17. APRIL 2023

PORTFOLIO 1

DATA2410 NETWORKING AND CLOUD COMPUTING

ALEX VU
S186969
OSLOMET - OSLO METROPOLITAN UNIVERSITY

Contents

1 Introduction

In today’s fast-paced digital world, the performance of networks is of a great importance as it directly affects the user’s experience and the overall efficiency of various applications and services. As global data traffics continues to grow exponentially, it is crucial to evaluate and optimize aspects of network performance to meet the ever-increasing demands.

One of the most commonly used tools for measuring network performance is iPerf. It is a open-source software designed to generate traffics between two endpoints in order to measure various network parameters, such as bandwidth, latency and packet loss (iPerf, n.d.).

The report will be divided into two main parts. The first part focuses on the implementation of a simplified version of iPerf, henceforth referred as Simpleperf. By utilising socket programming in Python, this program will be developed to measure network performance in the later tasks. The second part of the report will delve into the evaluation of network performance by using iPerf, Simpleperf and ping. These tools will be used to analyse the performance of various network links and configurations.

By exploring these topics, this report aims to address the challenges faced in network performance evaluation, such as network congestion, buffering, queueing, and packet losses. Through the implementation and analysis of various performance measurement tools, this report seeks to provide valuable insights and practical solutions to optimize network performance.

2 Simpleperf

One of the key components of this report is the implementation of Simpleperf. The purpose of this implementation is to design a simplified version of iPerf using socket programming in Python. The implementation of the program is designed to measure the performance of a network between a client and server over a TCP connection. The program is organised into several functions that handle argument parsing, validation, data formatting, and connection handling for both the server and the client. The functions include parse_args(), validate_args(), format_values(), format_num(), print_table(), handle_server(), start_server(), handle_client(), and start_client().

parse_args()

The first function defined in the program is parse_args(), which is responsible for parsing the command-line arguments provided by the user. In this case, the argparse module is used to define and parse the arguments. This ensures that a user can input various flags to set different options for the program. These flags include:

Flag	**Description**
-s	Enables the server mode
-c	Enables the client mode
-I	Sets the IP address of the server to bind to. The default value of this flag is '127.0.0.1'
-p	Sets the port number that the server will listen on or the client will connect to. The default value of this flag is 8088
-f	Sets the format for the summary data that will be printed where the default value is MB
-t	Sets the duration in seconds for which the data should be generated. The default value of this flag is 25
-i	Sets the interval in seconds at which statistics should be printed
-n	Sets the number of bytes that should be transferred by the client
-P	Sets the number between 1-5 of parallel connections to the server. The default value of this flag is 1

Once a flag is used, the function calls the validate_args() function to check for any invalid arguments. Finally, the function returns the parsed arguments, which will be used in later functions.

Depending on what the user inputs to the program, it will either run in server mode or in client mode. In server mode it will listen for incoming connections from clients, while in client mode it will connect to a server and send data.

To summarise, this function is responsible for parsing the arguments provided by a user, perform error checks and returning the parsed arguments for further use in the program.

validate_args()

In the text of the assignment, it has set mandatory tasks to validate the flags. To do this, this code has defined a function named validate_args(), which is responsible for checking whether arguments passed to the program are valid or not. The function takes the parsed as input and checks each argument one by one. If any of the arguments is invalid, the function will print an error message and exit the program.

The first check in the function is to ensure that the -s flag and -c flag are not enabled at the same time. If both flags are enabled, it is impossible to tell whether the program should run in server mode or client mode, so the function will exit with an error message.

The next check is to ensure that either the -s flag or the -c flag is enabled. If neither flag is enabled, the program will not know what mode to run in, so the function will exit with an error message.

The function will then check whether the IP address specified with the -b flag and the IP address specified with the -I flag are valid IP addresses. This is done using the ipaddress module, which raises a ValueError exception if the IP address is not valid (Python documentation, n.d.). If either IP address is not valid, the function will exit with an error message.

The function continues to check whether the port number specified with the -p flag is a valid port number. The assignment text has specified that the port numbers must be integers between 1024 and 65535.If the port number is outside of this range, the function will exit with an error message.

The next check is to ensure that the format specified with the -f flag is one of the three valid formats: B, KB, or MB. If the format is not valid, the function will exit with an error message.

The function will then check whether the value specified with the -t flag is a positive integer. If the value is less than 1, the function will exit with an error message.

The next check is to ensure that the value specified with the -i flag, if provided, is a positive integer. If the value is less than 1, the function will exit with an error message.

Next, the function will then check whether the format specified with the -n flag is valid. The format should be an integer followed by either B, KB, or MB. If the format is not valid, the function will exit with an error message.

Finally, the function checks whether the value specified with the -P flag is a valid number of parallel connections. The assignment has set that the value must be an integer between 1 and 5. If the value is not valid, the function will exit with an error message.

In summary, the validate_args() function performs basic error checking on the arguments passed to the program to ensure that they are valid. This helps to prevent the program from running with invalid input, which could result in unexpected behaviour or errors.

format_values()

The purpose of the format_values() function is to convert a given value to a specified format. This function is used in the main program to format the data generated by the server or the client to a desired format, in this case bytes (B), kilobytes (KB) or megabytes (MB).

The function takes in two arguments: value and input_format, where value is the value that needs to be formatted and input_format specifies the format in which the value needs to be converted. Next, the function begins by checking the input format. If it is "B", then the value is already in bytes and it returns the value as is. If the input format is "KB", the function converts the value to kilobytes by dividing it by 1000 and returns the result. Finally, if the input format is "MB", the function converts the value to megabytes by dividing it by 1000000 and returns the result.

After performing the conversion, the function will return the formatted value to the caller.

format_num()
The function format_num is responsible for extracting the integer value from a string that contains a unit (either B, KB, or MB) that is passed as an argument. This function is called when the -n flag is specified by the user to indicate the number of bytes that should be transferred by the client.

The function first checks if the input string contains a unit of KB or MB, and if it does, it extracts the integer value and multiplies it by 1000 or 1000000 respectively to convert it into bytes. If the input string contains a unit of B, it simply extracts the integer value and returns it as bytes.

Once the integer value is extracted, it is returned by the function to be used as the number of bytes to be transferred by the client.

print_table()
The purpose of the print_table() function is to print out the data values generated by the program to create a table row for a pleasant view. It sets the format for the table row by defining a string variable called format_row before it unpacks the list of data and format each item into the table row. Each column has a width of 20 characters. Finally, it prints the formatted table row to the console using the print method.

start_server()
The start_server() function is responsible for initialising a server that listens for incoming connections. To start off, the function extracts the IP address and port number from the -b and -p flags. It can also extract the specified format from the -f flag.

Next, the function creates a TCP socket using the socket() function from the socket module with the AF_INET and SOCK_STREAM constants as arguments. It binds the socket to the specified IP address and port number using the bind() method of the socket object. It then starts listening for incoming connections using the listen() method.

Once the server is listening, it will print out a message indicating that the server is listening. It then enters a loop that continuously waits for incoming connections using the accept() method. When a client connects, the function prints a message informing that the client has connected to the server. It will then create a new thread using the Thread() function from the threading module to handle the connection. The thread will pass the client socket, client address, and specified format as arguments. Then, the function will initiate the thread using the start() method. The connection is handled by the handle_server() function in a separate thread.

handle_server()
The handle_server() function is called by the former function in order to handle the packages it receives from the client. Once a client connects to the server, it receives the client socket and address as arguments along with the input format for the data.

It starts by setting the starting time at once a client connects and sends the bytes. The timer will be used later to calculate the duration of the transfer.

```
def handle_server(client_socket, client_address, input_format):
    ...
        ...
        total_received_bytes = 0

        while True:
            received_bytes = client_socket.recv(1000)

            total_received_bytes = total_received_bytes +
len(received_bytes)

            if "BYE" in received_bytes.decode():
                client_socket.sendall("ACK: BYE".encode())

                break
        ...
```

Figure 1: A sample of the handle_server() function showing how it receives the data. Retrieved from simpleperf.py

This sample of codes handles the receiving of data packages from the client. The server will continuously receive data from the client in chunks of 1000 bytes until it receives a message that consist of "BYE".

For each chunk of data received, the total number of bytes received is accumulated in the variable "total_received_bytes". The length of the received data is added to the total received bytes since the data is received in bytes.

Once the server receives a message that consist of “BYE”, it will send an acknowledgement, "ACK: BYE", back to the client. This acknowledges that the server has received their message and that the data transfer is complete before the loop breaks.

After the transfer is completed, the function calculates the duration of the transfer and formats the total number of received bytes into bytes, kilobytes, or megabytes based on the what format the user desired. The rate of incoming traffic is then calculated in megabits per second. The function then defines the headers of the table and the data of the table, which includes the client's IP address, the time interval, the number of received bytes, and the rate of incoming traffic. Finally, it prints out the values in a table format and closes the socket connection with the client.

To sum up, this function is responsible for receiving the data packages from the client and accumulating the number of bytes received. It uses a while loop to continuously receive bytes from the client until it receives a message that the transfer is complete. Once the message is received, the function calculates the total duration of the transfer, formats the total received bytes into the desired format and calculates the transfer rate in Mbps. Finally, it prints out the received data in a table format, and the socket connection with the client is closed.

start_client()

The start_client() function is responsible for connecting the client to the server. First, it starts by extracting the necessary information from the arguments that the user has inputted such as the IP address of the server, the port number, the duration of the transfer, the format of the output data, the interval, the number of bytes to be transferred and the number of parallel connections.

```
def start_client(args):
    ...
    for i in range(input_parallel):
        client_socket = socket(AF_INET, SOCK_STREAM)
        client_socket.connect((ip_address, port_number))
        print(f"Client connected with server {ip_address}:{port_number}")

        client_ip_address, client_port_number =
client_socket.getsockname()

        if i == input_parallel - 1:
            print_table(headers)

        thread = threading.Thread(target=handle_client,
args=(client_socket, client_ip_address, client_port_number, input_time,
input_format, input_interval_time, input_num))

        connection_list.append(thread)

        thread.start()
```

Figure 2: A sample of the start_client() function showing how it connects a client to a server. Retrieved from simpleperf.py

This sample creates a specified number of parallel connections to the server using a TCP socket. The number of parallel connections is defined by the user when they use the '-P' flag. The code will then iterate over each connection and creates a new thread for each connection using the handle_client() function. This will pass over the arguments to handle_client(), which will start the data transfer and the calculation. The threads are then appended to a list and initiated.

If there is no parallel connection, the code will still work as it will simply create one connection to the server and execute the handle_client() function on a single thread. This is due to that the '-P' flag has a default setting of one connection.

Once all threads have been initiated, the code awaits for all the threads to finish using the join() method. This ensures that the program waits for all the threads to finish executing before the it terminates.

handle_client()

The purpose of the handle_client() function is to send data from the client to the server. First, it starts by checking if the -n flag is enabled, which specifies the number of bytes to be sent. This is due to that the -n flag and the -t cannot run simultaneously because of how time is calculated in them. If it is enabled, it converts the value of -n to bytes using the format_num() function and voids the -t flag. Once that is complete, it will then set the starting time and initializes the values of the sent bytes.

```
def handle_client(client_socket, client_ip_address, client_port_number,
input_time, input_format, input_interval_time, input_num):
    ...
    total_sent_bytes = 0
    bytes = b"0" * 1000
    while(input_time is None and total_sent_bytes < num_bytes) or
(input_time is not None and (time.time() - start_time) < input_time):
        sending_bytes = client_socket.send(bytes)
        total_sent_bytes += sending_bytes
        ...
```

Figure 3: A sample of the handle_client() function showing how it sends data to the server. Retrieved from simpleperf.py

The function will then start a loop to send 1000 bytes to the server using the client_socket.send() method. During each iteration, it sends the bytes to the server and updates the total sent bytes. If the -n flag is enabled, the loop runs until the total number of sent bytes reaches the value defined by the flag. If the -t flag is enabled, the loop runs until the time elapsed since the start of the transfer is equal to the value defined by the flag.

```
def handle_client(client_socket, client_ip_address, client_port_number,
input_time, input_format, input_interval_time, input_num):
    ...
    start_time = time.time()
    current_time = start_time
    interval_sent_bytes = 0
    bytes = b"0" * 1000
    while(input_time is None and total_sent_bytes < num_bytes) or
(input_time is not None and (time.time() - start_time) < input_time):
        ...
        interval_sent_bytes = interval_sent_bytes + sending_bytes

        if input_interval_time is not None and (time.time() -
current_time >= input_interval_time):
            current_time = current_time + input_interval_time
            elapsed_time = current_time - start_time

            interval_rate = (interval_sent_bytes * 8e-6) /
input_interval_time

            interval_sent_bytes_format =
format_values(interval_sent_bytes, input_format)

            data = [[f"{client_ip_address}:{client_port_number}",
f"{elapsed_time - input_interval_time:.1f} - {elapsed_time:.1f}",
f"{interval_sent_bytes_format:.2f} {input_format}", f"{interval_rate:.2f}
Mbps"],]

            for row in data:
                print_table(row)

            interval_sent_bytes = 0
        ...
```

Figure 4: A sample of the handle_client() function showing how it sends data to the server in intervals. Retrieved from simpleperf.py

In this figure it shows what the function does if the -i flag is enabled. The if-statement ensures that the function will save the data of the transfer on a specific interval while the loop runs. Once that specified interval time has passed, it calculates the transfer rate for that interval. It will then format the interval data to be printed in a table row. The function then resets the interval sent bytes to zero for the next interval. This will continue until either the time or the amount of data has been reached.

```
def handle_client(client_socket, client_ip_address, client_port_number,
input_time, input_format, input_interval_time, input_num):
    ...
    while(input_time is None and total_sent_bytes < num_bytes) or
(input_time is not None and (time.time() - start_time) < input_time):
        ...
    client_socket.sendall("BYE".encode())

    response = client_socket.recv(1024).decode()

    if response == "ACK: BYE":
        end_time = time.time()
        duration = end_time - start_time

        total_sent_bytes_format = format_values(total_sent_bytes,
input_format)

        rate = (total_sent_bytes * 8e-6) / duration

        data = [[f"{client_ip_address}:{client_port_number}", f"0.0 -
{duration:.1f}", f"{total_sent_bytes_format:.2f} {input_format}",
f"{rate:.2f} Mbps"],]

        for row in data:
            print_table(row)

    client_socket.close()
```

Figure 5: A sample of the handle_client() function showing what it does after transferring the data. Retrieved from simpleperf.py

Once the loop completes, the function will send a message from the client that consist of “BYE” to the server to indicate that the transfer is complete and will then wait for an acknowledgement. Once the client receives an acknowledgement from the server, it calculates the duration of the transfer, formats the total sent bytes value in the specified format, calculates the rate in Mbps and prints the values in a table format. Finally, the client socket is closed.

To summarise, this function is responsible for transferring bytes from the client to the server and calculating various metrics related to the transfer. It uses various flags, such as -n and -t, to determine the amount of data to be sent and the duration of the transfer. It can also divide the metrics in intervals through the -i flag. Once the transfer is complete, it will send a message to the server to confirm that the transfer is done. It closes the socket once it receives an acknowledgment in response.

Main entry point

The last part of the code is the main entry point of the program. It checks if the argument specifies whether to run the program as a server or a client. If the -s flag is present, it starts the server by calling the start_server() function with the parsed command line arguments. If the -c flag is present, it starts the client by calling the start_client() function with the parsed command line arguments.

3 Experimental setup

In this report, Oracle VM Virtual Machine were used as the virtualisation software to run Ubuntu 22.04.2 LTS as the guest operating system. For the topology, the report used portfolio_topology.py that were provided in the assignment. In order to run the topology, Mininet were installed in the system.

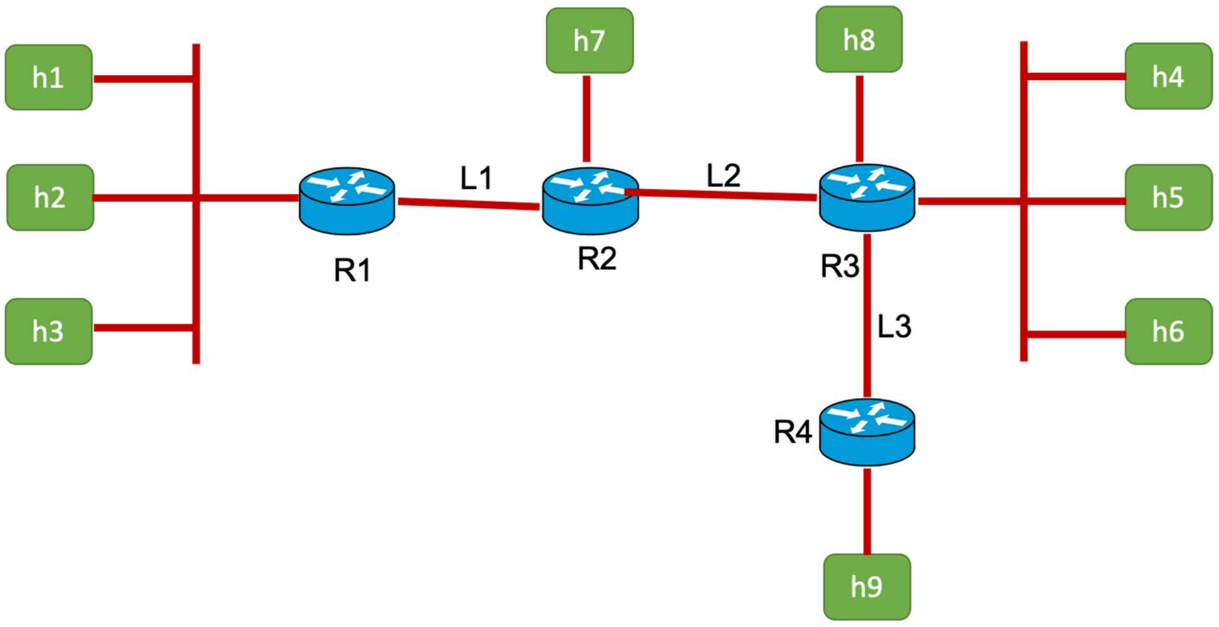

Figure 6: An illustration of portfolio_topology.py (Islam, 2023).

The topology consists of nine hosts and four routers that are interconnected through two switches. The host are labelled from H1 to H9, while the routers are labelled from R1 to R4. Furthermore, the topology is divided in several subnets. In subnet A, the hosts H1 to H3 are connected to switch s1, where s1 is connected to router R1. Subnet B connects the routers R1 and R2 with each other, while subnet C connects router R2 with host H7. In subnet D, the routers R2 and R3 connects with each other. Next subnet (which is only named subnet in the document) connects router R3 to host H8. In subnet E, the hosts H4 to H6 are connected to switch s2, who is connected to router R3. Subnet G connects the routers R4 and R4 with each other, while subnet I connects router R4 to host H9.

The purpose of this set up is to evaluate the performance of the network topology created using Mininet, in order to assess the efficiency for upcoming tasks. The report aims to provide insight to solve issues that occurs in network performance.

4 Performance evaluations

4.1 Network tools

In the experiment, three tools were used to evaluate the upcoming tasks: iPerf, ping and Simpleperf.

iPerf is a measuring tool for network performance can measure throughput, jitter, and packet loss by using TCP or UDP. This tool will only be used in the first test to measure the bandwidth in UDP, as stated in the assignment text.

Ping will be used to check whether a network device is reachable and to measure the round-trip time for packets to travel between two devices. It will be used to measure latency of the task,

The last tool, Simpleperf, will be used to measure throughput of the tasks.

4.2 Performance metrics

Bandwidth is the maximum amount of data that can be transferred over a network connection in a given period of time (Johnsen, 2022). By measuring the bandwidth, we can determine the maximum amount of data that can be transferred between two hosts in a given time, which can help in determining the performance of the network. In this report, the bandwidth will be given in megabits per seconds (Mbps).

In order to measure the latency, ping will be utilised. Ping works by sending a small packet to a specified IP address while it waits for a response. The time taken for the packet to travel to the destination and back is measured, which gives us an estimate of the latency. The average latency is calculated by sending multiple packets and taking the average of the response times (GeeksforGeeks, 2021).

To measure the throughput, bandwidth will be used as a measurement from the implemented Simpleperf. While bandwidth is maximum amount of data that can be transmitted over a network connection per unit of time, throughput is the actual amount of data that is successfully transmitted over in a given time (Kurose & Ross, 2016, p. 122).

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

The purpose of this experiment is to decide which rate is most optimal to send data when measuring the bandwidth in UDP mode.

The experiment begins by launching Mininet with the specified topology. Then, xterm is used to open terminals with the specific hosts as client-server pairs. In this case, the experiment will measure the bandwidth between H1 to H4, H4 to H9 and H7 to H9.

4.3.1 Results

The assignment has tasked to do three separate tests. Here are the following results:

Between H1 – H4:

[ID]	Interval	Transfer	Bandwidth	Jitter	Lost/Total Datagrams
[1]	0.0000-9.9999 sec	31.2 MBytes	26.2 Mbits/sec	0.027 ms	9/22294 (0.04%)

Between H1 – H9:

[ID]	Interval	Transfer	Bandwidth	Jitter	Lost/Total Datagrams
[1]	0.0000-10.0040 sec	18.7 MBytes	15.7 Mbits/sec	0.147 ms	6/13378 (0.045%)

Between H7 – H9:

```
 [ ID]              Interval          Transfer          Bandwidth          Jitter      Lost/Total Datagrams

 [  1]      0.0000-9.9991 sec     18.8 MBytes      15.7 Mbits/sec     0.010 ms           3/13378 (0.022%)
```

4.3.2 Discussion

One approach to measure the bandwidth is to determine the highest available bandwidth for each link in the network. In this instance, the assignment has provided a topology with a bandwidth of each links. The topology has specified three links: L1, L2, L3.

The first link is named L1 and is between the routers R1 and R2. It has specified a bandwidth of 40 Mbps. Next link is called L2 and lies between the routers R2 and R3. It has a bandwidth of 30 Mbps. The last link is labelled L3 and located between the routers R3 and R4, which has a bandwidth of 20 Mbps.

By knowing the bandwidth of each links, the experiment is able to avoid choosing a rate that is higher avoid congestion and packet losses. However, the rate must also not be too low, as it may leads to slower transfer speeds and longer transfer times.

The report will also be using another metric to find the optimal rate. By analysing the packet loss, we can determine the maximum sustainable rate without any packet loss. Packet loss occurs when one or more packets of data traveling across the network fail to reach their destination. This could be a consequence of exceeding the bandwidth (Kurose & Ross, 2016, p. 302). By varying the rate and monitoring packet losses, we can identify the rate at which packet loss is minimized and therefore the highest achievable bandwidth for the given link. In this case, the report aims to have a packet loss less than 1 % on each test.

For the first test between the hosts H1 and H4, the topology shows that the links between them is L1 and L2. Each links has a bandwidth of 40 Mbps and 30 Mbps respectively. Since the traffic passes through the two links, we need to determine the link with the smallest bandwidth capacity. In this case, the link L2 has the smallest bandwidth capacity of 30 Mbps, which means that the maximum bandwidth that can be transmitted across this link is 30 Mbps. To account for possible packet losses, the report has chosen a rate of 25 Mbps. The chosen rate is based on the fact that the links between H1 and H4 go through multiple routers, which increases the likelihood of packet loss due to network congestion or other issues.

The next test between the hosts H1 and H9, the topology shows that the traffic passes through the links L1, L2 and L3. They each have a bandwidth of 40 Mbps, 30 Mbps and 20 Mbps, respectively. In this instance, the link with the lowest bandwidth capacity is L2 with bandwidth of 20 Mbps. Since the traffic passes through every router in the topology, the chosen rate in this case is 15 Mbps.

In the last test between the hosts H7 and H9, the topology shows that the traffic passes through the links L2 and L3. Each of the links has a bandwidth of 30 Mbps and 20 Mbps respectively. This test is similar to the former test between the hosts H1 and H9, that they both pass L3 which has the lowest bandwidth capacity of 20 Mbps. The report has also chosen the same rate of 15 Mbps, however, in hindsight the rate could be slightly higher due to traffic passes one less router.

In cases where the topology is not known, one approach may be to perform a series of tests with different rates and analyse the results to estimate the maximum bandwidth of the network. This can be done by gradually increasing the rate until packet loss occurs and then reducing the rate until no packet loss occurs. The maximum rate that can be achieved without packet loss can be considered as the estimated maximum bandwidth of the network. However, this approach may not be practical since it requires several tests with different rates to be performed, which can be time consuming and resource intensive. Additionally, the results may not be very accurate since network conditions can vary over time and from one test to another.

4.4 Test case 2: Link latency and throughput

The purpose of this task is to analyse the latency and throughput between routers.

The experiment will still use the same topology in Mininet, but in this scenario, the latency and the throughput between R1 - R2, R2 - R3 and R3 - R4 will be measured in xterm. This will be done by using ping and Simpleperf.

4.4.1 Results

The assignment has tasked to do several tests. Here are the following results:

Link	Client	Server	Average round-trip time (RTT)	Throughput
L1	R1	R2	20.545 ms	36.50 Mbps
L2	R2	R3	40.157 ms	27.32 Mbps
L3	R3	R4	20.441 ms	18.38 Mbps

4.4.2 Discussion

The latency of the ping can be estimated based on the provided topology. This is done by calculating the round-trip time (RTT) of the packets sent between the routers. This will give an approximate measure of the time it takes for a packet to travel from one router to another and back again.

In the first link between the routers R1 and R2, the topology has stated that the delay is set at 10 ms. With the given delay of 10 ms, the round-trip time can be expected to be approximately 20 ms. It will take 10 ms for the packet to travel from R1 to R2 and another 10 ms for the packet to travel back from R2 to R1. This is consistent with the result from the test, where the average round-trip was 20.545 ms.

The next link between the routers R2 and R3, the delay is set at 20 ms in the topology. The estimate of the round-trip time will be about 40 ms, since the packet will use 20 ms to travel forward and back between the routers. This is also in correspondence with the results from the test, with an average round-trip time of 40.157 ms.

The last link between the routers R3 and R4, the delay is set at 10 ms in the topology. The expected round-trip time will be around 20 ms, due to it take 10 ms to travel to a router and back again. The results from the test shows an average round-trip time of 20.441 ms, which is in correlation with was estimated.

Based on the information obtained in the previous task, it was found that the bandwidth is found within the topology. The bandwidth of the three links was measured to be 40 Mbps, 30 Mbps, and 20 Mbps respectively. By using the bandwidth, the expected throughput between the respective hosts can be estimated.

The results of the experiment show that the throughput of the first link was 36.50 Mbps, while the second and third links recorded the throughputs of 27.32 Mbps and 18.38 Mbps, respectively. Overall, the results of the experiment show that the actual throughput of the links is slightly lower than the bandwidth. This is expected due to the bandwidth only depicts the maximum amount of data that can be transmitted while the throughput is the actual amount of data that is successfully transmitted over. The throughput will remain lower than the bandwidth unless the network operates at maximum performance.

4.5 Test case 3: Path latency and throughput

This task aims to analyse the latency and throughput between hosts with ping and Simpleperf.

The experiment will use the same method in the previous experiment, but instead of analysing the routers, this task will analyse the hosts.

4.5.1 Results

The assignment required several tests to be performed and the following are the results:

Client	Server	Average round-trip time (RTT)	Throughput
H1	H4	63.044 ms	27.03 Mbps
H1	H9	81.969 ms	16.18 Mbps
H7	H9	61.593 ms	17.53 Mbps

4.5.2 Discussion

Similar to the previous task, the latency can also be determined based on the given topology. The difference in this case is that the traffic passes through multiple network devices before reaching its destination.

In the first test between hosts H1 and H4, the packet will travel through the links L1 and L2. Each links has a delay of 10 ms and 20 ms, respectively. To calculate the expected latency, the delays of all links along the path must be added twice. In this case, the expected latency is calculated as to be approximately 60 ms. This aligns with the test results, which showed an average round-trip time of 20.545 ms.

For the next test from host H1 to host H9, the packet will travel through every links in the topology. The links has a delay of 10 ms, 20 ms and 10 ms, respectively. In this case, the total sum of the delay is 40 ms for one path. This means that the expected latency will be around 80 ms. The test result indicates that the average round-trip time was 81.969 ms, which is approximately what was expected.

In the last test from host H7 to host H9, the packet will travel through the links L2 and L3. Each links has a delay of 20 ms and 10 ms, respectively. Similar to the first test, the expected latency will be about 60 ms. According to the test results, the average round-trip time was 61.593 ms, which is roughly with what was expected.

As with the previous task, the throughput can be determined by examining the provided topology. Similar to the first case, the lowest bandwidth capacity needs to be identified because the traffic passes through multiple links.

According to the test results, the throughput for the first link was 27.03 Mbps, while the second and third links had throughputs of 16.18 Mbps and 17.53 Mbps, respectively. The network topology had previously revealed that link L2 has the lowest bandwidth capacity on the path from H1 to H4, while the last two pairs share the link with the lowest capacity on the way to H9. The links L2 and L3 have bandwidths of 30 Mbps and 20 Mbps, respectively. Overall, the test results suggest that the actual throughput of the links is slightly lower than the advertised bandwidth. Since the deviation is not significant, other programs consuming system resources could have affected the performance.

4.6 Test case 4: Effects of multiplexing and latency

The purpose of this case is to analyse the latency and throughput when multiple of hosts communicate simultaneously.

Multiplexing is a method by sharing a single communication channel among multiple users or applications. It allows multiple streams of data to be combined and transmitted over a single physical communication link or channel. When multiple pairs of hosts communicate simultaneously, they share the link's available bandwidth. Ideally, they would share the bandwidth equally, however, this is not often the case as some users may consume more bandwidth than others, which could lead to decreased performance and fairness concerns.

One method to measure the fairness in resource allocation is the Jain's Fairness Index (JFI). It ranges from 0 to 1, where 0 represents completely unfair allocation and 1 represents perfectly fair allocation. In the context of multiplexing, Jain's Fairness Index can be used to evaluate how fairly the available network bandwidth is being shared among different users or applications. This can be used to adjust network configurations or prioritising traffics (GeeksforGeeks, 2022). Jain's Fairness Index is calculated by using following formula:

$$JFI = \frac{(\sum_{i=1}^{n} x_i(t))^2}{n \sum_{i=1}^{n} x_i(t)^2}$$

where n is the number of connections and $x_i(t)$ is the throughput of the ith connection.

In this experiment, multiple of hosts will communicate with a specified server using ping and Simpleperf simultaneously in Mininet.

4.6.1 Results

The test case consists of four parts and here are the following results:

First test:

Client	Server	Average round-trip time (RTT)	Throughput
H1	H4	70.634 ms	12.50 Mbps
H2	H5	71.224 ms	16.08 Mbps

Second test:

Client	Server	Average round-trip time (RTT)	Throughput
H1	H4	73.207 ms	8.43 Mbps
H2	H5	69.814 ms	11.89 Mbps
H3	H6	69.574 ms	7.99 Mbps

Third test:

Client	Server	Average round-trip time (RTT)	Throughput
H1	H4	68.665 ms	16.00 Mbps
H7	H9	69.808 ms	12.19 Mbps

Fourth test:

Client	Server	Average round-trip time (RTT)	Throughput
H1	H4	67.283 ms	26.26 Mbps
H8	H9	24.681 ms	18.52 Mbps

4.6.2 Discussion

The first test showed that for the first pair of hosts, the average round-trip time was 70.634 ms, while the pair had a throughput of 16.08 Mbps. In contrast, the second pair had an average round-trip time of 71.224 ms and throughput of 16.08 Mbps. Compared to the previous test with the same pair of hosts, this case had an increase of almost 10 ms. This occurred because both pairs shared the same bandwidth and communicated simultaneously, causing packets from one pair to wait for the other pair's packet to be transmitted, resulting in a higher round-trip time. The second pair had a higher throughput than the first pair, indicating that they were likely given a larger share of the available bandwidth. Although the difference seemed significant, the Jain's Fairness Index showed a value of 0.98, indicating a high level of fairness. This means that the allocation of resources was fair and performance variation was minimal.

In the second test, three pairs had to be run simultaneously. Similar to the first test, the three pairs had around the same increase of time compared to the previous case. In this case, the average round-trip time of the pairs were 73.207 ms, 69.814 ms and 69.574 ms, respectively. However, the results suggest that adding more pairs of hosts to the network can have an impact on the throughput. The result gave the pairs a throughput of 8.43 Mbps, 11.89 Mbps and 7.99 Mbps. This could be due to increased competition for network resources as more hosts communicate simultaneously. The Jain's Fairness Index show a value of 0.85, which is relatively fair. To increase the fairness, one suggestion would be to reallocate additional resources to the first and third pair. However, it is worth mentioning that the results may not be precise due to the network conditions may differ over time and between tests.

During the third test, two pairs had to be run simultaneously, however, in this case they only share one bandwidth in the traffic. The results gave average round-trip times of 68.665 ms and 69.808 ms, respectively for the pairs. The measured throughput of the pairs shows 16.00 Mbps and 12.19 Mbps, respectively. In this instance, it is clear which link is the bottleneck, being link L2, which leads to queuing and resulting in these numbers. While the Jain's Fairness Index shows a high value of 0.98, it must be noted that the pairs do not share the whole traffic and may not represent the fairness in resources as they each may have different purpose.

Like in the previous test, in the fourth test, two pairs had to be run simultaneously where they do not share the whole traffic. This time, the average round-trip time of the pairs were 67.283 ms and 24.681 ms, respectively. Although the difference appears substantial, it is worth noting that the first pair travels through links L1 and L2, which have a combined one-way delay of 30 ms, while the second pair only traverses link L3 with a delay of 10 ms. The delay in milliseconds accumulates due to the queues at the points where the links intersect. Concerning the throughput of the pairs, the results show 26.26 Mbps and 18.52 Mbps, respectively. Similar to the round-trip time, the difference is caused by that the pairs do not share the bandwidth of the whole path they travel through. The Jain's Fairness Index show a high value of 0.97, but like in the previous test, it may not represent the fairness in resources.

To summarise, the results of the tests indicate that adding more hosts to the network can have an impact on the round-trip time and throughput, possibly due to increased competition for network resources. However, the Jain's Fairness Index suggests that the resources are being allocated fairly and equitably with little disparity in performance. Nonetheless, the results may not be highly precise due to the network conditions fluctuating over time and between tests.

4.7 Test case 5: Effects of parallel connections

The purpose of this case is to analyse the throughput when three pairs of hosts connect simultaneously. However, the first pair will have a parallel connection with the usage of the -P flag from Simpleperf. Otherwise, this case will use similar method as the previous case.

4.7.1 Results

Here is the result from the test:

Client	Server	Throughput
H1	H4	5.54 Mbps
H1	H4	6.57 Mbps
H2	H5	6.90 Mbps
H3	H6	5.29 Mbps

4.7.2 Discussion

Referring to the topology, it shows that the pairs will share every bandwidth along the path. The estimated throughput can be found by identifying the link with the lowest bandwidth capacity and dividing it by the number of connections. In this case, the link with the lowest capacity is link L2 with a bandwidth of 30 Mbps. The bandwidth of each pair is estimated to be approximately 7.5 Mbps, where the throughput will be slightly lower than that. The result from the test case shows that the pairs have a throughput of 5.54 Mbps, 6.57 Mbps, 6.90 Mbps and 5.29 Mbps, respectively, which is slightly lower than the estimation. The cause of this could be the greater competition for network resources as more hosts communicate simultaneously, similar to the previous tasks. This could lead to congestion due to the network is unable to handle the data that are being transmitted, which can cause a decline in performance. The Jain's Fairness Index of the throughput shows a high value of 0.98, indicating a fair distribution of resources. However, since the results may differ over time and between tests, the value may not be representative in this case. In general, it appears that the current task is similar to the previous one. The consequence of multiple pairs that connect simultaneously is that it can result in reduced network performance.

5 Conclusion

In conclusion, the tests conducted with Simpleperf have been successful in evaluating the network performance under various conditions. The tests allowed us to measure the average round-trip time and the throughput of different pairs of hosts connecting simultaneously. The results show that the performance is affected by the number of hosts that connect simultaneously, as well as the available bandwidth of the links.

An issue with test case 4 and 5, is that the tests were not run in exactly the same time. The tests were done by manually by hitting enter to start the tests on each terminal as fast as possible. In hindsight, a script could be developed to run the hosts simultaneously to eliminate this issue.

It has been observed that some students have faced issues while running their virtual machine to conduct the tests. It is important to note that the computer that has been used to write this report was loaned out to a couple of students. This has been done with the permission of the subject manager. The students have given feedback that the tests have been successful, but it is uncertain how the evaluation will affect the outcome of this report.

6 References

GeeksforGeeks. (2021). *What is Ping? - GeeksforGeeks*. Retrieved from https://www.geeksforgeeks.org/what-is-ping/

GeeksforGeeks. (2022). *TCP Fairness Measures - GeeksforGeeks*. Retrieved from https://www.geeksforgeeks.org/tcp-fairness-measures/

iPerf. (n.d.). *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*. Retrieved from https://iperf.fr/

Islam, S. (2023). *portfolio.png.* Retrieved from Canvas: https://oslomet.instructure.com/courses/25246

Johnsen, R. (2022). *båndbredde – Store norske leksikon*. Retrieved from https://snl.no/b%C3%A5ndbredde

Kurose, J., & Ross, K. (2016). *Computer Networking A Top-Down Approach, 7th Edition.* Pearson plc.

Python documentation. (n.d.). *ipaddress — IPv4/IPv6 manipulation library*. Retrieved from https://docs.python.org/3/library/ipaddress.html