16.04.23

Simpleperf and performance evaluation

Portfolio 1 - DATA2410
Networking and cloud computing

Jørgen Skontorp
s364516
Oslo Metropolitan University

1 Introduction

This assignment is all about the basics of network performance. I will explore how data is sent in a network and how bandwith, delay and packet loss impacts network performance. My goal is to show the basic mechanics of how networks perform through a series of performance evaluation tests.

In this assignment I have designed and implementet simpleperf, a simplified network measuring tool based on the iPerf tool. iPerf is a popular tool for measuring the maximum achievable bandwidth on IP networks. I programmed simpleperf in python using its socket module. I will use this tool, as well as others to test network performance.

My approach to this task was to spend a lot of time to design, program and implement my simpleperf tool. I based my design on a previous socket programming assignment, then implementing each functionality one by one. I was making sure everything worked according to specifications by countinously carrying out checks and optimize my code every step of the way. Before starting the performance tests I conducted several trials to check that the software and virtual network worked as it should. This would later prove to be worthwile and made the performance tests themselves easy. After gathering data I could link it to relevant theory.

A limitation of this experiment is, that like my program, these are simplified performance evaluations. These tests are only intended to show examples and tendencies of network concepts. The intent is not accuracy. However, these test provided a nice framework to discuss concepts and theories about network performance.

This paper first gives an in-depth insight of my simpleperf program and an overview of the network used in this project. Then an overview of the tools and metrics used in my tests, followed by five network performance tests. I discuss my findings for each test and link it to relevant network theories. Lastly I conclude my work.

2 Simpleperf

Simpleperf is a simplified version of the iperf tool for measuring network. It is programmed in python, using python's socket module which provides an interface to the Berkeley sockets API. Simpleperf uses TCP socket connections to measure network throughput between a server and client(s). The program has different flag arguments that gives you extra settings to tweak how you want the program to be ran. See the programs README.md file or type '*python3 simpleperf.py -h*' for a list of all the different arguments.

The program has different options for server mode and for client mode. In server mode there are an option for binding an IP address for the server, set port and choose byte-format for the result summary print. In client mode there are options for which IP address and port to connect, select byte-format for the result summary print, interval for how often the results should print and how many parallel connections the client should have. The arguably most important option for client mode is the time and number argument. The client keeps sending packets of 1000 bytes to the server for 25 seconds. However, with the time function one can edit amount of seconds the client will send packets. With the number argument one can decide a set amount of bytes/KB/MB to send, and the timer will be ignored. The client will then keep sending until the set amount is sent.

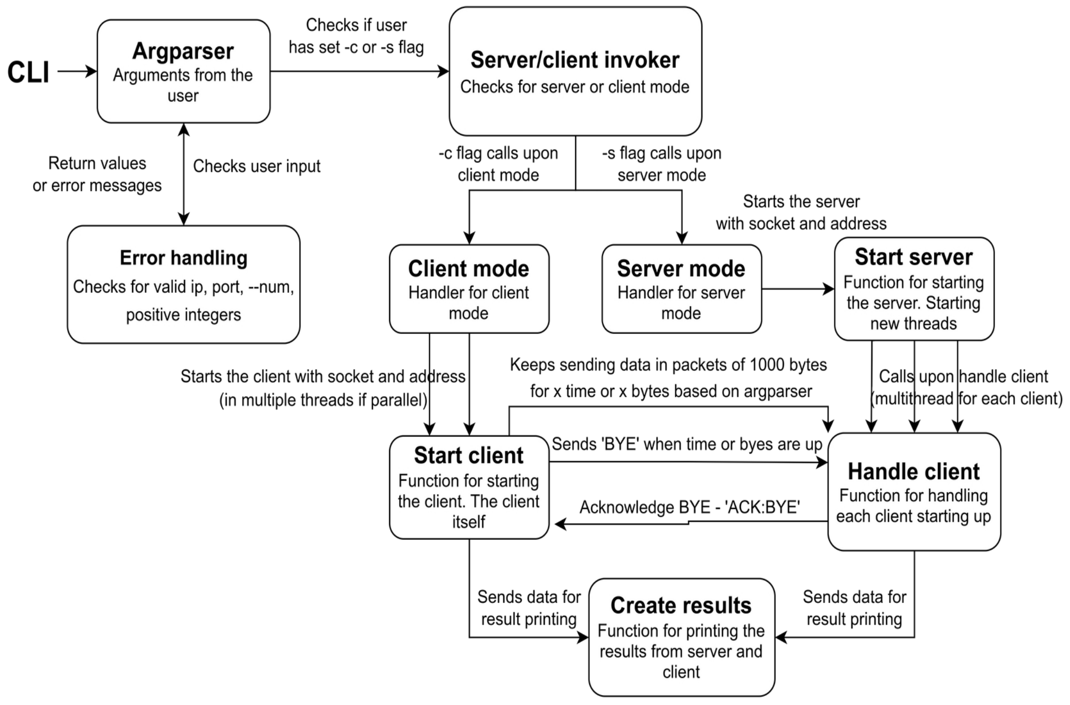
CLI
Argparser
Arguments from the user
Checks if user has set -c or -s flag
Server/client invoker
Checks for server or client mode
Return values or error messages
Checks user input
Error handling
Checks for valid ip, port, --num, positive integers
-c flag calls upon client mode
-s flag calls upon server mode
Client mode
Handler for client mode
Server mode
Handler for server mode
Starts the server with socket and address
Start server
Function for starting the server. Starting new threads
Starts the client with socket and address (in multiple threads if parallel)
Keeps sending data in packets of 1000 bytes for x time or x bytes based on argparser
Calls upon handle client (multithread for each client)
Start client
Function for starting the client. The client itself
Sends 'BYE' when time or byes are up
Acknowledge BYE - 'ACK:BYE'
Handle client
Function for handling each client starting up
Sends data for result printing
Create results
Function for printing the results from server and client
Sends data for result printing

Figure 1: *Chart over simpleperf program and functions*

Figure 1 shows a chart over my simpleperf program and all its functions and their relations. The user starts in the command line interface and writes arguments. The argparser library gives the user clear instructions and handles the arguments received. The user input is then sent to different error handling functions. If there is an error a message will be printed in console, or else the functions will return the values. The argparser then invokes the server or client based on the flag set by the user.

If there is a -c flag, the client_mode function is started. This function starts up threads based on how many parallel clients the user has set with the -P flag. The default is one thread. Parallel connections are just threads with more clients doing the same argument provided at the same time. The thread then calls upon the start_client function with a socket connection and address based on the provided IP address and port (or default values), and it tries to connect to the server.

If there is a -s flag, the server_mode function is started. It sets the IP address and port based on the user input (or default values) and then creates a new socket. The function then calls on the start_server function with socket and address as arguments. The start_server function listens for connections and tries to accept them. If there is a timeout, keyboard interrupt or the server otherwise can't accept the connection, there is printed an error message. If the client connects there is started a new thread for the client, and the handle_client function is called upon with the connection, client address and server address as arguments.

The communication between server and client are happening in the start_client and handle_client functions. The start_client function tries to connect with the server and generates an error message if it couldn't connect.

When connecting, the start_client function checks wether there is set a --num flag for a specified number of bytes. If there are, the function starts a while-loop that lasts for as long as there are bytes. For each iteration it checks if there are less than 1000 bytes left. When there are less than 1000 bytes, it sends the rest of the bytes and sets remaining bytes 0 to stop the loop. While there are more than 1000 bytes left, the client will keep sending packets of 1000 bytes to the server for as long as there are more bytes to be sent.

If there is not specified a --num flag, the client goes into time mode. The default is 25 seconds, but the user can set a defined time with the --time flag. The client then starts a while-loop that runs as long as the current time is less than the end time. The client then keeps sending packets with 1000 bytes as fast as it can until the time is up. Figure 2 illustrates how the client connects and sends data to the server until it reaches end time.

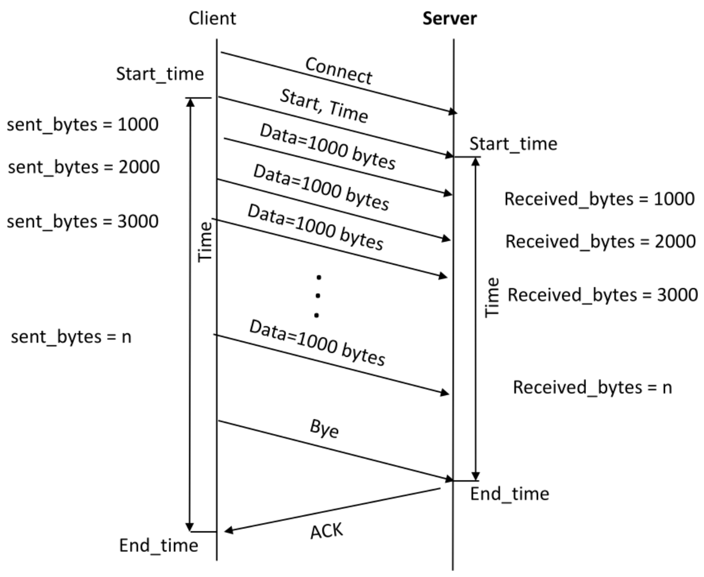

Figure 2: *Illustration of server-client communication (with default or time flag)*

When the start_client function is done sending data, whether the amount of bytes is sent or the time is up, it sends a 'BYE' message to the server, sends the results and awaits a response from the server as seen in figure 2.

The handle_client function on the server side is an infinite loop that tries to recieve packets of 1000 bytes over the socket connection. If it can't recieve data, an error message is printed. The function keeps track of time passed and amount of data recieved. The while-loop breaks when there are no data or the server recieves 'BYE' in the incomming data. When the server has recieved the BYE-message it sends back 'ACK:BYE' back to the client to acknowledge that the BYE-message is recieved. When the start_client function recieves the 'ACK:BYE' it shuts down.

Both handle_client and start_client sends their results to the create_results function. This function takes address, start time, interval time, elapsed time, data, and interval as arguments. The function then uses the data to calculate rate, sets the data in right format and then uses the PrettyTable library to create and print a result table based on the data sent from server and client. The function checks which mode sent the results and prints a table header accordingly. The server sends the socket address, start time, end time, elapsed time and how many bytes that are recieved. The client sends its address, the start time, how much time has elapsed (especially relevant when the --num mode is selected) and total amount of bytes sent. The client checks whether there is an interval flag set for printing results. If there is an interval, the function checks in the while loop whether the interval matches the time elapsed and then sends the current data to the create_results function, as well as interval time and interval mode. The create_result function then prints an interval table that sets the interval time based on the time elapsed.

3 Experimental setup

Figure 3 shows the network topology that I used for my experiments. The illustration shows that there are nine nodes, named h1 to h9. There are five subnetworks. Two subnets are made up of three nodes each (h1-h3 and h4-h6) connected by a switch to the router, while three subnets consists of single nodes (h7,h8,h9) connected to a router. The subnetworks are connected by four routers named R1 to R4. There are a total of three links between the routers named L1 to L3. The L1 link has a bandwith of 40Mb/s and 10ms delay, L2 has a bandwith of 30Mb/s and 20ms delay, L3 a bandwith of 20Mb/s and 10ms delay.

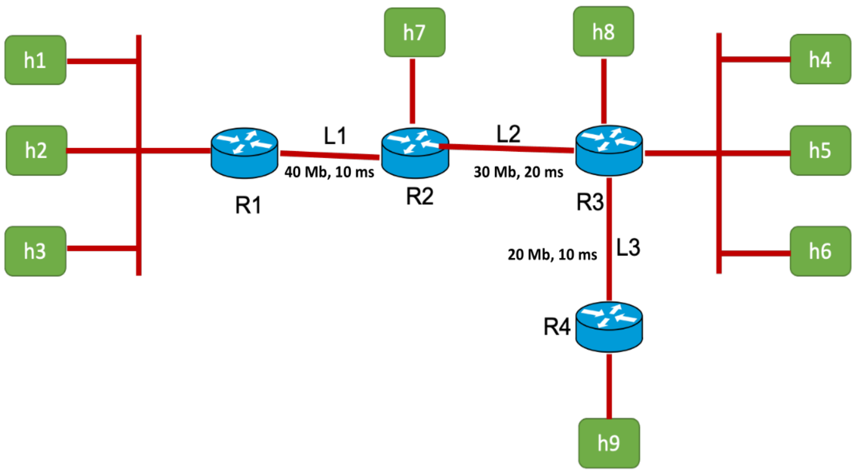

Figure 3: *The network topology handed out for this assignment*

Even though it is not specified in the illustration of my network topology, the code portfolio-topology.py shows that the different links also has a defined maximum queue size of the transmission queue between the two router pairs. This parameter controls the maximum number of packets that can be buffered in the transmission queue before they are transmitted. L1 has a max queue of 67 packets, L2 has 100 packets and L3 has 33 packets.

In our experiment we know the network topology beforehand. We were both handed out illustrations and the code with the details of the network. If I didn't know the network topology beforehand I would have needed to use tools to find it. I could for example have used tools like traceroute or different network discovery tools.

4 Performance evaluations

4.1 Network tools

In my experiment I used a virtual machine with different applications and tools.

Virtual machine

For this assignment I used Ubuntu 22.04.2 LTS Desktop in a virtual machine. I started using Oracle VM VirtualBox with 3 processor cores and 8 GB RAM. With this setup I ran into performance issues. The simpleperf program I made performed awful with very low throughput. The other tools also had performance issues. After trying the first test case, and doing a lot of troubleshooting, I found out that I had to change VM software. According to Mininet documentation, VMware supposedly runs Mininet faster than VirtualBox. I started the setup from scratch running VMware Workstation 17 Player with the same setup as mentioned earlier. Running tests with this VM fixed all the performance issues I had with my tools.

Applications

For the programming part of this assignment, I used VSCodium and python to develop my tool. I used git and a private repo to easily share my project and results between host and

VM guest, as well as having backups of my project. I used pip package installer to install PrettyTable to format my result prints.

In Ubuntu, I used Open VSwitch and Mininet to run the network topology program we were handed, so I could run the virtual network. Mininet uses xterm terminal emulator to communicate with the network.

Tools
The ping tool is used to test from a remote location, whether a particular host is up and reachable. It is often used to measure latency between the client host and the target host. I used the ping tool to ping from one node in the network to another with a count of 25 replies to measure the latency between nodes.

iPerf is a tool for measuring the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, buffers and protocols. For each test it reports the bandwidth, loss, and other parameters (*IPerf - The TCP, UDP and SCTP Network Bandwidth Measurement Tool*, n.d.). I used the tool in UDP mode where the client creates UDP streams of a specified bandwith. iPerf then prints a report with bandwith, packet loss and jitter. I used the tool to measure bandwith between client-server pairs.

Simpleperf is a program I programmed myself in Python, as described in section two in this report. Since simpleperf uses sockets with the TCP protocol it provides guaranteed delivery of packets, flow control and congestion-control. I used this program to measure basic throughput in Mpbs between different host-pairs for 25 seconds.

4.2 Performance metrics

In my test cases I will be using round-trip time (delay) and throughput (bandwith) as performance metrics.

The round-trip time, abbreviated to RTT, is the time it takes to send data from the sender to the recipent and back again. The RTT is being affected by all the different types of delays that occurs on the route data travels. These types of delays will be discussed later in section 4.5.2.

Ping works by sending ICMP (Internet Control Message Protocol) echo request packets, also called "ping", to the target host. It listens for ICMP echo response replies, also called "pong". The tool measures the delay between when the client sent the ping message and recieved the pong message. This is the RTT. Ping measures the RRT, records packet loss, and calculates a summary of multiple ping-pong exchanges (Kurose & Ross, 2022, p. 474). Packet loss can also impact the RTT in reality, since the packets need to be retransmitted. However, according to the ping manual, the default mode is to wait one second between each packet, and not using lost packets in the calculation of average RTT. Packet loss are however measured and printed in the results. The average RTT seen in my results are the result provided by the ping tool.

Throughput is the rate (bits/time unit) at which bits are being sent from sender to receiver. The throughput can be measured in average, the rate over longer period of time, or instantaneous to give the rate at a given point in time. The throughput is largely connected to the bandwith and the transmission rate. The bandwith is the theoretical maximum amount of data that can be transmitted, which I know from the network topology. The transmission rate is the actual speed of data transmission. Throughput is then the actual amount of data successfully transmitted over the link. I will discuss throughput more later in the text.

The throughput results from simpleperf are the averaged throughput measured for 25 seconds. By setting an interval flag one can also get the instantaneous throughput, but that is not used in these experiments. The throughput in simpleperf is calculated by sending packets of 1000 bits at a time over a TCP socket connection while measuring the time until a 'BYE' message is sent when the transmission is done.

4.3 Test case 1: measuring bandwidth with iperf in UDP mode

I used iPerf in this test case to measure the bandwith between three client-server pairs in the handed out network topology. By running the topology in Mininet and then opening terminals for the different nodes, I was able to use iperf in UDP mode to measure throughput with bandwith, jitter and packet loss.

Questions:
1. Which rate (X) would you choose to measure the bandwidth here? Explain your answers.
I looked at the chart of the network topology that were handed out. I looked at the route from hX to hY to see what the lowest bandwidth would be, and selected accordingly. This bandwith would be the bottleneck in the route, and increasing the rate beyond this would just create significant packet loss. For example: From h1 to h4 we are passing through R1-R2(40Mb) and R2-R3 (30Mb), and therefore I picked 30Mb to maximize bandwith while keeping packet loss at a minimum.

2. If you are asked to use iPerf in UDP mode to measure the bandwidth where you do not know anything about the network topology, what approach would you take to estimate the bandwidth? Do you think that this approach would be practical? Explain your answers.
Somewhat inspired by how congestion control in TCP works, I would pick a relatively low bandwith then work my way up from there to find a threshold for where there would be a significant packet loss. I would start at 1 MB, then double that and grow exponentially. When there is a significant loss I would work my way backwards by reducing with 1 MB, then decrease exponentially until the packet loss is less significant. When finding this threshold I would start incrementing to fine adjust. I don't think the approach is very practical because it would take a lot of trial and error. This would be very time-consuming on larger networks. However, it is more methodical than just taking wild guesses, and quicker than an incremental approach. In the "real world" one would use network discovery tools like nmap or Wireshark to analyze network traffic and make up a network topology.

4.3.1 Results

Nodes	Test rate	Transfer	Bandwith	Lost/Total	Packet loss
h1-h4	30MB	34.8 Mbytes	29.1Mbits/s	1923/26752	7.2%
h1-h9	20MB	23.2 Mbytes	19.4Mbits/s	1295/17836	7.3%
h7-h9	20MB	23.2 Mbytes	19.4Mbits/s	1305/17836	7.3%

4.3.2 Discussion

UDP is a protocol that is unreliable, offers no flow control or congestion control. That means that there is no requirement for packets to show up in order, or at all (Kurose & Ross, 2022, p. 122). Because of this I have included packet loss in my result table, as this is quite relevant when using the UDP protocol.

When running the three tests I got pretty similar results, with almost the same amount of packet loss. I found it interesting that h1-h9 and h7-h9 almost had the same result when both transfering 20 MB. I expected that there would be a bigger difference considering how h1-h9 must go through one more router. However, that link had a 40Mb bandwith capacity and 10ms delay, while I was sending 20 MB which the L1 link had enough capacity to handle. There was a slight difference between the results, but not as much as I would have imagined.

By keeping the send rate at the link bottleneck bandwith I was able to get bandwiths close to the bottleneck capasity. When setting the send rate larger than the bottlenecks, I still got quite high bandwith, but the packet loss could be well over 50% and more. This shows how UDP can keep sending at high rates because packet loss is tolerated. I don't have the server side results. These would be different from the client side because of packet loss.

4.4 Test case 2: link latency and throughput

In this test case I measured latency (RTT) and throughput on the three links between the routers in the network. I first used ping with a count of 25 on each of the three router pairs to find the average RTT. Then I ran my simpleperf program for 25 seconds on the same three pairs to measure throughput.

4.4.1 Results

Link	**Average RTT**	**Throughput**
L1 (R1-R2)	20.404ms	38.76 Mbps
L2 (R2-R3)	40.365ms	29.04 Mbps
L3 (R3-R4)	20.318ms	19.25 Mbps

4.4.2 Discussion

These results are as excpected from the network topology. Figure 3 shows the bandwith and delay on the links between the routers. To find the average RTT for the links, we can multiply the delay with two. This is because the "ping" message will first have a delay to the recieving router, and the "pong" message will have the same delay travelling back. This gives us 2x delay. My results show that the average RTT is the same as the link delay multiplied by two. There is a slight deviation from the theoretical RTT of aproximately 0.3-0.4ms. This could be because of the program itself or the software, like the virtual machine or Mininet. However, as the variation is <0.5ms, I consider it trivial.

My results are close to the theoretical bandwith limitations on the throughput side. My results are between 0.35-1.24 Mbps lower than the theoretical limit. There are several reasons why my results are a bit below the network bandwith. The main reason is probably my simpleperf program. Simpleperf sends packets of 1KB each iteration. In comparison, the iPerf tool sends 8KB in UDP-mode (*IPerf - The TCP, UDP and SCTP Network Bandwidth Measurement Tool*, n.d.). iPerf manages to be more efficent by running fewer iterations. My program is probably not fully code optimized. Simpleperf is written in python, a higher level language, while tools like iPerf are written in lower level languages like C to be more effective. The throughput result is also affected by the virtual machine. As mentioned earlier, I saw massive change in results when switching VM software. Even though VMware gave better and more accurate results, it would be safe to assume that this VM also impacts the results.

4.5 Test case 3: path Latency and throughput

In this test case I measured path latency and throughput between host pairs. As with the case above, I used ping with a count of 25 and simpleperf for 25 seconds. I did this for all the three host pairs.

4.5.1 Results

Node path	Average RTT	Throughput
h1-h4	60.951ms	25.00 Mbps
h1-h9	80.970ms	17.36 Mbps
h7-h9	60.650ms	17.66 Mbps

4.5.2 Discussion

I test different node paths for this experiment. The first path, h1-h4 travels through the links L1 and L2. The second path, h1-h9, goes through L1-L3. Lastly, h7-h9 travels through L2 and L3. Figure 3 shows the following bandwith and delay for these three links: L1 (40Mb, 10ms), L2 (30Mb, 20ms) and L3 (20Mb, 10ms).

From h1 to h4, the theoretical average RTT will then be (10ms + 20ms)*2 = 60ms. I got 60.951ms. The slowest bandwith on the route is L2, which will be the bottleneck. The throughput should therefore be around 30Mbps. The result shows 25Mbps. From h1 to h9, the theoretical average RTT will be (10ms + 20ms + 10ms)*2 = 80ms. The results show 80.970ms. The bottleneck for this path is L3, which should give a 20 Mbps throughput. The results show 17.36 Mbps. From h7 to h9, the theoretical average RTT will be (20ms + 10ms)*2 = 60ms. The results show 60.650ms. The bandwith bottleneck is also L3, which should give the same throughput as h1-h9. The results show 17.66 Mbps.

These results show an average RTT of 0.97-0.35ms more than the theoretical average based on the link delays. There are different types of delay in a network route as seen in figure 4. Processing delay in my experiment could be from determing where to direct the packet. However, as this is a virtual network this would probably be miniscule. Queuing delay could be a factor here. This delay is caused by packets waiting to be transmitted onto the link. If the trafic is heavy and many packets have to wait in queue, the delay will be longer. As seen in the network topology code, there is a set maximum queue for the links, which could influence the result. The transmission delay is the delay that accures from transmitting packets. In packet-switched networks there is common that a packet only

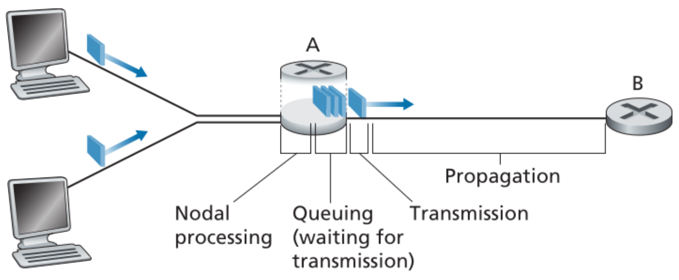

Figure 4: Different types of nodal delay (Kurose & Ross, 2022, p. 66)

can be transmitted after all the packets that have arrived before it have been transmitted. The transmission delay is calculated by bits divided by the transmission rate. Transmission delay are typically on the order of microseconds to milliseconds in practice (Kurose & Ross, 2022, p. 66). Lastly, the propogation delay which are determined by the time requried to travel from beginning to the end. This delay depends on the physical medium of the link and the length to travel. Since this is a virtual network, this delay would be none.

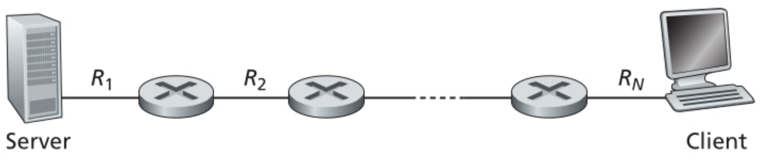

Figure 5: Throughput for a file transfer from server to client (Kurose & Ross, 2022, p. 74)

My results show that my throughput was at worst 5 Mbps slower than the link bandwith and 2.34 Mbps slower at best. Considering figure 5, the server can not pump bits through the links faster than the bps of link R_n. For a file transfer from server to client, the throughput for a file transfer is min{R_1, R_2,..., R_n}, which is the transmission rate of the bottleneck link along the path (Kurose & Ross, 2022, p. 75). When there is no other intervening traffic, the throughput should be aprocimately the same as the minimum transmission rate along the path from server to client. I don't have a definite answer to why my throughput result deviates some from the bottleneck link. My hypothesis is simply how my simpleperf program was written, that the code is written in python and not code optimized. Combined with the possibility of the virtual machine which could affect the results.

4.6 Test case 4: effects of multiplexing and latency

In this case I am looking at the effect of multiplexing where many hosts simultaneously communicate. I measure throughput and latency with simpleperf and ping simultaneously for different host pairs. This case is done in four different experiments: First, two pairs of hosts both using the links L1 and L2. Then one more pair using L1 and L2, for the total of three pairs. Then two pairs both sharing L2, but not L1 and L3. Lastly two pairs not sharing any links, but both going through the same router (R3).

4.6.1 Results

1. Two pairs of client-hosts (h1-h4, h2-h5) simultaneously

Node path	Average RTT	Throughput
h1-h4	71.362ms	14.90 Mbps
h2-h5	71.838ms	14.29 Mbps

2. Three pairs of client-hosts (h1-h4, h2-h5, h3-h6) simultaneously

Node path	Average RTT	Throughput
h1-h4	71.680ms	12.61 Mbps
h2-h5	71.044ms	9.22 Mbps
h3-h6	72.566ms	8.14 Mbps

3. Two pairs of client-hosts (h1-h4, h7-h9) simultaneously

Node path	Average RTT	Throughput
h1-h4	70.331ms	18.00 Mbps
h7-h9	69.089ms	10.84 Mbps

4. Two pairs of client-hosts (h1-h4, h8-h9) simultaneously

Node path	Average RTT	Throughput
h1-h4	69.058ms	27.92 Mbps
h8-h9	25.032ms	19.11 Mbps

4.6.2 Discussion

The job of delivering data to the correct socket is called demultiplexing. Gathering data chunks at the source host from different sockets, encapsulating data and passing them is called multiplexing (Kurose & Ross, 2022, p. 217). In the context of my experiment, the demultiplexing happening here is finding out which socket to deliver the data to, and vice versa for multiplexing. As mentioned earlier, when there are no intervening traffic, the throughput should be aprocimately the same as the bandwith of the bottleneck link. In this experiment however, there are intervening traffic to consider. This is where demultiplexing, multiplexing and TCP congestion-control comes into the picture.

TCP has a congestion-control mechanism. Each sender limit the rate at which it sends traffic into its connection based on the network congestion. If there are little congestion on the path between sender and reciever, TCP will increase its sending rate and vice versa if there is congestion along the path. The congestion-control mechanism works by setting a constraint on the rate which a sender can send traffic into the network. This is called the congestion window. At the start of a connection, this window is small, allowing only a few packages to be sent at a time. When the reciever is successfully recieving the data, this window increases, allowing more packets to be sent. When packets are dropped or delayed, this window decreases again to prevent congestion from occuring. There are different algorithms used for controlling the congestion window. I am not sure which one is used in pythons socket module.

A congestion-controll mechanism is said to be fair if the avarage transmission rate of each connection gets an approximately equal share of the link bandwith (Kurose & Ross, 2022, p. 306). Different congestion-controll algorithms tries to create fairness for the connections. One such algorithm is Additive Increase Multiplicative Decrease (AIMD). AIMD ensures fairness in the network by adjusting the sender's congestion window based on the network congestion. It works by the sender slowly increasing the amount of data transmitted until congestion occurs, then decrease the transmission rate to avoid more congestion. When the network is not congested, the congestion window increases linearly, which means the sender adds a constant number of packets to the window. When congestion is detected, the sender reduces the congestion window by half. This reduction is more significant than the increase, so the sender becomes more cautious about sending data after detecting congestion (Kurose & Ross, 2022, p. 301). This algorithm is designed to achieve fairness between multiple connections. This experiment shows congestion-control and fairness in action.

The throughput in experiment one shows that when running two host-pairs over L1 and L2, they share the link bottleneck bandwith of 30Mb equally. The same can be seen in experiment two where the three host-pairs shares the bandwith almost equally. The throughput was aproximately the same as the bandwith of the bottleneck link divided by *n* host-pairs, and could be considered fair. In experiment three, the host-pairs only share the L2-link. The bandwith distribution is split almost 2/3 and 1/3 between the two pairs. My hypothesis for why this is, is that h1-h4 gets a bigger portion of the bandwith because of a higher bandwith than h7-h9 throughout the route. Or it could be because h1-h4 starts out with both greater bandwith and delay, which can give it an unfair advantage. I am not sure if this is the reason, but we can see an example of unfair share of bandwith. In experiment four, the throughput is close to the bandwith of the bottleneck link. This is because the host-pairs don't share a link, only router R3.

The average RTT is about 10ms slower than the theoretical average across the board. I would have thought the RTT would be even slower because of congestion, but we can still see a clear effect of congestion and queue. h8-h9 performed a bit better at roughly 5ms slower, which could be attributed to no sharing of links with h1-h4 when running the experiment. However, we can see in experiment four that when not sharing a link but sharing a router, the throughput is high but the RTT is slower because of congestion in the shared router.

Even though the congestion-control algorithms are supposed to ensure fairness, this is not always the case, as seen in experiment four, and somewhat in experiment two. One reason for this in my case could be my timing when starting the clients. I started the client with the lowest host number first, and that can be seen in my results on experiment two, where h1 has a significantly greater throughput than h3. I think that if the connections ran for more time, the distribution would become more fair over time. 25 seconds is not much time, and perhaps if given enough time, the throughput would be more evenly distributed. The fairness provided by the algorithms are also under ideal scenarios, and in practice these are sometimes not met. It has been shown that when multiple connections share a bottleneck, the connections with the smaller RTT are able to grab bandwith at the link more quickly as it becomes free. They will end up with higher throughput than connections with larger RTT (Kurose & Ross, 2022, p. 307-308). This could maybe be the case for the throughput results in experiment three.

4.7 Test case 5: effects of parallel connections

In this test case I am looking at the effects of parallel connections. I am measuring both latency and throughput as per the other tests. I am running three pairs of hosts, where all of them uses the same link path. The first pair will run in parallel of two connections.

4.7.1 Results

Node path	**Throughput**
h1-h4 (1)	8.74 Mbps
h1-h4 (2)	7.10 Mbps
h2-h5	6.55 Mbps
h3-h6	7.07 Mbps

4.7.2 Discussion

Parallel TCP connections is a fairness problem. There is nothing to stop a TCP client from using multiple parallel connections. When a client uses multiple parallel connections, it gets a larger fraction of the bandwith (Kurose & Ross, 2022, p. 309). For example: Browsers have an unintended incentive to use multiple parallel TCP connections, rather than a single persistent one. The congestion-control aims to give each connection in the bottleneck link an equal share of the available bandwith. If there are *n* connections operating over a bottleneck link, then each connection gets 1/*n* of the bandwith. By opening multiple parallel connections to transport a single web page, the browser can cheat and grab more bandwith (Kurose & Ross, 2022, p. 144).

One can argue that this is what we are seeing in this test. If there was the same application running on h1-h4 in parallel connections, this application would get aproximately 2/4 of the bandwith, while h2 and h3 gets 1/4 each. Because the parallel client runs in its own thread it is being handled as an unique host. So even though this is a fair distribution on paper, h1-h4 ends up with double the amount of the total bandwith.

5 Conclusions

My design and implementation of simpleperf seemed to work as intended. My results were about the same as I would have expected. The program is not perfect of course, and there are some minor descrepencies of my results and the theoretical limits. However, they are close enough to be used as examples for network theories.

One big weakness in my simpleperf program, which were not tested in these cases, were my implementation of result printing in interval mode. The program prints the correct results, yet it prints out an individual table for each interval. This is not a problem when there is one client running, but when running many clients in parallel it is hard to keep track of which result belong to which client. I chose not to use time on this issue, but the table should be formatted in a way that gathered all the results from each client in separate tables.

The results of my tests showed how UDP packet loss works. They showed how performance was close to the theoretical limit when there were no other traffic in the network and how bandwith and delay set limits on the network. Further they demonstrated how transfers behaved with more traffic and congestions, and how traffic-congestion mechanisms works in the TCP protocol. My results also showed the concept of fairness when bandwith is shared, and the problem with parallel connections and fairness in TCP connections.

A limitation of this work is accuracy. From lack of code optimalization, virtual machines with sometimes questionable performance, the short amount of time each test was run and human error in regard to starting tests at the same time. This would all impact the accuracy of the results. Nevertheless, the point of this work is not accuracy, but to show different network concepts. And in that regard, I feel like I have succeeded.

6 References

Referenced with the APA Formatting and Style Guide 7th edition

iPerf—The TCP, UDP and SCTP network bandwidth measurement tool. (n.d.). Retrieved April 12, 2023, from https://iperf.fr/

Kurose, J. F., & Ross, K. W. (2022). *Computer networking: A top-down approach* (Eighth edition, global edition). Pearson.