DATA2410

NETWORK AND CLOUDCOMPUTING-

HOME EXAM.

Candidate number: 140

Emne Kode: DATA2410

Emnenavn: Datanettverk og skytjenester

Antall sider:

Innleveringsfrist: 19.05.2025

Introduction:

This is a report on the home exam for the subject DATA2410. The project focuses on implementing a "reliable transport protocol DRTP". This protocol is important for ensuring reliable transport, which means that it has secure control of which data is being sent and received, no duplications or drops. DRTP is built over UDP (User Datagram Protocol, which is an unreliable transport protocol.

DRTP uses a logic called Go-Back-N (GBN) for the packets transport. GBN is an important part of this application. GBN will, with the help of sliding window and retransmission, hold a solid flow where we can have control over what data is received and what we need to retransmit. GBN works differently by resending all the packets in a specific window, therefore ensuring that data are sent in the correct order.

There are more methods used by the more reliable data transport protocol, transmission control protocol TCP, but for this DRTP implementation we are going to use some that are like TCP as the three-way handshake, which will be the first phase of data transport, where we establish a connection between the server and the client. The second phase is the actual data transport where I will be using concepts like the sliding window and retransmission to have reliable transport. The third and last phase of DRTP is the connection tear down.

So, in this python program called application.py I implemented what DRTP would look like and used a JPG file as a test example. This application can be used to transport various file types but will need a special packet "metadata" that sends information about the file as the first packet and then supplies the content data in the data transport phase. I chose to overlook this part to gain simplicity of the code and to follow the exam instructions closely.

Implementation:

I have comments for each function on my application.py code. Here I will be explaining the implementation of the parts that are fundamental for DRTP.

The goal of DRTP is to deliver reliable file transfer over the inherently unreliable UDP.

- *Packet structure and header*

```
# the header is 8 bytes: this gives 2 bytes to (seq, ack, flags, win)
header_format = '!HHHH' # Network byte order, four unsigned shorts
```

```
def create_packet(seq, ack, flags, win, data=b''):
    header = struct.pack(header_format, seq, ack, flags, win)
    return header + data
```

DRTP packets carry both control and data. This header format ensures that both client and server interpret packet metadata identically. The 8-byte header format includes:

- Seq: sequence number that ensures in-order delivery.
- Ack: acknowledgment number that confirms receipt of packets.
- Flags: SYN, ACK, and FIN, for connection control.
- Win: window size for flow control.

- *Flag constants and their use.*

```
# some flag data
SYN = 1 << 3 # value 8 = 1000
ACK = 1 << 2 # value 4 = 0100
FIN = 1 << 1 # value 2 = 0010
```

since DRTP mimics the TCP's control flow using flags to manage connection states:

- SYN: initiates a connection.
- ACK: confirms successful receipt.
- FIN: terminates the connection.

Using bitwise operations where we give different values for the set bit "1" allows combining multiple flags in a single integer, which reduces complexity in packet parsing and promotes extensibility.

- *Window size and flow control*

```
WINDOW_SIZE = 3  # sliding window size (client)
MAX_SERVER_WINDOW = 15  # max window size server accpts
```

A sliding window allows multiple packets to be in transit before receiving an ACK. This improves throughput over high-latency networks.

- Client window 3: keeps implementation simple while demonstrating core DRTP behavior.
- Server max window 15: provides flexibility and ensures the server can throttle clients that send too aggressively.

- *Timeout and retransmission logic*

```
TIMEOUT = 0.4  # 400 ms timeout
```

```
for s in window:
    if time.time() - sent_time[s] > TIMEOUT:
        print(f"{timestamp()} -- retransmitting packet with seq ={s}")
        sock.sendto(packets[s],(server_ip,server_port))
        sent_time[s] = time.time()
```

DRTP ensures reliability by transmitting unacknowledged packets. This is especially critical for UDP where dropped packets are not detected by default. The sliding window combined with timeout gives us the Go-Back-N logic and retransmits the packets in said window.

- *Three-way handshake (connection setup)*

```
sock.sendto(create_packet(1, 0, SYN, WINDOW_SIZE), (server_ip,server_port))
print("SYN packet is sent")

try:
    data, _ = sock.recvfrom(PACKET_SIZE)
```

This is the first snippet of the three-way handshake starting at the client side. DRTP establishes a virtual connection using a handshake like TCP.

- Step1: Client sends SYN.
- Step2: server responds with SYN-ACK
- Step3: client responds with ACK.

This handshake ensures both sides are ready and negotiate window sizes, enabling fair and reliable transport. In the application.py code, used a limitation/error handing for the max limit on both sides, this ensures that if the client calls a window size higher than the max server window size. It aborts the transport, so the client can request a lower window size.

Client side:

```
_, ack, flags, server_window , payload = parse_packet(data)

if payload == b"REJECT":
    print("Server rejected connection: window size too large.")
    sock.close()
    return
```

server side:

```
seq, _, flags, client_window, _ = parse_packet(data)
syn, _, _ = parse_flags(flags)
if syn:
    print("SYN packet is received")

    if client_window > MAX_SERVER_WINDOW:
        print(f"Client's window size ({client_window}) exceeds server limit ({MAX_SERVER_WINDOW}). closing..")
        reject_msg = b"REJECT"
        sock.sendto(create_packet(0,0,0,0, reject_msg), (client))
        sock.close()
        return
```

- *File sending with sliding window.*

```
while len(window) < WINDOW_SIZE and not eof:
    data = f.read(TRANSFERRED_DATA_SIZE)
```

Each packet carries a TRANSFERRED_DATA_SIZE = PACKET_SIZE – HEADER_SIZE portion of the file. Packets are stored in a dictionary for retransmission if no ACK is received.

- *Simulated packet drops (for testing)*

```
if seq == discard_seq:
    print(f"{timestamp()} -- simulated drop of packet {seq}")
    discard_seq = -1 #only drop once
    continue
```

This allows testing how well DRTP handles loss.

It is essential to validate retransmission logic and robustness of ACK handling.

- *Connection teardown*

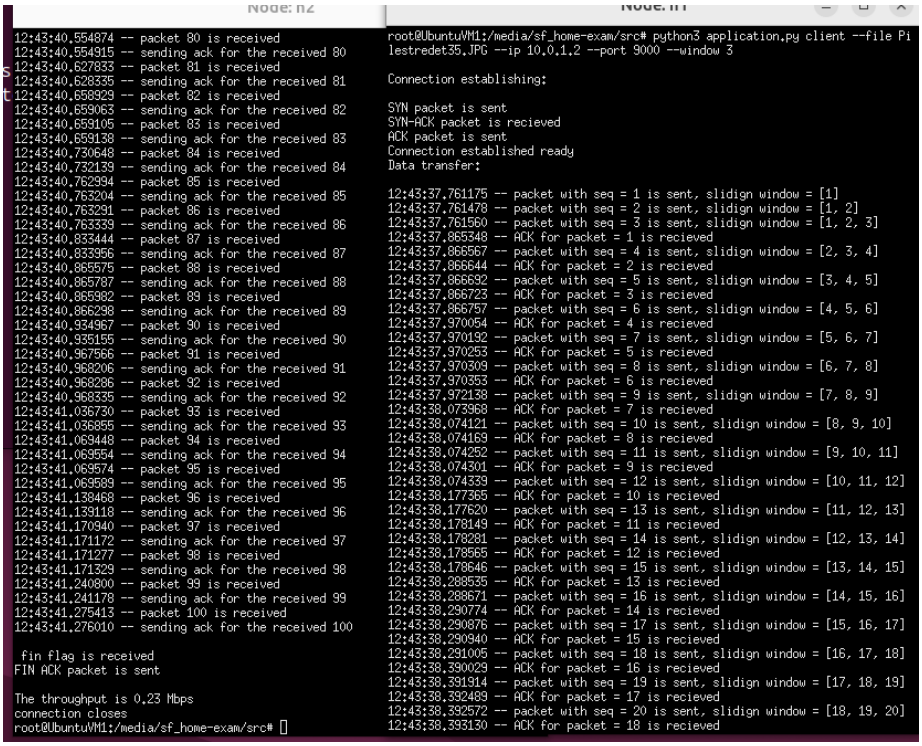
```
print("Connection teardown \n")
sock.sendto(create_packet(seq, 0, FIN, 0),(server_ip, server_port))
print("fin packet sent \n")
try:
    fin_ack, _= sock.recvfrom(PACKET_SIZE)
    _,_,flags, _, _ = parse_packet(fin_ack)
    _, ack_flag, _ = parse_flags(flags)
    if ack_flag:
        print("FIN PACKET IS RECIEVED ")
except socket.timeout:
    print("Timeout waiting for FIN ACK")
print ("connection closes")

sock.close()
```

A Fin packet initiates termination. If an ACK is received, the connection is closed, which ensures that both the client and server agree the transfer is complete.

Discussion:

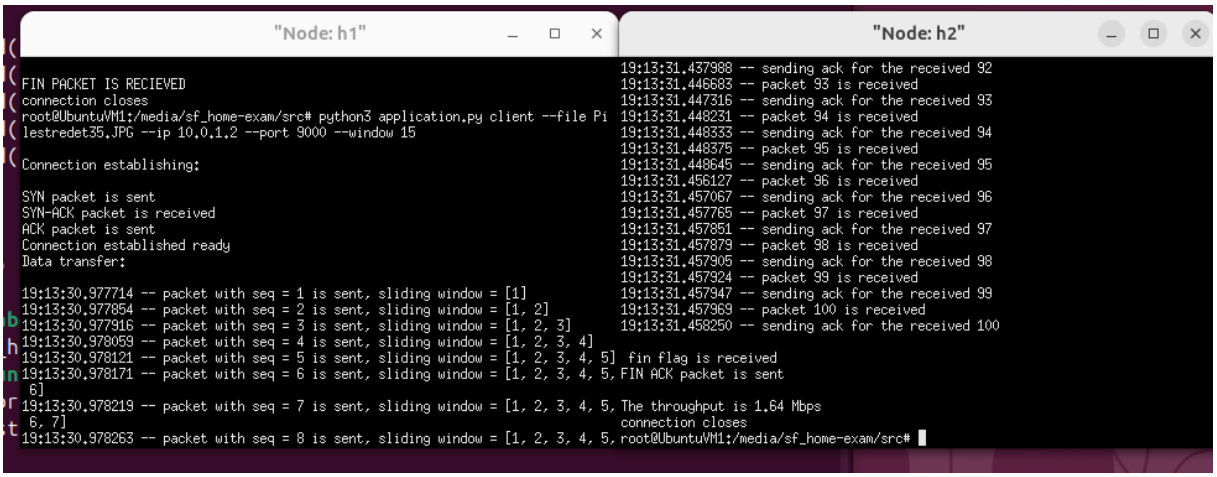
1. Testing application.py with sliding window 3, 5, 10, 15 and calculating the throughput. Any amount higher will result in a connection teardown. The RTT is 100(standard).
   I will be putting the results on a table for simplicity and readability.
   But here is an example picture for how my Mininet results look like for RTT 100 sin 3.



Sliding window size	throughput
3	0,22Mbps
5	0,37Mbps
10	0,66Mbps
15	1,04Mbps

Observing the result, we can see that the throughput for each sliding window size increases. The throughput is increasing because the higher sliding window size enables more packets to be in transmission at the same time. The sliding window also is an instruction to tell the clients how many packets it can send before waiting for an ACK. This is beneficial for handling delays.

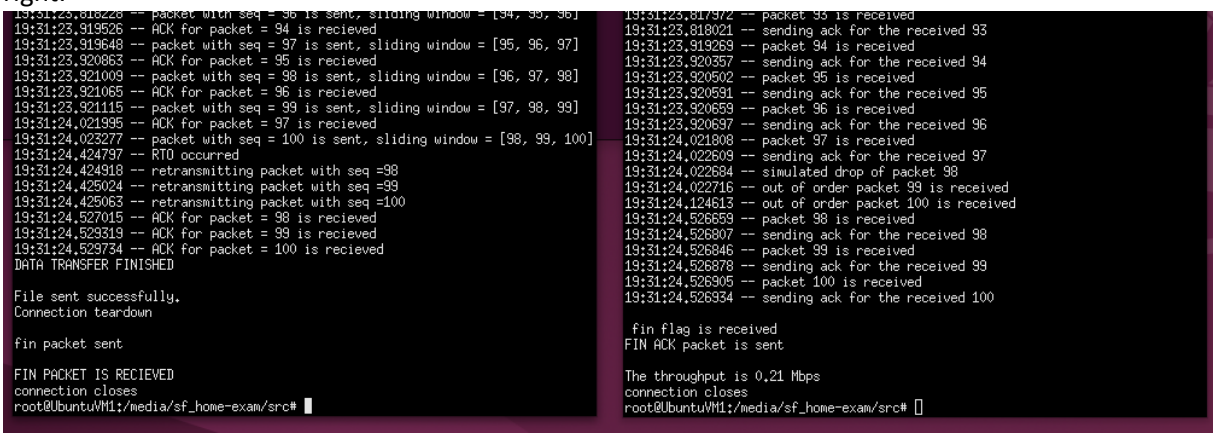
2. Testing application.py with the window sizes 3,5,10,15 and modifying the RTT to 50ms and 200ms. And then see how this affects the throughput.
   I will be putting the results on a table for simplicity and readability.
   But here is an example picture for how my Mininet results look like for the RTT 50 win 15.



Sliding window size	RTT	Throughput
3	50	0,40Mbps
5	50	0,50Mbps
10	50	1,34Mbps
15	50	1,64Mbps
3	200	0,11Mbps
5	200	0,19Mbps
10	200	0,39Mbps
15	200	0,54Mbps

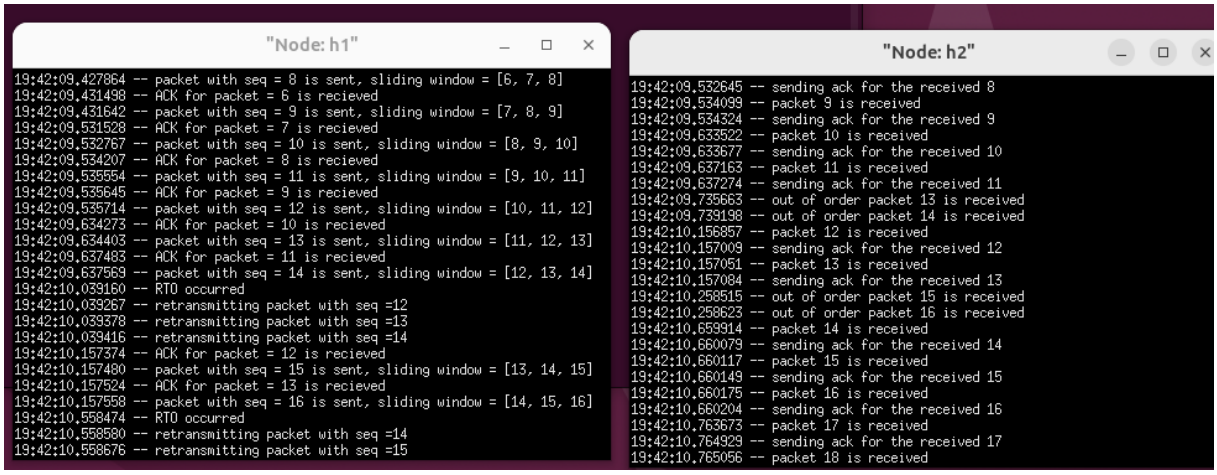
RTT- Round trip time for data transport. with the higher RTT we can notice that the throughput is significantly lower. Even though we change the sliding window size to a higher value to allow more packets to be in transmission simultaneously, we still have a lower throughput. The simple explanation for this would be that the longer the time is the lower throughput. When we calculate the throughput, we are dividing the inventory I by the time T. and therefore get this result with higher and lower RTT.

3. Here you can see how the server and client manage the lost packet. I changed the RTT back to 100ms. I chose to discard packet 98. The left one is the client and on the server is on the right.

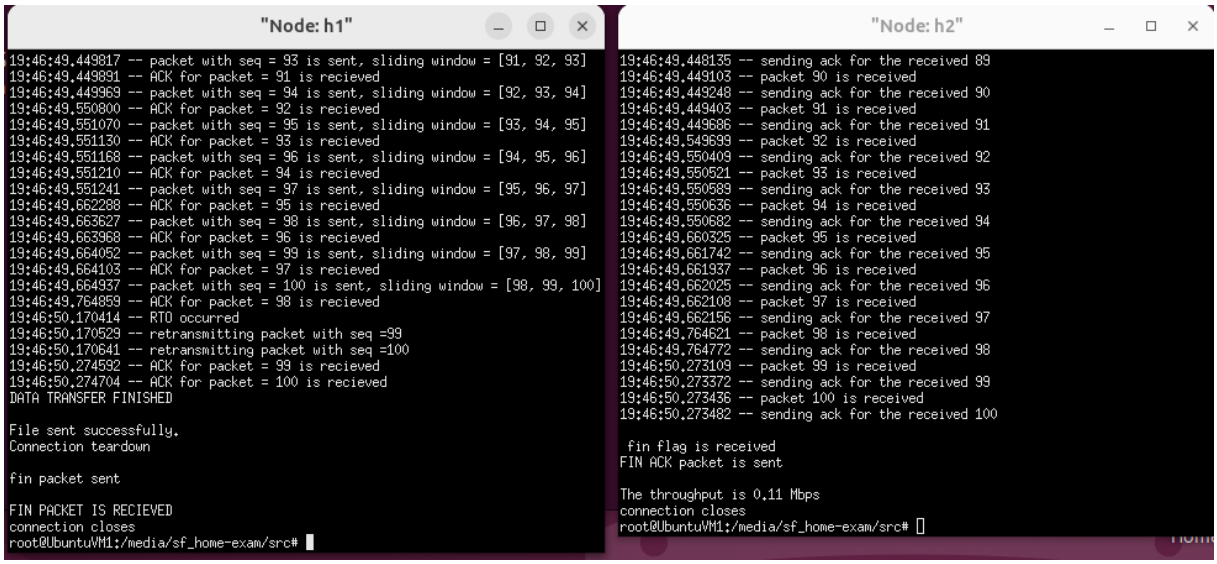


4. Results for 2%, 5% and 50% loss.

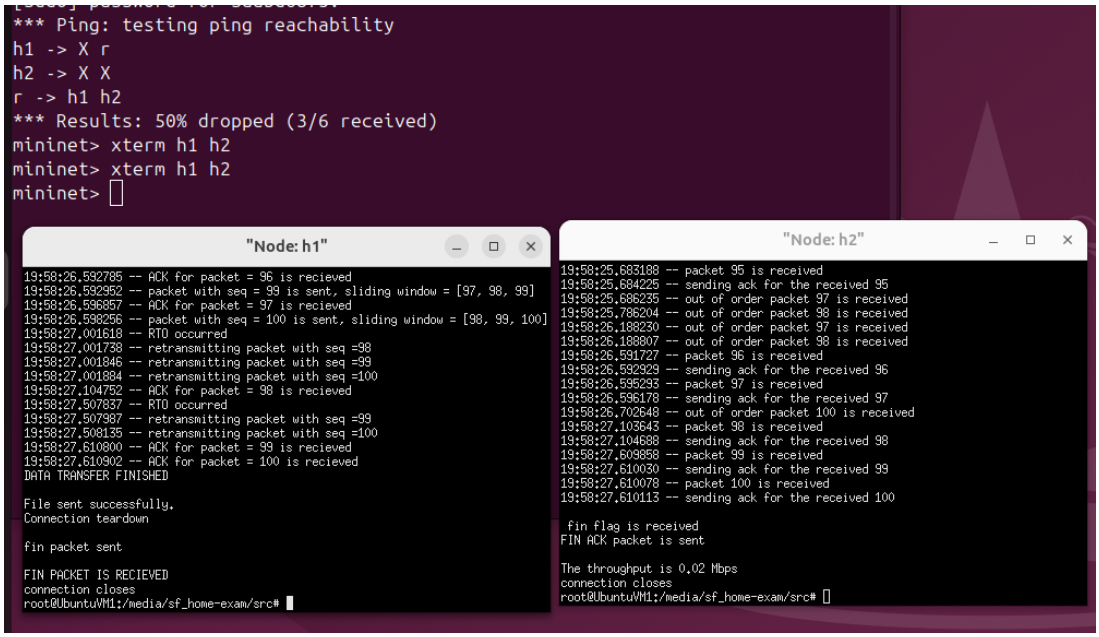
RTT 100 2% loss:



RTT 100 loss 5%:



RTT 100 loss 50%:



What I noticed was a big decrease in the throughput and since I have 100 packets to send my chosen file, RTO accrued almost as much as the loss percentage. Since we are using GBN logic, it resends all the packets in the window, and this takes time in 50% loss.

5. A loss of a FIN-ACK packet means that the connection close was not mutual, even though it still closes since the data are sent. But it decreases the reliability of the application.

Sources:

- DATA2410 course notes.
- Argparse from python. https://docs.python.org/3/howto/argparse.html
- The lecturers, Safiqul Islam, open GitHub code: https://github.com/safiqul/2410/blob/main/header/header.py
- Assignment two code.
- AI was only used to check error in the code.