




MAY 16, 2023

## PORTFOLIO 2

### DRTP IMPLEMENTATION

IBRAHIMA, YLLI, HARKI  
S362077, S364574, S355423  
Oslo Metropolitan University



1	INTRODUCTION	3
2	BACKGROUND	3
3	IMPLEMENTATION	6
4	DISCUSSION	9
5	CONCLUSIONS	11
6	REFERENCES	11

## 1 Introduction

### DATA2410 Reliable Transport Protocol (DRTP)

This group project involves implementing a simple transport protocol that contains an advancement of data delivery on top of the already existing UDP. User Datagram Protocol (UDP) is a transport layer protocol which is connectionless thus viewed as unreliable. The delivery of data packets is not guaranteed and UDP datagrams are sent without an acknowledging message because of it being connectionless and no proper connection between sender and receiver is established. Our aim is to ensure that our protocol delivers data smoothly and in sequence without missing data or producing/allowing duplicates. This will, in a few ways, be similar to the popular protocol, TCP. Our code will comprise of three reliable methods. The three reliable methods which will be implemented are: Stop and Wait, Go Back N(GBN) and selective repeat. The stop and wait protocol work as follows:

THE DRTP will use the foundation of a UDP with additional code for it to deliver data reliably. It would seemingly operate similar to how TCP operates. In this way, the file which will be transmitted from the server site to the client, would be received fully, in sequence and without any duplicated data.

Furthermore, the structure of the report will go as follows:

- Background theory about how we implement reliable methods into our DRTP.
- How the code is implemented and how the features work
- Discussions about the test cases where the code is put to use.
- And a conclusion that includes the summary of the project

## 2 Background

The DRTP implementation comprises of a few reliable methods that resembles features from TCP. The reason for this is that it would develop the UDP protocol and add more overall functionality to our code so that it works as similar as possible to how TCP works. One such feature is the “stop and wait” function. Data that is corrupted, misplaced or duplicated gets recovered with TCP. In order to ensure dependability, TCP assigns a sequence number to each byte it transmits and then demands an acknowledgment (ACK) from the receiving end. The data is retransmitted if the ACK is not received within the time given.

### Stop And Wait

Stop and wait is an important fundamental feature that ensures the reliability of transmission of data packets. The figure below shows an illustration of how the stop and wait function operates with furthermore explanation under.

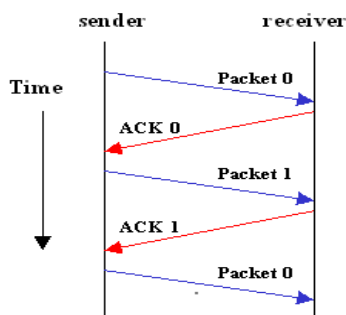
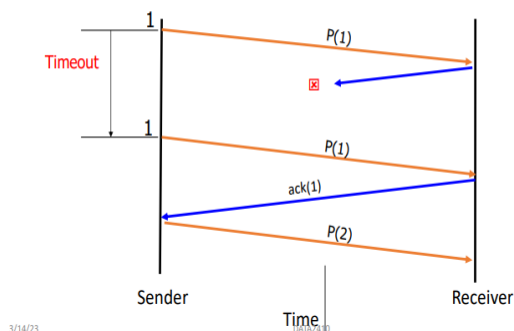


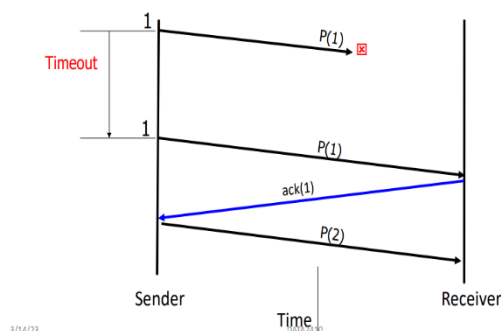
Figure 1. Stop and wait

The connection occurs between a sender and a receiver as Figure 1 shows. The sender transmits one packet and then waits for the receiver to send an acknowledge message. The receiver receives the data packet and send an acknowledgement so that the sender is notified that the packet it sent was delivered to the receiver. This continues so on and so forth until all the data is sent. If the sender does not receive the acknowledgment message within a specified time, it resends the packet. There are two cases in which the sender does not receive an acknowledgment message. These situations can happen when; The ACK message gets lost after the receiver has sent it out or when the data packet transmitted by the sender fails to be delivered to the receiver in which the receiver would not get the packet.<sup>[1]</sup>

Dealing with packet loss (of ack)



Dealing with packet loss

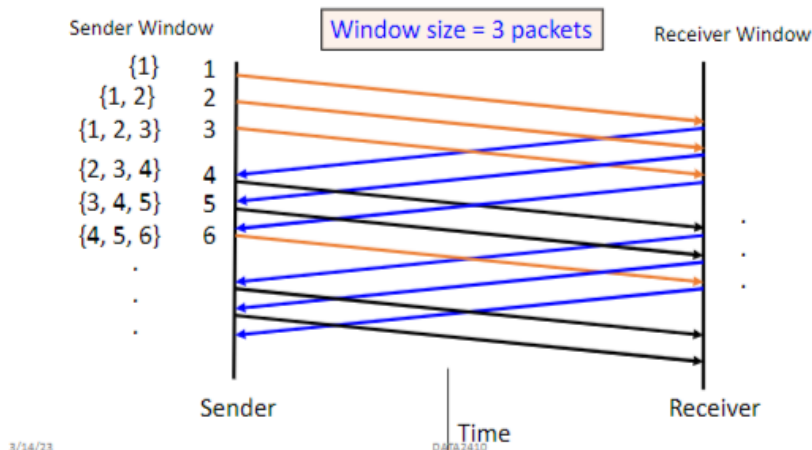


In cases of when the data packet is lost, the following will happen: When the timer expires, the sender assumes that the data is lost and then retransmits the packet. However, the receiver may have already received the packet and sent an acknowledgment message that was lost in transit. This can lead to duplicate packets being sent. To avoid this problem, sequence numbers are assigned to the data packets. By using these sequence numbers, it would be easy to determine if a packet is a duplicate.

## Go-Back-N (GBN)

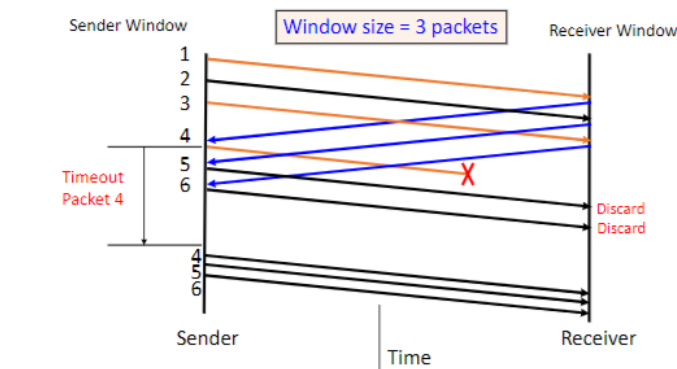
The second reliable method that was implemented into our code is called "Go Back N". This method is identical to that of Stop and Wait as they both send data packets and receive ACK-messages from the receiving point. Unlike Stop and Wait, GBN uses a method called pipelining. GBN transmits multiple packets before waiting for an ACK-message. Rather than sending an ack message after receiving the first packet like stop and wait, the receiving end sends an ack-message after receiving the last packet. The sender keeps track of the packets that have been sent but not yet acknowledged through a so called "window. The window determines how many packets can be sent before waiting for an acknowledgment message from the receiver. The receiver sends an acknowledgment message for the last packet it receives.

## GBN without loss



There are situations where data is lost in transmission. In such cases, the receiver discards all following packets until it receives the lost data. The sender will then continue to send packets until it receives an ack-message for the lost packet. Once the lost packet is received, the receiver sends an ack-message for the last packet it received.

## GBN with loss



## Selective repeat

Selective repeat (SRP) is similar to the GBN protocol. One main difference is that it uses buffers with both the receiver and sender. Also, both make use of a window of size. SRP works mainly well when the link between the sender and receiver is very unreliable. In this case, selectively retransmitting frames is more efficient than retransmitting all of them. This is because retransmission usually happens more frequently. Backward acknowledgements are also in use.

In selective repeat, when the sender transmits a packet, it waits for an acknowledgment (ACK) from the receiver. If an ACK is received, it indicates that the packet has been successfully received and the sender can move on to the next packet. However, if an ACK is not received within a certain time

period (referred to as the timeout period), the sender assumes that the packet has been lost and needs to be retransmitted.

To handle potential retransmissions, the sender maintains a buffer that stores the packets that have been sent but not yet acknowledged. This buffer allows the sender to retransmit specific packets that are lost or damaged, rather than resending the entire sequence of packets. The buffer holds these packets until the corresponding ACK is received or until the timeout period expires.

When a packet is successfully acknowledged by the receiver, it can be removed from the buffer, creating space for new packets to be sent. This process continues until all packets have been acknowledged and the data transmission is complete.

## Selective repeat with packet loss

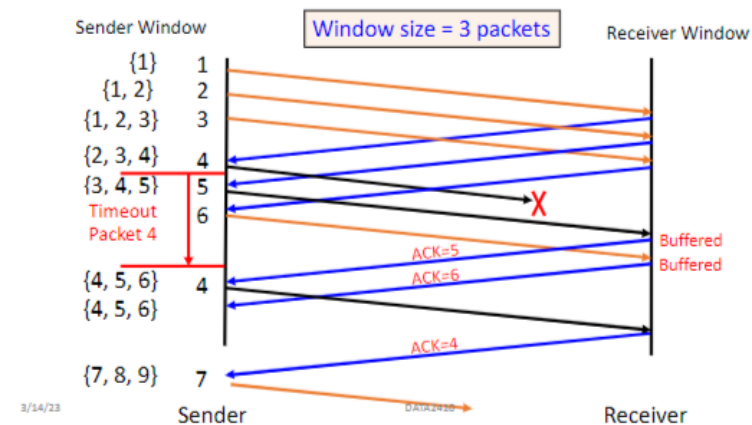


Figure 2 SR with packet loss

## 3 Implementation

Our code consists of three different implementations that originates from the TCP protocol. As described in the background, these implementations are: Stop and Wait, Go-Back-N (GBN) and Selective repeat (SR). Our code was unfortunately not fully implemented as we planned, so selective repeat was not applied. As for the other reliable methods, it was well implemented.

## Stop and Wait implementation

### Sender

Our code snippet including the stop and wait function from the sender site can be viewed in the

```
def stop_and_wait_sender(connection):  
    connection.settimeout(0.5)  
    seq, ack, flags, win = 1, 0, 0, 0  
    throughput = 0  
    message = args.filetransfer  
    f = open(message, "rb")  
  
    while True:  
        msg = f.read(1460)  
        throughput += len(msg)  
        start = time.time()  
  
        if msg == b'':  
            fin_packet = create_packet(0, 0, 2, 1, b'')  
            connection.send(fin_packet)  
            break  
  
        packet = create_packet(seq, ack, flags, win, msg)  
  
        ##### test case 3 (Skip seq) #####  
  
        if(seq == 1):  
            print("skip")  
        else:  
            connection.send(packet)  
            seq += 1  
            acknowledgment = b''
```

following illustration:

The function takes in one parameter, namely "connection". This is the connection formed from the client to the server. This connection is set to be active for 0,5 seconds.

The following variables:

- seq, ack, flags, win = 1, 0, 0, 0
- throughput = 0
- The sending message set to the input file
- F opening the message.

While these variables are set, and a connection is made. The following will occur:

The sender will read the file, and then divide it in chunks of data, namely packets. These packets will contain the above-mentioned variables that were set to 1, 0, 0, 0. Furthermore, these packets will be sent to the receiver and finally right after sending one packet to the receiver, the sender waits for an ack-message before sending the next packet to the receiver.

## The Stop and wait receiver

The receiver uses the server socket to take the message from the stop and wait sender and extract the bytes from the file that gets copied and returns it in the server function to get written down in the copy file in chunks (as shown below). When done copying a finish flag gets sent to the client side.

```
def stop_and_wait_reciever(connectionserver, teller):  
    while True:  
        message, clientaddress = connectionserver.recvfrom(1472)  
  
        h = message[:12] # we extract the header information of the packet  
        seq, ack, flags, win = parse_header(h) # gets the info for the header  
        message = message[12:]  
        syn1, ack1, fin1 = parse_flags(flags)  
  
        if fin1 > 0:  
            finAck_packet = create_packet(0, 0, 6, 1, b'')  
            connectionserver.sendto(finAck_packet, clientaddress)  
            break  
  
        print(f'seq={seq}, ack={ack}, flags={flags}, receiver-window={win}') # p  
        ackPacket = create_packet(0, seq, 4, 5, b'') # creates an acknowledgement  
  
        ##### test case 2 (ack skip) #####  
  
        #if(teller == 3):  
        #    print("Skips")  
        #else:  
        if message != "":  
            connectionserver.sendto(ackPacket, clientaddress)  
  
        return message  
  
if args.reliable == "sw":  
    f = open("Copy-"+ message, "wb")  
    teller = 0  
    while True:  
        msg = stop_and_wait_reciever(serverSocket, teller)  
        teller += 1  
        if msg == b'fin':  
            break  
        #print(msg)  
        if not msg:  
            break  
        msg  
        f.write(msg)  
    serverSocket.close()
```

The GBN sender



The GBN sender reads the bytes of the image and sends the packet in a window, in this case 5 and sends the packets to the receiver and then moves the window to the next packets here is an example:



The window moves when an acknowledgment is received. If an ack message is not received then the whole window gets wiped and sent again. When sent the packets have successfully arrived to the receiver side of GBN.

```
def GBN_sender(connection, feil):
    seq, ack, flags, win = 1,0,0,5

    message = args.filetransfer
    f = open(message, "rb")

    slidewindowData = []
    slidewindowSeq = []
    i = 0

    throughput = 0
    start = time.time()
    test = True

    while i != win:
        msg = f.read(1460)
        if msg == b'':
            connection.send("fin".encode())
            break
        packet = create_packet(seq, ack, flags, win, msg)
        slidewindowData.append(packet)
        slidewindowSeq.append(seq)
        connection.send(slidewindowData[i])
        i += 1
        seq += 1
```

## 4 Discussion

Test cases here and show how you handle losses, reordering and duplicate packets.

Testcase 1:

	Stop and wait	GBN
RTT 25	13.10 mb/s	Win=5(0.78 mb/s) Win = 10(1.07 mb/s) Win = 15(1.5 mb/s)
RTT 50	6.42 mb/s	Win=5(0.5 mb/s) Win=10(0.9 mb/s) Win=15(1.1 mb/s)
RTT 100	5.6 mb/s	Win=5(0.55 mb/s) Win=10(0.93 mb/s) Win=15(1.45 mb/s)

Testcase 2:

ACK skip for stop and wait.

Here the receiver side, when it gets packet number 4 it does not send an acknowledgement back to the client and the client resends the packet that did not get the acknowledgement.

```
"Node: h1"
[42, 43, 44, 45, 46]
[43, 44, 45, 46, 47]
[44, 45, 46, 47, 48]
throughput 0.611574551065722 mbps
root@mininet:/home/mininet/Desktop# python3 application.py -c -f Apollo Creed.j
py -i 10.0.1.2 -p 8080 -r sw
Client connected with 10.0.1.2 , port 8080
ACK: 0
ACK: 1
ACK: 2
No ACK received, resending packet
ACK: 3
ACK: 4
ACK: 5
ACK: 6
ACK: 7
ACK: 8
ACK: 9
ACK: 10
ACK: 11
ACK: 12
ACK: 13
ACK: 14
ACK: 15

"Node: h3"
seq, ack, flags, win = parse_header(h)
File "/home/mininet/Desktop/application.py", line 62, in parse_header
header_from_msg = unpack(header_format, header)
struct.error: unpack requires a buffer of 12 bytes
root@mininet:/home/mininet/Desktop# python3 application.py -s -i 10.0.1.2 -p 80
80 -r sw
A Simpleperf server is listening on port 8080
seq=1, ack=0, flags=0, receiver-window=0
seq=2, ack=1, flags=0, receiver-window=0
seq=3, ack=2, flags=0, receiver-window=0
seq=4, ack=3, flags=0, receiver-window=0
Skips
seq=4, ack=3, flags=0, receiver-window=0
seq=5, ack=4, flags=0, receiver-window=0
seq=6, ack=5, flags=0, receiver-window=0
seq=7, ack=6, flags=0, receiver-window=0
seq=8, ack=7, flags=0, receiver-window=0
seq=9, ack=8, flags=0, receiver-window=0
seq=10, ack=9, flags=0, receiver-window=0
seq=11, ack=10, flags=0, receiver-window=0
seq=12, ack=11, flags=0, receiver-window=0
seq=13, ack=12, flags=0, receiver-window=0
seq=14, ack=13, flags=0, receiver-window=0
```

ACK skip for packet 3. with GBN.

Here the receiver side does not send an acknowledgement for packet 4 and then the entire window gets resent.

```
"Node: h1"
Creating an acknowledgment packet:
this is an empty packet with no data =0
this is an acknowledgment packet of header size=12
seq=0, ack=1, flags=4, receiver-window=0
syn_flag = 0, fin_flag=0, and ack_flag=4
Client connected with 10.0.1.2 , port 8080
[1, 2, 3, 4, 5]
[2, 3, 4, 5, 6]
[3, 4, 5, 6, 7]
[4, 5, 6, 7, 8]
det skjedde en feil
[4, 5, 6, 7, 8]
[4, 5, 6, 7, 8]
[4, 5, 6, 7, 8]
[4, 5, 6, 7, 8]
[5, 6, 7, 8, 9]
[6, 7, 8, 9, 10]
[7, 8, 9, 10, 11]
[8, 9, 10, 11, 12]
[9, 10, 11, 12, 13]
[10, 11, 12, 13, 14]
[11, 12, 13, 14, 15]
[12, 13, 14, 15, 16]

"Node: h3"
Creating an acknowledgment packet:
this is an empty packet with no data =0
this is an acknowledgment packet of header size=12
seq=0, ack=1, flags=4, receiver-window=0
syn_flag = 0, fin_flag=0, and ack_flag=4
A Simpleperf server is listening on port 8080
seq=1, ack=0, flags=0, receiver-window=5
seq=2, ack=0, flags=0, receiver-window=5
seq=3, ack=0, flags=0, receiver-window=5
Skips
seq=4, ack=0, flags=0, receiver-window=5
seq=5, ack=0, flags=0, receiver-window=5
seq=6, ack=1, flags=0, receiver-window=5
seq=7, ack=2, flags=0, receiver-window=5
seq=8, ack=3, flags=0, receiver-window=5
seq=9, ack=4, flags=0, receiver-window=5
seq=10, ack=5, flags=0, receiver-window=5
seq=11, ack=6, flags=0, receiver-window=5
seq=12, ack=7, flags=0, receiver-window=5
seq=13, ack=8, flags=0, receiver-window=5
seq=14, ack=9, flags=0, receiver-window=5
seq=15, ack=10, flags=0, receiver-window=5
seq=16, ack=11, flags=0, receiver-window=5
seq=17, ack=12, flags=0, receiver-window=5
seq=18, ack=13, flags=0, receiver-window=5
seq=19, ack=14, flags=0, receiver-window=5
seq=20, ack=15, flags=0, receiver-window=5
seq=21, ack=16, flags=0, receiver-window=5
```

Testcase3:

Skip seq for stop and wait.

Here the first packet does not get sent so the server does not send anything. Then the client realizes that the acknowledgement was not received and resends the packet.

```
ACK: 47
The process is finished
throughput 2472.8794494954206 mbps
yllis@yllis-MacBook-Air src % python3 application.py -c -r sw -f Apollo Creed.jpg
Client connected with 127.0.0.1 , port 8088
skip
No ACK received, resending packet
ACK: 0
ACK: 1
ACK: 2
ACK: 3
ACK: 4
ACK: 5
ACK: 6
ACK: 7
ACK: 8
ACK: 9
ACK: 10
ACK: 11
ACK: 12
ACK: 13
ACK: 14

A Simpleperf server is listening on port 8088
seq=1, ack=0, flags=0, receiver-window=0
seq=2, ack=1, flags=0, receiver-window=0
seq=3, ack=2, flags=0, receiver-window=0
seq=4, ack=3, flags=0, receiver-window=0
seq=5, ack=4, flags=0, receiver-window=0
seq=6, ack=5, flags=0, receiver-window=0
seq=7, ack=6, flags=0, receiver-window=0
seq=8, ack=7, flags=0, receiver-window=0
seq=9, ack=8, flags=0, receiver-window=0
seq=10, ack=9, flags=0, receiver-window=0
seq=11, ack=10, flags=0, receiver-window=0
seq=12, ack=11, flags=0, receiver-window=0
seq=13, ack=12, flags=0, receiver-window=0
seq=14, ack=13, flags=0, receiver-window=0
seq=15, ack=14, flags=0, receiver-window=0
seq=16, ack=15, flags=0, receiver-window=0
seq=17, ack=16, flags=0, receiver-window=0
seq=18, ack=17, flags=0, receiver-window=0
seq=19, ack=18, flags=0, receiver-window=0
seq=20, ack=19, flags=0, receiver-window=0
seq=21, ack=20, flags=0, receiver-window=0
```

Skip seq for GBN.

Here, the client does not send the packet number 8 and then it resends what is in the window.

```

[35, 36, 37, 38]
[36, 37, 38, 39]
[37, 38, 39, 40]
[38, 39, 40, 41]
[39, 40, 41, 42]
[40, 41, 42, 43]
[41, 42, 43, 44]
[42, 43, 44, 45]
[43, 44, 45, 46]
[44, 45, 46, 47]
throughput 28.984084529868973 mbps
yllis@Yllis-MacBook-Air src % python3 application.py -c -r gbn -f Apollo Creed.jp
9
Client connected with 127.0.0.1 , port 8888
[1, 2, 3, 4, 5]
[2, 3, 4, 5, 6]
[3, 4, 5, 6, 7]
skip
[4, 5, 6, 7, 8]
[5, 6, 7, 8, 9]
[6, 7, 8, 9, 10]
[7, 8, 9, 10, 11]
[8, 9, 10, 11, 12]

seq=1, ack=0, flags=0, receiver-window=5
seq=2, ack=0, flags=0, receiver-window=5
seq=3, ack=0, flags=0, receiver-window=5
seq=4, ack=0, flags=0, receiver-window=5
seq=5, ack=0, flags=0, receiver-window=5
seq=6, ack=1, flags=0, receiver-window=5
seq=7, ack=2, flags=0, receiver-window=5
seq=8, ack=4, flags=0, receiver-window=5
seq=9, ack=5, flags=0, receiver-window=5
seq=10, ack=6, flags=0, receiver-window=5
seq=11, ack=7, flags=0, receiver-window=5
seq=12, ack=8, flags=0, receiver-window=5
seq=8, ack=4, flags=0, receiver-window=5
seq=9, ack=5, flags=0, receiver-window=5
seq=10, ack=6, flags=0, receiver-window=5
seq=11, ack=7, flags=0, receiver-window=5
seq=12, ack=8, flags=0, receiver-window=5
seq=13, ack=8, flags=0, receiver-window=5
seq=14, ack=9, flags=0, receiver-window=5
seq=15, ack=10, flags=0, receiver-window=5
seq=16, ack=11, flags=0, receiver-window=5
seq=17, ack=12, flags=0, receiver-window=5
seq=18, ack=13, flags=0, receiver-window=5

[22, 23, 24, 25, 26]
[23, 24, 25, 26, 27]
[24, 25, 26, 27, 28]
[25, 26, 27, 28, 29]
[26, 27, 28, 29, 30]
[27, 28, 29, 30, 31]
[28, 29, 30, 31, 32]
[29, 30, 31, 32, 33]
[30, 31, 32, 33, 34]
[31, 32, 33, 34, 35]
[32, 33, 34, 35, 36]
[33, 34, 35, 36, 37]
[34, 35, 36, 37, 38]
[35, 36, 37, 38, 39]
[36, 37, 38, 39, 40]
[37, 38, 39, 40, 41]
[38, 39, 40, 41, 42]
[39, 40, 41, 42, 43]
[40, 41, 42, 43, 44]
[41, 42, 43, 44, 45]
[42, 43, 44, 45, 46]
[43, 44, 45, 46, 47]

```

## 5 Conclusions

To conclude, we made an application that can take a file and make an identical copy. The application runs on three reliable methods: Stop and wait, Go Back-N, Selective repeat. These methods send chunks of the file in packets and writes it to a new file. How it works is that the methods have a sender function that reads and send the bits of the file to the other function that receives and writes it down. The application uses UDP which is unreliable, which causes packet loss and to prevent this the application runs a DRTP which makes it reliable, by resending the packets that was lost. By doing this the application can copy a file successfully without having any packet loss.

## 6 References (Optional)

[https://www.isi.edu/nsnam/DIRECTED\\_RESEARCH/DR\\_HYUNAH/D-Research/stop-n-wait.html](https://www.isi.edu/nsnam/DIRECTED_RESEARCH/DR_HYUNAH/D-Research/stop-n-wait.html)

### NOTE:

The report cannot exceed 20 pages, including the list of references. The page format must be A4 with 2 cm margins, single spacing and Arial, Calibri, Times New Roman or similar 11-point font.