| -----IPERF results----- | | |
| UDP : | 3.87 Gbits/sec | 3.85 Gbits/sec |
| UDP loss: | 1934/375256 | 1934/375256 |
| TCP: | 3.83 Gbits/sec | 3.83 Gbits/sec |
| -----Packet/byte counters----- | | |
| NIC counter----- | | |
| Packets: | rx:5725576 tx:491126 | tx:5725558 rx:491108 |
| Bytes: | rx:48411519886 tx:26587945 | tx:48411512340 rx:29532462 |
| Errors: | rx:0 tx:0 | tx:0 rx:0 |
| Kernel counter----- | | |
| Packets: | rx:5725576 tx:491126 | tx:5725558 rx:491108 |
| Errors: | rx:0 tx:0 | tx:0 rx:0 |

Figure 14: Output of the tool

The reason why the IP MTU is not included in the output of the tool is because it is one of the parameters the tool takes and it is supposed to be known.

4.5 Challenges

There are several issues that we encountered when we wrote and tested the tool:

- Using our program and the integrated tools without influencing CPU load.
 - This was mitigated by waiting for iperf to perform the tests and then cleanly exit. During the time that iperf performed the tests our application would be in blocking mode. This means

that the server resources will not be influenced while the Iperf tests are running.

- Counting network statistics without taking the overhead in to account which is introduced by the client/server communication.
 - This problem could be solved by sending the data only after the network statistics were read. This does not entirely solve the problem. Our tool and iperf still needs to communicate between client and server, which we calculated as to introduce a small overhead of 18 bytes. We subtracted these bytes from the network counters, which lead to even reporting on both sides.

5 Recommendations for optimal network performance

To obtain optimal network performance, one should verify the buffer sizes first. These can be found in `/etc/sysctl.conf`. As been said in sections 3.2.4 and 3.2.3, default buffer sizes in all major operating systems are just to small.

A second recommendation includes the use of jumbo frames. As one can see from figure 10 there is a clear performance difference in the different Ethernet MTUs which are used. Standard an Ethernet MTU size of 1500 bytes is used. This basically means that six times more headers needs to be generated, which takes some time.

If possible use a large IP MTU size. 8000 bytes is the default on the Internet. With an IP MTU size of 63k bytes, one could almost gain 100% of the speed.

When a multi cored CPU is part of the configuration, dedicate one or two threads to processing network traffic, as CPU load influences the network performance. In a setup with a single core/thread, one should try to keep the CPU load as low as possible.

Choose the right slot for the network interface card. For example, do not insert the external firewire card into the fastest PCIe slot, but use it for the network interface card instead.

If the system, which receives or transmits data at a high rate, has barely enough RAM to have all its processes and services running, dedicate some services to another system or add RAM. If not enough RAM is provided and the system starts swapping, network performance will decrease drastically.

6 Conclusions

In this section we review our research questions, like stated in section 1.1.

Is it possible to determine and classify the parameters which affect network performance?

We could define a large array of complex parameters that influence the network performance.

Fundamentally there are two main aspects where bottlenecks can occur, namely in software and hardware.

Hardware bottlenecks include CPU load, swapping from RAM to disk, PCIe bus speeds and network interface speeds.

Software parameters include both IP and Ethernet MTU, TCP window size, TCP buffer size, UDP buffer size, flow control and SACK.

From our findings we could conclude that parameters, like the IP MTU, influence throughput while other parameters, like the Ethernet MTU, influence the packet loss. Historically, the consent regarding IP MTU is that it should be small enough that fragmentation will not occur at the data link layer. One can see that for high performance(10 GBit) networking this principle is not effective anymore, with maximum speeds obtained with very large IP MTU sizes, i.e. 63k.

An anomaly is that the receiver was less influenced by CPU load and memory swapping, while one would expect that receiving data at a high rate would be more intensive, as of the checksum calculation and inspecting packet order. One possible explanation is that the process on the sender part runs in user space, while on the receiving side interrupts happen in the kernel space, having a higher priority.

Further, we noticed that the default settings for Linux are inappropriate for high performance networking. The buffers are optimized for hosts which are connected to networks with lower rates. It should be noted that RTT should play a very important part when choosing the appropriate buffer size, because using a low buffer size on a link with high RTT would cause dramatic losses of performance. Furthermore, choosing a low TCP buffer size on a high speed network would introduce a different bottleneck, and that is the large amount of kernel interrupts required to process the small data buffers.

Is it possible to develop a tool which monitors the parameters and can pinpoint the cause of the reduced network performance?

A high dynamic environment makes it hard to pinpoint bottlenecks. CPU load fluctuates all the time, just like memory usage. Therefore measurements need to be performed in a time frame which is wide enough to make the data reliable.

Gathering data from the static parameters, like configured Ethernet MTU and TCP buffer sizes, is more easy. Based on this information, it is possible to make recommendations to the end user.

7 Future research

Network performance monitoring and measuring is a very broad subject. Because of the limited time frame we could only perform a small part of this subject. Other possible approaches are:

Link bonding By using link bonding, a double link can be established between two end hosts. Using round robin, one could send the data simultaneously over both ends. Calculating the optimal IP MTU can be something to keep in mind. If an IP MTU of 63k is used for example, a 9k Ethernet frame is sent on one link, another 9k Ethernet frame is sent on the other link, and so on. In the end 3 Ethernet frames will be sent on one link, while four Ethernet frames will be sent on the other link. It is clear that the links will send the data out of balanced, and this might introduce decreased network performance.

Multiple streams Using multiple streams can improve performance, but can also introduce new issues. The packet reordering can for example be one of these issues. Two streams with packets of different sequence numbers will arrive and the reordering could be more complex.

Adding network devices Our test setup did not include any network devices. By adding different kinds of network devices in between, unexpected problems can come up.

Hard drive speed Iperf did not use the hard drive to read data from (although one can specify to read from a file) or to write data to. Some applications, though, require disk access. Disk speed and the bus speed to transfer data from and to the hard drive, can play a crucial role here.

Delay We did not include delay in our research because we would focus on the end points and the delay was to negligible. However, this is an essential topic to investigate.

8 Acknowledgements

We would like to thank SARA, especially our supervisors Freek Dijkstra and Ronald van der Pol, for providing us with the hardware to run our tests and valuable advice. During our project they steered us in the right direction and whenever we had questions their response was fast and prompt.

References

- [1] **Intel 5520 Chipset and Intel 5500 Chipset Datasheet**
<http://www.intel.com/assets/pdf/datasheet/321328.pdf>
- [2] **Intel Pentium Dual-Core Processor E6000 and E5000 Series Datasheet**
<http://download.intel.com/design/processor/datashts/320467.pdf>
- [3] **Iperf**
<http://iperf.sourceforge.net/>
- [4] **Comparison between network test tools** http://staff.science.uva.nl/~jblom/gigaport/tools/test_tools.html
- [5] **Ethtool man page**
<http://manpages.ubuntu.com/manpages/lucid/en/man8/ethtool.8.html>
- [6] **Netstat man page**
<http://manpages.ubuntu.com/manpages/lucid/en/man8/netstat.8.html>
- [7] **Lookbusy**
<http://www.devin.com/lookbusy/>
- [8] **Stress man page**
<http://manpages.ubuntu.com/manpages/lucid/man1/stress.1.html>
- [9] **Sysctl man page**
<http://linux.die.net/man/8/sysctl>
- [10] **Myri-10G NICs and Software**
http://www.myri.com/Myri-10G/documentation/Myri-10G_NICs+Software.pdf
- [11] **D1.1 Server Scalibility**
Authors: Ronald van der Pol, Freek Dijkstra
- [12] **TCP Extensions for High Performance**
<http://www.ietf.org/rfc/rfc1323>
- [13] **TCP performance tuning**
Author: Mattias Wadenstein
<http://www.acc.umu.se/~maswan/linux-netperf.txt>
- [14] **TCP Selective Acknowledgment Options**
<http://www.ietf.org/rfc/rfc2018>
- [15] **ESnet Network Performance Knowledge Base**
<http://fasterdata.es.net/>
- [16] **Flow Control in the Linux Network Stack**
<http://www.cl.cam.ac.uk/~pes20/Netsem/linuxnet.ps>

A Tool source code

A.1 main.rb

```
1  #!/usr/bin/ruby1.9
2  # Usage ./mtool <interface> [-c <ipaddress> ] [-s ]
3  # Example: On the server: ./mtool eth0 -s
4  #           On the client: ./mtool eth0 -c 10.0.0.1
5  #
6  # Mtool is a tool which helps to identify
       bottlenecks on high
7  # performance networks.
8  #
9  # This program is written by Alex Giurgiu (alex.
       giurgiu@os3.nl) and
10 # Jeroen Vanderauwera (jeroen.vanderauwera@os3.nl)
11
12 require 'logging.rb'
13 require 'netcom.rb'
14 require 'test.rb'
15
16
17 class Info
18   attr_reader :ipaddr, :lspeed, :ldetected, :lduplex
   , :flowcontrol, :info
19
20   def initialize(interface)
21     @interface = interface
22     self.gather(interface)
23   end
24
25
26   def gather(interface)
27     Log.inf("Starting data gathering for " +
28           interface)
29
30     @hostname = %x[hostname]
31     ethdata = %x[ethtool #{interface}]
32
33     ethdata.each_line do |d|
34       if d.include?("Speed")
35         @lspeed = d.split.last
36       elsif d.include?("Link detected")
37         @ldetected = d.split.last
38       elsif d.include?("Duplex")
39         @lduplex = d.split.last
40       end
41     end
42
43     ipdata = %x[ifconfig #{interface}]
```

```

43     @ipaddr = ipdata.split[6].split(': ')[1]
44
45     @flowcontrol = []
46     flowdata = %x[ethtool -a #{interface}]
47     flowdata.each_line do |d|
48         if d.include?("Autonegotiate")
49             @flowcontrol[0] = d.split(':').last.lstrip
50         elsif d.include?("RX")
51             @flowcontrol[1] = d.split(':').last.lstrip
52         elsif d.include?("TX")
53             @flowcontrol[2] = d.split(':').last.lstrip
54         end
55     end
56
57     #Ethernet MTU
58     ethmtudata = %x[ifconfig #{interface}].split
59     ethmtudata.each do |l|
60         if l.include?("MTU")
61             @ethmtu = l.split(":")[1]
62         end
63     end
64
65     ##
66     #####
67
68     # Reading values from /proc/sys/net/ipv4
69     #
70     ##
71     #####
72
73     # TCP socket buffer (TCP windows size)
74     lines = IO.readlines("/proc/sys/net/ipv4/
75         tcp_rmem")
76     first = lines.first
77     @tcpwsize = []
78     @tcpwsize[0] = first.split[0] # initial TCP
79         window size
80     @tcpwsize[1] = first.split[1] # default TCP
81         window size
82     @tcpwsize[2] = first.split[2] # max. TCP window
83         size
84
85     # TCP sending socket buffer
86     lines = IO.readlines("/proc/sys/net/ipv4/
87         tcp_wmem")
88     first = lines.first
89     @tcpssize = []
90     @tcpssize[0] = first.split[0] # initial TCP
91         sending buffer size

```

```
82     @tcpsbssize[1] = first.split[1] # default TCP
      sending buffer size
83     @tcpsbssize[2] = first.split[2] # max. TCP
      sending buffer size
84
85     # TCP buffer (For all connections)
86     lines = IO.readlines("/proc/sys/net/ipv4/tcp_mem
      ")
87     first = lines.first
88     @tcpbssize = []
89     @tcpbssize[0] = first.split[0] # initial TCP
      buffer size
90     @tcpbssize[1] = first.split[1] # default TCP
      buffer size
91     @tcpbssize[2] = first.split[2] # max. TCP buffer
      size
92
93     # TCP SACK
94     lines = IO.readlines("/proc/sys/net/ipv4/
      tcp_sack")
95     @tcpsack = lines.first
96     if @tcpsack == 1
97         @tcpsack = "On"
98     else
99         @tcpsack = "Off"
100    end
101
102    # UDP buffer (For all connections)
103    lines = IO.readlines("/proc/sys/net/ipv4/udp_mem
      ")
104    first = lines.first
105    @udpssize = []
106    @udpssize[0] = first.split[0] # initial UDP
      buffer size
107    @udpssize[1] = first.split[1] # default UDP
      buffer size
108    @udpssize[2] = first.split[2] # max. UDP buffer
      size
109
110    # UDP min receiving buffer
111    lines = IO.readlines("/proc/sys/net/ipv4/
      udp_rmem_min")
112    @udprb = lines.first
113
114    # UDP min sending buffer
115    lines = IO.readlines("/proc/sys/net/ipv4/
      udp_wmem_min")
116    @udpsb = lines.first
117
```

```

118     ##
119     #####
120     #   Reading PCI(e) slot speed   #
121     ##
122     #####
123     slot = %x[ethtool -i #{interface}]
124     slotid = ""
125     slot.each_line do |d|
126       if d.include?("bus-info")
127         temp = d.split.last.split(":")
128         slotid = temp[1] + ":" + temp[2] unless temp
129           [1].nil? || temp[2].nil?
130       end
131     end
132     lspcidata = %x[lspci -vvvv]
133     block = []
134     @slot = {}
135     lspcidata.each_line do |l|
136       l.chomp!
137       block << l
138       if l.empty?
139         if block[0].include?(slotid)
140           block.each do |b|
141             if b.include?("LnkSta")
142               tmp = b.split(":")[1].split(",")
143               @slot["speed"] = tmp[0].split[1]
144               @slot["width"] = tmp[1].split[1]
145             end
146           end
147         end
148       end
149       block = []
150     end
151     ##
152     #####
153
154     beforestats
155
156     @info = {}
157     @info["hostname"] = @hostname
158     @info["ipaddress"] = @ipaddr
159     @info["linkspeed"] = @lspeed
160     @info["linkup"] = @ldetected
161     @info["duplex"] = @lduplex
162     @info["flowcontrol"] = @flowcontrol
163     @info['ethmtu'] = @ethmtu
164     @info['slot'] = @slot

```

```
161     @info["tcpwsize"] = @tcpwsize
162     @info["tcpbsize"] = @tcpbsize
163     @info["tcpsbsize"] = @tcpsbsize
164     @info["tcpsack"] = @tcpsack
165     @info["udpbsize"] = @udpbsize
166     @info["udprb"] = @udprb
167     @info["udpsb"] = @udpsb
168
169     @info = sanitizehash(@info)
170 end
171
172 def sanitizehash(info)
173   info.each_pair do |k,v|
174     if v.nil?
175       info[k] = v = "n/a"
176     elsif v.kind_of? Hash
177       info[k] = sanitizehash(v)
178     elsif v.kind_of? Array
179       info[k] = sanitizearray(v)
180     elsif v.kind_of? String
181       info[k]=v.chomp.lstrip.lstrip
182     end
183   end
184   return info
185 end
186
187 # Count bytes and packets in the NIC
188 def getNicCounters (interface)
189   lines = IO.readlines("/proc/net/dev")
190   niccounter = {}
191   lines.each do |l|
192     if l =~ #{interface}/
193       niccounter["rxp"] = l.split(':').last.split
194         [0] # Received packets
195       niccounter["rxb"] = l.split(':').last.split
196         [1] # Received bytes
197       niccounter["rxe"] = l.split(':').last.split
198         [2] # Received errors
199       niccounter["txp"] = l.split(':').last.split
200         [8] # Transmitted packets
201       niccounter["txb"] = l.split(':').last.split
202         [9] # Transmitted bytes
203       niccounter["txe"] = l.split(':').last.split
204         [10] # Transmitted errors
205     end
206   end
207   return niccounter
208 end
209
210 # Count bytes and packets in the kernel
```

```

205     def getKernelCounters (interface)
206         temp = %x[netstat -i]
207         kernelcounter = {}
208         temp.each_line do |l|
209             if l =~ /#{interface}/
210                 kernelcounter["rxp"]
211                     = l.split[3] #
212                         Received
213                             packets
214                 kernelcounter["rxs"]
215                     = l.split[4] #
216                         Received errors
217                 kernelcounter["txp"]
218                     = l.split[7] #
219                         Transmitted
220                             packets
221                 kernelcounter["txs"]
222                     = l.split[8] #
223                         Transmitted
224                             errors
225             end
226         end
227     end
228     return kernelcounter
229 end
230
231 # Subtract to reset counters
232 def subtractCounters (after, before)
233     result = {}
234
235     after.each_key do |k|
236         result[k] = after[k].to_i - before[k].to_i
237     end
238     result.each_key do |k|
239         result[k] = result[k].to_s
240     end
241     return result
242 end
243
244 def sanitizearray(info)
245     info.each do |a|
246         if a.nil?
247             info[info.index(a)] = a = "n/a"
248         elsif a.kind_of? Hash
249             info[info.index(a)] = sanitizehash(a)
250         elsif a.kind_of? Array
251             info[info.index(a)] = sanitizearray(a)
252         elsif a.kind_of? String
253             info[info.index(a)] = a.chomp.lstrip.lstrip
254         end
255     end
256 end

```

```
244     return info
245   end
246
247   def beforestats
248     @beforenic = getNicCounters(@interface)
249     @beforekernel = getKernelCounters(
250       @interface)
251   end
252
253   def afterstats
254     afternic = getNicCounters(@interface)
255     afterkernel = getKernelCounters(@interface)
256
257     @resultnic = subtractCounters(afternic,
258       @beforenic)
259     @resultkernel = subtractCounters(afterkernel
260       , @beforekernel)
261     @info["niccounter"] = @resultnic
262     @info["kernelcounter"] = @resultkernel
263   end
264
265   def showinfo
266     puts @ipaddr
267     puts @lspeed
268     puts @ldetected
269     puts @lduplex
270     puts @ethmtu
271     puts @flowcontrol.to_s
272     puts @tcpwsize.to_s
273     puts @tcpbsize.to_s
274     puts @tcpsbsize.to_s
275     puts @tcpsack
276     puts @udpysize.to_s
277     puts @udprb
278     puts @udpsb
279     puts @slot.to_s
280   end
281
282   end
283
284   if ARGV.empty?
285     Log.error("Usage: ./mtool <interface> [-c <
286       ipaddress> ] [-s ]")
287   else
288     interface = ARGV[0]
289
290     if ARGV[1] == '-c'
291       clientth = Thread.new { @client = Client.new(
292         ARGV[2]) }
```

```

289   intdata = Info.new (interface)
290   @client.nettest(ARGV[2], 50, intdata.info["
      ipaddress"])
291   intdata.afterstats
292   @client.send(intdata.info)
293   @client.killrpc
294
295   clientth.join
296   elsif ARGV[1] == '-s'
297     serverdata = Info.new (interface)
298     serverth = Thread.new { @server = Server.new(
      serverdata.ipaddr) }
299     serverth.join
300     serverdata.afterstats
301     puts @server.clientdata.size.to_s
302     puts @server.clientdata[0].to_s
303     puts "====="
304     puts @server.clientdata[1].to_s
305     puts "====="
306     puts @server.clientdata[3].to_s
307     clientinfo = @server.clientdata[2].merge(
      @server.clientdata[0])
308     serverinfo = serverdata.info
309     puts "====="
310     puts clientinfo
311     puts serverinfo
312     puts "====="
313     Compare.info(serverinfo, clientinfo)
314   end
315
316   #rescue => msg
317     #Log.error("Something went wrong :( Terminating
      ... [" + msg.to_s + "]")
318   #end
319
320 end

```

A.2 *compare.rb*

```

1  class Compare
2
3  def self.info(serverdata, clientdata)
4    @serverdata = serverdata
5    @clientdata = clientdata
6    self.callChecks
7    self.display
8  end
9
10 def self.display
11

```

```

12     puts makeline("", "*Server(" + @serverdata['
        hostname'] + ")*", "*Client(" + @clientdata
        ['hostname'] + ")*")
13     puts "-----General NIC
        information
        -----"
14     puts makeline("Ip address:", @serverdata['
        ipaddress'], @clientdata['ipaddress'])
15     puts makeline("Link speed:", @serverdata['
        linkspeed'], @clientdata['linkspeed'])
16     puts makeline("Link up [Yes/No]:", @serverdata['
        linkup'], @clientdata['linkup'])
17     puts makeline("Duplex [Full/Half]:", @serverdata
        ['duplex'], @clientdata['duplex'])
18     puts makeline("Flow control [On/Off]:", "autoneg
        : " + @serverdata['flowcontrol'][0] + " rx:"
        + @serverdata['flowcontrol'][1] + " tx:" +
        @serverdata['flowcontrol'][2], "autoneg:" +
        @clientdata['flowcontrol'][0] + " rx:" +
        @clientdata['flowcontrol'][1] + " tx:" +
        @clientdata['flowcontrol'][2] )
19
20     puts makeline("Ethernet MTU:", @serverdata['
        ethmtu'].chomp.lstrip.lstrip, @clientdata['
        ethmtu'].chomp.lstrip.lstrip )
21     puts makeline("PCIe slot:", @serverdata['slot']
        ['speed'] + " " + @serverdata['slot']['width'
        ], @clientdata['slot']['speed'] + " " +
        @clientdata['slot']['width'] )
22
23     puts "-----TCP settings
        -----"
24     puts makeline("TCP window size [Initial Default
        Maximu]:", @serverdata['tcpwsize'][0] + " "
        + @serverdata['tcpwsize'][1] + " " +
        @serverdata['tcpwsize'][2], @clientdata['
        tcpwsize'][0] + " " + @clientdata['tcpwsize'
        ][1] + " " + @clientdata['tcpwsize'][2] )
25     puts makeline("TCP buffer size [Initial Default
        Maximu]:", @serverdata['tcpbsize'][0] + " "
        + @serverdata['tcpbsize'][1] + " " +
        @serverdata['tcpbsize'][2], @clientdata['
        tcpbsize'][0] + " " + @clientdata['tcpbsize'
        ][1] + " " + @clientdata['tcpbsize'][2] )
26     puts makeline("TCP sending buffer [Initial
        Default Maximu]:", @serverdata['tcpsbsize'
        ][0] + " " + @serverdata['tcpsbsize'][1] + "
        " + @serverdata['tcpsbsize'][2],

```

```

        @clientdata['tcpsbssize'][0] + " " +
        @clientdata['tcpsbssize'][1] + " " +
        @clientdata['tcpsbssize'][2] )
27 puts makeline("TCP SACK [On/Off]:", @serverdata[
    'tcpsack'], @clientdata['tcpsack'])
28
29
30 puts "-----UDP settings
    "
31 puts makeline("UDP buffer size [Initial Default
    Maximum]:", @serverdata['udpbsize'][0] + " "
    + @serverdata['udpbsize'][1] + " " +
    @serverdata['udpbsize'][2], @clientdata['
    udpbsize'][0] + " " + @clientdata['udpbsize'
    ][1] + " " + @clientdata['udpbsize'][2] )
32 puts makeline("UDP min. receiving buffer:",
    @serverdata['udprb'], @clientdata['udprb'] )
33 puts makeline("UDP min. sending buffer:",
    @serverdata['udpsb'], @clientdata['udpsb'] )
34
35 puts "-----IPERF results
    "
36 puts makeline("UDP :", @clientdata['udpclient'],
    @clientdata['udpserver'])
37 puts makeline("UDP loss:", @clientdata['udploss'
    ], @clientdata['udploss'])
38 puts makeline("TCP:", @clientdata['tcp'],
    @clientdata['tcp'])
39
40 puts "-----Packet/byte counters
    "
41 puts "NIC counter
    "
42 puts makeline( "Packets: " , "rx:" + @serverdata
    ['niccounter']['rxb'], "tx:" + @clientdata['
    niccounter']['txb'])
43 puts makeline( " " , "tx:" + @serverdata['
    niccounter']['txb'], "rx:" + @clientdata['
    niccounter']['rxb'])
44 puts "
    "
45 puts makeline( "Bytes: " , "rx:" + @serverdata['
    niccounter']['rxp'], "tx:" + @clientdata['
    niccounter']['txp'])

```

```

46     puts makeline( "", "tx:" + @serverdata['
          niccounter']['txp'], "rx:" + @clientdata['
          niccounter']['rxp'])
47     puts "
          -----
          "
48     puts makeline( "Errors: ", "rx:" + @serverdata['
          niccounter']['rx'], "tx:" + @clientdata['
          niccounter']['txe'])
49     puts makeline( "", "tx:" + @serverdata['
          niccounter']['txe'], "rx:" + @clientdata['
          niccounter']['rx'])
50     puts "Kernel counter
          -----
          "
51     puts makeline( "Packets: ", "rx:" + @serverdata['
          kernelcounter']['rxp'], "tx:" + @clientdata[
          'kernelcounter']['txp'] )
52     puts makeline( "", "tx:" + @serverdata['
          kernelcounter']['txp'], "rx:" + @clientdata[
          'kernelcounter']['rxp'] )
53     puts "
          -----
          "
54     puts makeline( "Errors: ", "rx:" + @serverdata['
          kernelcounter']['rx'], "tx:" + @clientdata[
          'kernelcounter']['txe'] )
55     puts makeline( "", "tx:" + @serverdata['
          kernelcounter']['txe'], "rx:" + @clientdata[
          'kernelcounter']['rx'])
56
57
58     end
59
60     def self.wspacel(nr, str)
61         wsp = ""
62         nr.times do
63             wsp.concat(" ")
64         end
65         return wsp.concat(str)
66     end
67
68     def self.makeline(col1, col2, col3)
69         col2 = wspacel(50-col1.length, col2)
70         if col2.include?("\033[0m")
71             col3 = wspacel(80-col2.length-col1.length + 9,
              col3)
72         else
73             col3 = wspacel(80-col2.length-col1.length,
              col3)

```

```
74     end
75
76
77     return col1 + col2 + col3
78
79 end
80
81 def self.callChecks
82     serverlinkspeed = @serverdata['linkspeed'].sub(
83         "Mb/s", "" ).to_i
84     clientlinkspeed = @clientdata['linkspeed'].sub(
85         "Mb/s", "" ).to_i
86     self.checks(@serverdata, serverlinkspeed,
87         clientlinkspeed)
88     self.checks(@clientdata, clientlinkspeed,
89         serverlinkspeed)
90 end
91
92 def self.checks (data, thislinkspeed,
93     otherlinkspeed)
94     # General NIC information checks
95     data['linkspeed'] = self.red(data['linkspeed'])
96     unless thislinkspeed >= otherlinkspeed
97         data['duplex'] = self.red(data['duplex']) unless
98             data['duplex'] != "Half"
99     end
100     for i in (0..2)
101         data['flowcontrol'][i] = self.red(data['
102             flowcontrol'][i]) unless data['flowcontrol
103             '][i] != "off"
104     end
105     data['ethmtu'] = self.red(data['ethmtu']) unless
106         data['ethmtu'] >= "9000"
107     slotspeed = data['slot']['speed'].sub( "GT/s", "
108         ").to_i * data['slot']['width'].sub( "x", ""
109         ).to_i * 1000
110     data['slot']['speed'] = self.red(data['slot']['
111         speed']) unless slotspeed >= thislinkspeed
112     data['slot']['width'] = self.red(data['slot']['
113         width']) unless slotspeed >= thislinkspeed
114
115     # TCP settings checks
116     tcpsettings = Array[ "4096", "16384", "1048576"
117         ]
118     for i in (0..2)
119         data['tcpwsize'][i] = self.red(data['tcpwsize'
120             '][i]) unless data['tcpwsize'][i] >=
121             tcpsettings[i]
122     end
123     for i in (0..2)
```

```

106     data['tcpbsize'][i] = self.red(data['tcpbsize'
107     ] [i]) unless data['tcpbsize'][i] >=
108     tcpsettings[i]
109     end
110   for i in (0..2)
111     data['tcpbsize'][i] = self.red(data['tcpbsize'
112     ] [i]) unless data['tcpbsize'][i] >=
113     tcpsettings[i]*4
114   end
115   data['tcpsack'] = self.red(data['tcpsack'])
116   unless data['tcpsack'] != "Off"
117
118   # UDP settings checks
119   for i in (0..2)
120     data['udpbsize'][i] = self.red(data['udpbsize'
121     ] [i]) unless data['udpbsize'][i] >=
122     tcpsettings[i]*4
123   end
124   data['udprb'][i] = self.red(data['udprb'])
125   unless data['udprb'] >= "4096"
126   data['udpsb'][i] = self.red(data['udpsb'])
127   unless data['udpsb'] >= "4096"
128
129   end
130
131   # Color the command line output to normal
132   def self.colorize(text, color_code)
133     "#{color_code}#{text}\033[0m"
134   end
135
136   # Color the command line output to red
137   def self.red(text); self.colorize(text, "\033[31m"
138   ); end
139
140 end

```

A.3 netcom.rb

```

1 require 'xmlrpc/client'
2 require "xmlrpc/server"
3 require 'webrick'
4 require 'compare.rb'
5
6
7
8 class Client
9
10   def initialize(ip, port=12333)
11     @server = XMLRPC::Client.new(ip, "/netmes", port
12     )

```

```

12     @ip = ip
13     @port = port
14 end
15
16 def send(data)
17     Log.inf ("Sending gathered data.")
18     @server.call("data", data)
19     Log.inf @server.call("data", data), @ip
20 end
21
22 def nettest(ip,time=20, iplocal)
23     Log.inf ("Sending start iperf server call")
24     Log.inf @server.call("startiperfs"), ip #start
        the TCP and UPD iperf servers on the server
25     Log.inf ("Start iperf server call sent")
26     sleep(2)
27     Log.inf ("Starting iperf locally")
28     c2s = Test.new("client",ip, '63k', '512k', '
        1000M', 50) #start the TCP and UPD iperf
        clients on the client
29     renew
30     Log.inf @server.call("stopiperfs"), ip #stop
        the TCP and UPD iperf servers on the server
31     send(c2s.clientdata)
32     killrpc
33 end
34
35 def killrpc
36     renew
37     Log.inf @server.call("stoprpcs"), @ip
38 end
39
40 def renew
41     @server = XMLRPC::Client.new(@ip, "/netmes",
        @port)
42 end
43
44 end
45
46
47 class Server
48     attr_accessor :clientdata, :serverdata
49
50     def initialize(ip, port=12333)
51
52         @clientdata = []
53         servlet = XMLRPC::WEBrickServlet.new
54         servlet.add_handler("data") do |data|
55             puts data.to_s
56             @clientdata << data

```

```

57     Log.inf "Data received from client. "#XML-RPC
        stoping."
58     "Data received"
59     #@server.stop
60 end
61
62 servlet.add_handler("startiperfs") do
63     @c2s = Test.new('server' ,ip)
64     "Iperf servers started"
65 end
66
67     servlet.add_handler("startiperfc") do |ip|
68         @s2c = Test.new('client' ,ip)
69         @serverdata = @s2c.clientdata
70         "Iperf client started"
71     end
72
73 servlet.add_handler("stopiperfs") do
74     Thread.kill(@c2s.ths1)
75     Thread.kill(@c2s.ths2)
76     %x[killall -9 iperf]
77     "Iperf servers stoped"
78 end
79
80 servlet.add_handler("stoprpcs") do
81     @server.stop
82     Log.inf "RPC server stoped"
83     "RPC server stoped"
84 end
85
86 Log.inf("Starting XML-RPC server")
87
88 @server=WEBrick::HTTPServer.new(:Port => port)
89 trap("INT"){ server.shutdown }
90 @server.mount("/netmes", servlet)
91 @server.start
92
93 #rescue => msg
94     #Log.error("Failed to start the RPC server.
        Terminating... [" + msg.to_s + "]" )
95 #end
96
97 #Log.inf("XML-RPC server started on " + ip + ":"
        + port.to_s)
98 end
99
100 end

```

A.4 logging.rb

```
1 class Log
2   def self.inf(data, from="local")
3     puts "[INFO](\"+from+\")==> " + data
4     return data
5   end
6
7   def self.error(data, from="local")
8     puts "[ERROR](\"+from+\")==> " + data
9     return data
10  end
11 end
```