



16. MAY 2023

PORTFOLIO 2  
DATA2410

236 – 242 - 131

OSLOMET

Ellen Gleditschs hus, Pilestredet 35, 0166 Oslo



## Table of Contents

Introduction.....	2
Acronyms.....	2
Background.....	2
UDP – User Datagram Protocol.....	2
Stop-n-wait .....	3
Go-Back-N.....	3
Selective repeat.....	3
Go-Back-N and Selective repeat.....	4
Implementation.....	4
Task 1 - Throughput with varying window sizes and RTTs .....	5
Stop-n-wait .....	5
Results.....	5
Discussion.....	6
GO BACK N .....	6
RESULTS.....	7
DISCUSSION .....	8
GO BACK N WITH SELECTIVE REPEAT.....	8
RESULTS.....	8
DISCUSSION .....	10
Task 2 – Test case (Packet loss).....	10
Stop-and-wait.....	10
Go-Back-N .....	11
Go-Back-N-SR.....	11
Task 3 – Test case (out-of-order delivery).....	12
Go-back-N.....	12
Go-Back-N-SR.....	12
Task 4 – Test case (efficiency).....	12
Stop-and-wait.....	12
Results.....	12
Discussion.....	13
Go-back-N.....	13
Results.....	13
Discussion.....	13

Go-back-N-SR .....	14
Conclusion .....	14
References .....	14

## Introduction

Reliable data transfer is an essential piece of file transferring applications, allowing safe and efficient transmissions of data over less reliable networks. In this report, we will explore the implementation of reliable protocols for a simple file transferring application over UDP. To overcome the shortcomings of the connectionless transport protocol, we will implement a range of protocols such as Stop-and-wait, Go-back-N, and Selective-Repeat. We will also undergo the implementation and evaluation of these protocols, with the aim to provide insights into their effectiveness in addressing the challenges of transmitting data over unreliable networks.

## Acronyms

Here you will find a list of the most frequently used acronyms in this assignment:

D RTP = DATA2410 Reliable Transport Protocol

UDP = User Datagram Protocol

TCP = Transmission Control Protocol

ACK = Acknowledgement

RTT = Round-trip time

SR = Selective repeat

GBN = Go-back-n

GBN-SR = Go-back-n and selective repeat

## Background

Before heading into the implementation details of this assignment, we would like to give you a brief explanation of some of the most important terms that will be used continuously in this portfolio.

### UDP – User Datagram Protocol

A user datagram protocol is a transport layer protocol that is frequently used in computer networks. UDP is a connectionless protocol, this means that it does not establish a specific connection before transferring

the data. This type of connection is mostly used in applications that need to transfer data as quickly as possible, for example online games and real-time communication. The downside with this protocol is known to be unreliable compared to TCP.

### Stop-n-wait

The term, stop-n-wait, is a control mechanism that often gets implemented on top of UDP to provide a reliable data transmission. Stop-n-wait like this; the sender transmits a single data packet and waits for acknowledgment from the receiver before it sends the next packet. Then the receiver sends an acknowledgment back to the sender to confirm that the reception was successful. If the sender does not receive the acknowledgment from the receiver within a defined time period, it assumes that the packet was either lost or damaged and then resends the packet.

### Go-Back-N

Similar to the stop-n-wait, go-back-n is used with the UDP for reliable data transmission. It provides a way for the sender to transmit multiple packets without waiting for the acknowledgements from the receiver. This protocol will help improve the efficiency of the transmissions. Below you will find a brief, but precise two-step description of how go-back-n works.

1. The sender divides the data into packets and starts sending them one by one, without waiting for acknowledgment from the sender
2. The receiver receives the packets and checks them for errors. If everything checks out, the receiver sends an acknowledgment back to the sender. If there is a packet that is damaged or missing, the receiver sends a negative acknowledgment.

### Selective repeat

Selective repeat is a technique that is most commonly used in data communication protocols to provide reliability. When using selective repeat, you allow the receiver to request only the damaged or missing packets, instead of the whole set of packets. Below you will find a brief, but precise description of how selective repeat works.

1. The sender divides the data into packets and starts sending them one by one, without waiting for acknowledgment from the sender.
2. The receiver receives the packets out of order due to the nature of the protocol.
3. The receiver maintains a buffer to store the packets until all of the packets have arrived. When storing the packets in the buffer, the receiver is able to reorder the packets correctly once everything has arrived.
4. The receiver receives the packets and checks them for errors. If everything checks out, the receiver sends an acknowledgment back to the sender. If there is a packet that is damaged or missing, the receiver does not send an acknowledgment.
5. When the sender receives the acknowledgment, it knows which packets were received and which never made it to the receiver. The sender then retransmits the lost/damaged packets.

## Go-Back-N and Selective repeat

GBN-SR is often referred to as a hybrid protocol because it combines all features of the GBN protocol and the SR protocol. By combining the features of the GBN protocol and the SR protocol GBN-SR reduces the number of unnecessary retransmissions and optimizes the efficiency of the transfers.

1. The sender maintains a window of packets. These packets can then be transmitted without waiting for acknowledgements from the receiver.
2. The receiver sends cumulative acknowledgements indicating the highest consecutive packet it has received without any damage. (Similar to GBN)
  - a. In addition to this, the receiver also sends individual acknowledgements for each packet. (Similar to RN)
  - b. Allows the sender to identify the specific packets that are missing or damaged.
3. The sender retransmits the missing and/or damaged packets based on either the cumulative acknowledgment or the individual acknowledgement from the receiver.
4. If the receiver receives a duplicate, it discards it. This is known as packet discard and prevents unnecessary retransmission.

## Implementation

The python file named 'dtrp.py' can be invoked as either a client (sender) or a server (receiver), allowing the user to transfer a file through few and simple command-line arguments. The tool also allows users to measure throughput (packet/sec) by sending any file between the client and server using any of the three implemented transferring protocols.

To counter the connectionless protocol 'UDP', we have implemented headers as additional data at the beginning of each packet or segment. The headers provide information about the data being transmitted, and certain other fields contain information for the initialization of a new connection or the termination of a connection. Here is an illustration of the fields in the header.

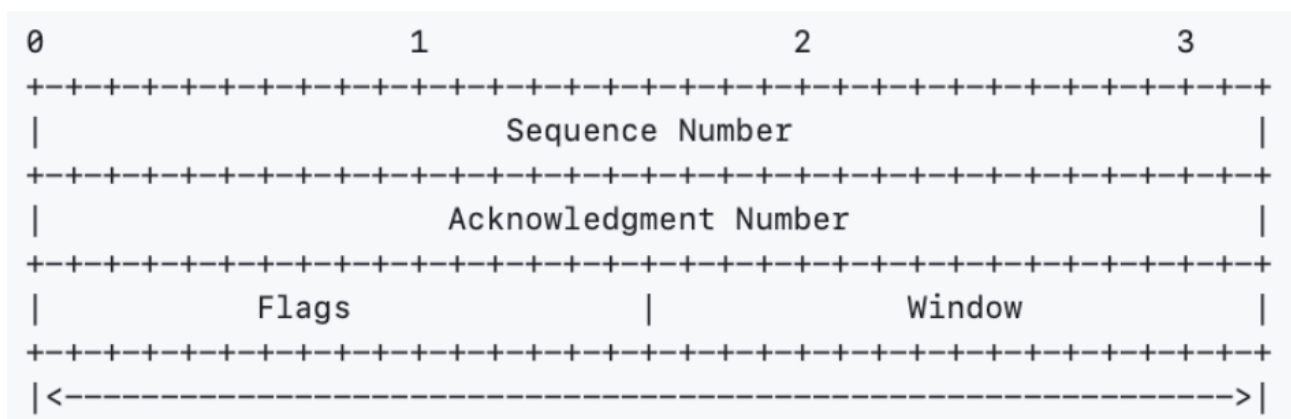


Figure 1 Header (Islam, Canvas)

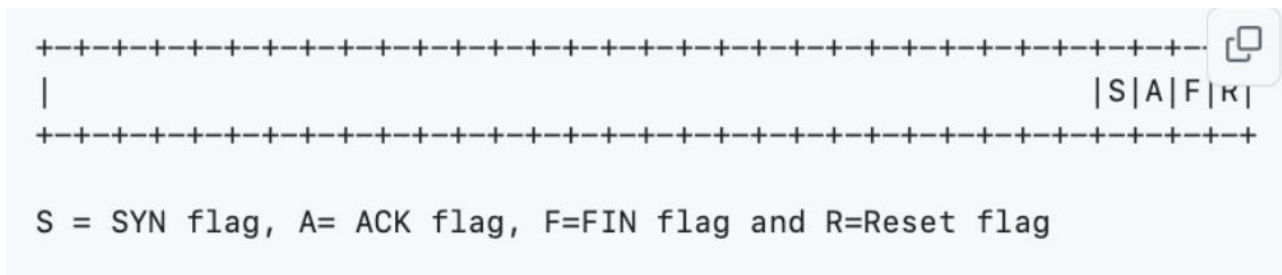


Figure 2 Flags (Islam, Canvas)

Once the input parameters are provided and the application is ran, dtrp.py will automatically implement the reliable data transfer protocol requested, attempt to establish a connection with requested client, then ensuring a reliable and accurate transfer over the UDP connection.

## Task 1 - Throughput with varying window sizes and RTTs

### Stop-n-wait

#### Results

Server – python3 dtrp.py -s -f image.jpg -r saw

Client – python3 dtrp.py -c -f img.jpg -r saw

```

"Node: h1"
('10.0.1.2', 53226) +[PACKET #126]
('10.0.1.2', 53226) <-[ACK]
('10.0.1.2', 53226) +[PACKET #127]
('10.0.1.2', 53226) <-[ACK]
('10.0.1.2', 53226) +[PACKET #128]
('10.0.1.2', 53226) <-[ACK]
('10.0.1.2', 53226) +[PACKET #129]
('10.0.1.2', 53226) <-[ACK]
('10.0.1.2', 53226) +[PACKET #130]
('10.0.1.2', 53226) <-[ACK]
('10.0.1.2', 53226) +[PACKET #131]
('10.0.1.2', 53226) <-[ACK]
('10.0.1.2', 53226) +[PACKET #132]
('10.0.1.2', 53226) <-[ACK]
('10.0.1.2', 53226) +[PACKET #133]
('10.0.1.2', 53226) <-[ACK]
('10.0.1.2', 53226) +[PACKET #134]
('10.0.1.2', 53226) <-[ACK]
('10.0.1.2', 53226) +[PACKET #135]
('10.0.1.2', 53226) <-[ACK]
('10.0.1.2', 53226) +[PACKET #136]
('10.0.1.2', 53226) <-[ACK]
('10.0.1.2', 53226) +[FIN]
Receiver throughput (Send and wait): 32.82693160548999 packets/s
('10.0.1.2', 53226) <-[ACK]
Client ('10.0.1.2', 53226) has disconnected

"Node: h3"
10.0.0.1 <-[PACKET #126]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #127]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #128]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #129]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #130]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #131]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #132]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #133]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #134]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #135]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #136]
10.0.0.1 +[ACK]
Sender throughput (Send and wait): 32.83485045783125 packets/s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...

```

Figure 3 RTT = 25 ms, Throughput = 32.83 packets/s

```

"Node: h1"
('10.0.1.2', 37548) <-[ACK]
('10.0.1.2', 37548) +[PACKET #128]
('10.0.1.2', 37548) <-[ACK]
('10.0.1.2', 37548) +[PACKET #129]
('10.0.1.2', 37548) <-[ACK]
('10.0.1.2', 37548) +[PACKET #130]
('10.0.1.2', 37548) <-[ACK]
('10.0.1.2', 37548) +[PACKET #131]
('10.0.1.2', 37548) <-[ACK]
('10.0.1.2', 37548) +[PACKET #132]
('10.0.1.2', 37548) <-[ACK]
('10.0.1.2', 37548) +[PACKET #133]
('10.0.1.2', 37548) <-[ACK]
('10.0.1.2', 37548) +[PACKET #134]
('10.0.1.2', 37548) <-[ACK]
('10.0.1.2', 37548) +[PACKET #135]
('10.0.1.2', 37548) <-[ACK]
('10.0.1.2', 37548) +[PACKET #136]
('10.0.1.2', 37548) <-[ACK]
('10.0.1.2', 37548) +[FIN]
Receiver throughput (Send and wait): 17.428582425595373 packets/s
('10.0.1.2', 37548) <-[ACK]
Client ('10.0.1.2', 37548) has disconnected
root@leon-VirtualBox:/home/leon/Desktop/Share#

"Node: h3"
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #128]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #129]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #130]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #131]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #132]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #133]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #134]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #135]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #136]
10.0.0.1 +[ACK]
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 4 RTT = 50ms, Throughput = 17.43 packets/s

```

"Node: h1"
('10.0.1.2', 45682) <-[ACK]
('10.0.1.2', 45682) +[PACKET #128]
('10.0.1.2', 45682) <-[ACK]
('10.0.1.2', 45682) +[PACKET #129]
('10.0.1.2', 45682) <-[ACK]
('10.0.1.2', 45682) +[PACKET #130]
('10.0.1.2', 45682) <-[ACK]
('10.0.1.2', 45682) +[PACKET #131]
('10.0.1.2', 45682) <-[ACK]
('10.0.1.2', 45682) +[PACKET #132]
('10.0.1.2', 45682) <-[ACK]
('10.0.1.2', 45682) +[PACKET #133]
('10.0.1.2', 45682) <-[ACK]
('10.0.1.2', 45682) +[PACKET #134]
('10.0.1.2', 45682) <-[ACK]
('10.0.1.2', 45682) +[PACKET #135]
('10.0.1.2', 45682) <-[ACK]
('10.0.1.2', 45682) +[PACKET #136]
('10.0.1.2', 45682) <-[ACK]
('10.0.1.2', 45682) +[FIN]
Receiver throughput (Send and wait): 9.160131947938115 packets/s
('10.0.1.2', 45682) <-[ACK]
Client ('10.0.1.2', 45682) has disconnected
root@leon-VirtualBox:/home/leon/Desktop/Share#

"Node: h3"
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #128]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #129]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #130]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #131]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #132]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #133]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #134]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #135]
10.0.0.1 +[ACK]
10.0.0.1 <-[PACKET #136]
10.0.0.1 +[ACK]
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 5 RTT = 100ms, Throughput = 9.16 packets/s

## Discussion

The time it takes for the sender to receive an ACK after sending a packet depends on the RTT. The higher the value of RTT, the longer it takes for packets to reach the recipient and vice versa.

When using the stop-and-wait protocol, the sender cannot send a new packet until it receives an ACK for the previous packet sent. Therefore, increasing transmission delay and reducing the throughput.

If a packet were to be lost, the sender would have to wait for a timeout before retransmitting. The timeout period in our case varies by the time it takes for a packet to be sent and an ACK to be received multiplied by 4. This results in an even longer time-out period, leading to longer retransmission times and longer delays.

Through testing, we can tell that each time the RTT is multiplied by 2, the throughput is approximately halved. This is as mentioned, due to longer RTT leading to longer delays between transmissions, and in some cases longer retransmission times.

In summary, RTT is an important parameter that affects both network performance and especially the stop-and-wait protocol. A longer RTT leads to longer delays, longer retransmission times and reduced throughput.

[GO BACK N](#)

## RESULTS

```
('10.0.1.2', 57100) <[DUPACK #134]
Receiver throughput (Go back N): 25.81272470857927 packets/s
('10.0.1.2', 57100) +[FIN]
('10.0.1.2', 57100) <[ACK]
Client ('10.0.1.2', 57100) has disconnected
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -s -f image.jpg -r gbn
10.0.0.1 +[ACK]
Sender throughput (Go back N): 20.911413867075073 packets/s
10.0.0.1 <[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -c -f img.jpg -r gbn -w 5
```

Figure 6 RTT = 25ms, WIN = 5, Throughput = 20.91 packets/s

```
Receiver throughput (Go back N): 33.34948091230449 packets/s
('10.0.1.2', 33110) +[FIN]
('10.0.1.2', 33110) <[ACK]
Client ('10.0.1.2', 33110) has disconnected
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -s -f image.jpg -r gbn
Sender throughput (Go back N): 29.077838431491394 packets/s
10.0.0.1 <[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -c -f img.jpg -r gbn -w 10
```

Figure 7 RTT = 25ms, WIN = 10, Throughput = 29.08 packets/s

```
('10.0.1.2', 55006) <[DUPACK #124]
Receiver throughput (Go back N): 40.22214412695257 packets/s
('10.0.1.2', 55006) +[FIN]
('10.0.1.2', 55006) <[ACK]
Client ('10.0.1.2', 55006) has disconnected
root@leon-VirtualBox:/home/leon/Desktop/Share# 
Sender throughput (Go back N): 36.31654303368321 packets/s
10.0.0.1 <[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -c -f img.jpg -r gbn -w 15
```

Figure 8 RTT = 25ms, WIN = 15, Throughput = 36.32 packets/s

```
Receiver throughput (Go back N): 14.081658889102671 packets/s
('10.0.1.2', 60718) +[FIN]
('10.0.1.2', 60718) <[ACK]
Client ('10.0.1.2', 60718) has disconnected
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -s -f image.jpg -r gbn
Sender throughput (Go back N): 11.396829080297852 packets/s
10.0.0.1 <[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -c -f img.jpg -r gbn -w 5
```

Figure 9 RTT = 50ms, WIN = 5, Throughput = 11.40 packets/s

```
Receiver throughput (Go back N): 20.599749378573453 packets/s
('10.0.1.2', 54936) +[FIN]
('10.0.1.2', 54936) <[ACK]
Client ('10.0.1.2', 54936) has disconnected
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -s -f image.jpg -r gbn
Sender throughput (Go back N): 17.829505572750197 packets/s
10.0.0.1 <[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -c -f img.jpg -r gbn -w 10
```

Figure 10 RTT = 50ms, WIN = 10, Throughput = 17.83 packets/s

```
Receiver throughput (Go back N): 26.754107641676637 packets/s
('10.0.1.2', 55034) +[FIN]
('10.0.1.2', 55034) <[ACK]
Client ('10.0.1.2', 55034) has disconnected
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -s -f image.jpg -r gbn
Sender throughput (Go back N): 24.190574736758613 packets/s
10.0.0.1 <[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -c -f img.jpg -r gbn -w 15
```

Figure 11 RTT = 50ms, WIN = 15, Throughput = 24.20 packets/s

```
Receiver throughput (Go back N): 8.204548600692615 packets/s
('10.0.1.2', 33498) +[FIN]
('10.0.1.2', 33498) <[ACK]
Client ('10.0.1.2', 33498) has disconnected
root@leon-VirtualBox:/home/leon/Desktop/Share# 
Sender throughput (Go back N): 6.686663680832723 packets/s
10.0.0.1 <[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#
```

Figure 12 RTT = 100ms, WIN = 5, Throughput = 6.69 packets/s



```

Receiver throughput (Go back N): 11.172069973303488 packets/s
('10.0.1.2', 53282) +[FIN]
('10.0.1.2', 53282) <-[ACK]
Client ('10.0.1.2', 53282) has disconnected
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -s -f image.jpg
-r gbn

Sender throughput (Go back N): 9.83825204608406 packets/s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -c -f img.jpg -r
gbn -w 10

```

Figure 13 RTT = 100ms, WIN = 10, Throughput = 9.83 packets/s

```

Receiver throughput (Go back N): 15.359102566702417 packets/s
('10.0.1.2', 57626) +[FIN]
('10.0.1.2', 57626) <-[ACK]
Client ('10.0.1.2', 57626) has disconnected
root@leon-VirtualBox:/home/leon/Desktop/Share#

Sender throughput (Go back N): 13.915808522557715 packets/s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 14 RTT = 100ms, WIN = 15, Throughput = 13.92 packets/s

## DISCUSSION

RTT and window size are important parameters that affect the performance of the protocol 'Go-back-N'. The effect of increasing RTT is the same as above, a doubled increase in RTT results in an approx. halving of the throughput. Whilst an increase in window size provides a larger number of packets to be transmitted, thus lowering the latency, and increasing the throughput.

An important thing to notice, is that on the system that was used to test the application, we had some issues with packets loss and loss of acks, affecting the throughputs all throughout testing. If we look pass the issues, we can tell that by each time we doubled the number of packets transmitted (increased window size), the throughput nearly doubled as well.

The relationship between RTT and window size is important because a longer RTT results into longer delays before receiving ACKs and can also cause the sender to transmit more packets than receivable, resulting to more retransmissions and lower throughput. For that reason, a reasonably larger window size may compensate for the longer RTT, maintaining optimal performance.

However, if the window size is unreasonable or too large relative to the RTT, the chance of packet loss increases, leading into more retransmissions and longer delays. A reasonable window size for certain RTTs depends thus on network conditions, size of packets and the reliability of the network.

In summary, RTT and window size are important parameters that affect network performance and the Go-Back-N protocol's performance. A longer RTT may require a bigger window size, but if the window size is too large compared to the RTT, the chance of packet loss' increases and the throughput decreases.

## GO BACK N WITH SELECTIVE REPEAT

## RESULTS

```

('10.0.1.2', 44810) <-[ACK]
Client ('10.0.1.2', 44810) has disconnected
Receiver throughput (Go back N): 32.64679208227029 packets/s
('10.0.1.2', 44810) +[FIN]
root@leon-VirtualBox:/home/leon/Desktop/Share# python3 dtrp.py -s -f image.jpg
-r gbn-sr

Sender throughput (Go back N with Selective Repeat): 32.653930711139985 packets/
s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 15 RTT = 25ms, WIN = 5, Throughput = 32.65 packets/s

```

('10.0.1.2', 56904) <-[ACK]
('10.0.1.2', 56904) <-[ACK]
Client ('10.0.1.2', 56904) has disconnected
Receiver throughput (Go back N): 33.226464963208414 packets/s
('10.0.1.2', 56904) +[FIN]
root@leon-VirtualBox:/home/leon/Desktop/Share#
Sender throughput (Go back N with Selective Repeat): 33.228760824425784 packets/s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 16 RTT = 25ms, WIN = 10, Throughput = 33.23 packets/s

```

('10.0.1.2', 57992) <-[ACK]
Client ('10.0.1.2', 57992) has disconnected
Receiver throughput (Go back N): 32.49094135353787 packets/s
('10.0.1.2', 57992) +[FIN]
root@leon-VirtualBox:/home/leon/Desktop/Share#
Sender throughput (Go back N with Selective Repeat): 32.55310054220989 packets/s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 17 RTT = 25ms, WIN = 15, Throughput = 32.55 packets/s

```

('10.0.1.2', 57264) <-[ACK]
('10.0.1.2', 57264) <-[ACK]
Client ('10.0.1.2', 57264) has disconnected
Receiver throughput (Go back N): 17.338140011887656 packets/s
('10.0.1.2', 57264) +[FIN]
root@leon-VirtualBox:/home/leon/Desktop/Share#
Sender throughput (Go back N with Selective Repeat): 17.332331846173517 packets/s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 18 RTT = 50ms, WIN = 5, Throughput = 17.33 packets/s

```

('10.0.1.2', 37542) <-[ACK]
Client ('10.0.1.2', 37542) has disconnected
Receiver throughput (Go back N): 17.39895034565076 packets/s
('10.0.1.2', 37542) +[FIN]
root@leon-VirtualBox:/home/leon/Desktop/Share#
Sender throughput (Go back N with Selective Repeat): 17.39126791039602 packets/s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 19 RTT = 50ms, WIN = 10, Throughput = 17.39 packets/s

```

('10.0.1.2', 36512) <-[ACK]
Client ('10.0.1.2', 36512) has disconnected
Receiver throughput (Go back N): 17.354093950813123 packets/s
('10.0.1.2', 36512) +[FIN]
root@leon-VirtualBox:/home/leon/Desktop/Share#
Sender throughput (Go back N with Selective Repeat): 17.35076756549525 packets/s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 20 RTT = 50ms, WIN = 15, Throughput = 17.35 packets/s

```

('10.0.1.2', 34630) <-[ACK]
('10.0.1.2', 34630) <-[ACK]
Client ('10.0.1.2', 34630) has disconnected
Receiver throughput (Go back N): 8.944955364782375 packets/s
('10.0.1.2', 34630) +[FIN]
root@leon-VirtualBox:/home/leon/Desktop/Share#
Sender throughput (Go back N with Selective Repeat): 8.952063515897933 packets/s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 21 RTT = 100ms, WIN = 5, Throughput = 8.95 packets/s

```

('10.0.1.2', 60862) <-[ACK]
('10.0.1.2', 60862) <-[ACK]
Client ('10.0.1.2', 60862) has disconnected
Receiver throughput (Go back N): 9.074118150548243 packets/s
('10.0.1.2', 60862) +[FIN]
root@leon-VirtualBox:/home/leon/Desktop/Share#
Sender throughput (Go back N with Selective Repeat): 9.072199571888818 packets/s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 22 RTT = 100ms, WIN = 10, Throughput = 9.07 packets/s

```

('10.0.1.2', 43786) <-[ACK]
Client ('10.0.1.2', 43786) has disconnected
Receiver throughput (Go back N): 9.08642759942957 packets/s
('10.0.1.2', 43786) +[FIN]
root@leon-VirtualBox:/home/leon/Desktop/Share#
Sender throughput (Go back N with Selective Repeat): 9.084000386084714 packets/s
10.0.0.1 <-[FIN]
10.0.0.1 +[ACK]
Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#

```

Figure 23 RTT = 100ms, WIN = 10, Throughput = 9.07 packets/s

## DISCUSSION

Large disclaimer regarding the Selective Repeat protocol or GBN-SR. Initially, SR() was working just fine, until we began testing it on the given topology. Issues like [FIN] being sent after the 20<sup>th</sup> packet (out of 136 packets), both server and client getting locked on a certain line of code, timeout not working and loss of acks and no timeout response.

We tried working on the code from scratch and finally got it working after a while. But the issue now is that it is clear as day wrong, and I have a huge suspicion that we've just recreated a stop-and-wait protocol. Nevertheless, we are continuing the discussion without results, but instead arguments and examples we have learnt so far.

By implementing the Selective-Repeat to the previous Go-Back-N, you will have a generally faster protocol when it comes to error recovery. In GBN(), when a packet is lost, all following packets in current sliding window are also discarded and retransmitted. On the other hand, SR() ensures that only the lost packets are retransmitted whilst the rest of the packets are normally processed, allowing for more efficient use of the available bandwidth.

To further optimize the protocol, one can allow SR() a smaller window size, thus reducing the probability of packet loss and retransmissions, resulting in faster transmission rate and throughput.

However, the actual performance of both protocols are heavily dependent on various network factors such as packet size, error rate and so on. Therefore, it is essential to evaluate the performance of each protocol in various network scenarios to further conclude their efficiency and relative speed.

## Task 2 – Test case (Packet loss)

See README.txt for how we used tc-netem to emulate test-cases

## Stop-and-wait

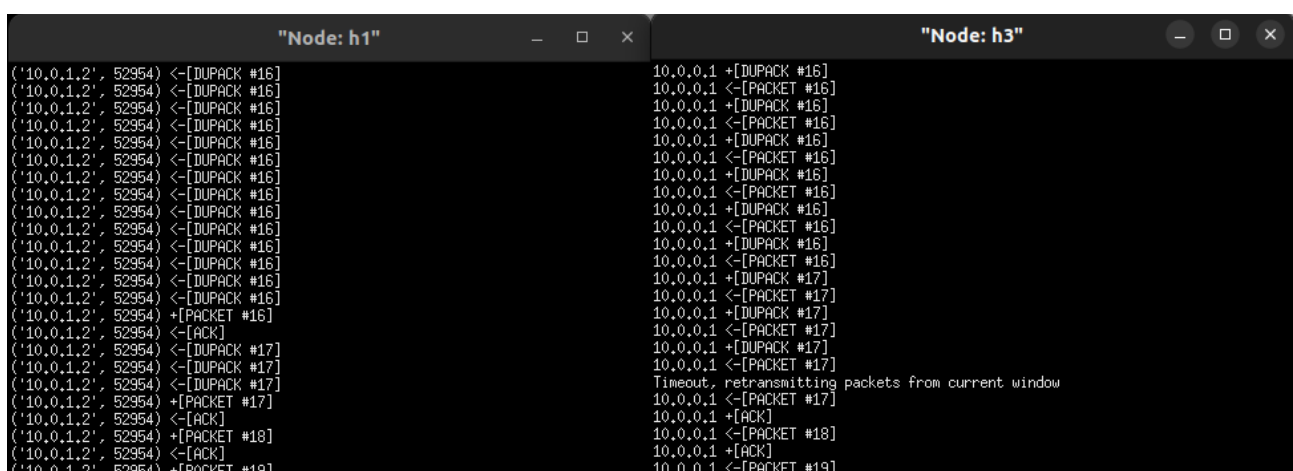


Figure 24 RTT = 25ms, Loss = 10%

## Go-Back-N

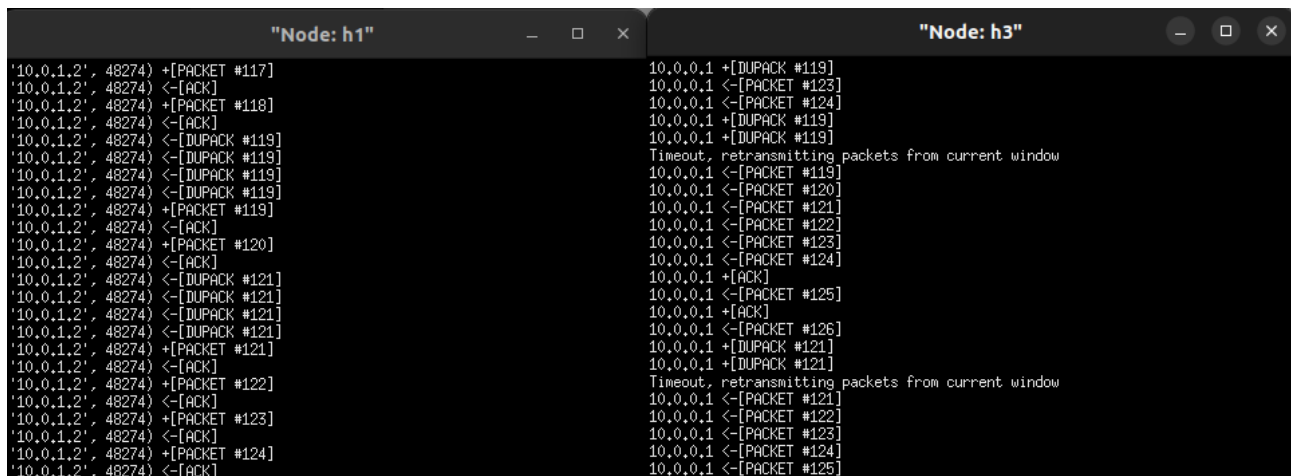


Figure 25 RTT = 25ms, Loss = 10%, Win = 5

## Go-Back-N-SR

The code for this protocol is incorrect and equal to stop-and-wait. Therefore, it will not be tested as it a false representation of how the actual protocol can perform.

Here is an example of what it would look like during a packet loss using Go-back-N with Selective-Repeat

Sender window = [0, 1, 2, 3, 4] 5, 6, 7, 0, 1, 2

Sender sends Packet #0

Receiver receives Packet #0 and sends ACK

Receiver window = 0 [1, 2, 3, 4, 5] 6, 7

Sender gets ACK for Packet #0

Sender window = 0 [1, 2, 3, 4, 5] 6, 7, 0, 1, 2

Sender sends Packet #1 which gets lost

Sender sends Packet #2

Receiver waits but only receives Packet #2 out of order and sends NAK #1 (or DUPACK #1)

Sender sends Packet #3

Sender gets NAK #1 and resends Packet #1

Receiver gets Packet #1 and sends ACK for packet #1, #2 and #3

Receiver window = 0, 1, 2, 3 [4, 5, 6, 7]

Sender gets ACK for packet #1, #2 and #3

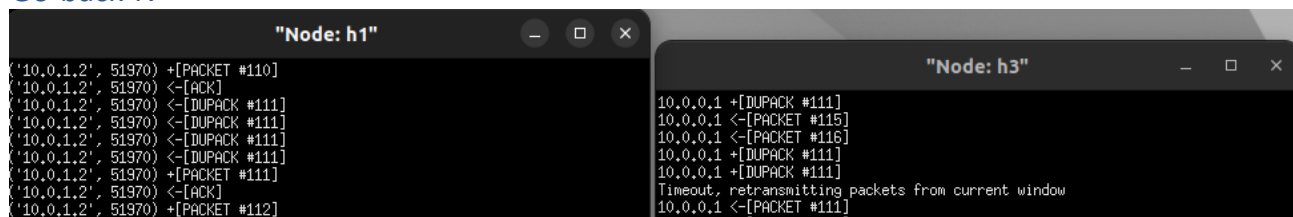
Sender window = 0, 1, 2, 3 [4, 5, 6, 7, 0] 1 2

.... etc

### Task 3 – Test case (out-of-order delivery)

See README.txt for how we used tc-netem to emulate test-cases

#### Go-back-N



```
"Node: h1"
10.0.0.1.2' 51970) +[PACKET #110]
10.0.0.1.2' 51970) <-[ACK]
10.0.0.1.2' 51970) <-[DUPACK #111]
10.0.0.1.2' 51970) <-[DUPACK #111]
10.0.0.1.2' 51970) <-[DUPACK #111]
10.0.0.1.2' 51970) <-[DUPACK #111]
10.0.0.1.2' 51970) <-[DUPACK #111]
10.0.0.1.2' 51970) +[PACKET #111]
10.0.0.1.2' 51970) <-[ACK]
10.0.0.1.2' 51970) +[PACKET #112]

"Node: h3"
10.0.0.1 +[DUPACK #111]
10.0.0.1 <-[PACKET #115]
10.0.0.1 <-[PACKET #116]
10.0.0.1 +[PACKET #111]
10.0.0.1 +[DUPACK #111]
Timeout, retransmitting packets from current window
10.0.0.1 <-[PACKET #111]
```

Figure 26 RTT = 25ms, Loss = 10%, Win = 5

With the increased Loss, we can tell that the sender got out of order and started sending PACKET #116 even after receiving DUPACK #111. The sender received no ACKS and was handled through timeout-exception by retransmitting the correct PACKET (#111).

#### Go-Back-N-SR

See last task under 'Go-Back-N-SR'.

### Task 4 – Test case (efficiency)

#### Stop-and-wait

#### Results

"Node: h1"	"Node: h3"
('10.0.1.2', 48746) +[PACKET #132]	Timeout, retransmitting packets from current window
('10.0.1.2', 48746) <-[ACK]	10.0.0.1 <-[PACKET #134]
('10.0.1.2', 48746) <-[DUPACK #133]	10.0.0.1 +[DUPACK #134]
('10.0.1.2', 48746) +[PACKET #133]	10.0.0.1 <-[PACKET #134]
('10.0.1.2', 48746) <-[ACK]	10.0.0.1 +[ACK]
('10.0.1.2', 48746) <-[DUPACK #134]	10.0.0.1 <-[PACKET #135]
('10.0.1.2', 48746) +[PACKET #134]	10.0.0.1 +[DUPACK #135]
('10.0.1.2', 48746) <-[ACK]	10.0.0.1 <-[PACKET #135]
('10.0.1.2', 48746) <-[DUPACK #135]	10.0.0.1 +[DUPACK #135]
('10.0.1.2', 48746) <-[DUPACK #135]	10.0.0.1 <-[PACKET #135]
('10.0.1.2', 48746) +[PACKET #135]	Timeout, retransmitting packets from current window
('10.0.1.2', 48746) <-[ACK]	10.0.0.1 <-[PACKET #135]
('10.0.1.2', 48746) <-[DUPACK #136]	Timeout, retransmitting packets from current window
('10.0.1.2', 48746) <-[DUPACK #136]	10.0.0.1 <-[PACKET #135]
('10.0.1.2', 48746) <-[DUPACK #136]	10.0.0.1 +[ACK]
('10.0.1.2', 48746) +[PACKET #136]	10.0.0.1 <-[PACKET #136]
('10.0.1.2', 48746) <-[ACK]	10.0.0.1 +[DUPACK #136]
('10.0.1.2', 48746) <-[DUPACK #137]	10.0.0.1 <-[PACKET #136]
('10.0.1.2', 48746) +[FIN]	10.0.0.1 +[ACK]
Receiver throughput (Send and wait): 20,86204115460348 packets/s	Sender throughput (Send and wait): 19,589993464438916 packets/s
('10.0.1.2', 48746) <-[ACK]	10.0.0.1 <-[FIN]
Client ('10.0.1.2', 48746) has disconnected	10.0.0.1 +[ACK]
root@leon-VirtualBox:/home/leon/Desktop/Share#	Closing...
	root@leon-VirtualBox:/home/leon/Desktop/Share#

Figure 27 RTT = 25ms, Loss = 5%

## Discussion

While the protocol is straightforward, it is not efficient in handling packet loss, duplicates, and loss of ACKs. The protocol can handle loss of an ACK by retransmitting the data packet until the ACK is received, though being quite inefficient in cases where the RTT is high and the time it takes increases. If a packet is lost, the same procedure applies – time out and retransmission of packet, inefficient for high-latency networks. Duplicates can be handled by ignoring them. However, if a duplicate is interpreted as a new packet, it can cause unnecessary retransmissions and increased latency.

Previously, with a RTT of 25ms and loss of 0%, we got a throughput of around 30 packets/sec. In this case, we got a throughput of around 20 packets/s with a loss of 5% instead.

In summary, Stop-and-wait can handle these issues, but not as efficiently as other protocols such as Go-back-N and Selective-Repeat.

## Go-back-N

### Results

Receiver throughput (Go back N): 23,257725579426918 packets/s	Sender throughput (Go back N): 18,768919019606184 packets/s
('10.0.1.2', 44258) +[FIN]	10.0.0.1 <-[FIN]
('10.0.1.2', 44258) <-[ACK]	10.0.0.1 +[ACK]
Client ('10.0.1.2', 44258) has disconnected	Closing...
root@leon-VirtualBox:/home/leon/Desktop/Share#	root@leon-VirtualBox:/home/leon/Desktop/Share#

Figure 28 RTT = 25ms, Win = 5, Loss = 5%

## Discussion

Go-back-N, like Stop-and-wait is also vulnerable to packet loss, loss of ACKs, and duplicates. In the case of packet loss, the sender retransmits all packets from the lost packet and waits for each of their corresponding ACKs. Resulting in poor performance, especially in networks with high loss. A loss of ACK(s) cannot be detected by the sender and may give the sender the idea that all packets up to that point have successfully been transmitted, which may result in duplicate packets. Duplicates, however are more

efficiently handled by the protocol, since the receiver maintains a window of packets to be expected, allowing duplicates to be simply ignored and instead transmit DUPACK to sender.

Previously, with a RTT of 25ms and loss of 0%, we got a throughput of around 20 packets/sec. In this case, we got a slight decrease resulting in a throughput of 18.7 packets/sec using a loss of 5% instead.

Overall, a reliable protocol that works better in networks with low loss and suffers from poor performance in high-loss networks.

## Go-back-N-SR

Not able to test. Same reason as before.

## Conclusion

Installing reliable data transfer protocols to an UDP-based file transferring application can provide enhanced performance and robustness. With the implementation of the protocols Stop-and-wait, Go-back-N, and Selective-Repeat, the application can ensure the user that the data transferred – is received accurately, even in the presence of errors and congestion. Furthermore, allowing the user to switch between protocols depending on network conditions can result into better performance, also giving the user a reliable and efficient means of transferring files over less reliable networks.

## References

### References

Islam, S. (n.d.). *Canvas*. Retrieved from portfolio-2-guidelines.pdf:

[https://oslomet.instructure.com/courses/25246/files/3189155?module\\_item\\_id=540504](https://oslomet.instructure.com/courses/25246/files/3189155?module_item_id=540504)

Islam, S. (n.d.). *Github*. Retrieved from header.py: <https://github.com/safiqul/2410/tree/main/header>

Lutkevich, B. (2021, October). *TechTarget*. Retrieved from Transmission Control Protocol (TCP):

<https://www.techtarget.com/searchnetworking/definition/TCP>

Rosencrance, L. (2021, October). *TechTarget*. Retrieved from User Datagram Protocol (UDP):

<https://www.techtarget.com/searchnetworking/definition/UDP-User-Datagram-Protocol>