




MAY 16, 2023

# PORTIFOLIO 2 REPORT

RELIABILITY OVER UDP

GLENN ANDRÉ HANSEN SURLAND, MOSTAFA FIRAS HADAD, ZUBEYR  
MUSTAFA AHMED, JAN AALBORG ERIKSEN

S354604, S351891, S356342, S354568  
Oslo Metropolitan University



1	INTRODUCTION	2
2	BACKGROUND	2
3	RELIABILITY FUNCTIONS	3
3.1	Stop-and-wait	3
3.2	Go-back-N	3
3.3	Selective repeat	4
4	IMPLEMENTATION	4
4.1	Sockets	4
4.2	argparse	4
4.3	Time	5
4.4	Struct	5
4.5	Server	5
4.6	Client	5
4.7	Filename	5
4.8	Reliability methods	6
4.9	Test cases	6
5	DISCUSSIONS	7
5.1	Throughput values with different window sizes and RTT	7
5.1.1	Stop and wait with different RTTs	7
5.1.2	Go-Back-N with different windows and RTTs	8
5.1.3	Selective Repeat with different windows and RTTs	9
5.2	Skipping an ACK	9
5.3	Skipping a sequence number	11
5.4	Handling of losses, reordering and duplicates	12
6	CONCLUSIONS	13
7	REFERENCES	13

# 1 Introduction

This report will include our work on the portfolio 2 assignment in the DATA2410 course. This assignment is a part of the course exam and contains a file transfer application made with the UDP transport protocol. UDP is an unreliable transport protocol, which is why we made reliability functions to make sure our file was received correctly.

With this application we are building reliability on top of a non-reliable transport protocol. We will discuss why this is important, and how it is done. We will look at the background for this project, as well as the implementation of the application.py program. We will delve into the different reliability functions and how these are implemented, as well as how they work. Lastly, we will discuss the different test cases and explain how they work before we finish up with our conclusion.

This assignment is closely connected to the portfolio 1 assignment we completed earlier this semester. Different code blocks and explanations may be taken from any of our four portfolio 1 assignments.

When figuring out how to create the application.py program, we use the knowledge accumulated over the last semester from lectures, labs, and the syllabus for this course. The syllabus is taken from the book: "Computer Networking A Top-Down Approach" written by James F. Kurose and Keith W. Ross (Kurose & Ross, 2022).

During the reading of this report, we hope that you will get a better understanding of our solution to the problem at hand, namely building reliability over a non-reliable transport protocol. We hope that you find our findings and solutions interesting.

# 2 Background

Application.py was made as a combined efforts of all the four people in our group. The background for its creation is as stated in the introduction, as a part of the portfolio 2 assignment in the DATA2410 course. As reliability is something we always wish for when it comes to file transfer, it is a good exercise to implement it for ourselves. UDP and TCP are the two transport layer protocols we have tested out during this course, and they function completely opposite when it comes to reliability. TCP is the reliable protocol and will automatically make sure that every packet is sent and received correctly during a transfer. Whereas UDP is a non-reliable protocol and will continue to send data no matter the loss.

To make UDP reliable, we construct different protocols on top of UDP to ensure the reliable delivery of the packets. These protocols are common in network delivery applications and are called Stop-and-Wait, Go-back-N, and Selective Repeat. We will go further in detail on all these protocols in section 3.

## 3 Reliability functions

Stop-and-wait, Go-back-N and selective repeat, it is a form of automatic repeat request (ARQ) protocol that is used to ensure that data packets are delivered correctly to the server. ARQ, in computer networking, is a form of reliable data transfer protocols based on retransmission to ensure reliable data transfer (Kurose & Ross, 2022).

### 3.1 Stop-and-wait

The Stop-and-Wait protocol is a simple and reliable method to ensure reliability during transmission between two devices, in this example, a server and a client. In this protocol, the client sends a packet of data to the server, then waits for an ACK from the server to confirm that the transmission of the packet was successful. If the client does not receive an ACK within a certain timeout period, it assumes that the packet was lost and retransmits the same packet.

Once the server receives the packet, it sends an ACK message to the client to acknowledge that it has received the packet successfully. If the server detects an error in the packet, like it being of the wrong sequence number, it discards the packet and requests that the client resend the correct packet.

The Stop-and-Wait protocol is reliable because it ensures that each packet is received and acknowledged before sending the next packet. It also ensures that lost packets are retransmitted, and error-free packets are not retransmitted unnecessarily. However, this protocol can be slow and inefficient, especially for long-distance transmissions or when network traffic is high, as it requires the client to wait for an ACK message before sending the next packet (Kurose & Ross, 2022).

### 3.2 Go-back-N

The Go-Back-N (GBN) method is a protocol used for reliable data transmission, and commonly used for communication channels that may have errors, such as noise or congestion.

In the GBN method, the client can transmit a sequence of packets through what we call a window. Within this window we specify a set number of packets that can be sent at a time. Each packet is numbered sequentially, to ensure that every packet is delivered in the right order. The server acknowledges each packet by sending an acknowledgment message back to the client. If the client does not receive an acknowledgment message for a packet within a certain time frame, it assumes that the packet was lost or corrupted, and retransmits that packet and all the subsequent packets in the window. This is where the name "Go-Back-N" comes from. The client goes back N packets depending on which packet was lost/corrupted relative to where it was when it received the NACK, or the timeout happened. The window controls how many packets the client can send at a time without waiting for an acknowledgment, while the server will discard all corrupted packets or packets received out of order.

The GBN method is efficient because it allows the client to send multiple packets without waiting for an acknowledgment for each packet. However, it can lead to inefficient use of bandwidth if packets are lost or corrupted frequently, as the client will have to retransmit multiple packets. Lastly, this method assumes that packets will be lost or corrupted in sequence, which often isn't the case in practice.

### 3.3 Selective repeat

Selective repeat (SR) is based on the same principles as GBN. In the SR, the client maintains a window of packets it can send to the server. Unlike in the GBN method, in SR the server can acknowledge individual packets rather than only entire sequences. The client can send multiple packets at a time without waiting for an acknowledgment, and if a packet is lost or corrupted, only that packet needs to be retransmitted, rather than the entire sequence. This is done by using a buffer, which allows the server to reorder the packets if they arrive out of sequence. The server will send a NACK for the packet that was lost/corrupted, and an ACK for the packets it received correctly.

The SR method is efficient because it allows the client to send multiple packets without waiting for an acknowledgment after each packet, just like GBN. It is also more efficient in terms of bandwidth usage because it only retransmits the lost or corrupted packets, rather than entire sequences. Additionally, the method allows the server to selectively acknowledge received packets, allowing for more efficient data transfer. However, it requires more complex buffering and tracking mechanisms at both the client and server, compared to the GBN method.

This makes the GBN method favourable when the connection has a low error rate, and SR favourable when the error rate is high. If GBN is used with a high error rate we will be wasting bandwidth by retransmitting multiple packets with each loss, whereas SR will minimize the number of retransmissions. As SR has a complex buffering system, it is not favourable with a low error rate, but GBN is nice when the error rate is low as we don't need to think about the wasted bandwidth.

## 4 Implementation

### 4.1 Sockets

In application.py we have used sockets as the medium from/to which we transfer data. Sockets are a library in python that allows us to send and receive data through these sockets. With sockets we can use a wide range of features and communication types to transfer data. We want to transfer over the internet, so we use the 'AF\_INET' attribute to specify this. By changing this attribute, sockets also allow us to send data over for example Bluetooth or IPv6, using the 'AF\_BLUETOOTH' and 'AF\_INET6' respectively. The second attribute sockets take in when you create the socket is the form of transmission. In portfolio 1 we used 'SOCK\_STREAM' for TCP, but in this case, we used 'SOCK\_DGRAM' for UDP.

### 4.2 argparse

When you want to start application.py you can do it in your preferred terminal. The application is made to be user-friendly with command line arguments using the argparse library. With the argparse library, you can easily enter command line arguments, also called flags, that we have specified in the code. Examples of these flags are '-f, -r, -s, -c'. These flags are what decides how the program will run. Every flag has a long version by default, and using argparse, you can also set short versions to make the command line arguments even more user-friendly. As an example of this, the '-f' flag is the short form of '--filename'. If you want to use the '-f' flag when running the program, you can choose if you want to use the short, or the long form of the argument. It also provides a flexible and customizable way to parse command line arguments, generate usage messages, and handle errors.

### 4.3 Time

The time function in python is used to get a value for a specific moment in time. It does this by returning the number of seconds since the Unix epoch, which is January 1, 1970, 00:00:00 UTC. This has many applications, but in our program, we use it to calculate the time it takes to transfer the data. We start by setting a start\_time variable before the function computes and set an elapsed\_time variable at the end of the data transfer. This elapsed\_time variable takes the time after the transfer and subtracts the start\_time, which gives us the total time of the transfer. Then you use another variable, called throughput, which divides the transferred files' size by the elapsed\_time. This gives us the throughput for the transfer.

### 4.4 Struct

Struct is a python library at the center of this application. It is a module that provides functions to convert native Python data types such as integers, floats, and strings to and from C struct represented as Python bytes objects (*Struct — Interpret Bytes as Packed Binary Data*, 2023). It is used to interpret and manipulate binary data stored in byte strings. In this program this library is used to pack and unpack header data, which is at the core functionality to make the transfer reliable.

### 4.5 Server

By using the '-s' flag you run the program in server mode. Server mode is considered another machine from the client, which we will talk about in the next section. The server's mission is to receive data sent by the client. When receiving data from the client, the server will also send acknowledgement packets (ACKs) back to the client as one stage of the reliability of the transfer. How this works will be decided by which form of reliability the user wants to use. More detail on how the server works differently depending on reliability method will be given in section 5.

### 4.6 Client

To initiate the program in the client mode, we use '-c'. The client is responsible for sending the image/data to the server. As stated in the previous section, the client and the server are considered two different machines, even though they are in the same file. This makes it possible to transfer a file from the client to the server even though they are in the same directory. Since they are in the same directory though, we must make sure that we change the name of the file when it is sent.

### 4.7 Filename

'-f' is the flag to choose the filename. This has to neat implications; you can set the name of the received file, and you can choose which file you want to send through the same flag.

When using the '-f' flag on the server side, we specify the name of the file that we create before we start to receive data. This makes us able to choose the name, and the type of file we want to make. Since we are transferring an image file in this program, we usually end the name with .jpg, but if you want to transfer a text file later, you can simply end the name of the file with .txt instead. By being able to choose the name of the file we can make sure that the sent file doesn't get overwritten. Since everything is located in the same directory, we would overwrite the file we wanted to send if we named the received file the same as the sent file.

On the client side, the '-f' flag is simply used to choose which file you want to send. In this application, we have three pictures, 'test\_cat.jpg', 'test\_dog.jpg', and 'highRes.jpg'. The reason why we have the two 'test\_' pictures was to test and see if the '-f' flag worked properly on the client side. We also added the 'highRes.jpg' file to test the application with a larger file. This being said, to keep the transfers consistent in section 5, we always used the 'test\_cat.jpg' picture as our transfer file.

## 4.8 Reliability methods

There are three reliability methods in this program. Here we will explain how the '-r' flag works with these methods. See section 3 for more details on how each method works.

When choosing the '-r' flag you have three choices to choose from, one for each of the methods, stop-and-wait, GBN and SR. To call the stop and wait method you would write it like this in the command line: "-r StopAndWait". The '-r' flag is parsed through the argument parser with the 'choices' attribute to make the user able to choose from a tuple of arguments that is implemented in the code (being the three reliability methods). When the user chooses one of the arguments, the program will run the code associated with the reliability method chosen. This way, the program functions differently depending on the method chosen.

## 4.9 Test cases

To make sure that the three reliability protocols are working as expected, we added the common argument '-t' to add two test cases. These test cases are 'skip\_ack' and 'skip\_seq', which makes the program skip an acknowledgement packet and a sequence number packet respectively.

'skip\_ack' is the first test case and makes the server skip an acknowledgement packet and make the client retransmit the packet that didn't get acknowledged. This test case is only usable on the server, because the client is the one sending the data and is not sending acknowledgements. As an example, if the client sends packet '1, 2, 3, 4, 5' this test case will skip returning an acknowledgement for one of these packets and the client will resend the packet that wasn't acknowledged.

The 'skip\_seq' test case on the other hand, is a test case to check reliability on the client side. This test case makes the client skip a sequence number to test for in sequence delivery. If the server receives packets in sequence '1, 2, 3, 5', it should know that packet 4 wasn't delivered and should request the 4<sup>th</sup> packet, making the client retransmit this 4<sup>th</sup> packet. Depending on if you run this test with GBN or SR the server will either drop the packets received out of order, or buffer them until it has received the lost packets. GBN here will drop the packets, and SR will buffer the packets.

## 5 Discussions

In this section we discuss how application.py performed under different environments in the simple-topo.py topology. In the first set of test cases we needed to test the application with three different RTT values. For this we changed the 'delay' value in the topology to get the wanted RTT. As the link between our two hosts in the topology consists of host 1, router 2 and host 3, we have two links with delay values. RTT is the time a packet uses from one host to another and back again (one round trip), so the  $RTT = \text{delay} * 2$ . As we have two links, we need to divide the delay on 2 again and insert the same value in both the 'delay' attributes in the topology. This means that to get a RTT of 25, 50, and 100, we need to set each of the two 'delay' attributes in the links to 6.25, 12.5, and 25 respectively.

In Table 1 you can see our results for three ping tests we did to make sure the delay was correctly set for the tests in 5.1 – 5.4. As we can see, some of the numbers are a few ms higher than the expected delay. There could be multiple reasons for this, some examples are network congestion, hardware limitations, and/or network protocols, but as there are only a few ms more delay than expected, this is a good result.

Ping test	RTT 25 (Delay 6.25)	RTT 50 (Delay 12.5)	RTT 100 (Delay 25)
Average RTT	25.17 ms	52.634 ms	103.066 ms
RTT min	24.353 ms	50.719 ms	101.226 ms
RTT max	28.958 ms	55.595 ms	104.514 ms
RTT mdev	1.082 ms	1.313 ms	0.942 ms

Table 1: Ping values to ensure delay was set correctly.

### 5.1 Throughput values with different window sizes and RTT

#### 5.1.1 Stop and wait with different RTTs

In the first test case we used the stop and wait protocol with the three different RTT values of 25, 50 and 100. Our results can be seen in Table 2. As we can see from the results, the more delay/RTT, the slower is the transfer, as expected. We can also see that the time it took to finish the transfer was quite stable, as the time nearly doubled every time we doubled the RTT. As with all networks, transfer speeds do vary from transfer to transfer, but our results are clearly within logical reach.

Both the links in the topology have a 100 Mbps transfer speed, and the delay value we put for each of the tests. As 100 Mbps equals to 12.5 MB/s, and when we factor in the delay of 25 ms per packet, we can see that our result is as expected. An easy example to show this is our result with RTT of 100 ms.

- 100 ms equals to 0.1 second.
- The packets we send are 1460 bytes of data.
- The file we send is roughly 105 kB (107 329 bytes)
- The total number of packets sent is 73.

With this information we can calculate that to send 73 packets with a 0.1 second for each of the packets to arrive, we would need roughly 7.3 seconds to send every packet. This information tells us that the delay is our bottleneck. Since each packet needs to be acknowledged, the transfer takes a lot longer to finish than the bandwidth would let us to believe. Had there been no delay on the link, the transfer of 105 kB on a link with bandwidth of 12.5 MB/s wouldn't take more than a couple of ms. The same goes for all the other RTT values.



Stop and wait	RTT 25	RTT 50	RTT 100
Time	2.03 sec	3.93 sec	7.63 sec
Throughput	52.73 kB/s	27.28 kB/s	14.07 kB/s

Table 2: Results of Stop and Wait protocol with RTT of 25, 50, and 100.

### 5.1.2 Go-Back-N with different windows and RTTs

In this second test we use the Go-Back-N protocol to test the throughput, as with Stop and Wait in the last section. GBN uses windows to send X number of packets at a time, in our cases, 5, 10, and 15. In Table 3 – 5 you can see our results for these tests.

As expected, we can see the same trend as with Stop and Wait, being that the higher value for the RTT, the slower is the transfer. Interestingly, the transfer speed with the GBN protocol is way faster than the Stop and Wait protocol. From section 5.1.1 we found out that the bandwidth between h1 – h3 is 12.5 MB/s, and we can tell that there still is a bottleneck elsewhere as we haven't reached these numbers in any of these tests. If we look at our bottleneck from the last test, the delay, we know that the GBN protocol sends a window of packets at a time, then wait for a response from the server to move the window forward and send the next window of packets. This way the client can send 5, 10, and 15 packets respectively, to the server without waiting for an acknowledgement. This speeds up the process of sending packets with 5, 10 and 15 times from our Stop and Wait protocol.

We can calculate this by using our example from the Stop and Wait calculation. At RTT 100, it took 7.63 seconds to transfer the data. If we divide 7.63 by 15 to make it 15 times faster than the Stop and Wait protocol, we get 0.506 seconds. This fact is reflected in our 0.52 seconds of transfer time with the GBN protocol with window size of 15 and the same RTT value. This proves yet again that the delay between h1 and h3 is the bottleneck of this transfer. Another factor that adds a small amount of delay for the transfer with both GBN and SR are the prints. When sending and receiving acks and seq, the program will print this information for each packet, which adds the small amount of delay.

Go-Back-N, window = 5	RTT 25	RTT 50	RTT 100
Time	0.43 sec	0.78 sec	1.53 sec
Throughput	246.06 kB/s	136.46 kB/s	59.83 kB/s

Table 3: Results of Go-Back-N with window size of 5 and RTT of 25, 50 and 100.

Go-Back-N, window = 10	RTT 25	RTT 50	RTT 100
Time	0.24 sec	0.44 sec	0.87 sec
Throughput	433.66 kB/s	242.30 kB/s	123.06 kB/s

Table 4: Results of Go-Back-N with window size of 10 and RTT of 25, 50 and 100.

Go-Back-N, window = 15	RTT 25	RTT 50	RTT 100
Time	0.19 sec	0.26 sec	0.52 sec
Throughput	539.72 kB/s	387.69 kB/s	203.29 kB/s

Table 5: Results of Go-Back-N with window size of 15 and RTT of 25, 50 and 100.

### 5.1.3 Selective Repeat with different windows and RTTs

Lastly with the throughput tests we have the SR tests. These tests are the same as the GBN in the last section, as GBN and SR work in very similar ways. They both send packets in windows, and as such, we tested SR with 5, 10, and 15 window size, and with 25, 50, and 100 RTT for each of the window sizes.

From our results, shown in Table 6 – 8, we can see that these results are practically the same as with the GBN tests. Between the GBN and the SR tests, the max difference in value is 7 ms in transfer speeds. This difference is miniscule and is only happening because of minor differences between each transfer in a network. This is an expected result as these protocols are very similar. The only difference between these two protocols is that GBN will discard any out of order packets, while SR will buffer them. This makes SR better than GBN in congested networks with higher error rates, as the client won't have to resend packets that were received out of order.

Lastly, if we look at the throughput values of these transfers, we can tell that we are getting roughly 0.5 MB/s throughput with a window size of 15 and RTT of 25, which is the highest throughput we have gotten in any of these tests. We now know that to reach the expected 12.5 MB/s bandwidth, we need to eliminate our bottlenecks as much as possible. These being window sizes and delay. The higher the window size and with a lower delay, the faster the transfer will finish.

Selective repeat, window = 5	RTT 25	RTT 50	RTT 100
Time	0.41 sec	0.84 sec	1.60 sec
Throughput	260.98 kB/s	126.55 kB/s	66.85 kB/s

Table 6: Results of Selective Repeat with window size of 5 and RTT of 25, 50 and 100.

Selective repeat, window = 10	RTT 25	RTT 50	RTT 100
Time	0.25 sec	0.48 sec	0.83 sec
Throughput	423.41 kB/s	221.76 kB/s	128.14 kB/s

Table 7: Results of Selective Repeat with window size of 10 and RTT of 25, 50 and 100.

Selective repeat, window = 15	RTT 25	RTT 50	RTT 100
Time	0.18 sec	0.27 sec	0.57 sec
Throughput	591.18 kB/s	390.38 kB/s	186.12 kB/s

Table 8: Results of Selective Repeat with window size of 15 and RTT of 25, 50 and 100.

## 5.2 Skipping an ACK

In this test case we use the '-t' flag to test the reliability of our program. These tests are all done with the same default settings, being 5ms of delay (20ms of RTT), and window size of 5 for both GBN and SR. In our program, it is the client that sends data to the server. This means that it's the server that sends an acknowledgement packet back to the client for each packet received. Thus, the '-t' flag in this test case is only used on the server.

As shown in Figure 1, the server receives packets as normally, until it skips an acknowledgement packet. This causes the client to retransmit the packet when it didn't receive the acknowledgement. When the client receives the acknowledgement for the retransmitted packet, it continues to transmit the rest of the packets before it gracefully closes the connection.

If we compare this transmission to the one in section 5.1.1, we can tell that the throughput of the transfer is lower in this test than the earlier test. For this test we used a lower RTT value (20ms) than in the transmission in section 5.1.1 (25ms), yet we got a lower throughput of 35.85 kB/s to 52.73 kB/s, why is that? The answer is in the protocol. The stop and wait protocol do exactly that, it stops and waits for the acknowledgement to arrive. When the acknowledgement doesn't arrive, it waits a set amount of time before it retransmits the packet, wasting time and lowering the throughput value.

```

"Node: h1"
root@lubuntu:/home/lubuntu/Desktop/portfolio2# python3 application2.py -c -f t
est_cat.jpg -r StopAndWait
Received SYN-ACK
Resending packet!
-----
TIME: 2.99 second(s)
FILE SIZE: 107.33 kB
THROUGHPUT: 35.85 kB per second
Sending FIN
Final ACK received, closing the connection!
root@lubuntu:/home/lubuntu/Desktop/portfolio2#

"Node: h3"
root@lubuntu:/home/lubuntu/Desktop/portfolio2# python3 application2.py -s -f t
ransferred_file.jpg -r StopAndWait -t skip_ack
Connected
SYN received!
ACK received!
Skipping ACK
FIN received, sending last ACK!
root@lubuntu:/home/lubuntu/Desktop/portfolio2#

```

Figure 1: Stop and Wait with server skipping an ACK.

Next, we have the 'skip\_ack' test with GBN. From Figure 2 we can see how this protocol reacts to the test case. As always, the client connects to the server, and they exchange SYN-ACK to secure the connection. The transfer is normal until the server skips an acknowledgement. The client starts with sending the first 5 packets, specified by the window. When it doesn't receive the acknowledgement for the first packet, it retransmits all the packets in the first window. Then it receives all the acknowledgements for the first window and the transmission continues like normal. The time for this transmission was 0.85 seconds, and the throughput was 124.89 kBps.

If we compare this result to the one in section 5.1.2, we can see that our 'skip\_ack' transmission took almost exactly twice as long. As discussed earlier, GBN is not a good protocol in networks with high error rates, unlike SR. When the server skips the first acknowledgement in a window, the client must retransmit all the packets in that window, making it very inefficient. Even though the transmission took twice as long, its still under 1 second of transfer time, which again could be normal variations in transfer times.

```

"Node: h1"
TIME: 2.99 second(s)
FILE SIZE: 107.33 kB
THROUGHPUT: 35.85 kB per second
Sending FIN
Final ACK received, closing the connection!
root@lubuntu:/home/lubuntu/Desktop/portfolio2# python3 application2.py -c -f t
est_cat.jpg -r GoBackN
Received SYN-ACK
Sending seq=1
Sending seq=2
Sending seq=3
Sending seq=4
Sending seq=5
Received ack=2
Received ack=3
Received ack=4
Received ack=5
Resending packet!
Sending seq=5
Sending seq=6
Sending seq=7
Sending seq=8
Sending seq=9
Received ack=6

"Node: h3"
root@lubuntu:/home/lubuntu/Desktop/portfolio2# python3 application2.py -s -f t
ransferred_file.jpg -r GoBackN -t skip_ack
Connected
SYN received!
ACK received!
Skipping ACK
FIN received, sending last ACK!
root@lubuntu:/home/lubuntu/Desktop/portfolio2#

```

Figure 2: Go-Back-N with server skipping an ACK.

Lastly for this test case we have the SR protocol. Figure 3 shows us how the client retransmitted packet 5, which was skipped by the server, and how the server received the 5<sup>th</sup> packet again afterwards. The throughput of this transfer was a little lower than the GBN protocol, but this is negligible and is due to variations in transfer times.

```

"Node: h1"
Sending seq=58
Sending seq=59
Sending seq=60
Sending seq=61
Sending seq=62
Sending seq=63
Sending seq=64
Sending seq=65
Sending seq=66
Sending seq=67
Sending seq=68
Sending seq=69
Sending seq=70
Sending seq=71
Sending seq=72
Sending seq=73
Sending seq=74
-----
TIME: 1,04 second(s)
FILE SIZE: 107,33 kB
THROUGHPUT: 102,90 kB per second
Sending FIN
Final ACK received, closing the connection!
root@lubuntu:/home/lubuntu/Desktop/portfolio2#

"Node: h3"
Received seq=74
FIN received!
root@lubuntu:/home/lubuntu/Desktop/portfolio2# python3 application2.py -s -f t
ransfered_file.jpg -r SelectiveRepeat -t skip_ack
Connected
SYN received!
Received seq=1
Received seq=2
Received seq=3
Received seq=4
Received seq=5
Skipping ACK
Received seq=5
Received seq=6
Received seq=7
Received seq=8
Received seq=9
Received seq=10
Received seq=11
Received seq=12
Received seq=13
Received seq=14
Received seq=15
Received seq=16

```

Figure 3: Selective Repeat with server skipping an ACK.

Test the program by skipping an ack to trigger a retransmission. Test with all 3 reliability functions

### 5.3 Skipping a sequence number

This next test case is the 'skip\_seq' test. This test is used on the client to skip a sequence number to make sure that the server can handle multiple packets of the same sequence number and out-of-order delivery. This test is done with GBN and SR, and not the Stop and Wait protocol. When the client skips the sequence number with sequence number 3, as shown in Figure 4 and Figure 5, the server will handle the out-of-order delivery in different ways depending on the protocol it uses. As discussed earlier, the GBN method will discard all packets received out-of-order, while the SR method will buffer these packets and reorder them when receiving the older packets.

In the GBN example in Figure 4 we can see that the client skipped the 3<sup>rd</sup> packet. On the server side, the server states that it received the 4<sup>th</sup> packet but expected the 3<sup>rd</sup> packet. Following, the server received the 5<sup>th</sup> packet but was expected to receive the 3<sup>rd</sup> packet again. Since the client sends packets in a window of 5 packets before receiving acknowledgements, this makes the server receive 2 packets that gets discarded. Then the client resends the 3<sup>rd</sup>, 4<sup>th</sup>, and 5<sup>th</sup> packet again, and the transfer continues as normal. The time for this transfer was 0.93 seconds, and the throughput was 115.16 kbps, quite like the 'skip\_ack' test case for GBN.

```

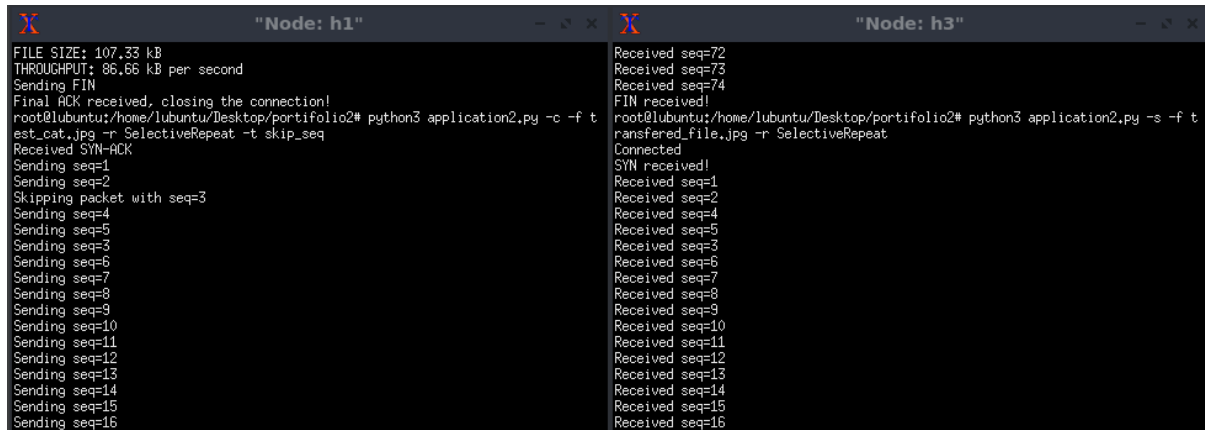
"Node: h1"
Sending FIN
Final ACK received, closing the connection!
root@lubuntu:/home/lubuntu/Desktop/portfolio2# python3 application2.py -s -f t
est_cat.jpg -r GoBackN -t skip_seq
Received SYN-ACK
Sending seq=1
Sending seq=2
Skipping packet with seq=3
Sending seq=4
Sending seq=5
Received ack=2
Received ack=3
Resending packet!
Sending seq=3
Sending seq=4
Sending seq=5
Sending seq=6
Sending seq=7
Received ack=4
Received ack=5
Received ack=6
Received ack=7
Received ack=8
Sending seq=8

"Node: h3"
Received seq=61
Received seq=62
Received seq=63
Received seq=64
Received seq=65
Received seq=66
Received seq=67
Received seq=68
Received seq=69
Received seq=70
Received seq=71
Received seq=72
Received seq=73
Received seq=74
FIN received!
root@lubuntu:/home/lubuntu/Desktop/portfolio2# python3 application2.py -s -f t
ransfered_file.jpg -r GoBackN
Connected
SYN received!
ACK received!
Received 4, expected 3
Received 5, expected 3
FIN received, sending last ACK!
root@lubuntu:/home/lubuntu/Desktop/portfolio2#

```

Figure 4: Go-Back-N with client skipping a SEQ\_NUM.

For the last test with 'skip\_seq', we use the SR protocol. Similar to the same test case with GBN, we can look at Figure 5 to see how the server handles out-of-order delivery. Unlike the GBN protocol, SR keeps the out-of-order delivered packet and reorders them when the window is received, instead of discarding them. This makes SR a better choice for networks that have a high error rate, as the client don't have to send this out-of-order packet multiple times. The time for this transfer was 0.90 seconds, and the throughput was 118.00 kBps, pretty much the same as the 'skip\_seq' test with GBN.



```

Node: h1
FILE SIZE: 107.33 kB
THROUGHPUT: 86.66 kB per second
Sending FIN
Final ACK received, closing the connection!
root@ubuntu:/home/lubuntu/Desktop/portfolio2# python3 application2.py -c f t
est_cat.jpg -r SelectiveRepeat -t skip_seq
Received SYN-ACK
Sending seq=1
Sending seq=2
Skipping packet with seq=3
Sending seq=4
Sending seq=5
Sending seq=3
Sending seq=6
Sending seq=7
Sending seq=8
Sending seq=9
Sending seq=10
Sending seq=11
Sending seq=12
Sending seq=13
Sending seq=14
Sending seq=15
Sending seq=16

Node: h3
Received seq=72
Received seq=73
Received seq=74
FIN received!
root@ubuntu:/home/lubuntu/Desktop/portfolio2# python3 application2.py -s -f t
ransferred_file.jpg -r SelectiveRepeat
Connected
SYN received!
Received seq=1
Received seq=2
Received seq=4
Received seq=5
Received seq=3
Received seq=6
Received seq=7
Received seq=8
Received seq=9
Received seq=10
Received seq=11
Received seq=12
Received seq=13
Received seq=14
Received seq=15
Received seq=16
```

Figure 5: Selective Repeat with client skipping a SEQ\_NUM.

## 5.4 Handling of losses, reordering and duplicates

By using our test cases and the different protocols we can see that even though they can't all handle losses, reordering and duplications, together, they cover all of this.

All three of our protocols handle losses. This is the principle of reliability. When a packet is lost, this being a data packet or an acknowledgement packet, the client will always retransmit the lost packet, either after a timeout, or after an acknowledgement for a later packet if the server received packets after the lost packet, like in GBN and SR.

When it comes to reordering, this is shown by the 'skip\_seq' test case. When the client skips a sequence number, the SR protocol will buffer the packets after the lost one and reorder them (on the server side) when the window is received. GBN on the other hand, does not handle reordering as this protocol just discards the packets instead.

Lastly, we can tell that the server handles duplication of packets with the GBN and SR protocols when we use the 'skip\_ack' test case. With GBN for example, the server will receive packet 1, 2, 4, 5, with the 3<sup>rd</sup> packet skipped. The server will then discard the 4<sup>th</sup> and 5<sup>th</sup> packet, before receiving them once again in the next transmission from the client. This means that the GBN handles these duplications by discarding the first instance of receiving them. SR will handle duplications if the acknowledgement packet for data packet is not received by the client, as the client will then retransmit the packet after a timeout.

## 6 Conclusions

To conclude this report, we have made, tested, and discussed how to build reliability on top of an already existing transport protocol, being UDP. Being able to make reliability is important for file transfers, as we always want to send and receive data reliably. We have discussed our findings after testing our application in a premade topology in Mininet and found that our results are within expectations of reality.

## 7 References

KUROSE, James F., ROSS, Keith W.. (2022). Computer Networking: a top-down approach, 8th edition (8th Edition). USA: Pearson.

*struct—Interpret bytes as packed binary data.* (2023). Python Documentation. Retrieved 2023, from <https://docs.python.org/3/library/struct.html>