APRIL 2, 2023

PORTFOLIO 1

A SIMPLEPERF TOOL

DIBA SHISHEGAR
S358980
Oslo Metropolitan University

1 Introduction

In this portfolio I will dive into how networks communicate, and the tools used to measure it. I developed a simplified version of iPerf, a popular tool used for measuring network throughput. I called my version "simpleperf".

The main focus of my portfolio is to create a lightweight tool that could send and receive data between a client and a server using sockets in order to communicate with our topology. It was also to understand how networks communicate using my lightweight tool. Such tools are important in today's world because data transmission needs to be reliable and fast to please everyone, from users to developers. This report describes how I developed simpleperf, which was inspired by iPerf, and how it works.

My approach involved creating an application that has two modes: server and client. Simpleperf runs on a virtual network managed by Mininet within a virtual machine. It's important to note that simpleperf has some limitations due to it being a simplified version of iPerf. However, simpleperf serves as a good starting point for understanding network performance evaluation.

The rest of this report is structured as follows:

- In Section 2, I will explain how I built simpleperf and how the server and client communicate.
- Section 3 describes the virtual network environment I used to test simpleperf.
- Section 4 presents the performance evaluations, including the network tools used (such as iPerf and ping), detailed test cases with various scenarios. Each test case includes an explanation of the experiment, results (such as average RTT and throughput), and a discussion of the findings.
- Finally, in Section 5, I will give a brief conclusion summarizing the most important results and their meaning and I'll also discuss the limitations of my work. I'll conclude by addressing any unresolved issues or problems.
- The References section, Section 6, lists the sources I used during the project.

In summary, this report describes my experience developing simpleperf, a simplified network throughput measurement tool.

2 Simpleperf

Simpleperf is a python-based network tool designed to measure network throughput and bandwidth between a client and a server. This portfolio will explain this implementation in detail, with a focus on the server, client and the main function as well as the communication between the server and client.

1.Server Function

The server function is responsible for receiving and sending data back to the client. It begins by creating a socket using the socket library and binding it to a specific host and port number. The server then listens for incoming connections, handling them using following steps:

A. Accepting incoming clients and managing their connections within a context manager (with conn). This context manager makes sure the connection closes automatically when the “with” block is exited. This helps to prevent resource leaks (the program won’t release its resources resulting in reduced performance, increased memory consumption, system instability) and simplifies the code. To handle multiple client connections simultaneously, a “def parallel()” function is created, which takes the client’s connection as an argument and processes each connection in a separate thread.

B. Receiving data from the client and sending it back. The server uses the “recv()” and “sendall()” methods to achieve this.

C. Detecting the termination message (“BYE”) from the client. The server checks if the received data contains the “BYE” message using “if “BYE” in data.decode()”. I check it this way instead of “if data.decode() == BYE” because the message might be lost or mixed with other data during the transfer.

D. Acknowledging the termination message and closing the connection. The server sends an acknowledgment message (“ACK: “BYE”) back to the client and closes the connection.

2 .Client function
The client function connects to the server, sends data, and measures network performance. It performs the following tasks:

A. Connecting to the server using a socket, specifying the server’s host and port number.

B. Initializing a start time variable, a total data counter, and a duration variable to keep track of the data transfer, elapsed time, and the maximum duration for the transfer.

C. Entering a loop that sends and receives data until either the specified transfer amount is reached or the duration has passed. The client sends 1000 bytes of data in each iteration using “sendall()” method. The loop continues as long as the elapsed time is less than the duration parameter.

D. Measuring network performance by calculating the interval statistics, such as throughput and bandwidth, and printing them to the terminal using a formatted output. The client also checks if the specified transfer amount has been reached or if the duration has passed and if it has it proceeds to the next step.

E. Terminating the transfer by sending “BYE” message to the server, receiving an acknowledgment message and calculating the final statistics, also in a formatted manner. If the transfer amount or duration condition is met the client sends the termination message and closes the connection.

3 . **Main function**
The main function is responsible for parsing command line arguments, setting up necessary parameters and calling the server or client functions based on the user’s choice. It performs the following tasks:

A . Parsing command line arguments using argparse to define and process the required and optional arguments.

B . Validating the provided arguments using regex or other input validation techniques to ensure that they fulfill the expected format and values.

C . Handling error checking such as ensuring that the user provides valid host and port numbers, correct time and parallel and displaying appropriate error messages in each case.

D . Providing usage information if the user does not fulfill any command line arguments, guiding them on how to correctly use the simpleperf tool.

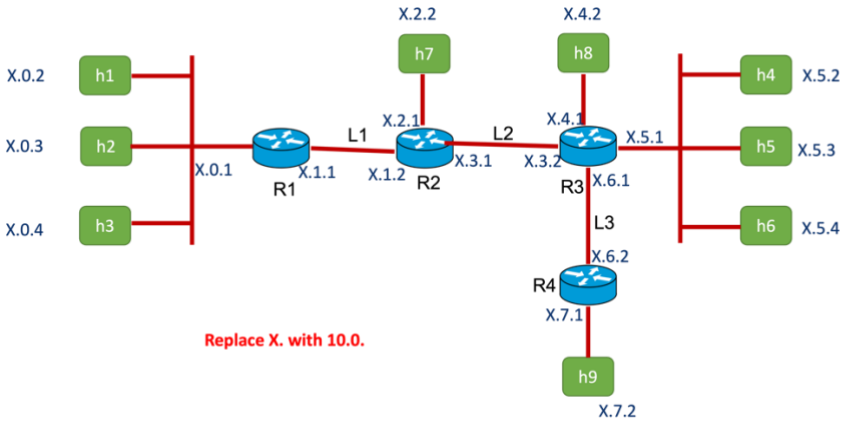
E . Calling the server or client function based on the user’s choice and passing the required parameters to them.

Communication using sockets
The communication between the server and the client is made possible by using sockets. Sockets provide a reliable communication channel for sending and receiving data. The communication process in simpleperf consist of the following steps:

1. The server creates a socket using the socket() function and binds it to a specific host and port number using the bind() method.
2. The server listens for incoming connections using the listen() method, with a "backlog" value, it determines how many connection requests can be held in a queue while the server is busy handling other requests.
3. The client creates a socket using the socket.socket() function and connects to the server using the connect() method, specifying the server's host and port number.
4. Once a client connects the server accepts the connection using accept() method which returns a new socket object representing the client connection.
5. The client sends 1000 bytes of data to the server in each iteration using the sendall() method.
6. The server receives the data using the recv() method and sends it back to the client using the sendall() method.
7. Both the server and the client can send and receive data simultaneously using the connected socket, enabling a "two-way" communication.
8. The termination message ("BYE") is sent by the client to the server to indicate that the transfer is completed. The server acknowledges the message by sending an ACK message back to the client.
9. The server and client close the connection using the close() method on their respective sockets.

3 Experimental setup

How I evaluated my simpleperf tool was with a virtual network managed by Mininet within a virtual machine, called ubuntu. I uploaded the network/topology to my ubuntu machine and used the terminal to run my simpleperf code. The topology that was used was:



When everything was uploaded, the topology and my simpleperf code, to the ubuntu machine I could now run the terminal and see the raw data. I used the terminal and typed in "sudo fuser -k 6653/tcp" then I ran "sudo pyhton portfolio-topology.py" to access the topology I just uploaded to the machine. After I had done this, I could see the terminal writing out the topology. I could now use mininet. The screen would show "mininet>". I would then type in for example "mininet > xterm r1 r2" for test case 2, then two terminal windows would pop up, r1's and r2's terminal windows to be specific, and I could run my simpleperf code. If we continue with the test case 2 example the right code to access my simpleperf code would be to run "python3 simpleperf.py -s -b 10.0.1.2 -p 8088" on r2's terminal window (server) and "python3 simpleperf.py -c -I 10.0.1.2 -p 8088 -t 25" on r1's terminal window (client). By clicking enter the raw data would present itself.

With the help of Mininet and ubuntu I was able to run my simpleperf tool on a network and get the raw data I need in order to measure different throughputs.

4 Performance evaluations

4.1 Network tools

The tools that I used for my experiment was ping, iperf and simpleperf. I also used Wireshark to see if the correct amount of data (1000 bytes) was being transferred between my server and client. I used iperf to measure the bandwidth in UDP mode for test 1. To check the link latency, I would use ping for test case 2,3,4 and 5 while also using my own simpleperf code to test case 2,3,4, and 5 to check the transfer and bandwidth.

4.2 Performance metrics

I used rate/ bandwidth, and ping /RTT to evaluate my simpleperf tool. By looking at these performance metrics I could see if my simpleperf tool was accurate enough. I already have access to iperf, and by comparing the results from iperf with my simpleperf tool I could tell if my code was on the right track. I also had the topology to assist me regarding the right bandwidth.

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

To measure the bandwidth with iperf in UDP mode I had to run three separate iperf tests. The first test was between host 1-4, second was between host 1-9 and lastly between host 7-9. On my ubuntu machines terminal I would first type in “sudo fuser -k 6653/tcp” then “sudo pyhton portfolio-topology.p” to access the correct topology for all the test cases. For the first scenario the client is h1 and the server is h4. In the terminal I wrote “xterm h1 h4” to get two terminal windows, one for h1 and one for h4. In the server window (h4) I wrote the code: “iperf -s -u” and for the client window (h1) I wrote: iperf -c 10.0.5.2 -u -b 90M. What does these lines mean?

The -s implies it is in server mode, ready to listen, and the -u implies it is a UDP connection. On the client side, the -c implies it is in client mode, ready to send back data, and -u still implies it is a UDP connection while -b specifies which rate I would like to send my data with, aka the bandwidth. In my first try I tried with 90M. As we can see, the actual bandwidth is at 29.1 Mbits/sec, with a 69% packet loss, as well as a relatively low jitter, so why did I lose so many packets? Why is the bandwidth at 29.1 M when I specified the bandwidth to be 90M? This is because we need to take a closer look at the topology. In the topology we can see that between host 1 and host 4 there are some routers between them. Between router 1 and 2 we see that the bandwidth is at 40 Mb with a latency of 10 ms and between router 2-3 the bandwidth is at 30 Mb per 20 ms. With our first try we set the bandwidth to be 90M, and as we saw we had a packet loss of 69% and the actual bandwidth was set to 29.1M. This is the result of a buffer overload at each router, and because it’s over UDP we lose these packets. If we try to send data in a lower rate, let us say 25M, we can see that we have 0% packet loss. This is because the rate that we are sending is lower than the slowest bandwidth link in the topology (lower than the 30 M between router 2 and 3). Therefore, I would choose any rate less than 30 M to measure the bandwidth with.

When it comes to estimating the bandwidth without knowing anything about the topology, I would start the same way. I would set the bandwidth to be 90M and see that the packet loss is high. I would then look at the actual bandwidth and see that it says “29.1 Mbits/sec”. This tells me that I should set the bandwidth to be a little lower, so I try with 25M, and now I see 0% packet loss and even lower

jitter than before. One could also just set the bandwidth to be different numbers until we see 0% packet loss, but this could take a while since you must guess your way to the right answer.

I would then proceed to do the exact same test for host 1-9 and host 7-9. The only difference is that the bandwidth is even lower (20M) so to get 0% packet loss we must have a rate lower than that. I chose rate 15M for both host 1-9 and 7-9 and got 0% packet loss with the actual bandwidth being 15.7 Mbits/sec for both host 1-9 and host 7-9.

4.4 Test case 2: link latency and throughput

To measure the RTT and bandwidth I used ping and simpleperf. First, I wanted to check the RTT and bandwidth between routers 1 and 2. To do this I typed in "xterm r1" then I wrote "ping 10.0.1.2 -c 25" in the black terminal window so I could get the RTT from router 1 to router 2. After I got the average RTT which was 22.992 ms I also needed the bandwidth between the two routers. To get the bandwidth I needed to use my own simpleperf tool. I typed in "xterm r1 r2" in the ubuntu terminal then I proceeded to type in "python3 simpleperf.py -s -b 10.0.1.2 -p 8088" for r2's window (the server) then "python3 simpleperf.py -c -I 10.0.1.2 -p 8088 -t 25" for r1's window (client). The bandwidth between router 1 and 2 is 38.14 Mbps. The RTT is low, and the bandwidth is high with a low latency. I also checked the topology and saw that between router 1 and 2 there are not too many other devises along the network path, it's a simple path where router 1 only has one other link and router 2 has two other links, so the results make sense.

For the RTT and bandwidth between router 2 and 3 I proceeded to repeat the same steps as I did for router 1 and 2. I just had to change "xterm r2 r3" and had to use r3's ip address which is 10.0.3.2. The RTT was 44.657 ms and the bandwidth was 28.46 Mbps. Here we can see that the RTT is higher with a lower bandwidth compared to router 1 and 2. The reason the RTT is higher, is for two reasons. Reason one being there are more devices along the network path resulting in higher RTT, router 2 has two other links while router 3 has three other links. Reason two being the latency is higher than between router 1 and 2. The latency is 20 ms between router 2 and 3 while it's only 10 ms between router 1 and 2. Again the results make sense, higher RTT with lower bandwidth, along with a higher latency, and more devices along the network path.

For router 3 and 4 (used r4's ip address 10.0.6.2) the RTT was 22.262 ms and bandwidth 19.08 Mbps. Both the RTT and bandwidth are low, why? If we take a look at the topology again, we can see that there are a few other devices in the network path, similar to test 1. Router 3 has three other links which is a lot but router 4 only has one other link so it makes up for router 3 having so many. The latency is also low (10 ms) like test 1. Therefore, the RTT is low while having a low bandwidth with a low latency, and a simple network path. The results make sense once again.

4.5 Test case 3: path Latency and throughput

Test case 3 reminds of test case 2, but this time we are interested in sending from the hosts and not the routers. I follow the same steps as test case 2, I ping from h1 to h4, using the ip address 10.0.5.2 (h4's ip address), using the code "ping 10.0.5.2 -c 25. Before pressing enter I would expect the latency/ average RTT to be around 66 ms. My reasoning is because in test case 2 we saw that the RTT between router 1 and 2 was ca 22 ms and between router 2 and 3 it was ca 44 ms, and now in our new test case the data is not only going to travel between one router but three. Looking at our topology we see that h1 is on the left side and h4 is at the complete opposite side, and between these two hosts are three routers that we already know the RTT for. I then thought to add router 1 and 2's RTT with router 2 and 3's, since that is h1

and h4's path to each other. After pressing enter I see that my expectations were correct and the RTT came out to be on average 66.651 ms.

To measure the bandwidth between h1 and h4 I followed my same steps in test case 2. I typed "xterm h1 h4" then typed in "python3 simpleperf.py -s -b 10.0.5.2 -p 8088" for h4's terminal window and "python3 simpleperf.py -c -I 10.0.5.2 -p 8088 -t 25" for h1's terminal window. The expected bandwidth between host 1 and host 4 is around 30 Mbps, because looking at the topology we see that the lowest bandwidth link between the routers in h1 and h4' path is 30 Mbps, so the bandwidth between h1 and h4 must be the lowest bandwidth number. After pressing enter I see that my expectations were right, the bandwidth is 28.35 Mbps between host 1 and host 4.

For host 1 to host 9 the procedure is the same as for host 1 and host 4 but changed the ip address to 10.0.7.2 (h9's ip address). I expected the RTT between the hosts to be around 88 ms, because from test case 2 we already know the RTT between r1-r2 to be 22 ms, for r2-r3 to be 44ms and the newest addition to h1-h9's path is r3-r4's RTT which is 22 ms. If we add all these three RTT together, 22 ms + 44 ms + 22 ms, we get 88ms. After pressing enter I get average RTT to be 89.892 ms. Expectations are correct.

The bandwidth between h1-h9 is expected to be around 20 Mbps. Looking at the topology we can see that the path between h1 and h9's lowest bandwidth rate is 20 Mbps. Pressing enter we can see again that the expectations were correct with a bandwidth of 18.82 Mbps.

RTT between h7-h9 (using the same ip address 10.0.7.2) is expected to be around 66 ms because again we know the RTT from r1-r2 to be 44 ms and from r3-r4 to be 22 ms and adding these two together gives us a RTT of 66 ms which is h7's path to h9. The actual RTT is 65.701 ms, expectations are met.

Bandwidth expectations for h7-h9 is to be around ca 20 Mbps since on the path from h7-h9 it's the lowest bandwidth between router 3-4. The actual bandwidth is 18.96 Mbps, and we can see that we were right about our expectations.

4.6 Test case 4: effects of multiplexing and latency

In test case 4 I had to measure average RTT and bandwidth for two pairs of hosts running simultaneously. This means running 6 terminal windows at the same time. On ubuntu's terminal I typed "xterm h1 h1 h2 h2" then "xterm h4 h5". One pair of h1 and h2 is going to measure the RTT while the other pair is going to measure the bandwidth with h2 and h5, the same time I measure the RTT. On h1's terminal window I would write in "ping 10.0.5.2 -c 25" and on h2 "ping 10.0.5.3 -c 25". This was RTT measurements between h1-h4 and h2-h5. I would not press enter, just have them ready. Then I would write in "python3 simpleperf.py -s -b 10.0.5.2 -p 8088" into h4's terminal window (server) and "python3 simpleperf.py -c -I 10.0.5.2 -p 8088 -t 25" for h1's window (client). I would still not press enter and just have it ready with the pings. The last pair of hosts I would just repeat the last step again, for h5 "python3 simpleperf.py -s -b 10.0.5.3 -p 8088" (server) and for h2 "python3 simpleperf.py -c -I 10.0.5.3 -p 8088 -t 25" (client). Now all 6 terminal windows are ready to run, I would quickly press the enter button for each terminal window and wait for the results.

The average RTT between h1-h4 was 103.030 ms while the bandwidth was 18.54 Mbps. Average RTT for h2-h5 was 105.140 ms and bandwidth was 10.37 Mbps. As we can see we have high RTTs and low bandwidths. If we run h1-h4 by itself we have seen in our previous tests that the RTT is 66 ms and bandwidth 28.35 Mbps, so how come the RTT has become so high and the bandwidth so low from just running simultaneously with another pair of hosts? In my case, both pairs of hosts (h1-h4 and h2-h5)

use the same link between r2 and r3 to communicate with each other, and this link has a capacity of 30 Mbps, so h1 and h2 must share this link between themselves resulting in one of them having ca 20Mbps and the other one getting ca 10 Mbps. When both pairs of hosts communicate simultaneously, the total traffic demand on the link may exceed its capacity, leading to network congestion. Due to these network congestions, we have gotten queuing delays and retransmissions (because it's a TCP connection) and as a result, the available bandwidth is shared between the two pairs of hosts, and the RTT may increase, resulting in degraded network performance.

Let us try this again but for three pairs of hosts this time. Average RTT for h1-h4 is 102.490 ms, h2-h5 is 105.231ms, and h3-h6 is 106.894 ms. While their bandwidths are: h1-h4 is 15.52 Mbps, h2-h5 is 7.05 Mbps and h3-h6 is 7.19 Mbps. As we can see the RTT are ca the same as before, still high. When it comes to their bandwidths, their paths are still the same all three pair of hosts must share their network path. The lowest bandwidth capacity is 30 Mbps, according to our topology. If we look at the first pair of hosts, h1, we can see that it got ca 15Mbps, this is most likely because I pressed enter first on its terminal window when I ran the test. Anyways we see that the 30Mbps had to be shared between first h1, then h2, which got the other half of the 30 Mbps, then h3 came right after and got h2's half of the 30Mbps, resulting in h2 and h3 sharing h2's 15Mbps and getting ca 7Mbps each.

I repeat this process for h1-h4 and h7-h9. Average RTT for h1-h4 is 101.179 ms and for h7-h9 it is 101.075 ms. The bandwidth for h1-h4 is 18.10 Mbps and for h7-h9 it is 10.47 Mbps. Again, we see that the RTT is high because of congestion and because their network path is ca the same length (long), and the bandwidth is low. H1-h4 and h7-h9 share the same network path only between r2-r3, and here we see that h1-h4 gets ca 18Mbps which is ca half of the slowest bandwidth capacity for h1-h4s path which is 30 M while for h7-h9 the slowest bandwidth capacity is 20M which h7-h9 also got ca the half of.

Repeating this for the last time between h1-h4 and h8-h9. Average RTT for h1-h4 is 93.078 ms which we've seen is ca what it's been the whole way, but for h8-h9 it is 42.032 ms. Here the RTT for the last pair of hosts have dropped. This is because the path between h8-h9 is relatively short, and they only go through one router, which also means less latency. When it comes to bandwidth, h1-h4 has 28.33 Mbps and h8-h9 has 19.03 Mbps. Here we can see that each path has almost gotten the full capacity of the slowest bandwidth link. This is because these two pair of hosts don't share the path r2-r3 as the tests before has. Because they don't share any routers there is no congestion and they can take up the full capacity of the slowest bandwidth link.

4.7 Test case 5: effects of parallel connections

In test case 5 I ran three pairs of hosts at the same time, one of them having a parallel connection. H1, h2, and h3 were the clients and h1 was the client running a parallel connection. H4, h5 and h6 were the servers. For h1 and h2 I used the code "python3 simpleperf.py -c -I 10.0.5.3/4 -p 8080/8" and for h1 "python3 simpleperf.py -c -I 10.0.5.2 -p 8082 -P 2". For the servers I ran the basic server code "python3 simpleperf.py -s -b 10.0.5.2/3/4 -p 8080/8/2". After having my servers ready and listening for a connection I hit enter on all three of my clients at the same time, as best as I could.

For the parallel connection (h1-h4) the bandwidth for the first connection was 7.23 Mbps and for the second connection it was 6.54 Mbps. Between h2-h5 the bandwidth was 8.59 Mbps and for the last hosts, h3-h6, it was 7.13 Mbps. If we compare my number to the topology we see that my results seem logical. The slowest bandwidth link we have to look at is still 30 Mbps, so if we take 30 Mbps divided by 4 (its 4 connections going through the same links) we get 7,5 Mbps. Each connection should have a bandwidth around 7,5 Mbps, as we see they are all around 7,5 Mbps +-. It's also worth to mention that the reason each connection has to share the lowest bandwidth link is again because of network congestion.

5 Conclusions

In conclusion, the experiments preformed using my simpleperf tool showed that the network bandwidth was successfully distributed among multiple hosts on the network. The tool proved to be an efficient way to simulate network traffic and measure network performance. However, some limitations were identified during the testing process, such as the need for more synchronized command execution (test 4) and possible inaccuracies due to the limitations of the tool itself.
For test 4, we saw the bandwidth being distributed almost evenly but not quite. Having host h1-h4 having a bandwidth capacity of 20 Mbps while host h2-h5 had 10 Mbps. This should have been more even, around 15 Mbps each.

Nonetheless, the results from the testing still provide valuable insight into the network performance and can be used as a starting point for further optimization of the network infrastructure.

6 References (Optional)

- Lekhonkhobe, Tshepang. "Argparse Tutorial." *Python Documentation*, https://docs.python.org/3/howto/argparse.html. Accessed on 19.03.2023.
- ProgrammingKnowledge, director. *Python Argparse and Command Line Arguments*. *YouTube*, YouTube, 27 June 2020, https://www.youtube.com/watch?v=f7eQ2lQv-NI . Accessed 19.032023.
- Satyam, Kumar. "Python: Time.monotonic() Method." *GeeksforGeeks*, GeeksforGeeks, https://www.geeksforgeeks.org/python-time-monotonic-method/ . Accessed 23.03.2023.
- ThomasThomas 4, et al. "How Do I Get Monotonic Time Durations in Python?" *Stack Overflow*, 2010, https://stackoverflow.com/questions/1205722/how-do-i-get-monotonic-time-durations-in-python . Accessed on 23.03.2023
- "Time - Time Access and Conversions." *Python Documentation*, https://docs.python.org/3/library/time.html. Accessed on. 26.03.2023
- "What Is Network Congestion? Common Causes and How to Fix Them?" *GeeksforGeeks*, GeeksforGeeks, 19 Feb. 2023, https://www.geeksforgeeks.org/what-is-network-congestion-common-causes-and-how-to-fix-them/. Accessed on 02.04.2023
- Team, IR. "A Guide to Network Congestion: Causes and Solutions I IR." *A Guide To Network Congestion: Causes and Solutions I IR*, https://www.ir.com/guides/network-congestion. Accessed on 03.04.2023