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.

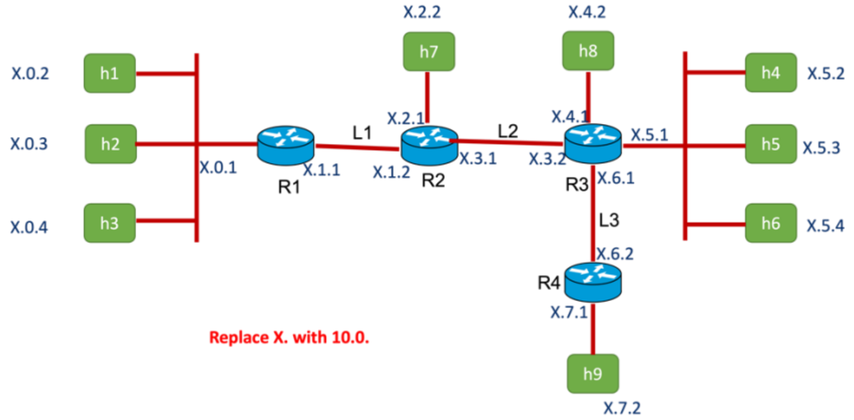
E . Calling the server or client function based on the user’s choice and passing the required parameters to them.

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/RTT (in ms), 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 (in Mbps).

4.2 Performance metrics

I used rate/bandwidth, and latency/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.

Summarizing table

	h1-h4 (90M)	**h1-h4 (25M)**	**h1-h9 (90M)**	**h1-h9 (25M)**	**h1-h9 (15M)**	**h7-h9 (90M)**	**h7-h9 (15M)**
Bandwidth	29.1 Mbps	26.2 Mbps	19.4 Mbps	19.4 Mbps	15.7 Mbps	17.4 Mbps	15.7 Mbps
Packet loss	69%	0%	79%	26%	0%	81%	0%

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. 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, test 1 between h1-h4, 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 h1-h9. 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. First, I tried with 90M for h1-h9 and got bandwidth at 19.4 Mbps with 79% data loss, as expected. Then I tried with 25M and got a bandwidth of 19.4 Mbps and data loss at 26%. Still too high packet loss, therefore I tried with 15M and now we can see that I have 0% packet loss because the rate I chose is lower than 20 Mbps (lowest bandwidth link in the topology for h1-h9's path).

For h7-h9 I tried again with 90 M and had a bandwidth of 17.4 Mbps and a packet loss at 81%. Again, looking at the topology I need to stay under 20 M to get 0% packet loss. I tried one last time with 15 M and finally got 0% packet loss.

4.4 Test case 2: link latency and throughput

Summarizing table

	r1-r2	r2-r3	r3-r4
Average latency/RTT	22.992 ms	44.657 ms	22.262 ms
Bandwidth	38.14 Mbps	28.46 Mbps	19.08 Mbps

To measure the RTT and bandwidth between r1-r2 I used ping and simpleperf. 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, and it has a latency of 10 ms one way (10 ms + 10 ms = 20 ms total RTT, ca. what I got), so a RTT of ca 20 ms makes sense and matches the topology, same for the bandwidth, it matches the topology.

For the RTT and bandwidth between router 2 and 3 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 each way 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 the RTT was 22.262 ms (again around 10 ms each way, matches the topology) 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 each way) 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

Summarizing table

	h1-h4	h1-h9	h7-h9
Average latency/RTT	66.651 ms	89.892 ms	65.701 ms
Bandwidth	28.35 Mbps	18.82 Mbps	18.96 Mbps

Test case 3 reminds of test case 2, but this time we are interested in sending from the hosts and not the routers. Before knowing the results, 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 1-2 with router 2-3's RTT together, 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.

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 I expected the RTT to be around 88 ms. My reason is 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

Summarizing table running h1-h4 and h2-h5 simultaneously

	h1-h4	**h2-h5**
Average latency/RTT	103.030 ms	105.140 ms
Bandwidth	18.54 Mbps	10.37 Mbps

Running h1-h4, h2-h5 and h3-h6 simultaneously

	h1-h4	**h2-h5**	**h3-h6**
Average latency/RTT	102.490 ms	105.231 ms	106.894 ms
Bandwidth	15.52 Mbps	7.05 Mbps	7.19 Mbps

Running h1-h4 and h7-h9 simultaneously

	h1-h4	**h7-h9**
Average latency/RTT	101.179 ms	101.075 ms
Bandwidth	18.10 Mbps	10.47 Mbps

Running h1-h4 and h8-h9 simultaneously

	h1-h4	**h8-h9**
Average latency/RTT	93.078 ms	42.032 ms
Bandwidth	28.33 Mbps	19.03 Mbps

The results for test 1 show that 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 only run a ping from h1-h4 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 while pinging? 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. H1-h4 and h2-h5 must compete for this link between themselves resulting in one of them having ca. 18 Mbps and the other one getting ca. 10 Mbps.

Furthermore, 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 in addition to the propagation delays (very little, almost no effect on simpleperf) from only pinging and retransmissions (because it's a TCP connection). As a result, the available bandwidth is shared between the two pairs of hosts, and the RTT may increase, resulting in degraded network performance.

It's worth to mention that the RTT is also ca 40 ms higher for both hosts running simultaneously. This is not by chance, looking at the topology we see that the latency is 20 ms for the slowest bandwidth link between the network path between h1-h4 & h2-h5. This means if the max queuing is exceeded, there will be 20 ms delay added both ways for the packets (40 ms in total). We add 40 ms to h1-h4 RTT which is ca 66 ms + ca 40 ms (delayed time) = ca. 100 ms (real answer 103.030 ms). We try the same for h2-h5: RTT by itself around 66 ms + ca 40 ms (delayed time) = ca. 100 ms(real answer 105.140 ms). This delayed time we have added to the hosts RTT is the buffer time, we will see this again for all the other cases too.

Test 2 consists of 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 because of network congestion because they share the exact same network path. This results in the buffer time of ca 40 ms added to the hosts original RTT we got by just pinging, resulting in higher RTT. 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-h4, 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. Furthermore, we see that the 30Mbps had to be shared between first, h1-h4, then between h2-h5, which got the other half of the 30 Mbps. Right after that h3-h6 came and got h2-h5's half of the 30Mbps, resulting in h2-h5 and h3-h6 sharing 15Mbps and getting ca. 7,5Mbps each.

On to test 3: 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 because their network path is shared to an extent so because of this a buffer time of 40 ms is added to both hosts RTT while the bandwidth is low. H1-h4 and h7-h9 only share the same network path between r2-r3, and because they share the same path both pairs of hosts get around half the bandwidth of their slowest bandwidth link. H1-h4's slowest link is 30 Mbps, therefore it gets ca. 18 Mbps. H7-h9's slowest link is 20 Mbps, resulting in it getting ca 10 Mbps.

Lastly test 4: h1-h4 and h8-h9. Average RTT for h1-h4 is 93.078 ms which has dropped a little in ms, and 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 compete for one router, which also means less latency. They also share a shorter network path together compared to the other tests. So here the buffer time is not the same for both paths. For h1-h4 it's still an added 40 ms delay (ca 66 ms + 40 ms = ca 100 ms) but for h2-h5 the added delay is now 20 ms (look at the slowest bandwidth link between h8-h9, it's 10 ms each way) so we find the equation for h8-h9 RTT by first running ping solo from h8-h9 which is ca 20 ms, then add the buffer time of 20 ms and get ca 40 ms. 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. Since they don't share any routers there is no network congestion and they can take up the full capacity of the slowest bandwidth link.

Furthermore, I would also bring up the Jain's fairness index. The Jain's fairness index tells us how fair the distribution of the bandwidth between the hosts are. This is helpful because the index will solidify if the bandwidth is fair or not. Let us look at test case 4 test 1, the bandwidth for h1-h4 is 18.54 Mbps and for h2-h5 is 10.37 Mbps. By just looking at these numbers we would think the distribution is unfair, but the Jain's fairness index is 0.92 so it tells us that the distribution is fair. Therefore, the reason for it not being around the same bandwidth is affected by other factors as I've explained earlier, congestion, synchronized command execution and even my own simpleperf code not being optimized.

Jain's fairness index h1-h4 & h2-h5 & h3-h6: 0.86

Jain's fairness index h1-h4 & h7-h9: 0.93

Jain's fairness index h1-h4 & h8-h9: 0.96

As we can see, all of the bandwidths are distributed fairly between the hosts.

4.7 Test case 5: effects of parallel connections

Summarizing table

	h1-h4 (-P 2)	**h2-h5**	**h3-h6**
Bandwidth	7.23 Mbps 6.54 Mbps	8.59 Mbps	7.13 Mbps

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 numbers 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 network path to reach their respective server) 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, since they now all compete for the exact same network path.

Referencing to my earlier explanation about Jain's fairness index, we can also use it in test case 5. Just like test case 4, we run hosts simultaneously therefore we can use the index to get an indication about the bandwidth being fairly distributed or not. Using the formula, we have an index of 0.98, meaning the bandwidth is fairly distributed.

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. Besides this I did not encounter any limitations or other difficulties that are worth mentioning.

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.

Disclaimer:
Gaute Kjellstadli used my MacBook Pro (m1 chip) to run his own code on my ubuntu machine. The tests seem to have been working but I do not know if this have had any negative impact on my test results etc.

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