Reliable Transport Protocol (DRTP)

Cadidate number: 267
Course code: DATA2410
Course name: Datanettverk og skytjenester
Education program: Informasjonsteknologi Bachelor
Total word count: 1687
Deadline: 21.05.2024 – 12:00

OSLO METROPOLITAN UNIVERSITY
STORBYUNIVERSITETET

Introduction

The purpose of this project is to create a simple application for transporting a file (data) from client to server. It is a UDP based application that does not have any built-in reliability protocols like TCP socket programming gives, but rather a self-made reliability protocol to ensure that the data client provides will be delivered without duplicate packets or packet loss / missing data. The program has a packet size of 1 KB as specified in the assignment requirements and has statements to handle and recognise flags in headers and respond accordingly to the given data. It uses a sliding window protocol to allow for user to choose how big the sliding window should be when invoking the client. All output and prints are based on the commands provided and output given in the assignment text, under sections “A simple file transfer server”, “A simple file transfer client” and “Suggestions”.

Implementation

Argument parsing

To be able to invoke server, client and other command line arguments I implemented argument parsing in accordance with the table provided in the argument details. The formatting of the arguments were taken from sources provided in the DATA2410 Canvas modules (*Argparse — Parser for Command-Line Options, Arguments and Sub-Commands*, n.d.) and from previous obligatory assignment work. For example, the parse arguments for invoking a server are implemented like so:

```
parser.add_argument('-s', '--server', action='store_true', help='enable the server mode')
parser.add_argument('-c', '--client', action='store_true', help='enable the client mode')
```

The relevant arguments also have default values definitions, like port defaulting to 8080 and window size defaulting to 3.

Input checks

To be able to control whether the user properly invokes modes in the program with a correct port, there is a *port_check(args.port)* function. This function checks if the port number input is in accordance with our given range and returns True if it is. Returning false will print a misinput error and exit the program.
The program will also check if command line arguments are used correctly. This includes checking if someone tries to invoke server and client at the same time, invokes neither in an argument, leaves out a file and if they specify the name of the output file using the -f command.
The code does not include specific format checks for IP as I was unable to implement it before the deadline. It won't work with wrong formatting as it needs a valid IP address to start server, but there isn't a specific error handling in the code that makes the program respond this way.

Packet handling

There are two main functions used to handle packets sent between client and server, a function that unpacks received packets and a function that packs packets to be sent. These are constructed so you can manipulate what they contain using SYN, ACK and FIN binary constants. Both functions use the struct library to restructure packets based on our pre-defined formatting rules. For example, struct is used to construct a packet like so:

```
def create_packet(seq_nr, ack_nr, flags, data=b""):
    header = struct.pack(HEADER_FORMAT, seq_nr, ack_nr, flags)
    return header + data
```

For the different types of packets needed to be sent between server and client, this function is called with flag variables and defined in a packet variable:

```
if flags & SYN:
    print("SYN packet received")
    syn_ack_packet = create_packet(0, 0, SYN | ACK)
```

This lets us recognise the contents of different types of packets and have the program send the correct types of packets between server and client based on what it receives. The code is made to print confirmations of every packet sent, received and retransmitted, as well as notify timeouts while waiting. Program will wait 500ms for a packet before timing out.

Data Transfer

On client side, data transfer is handled using a few different variables to keep track of how much data we have transferred and what is coming up next. This is where the sliding window is implemented as well.

```
while bytes_sent < file_size or window:
    while len(window) < window_size  and bytes_sent < file_size:
        data = f.read(DATA_SIZE)
        if not data:
            break
```

The main handling of when to stop data transfer happens in the code snippet above. The program will use its tracking variables to compare against the whole file and decide whether to keep transferring. If there's no data left to send, it will end the loop, but if there's more, it will continue to send packets and increment the tracker variables/sliding window. There's an exception for timeout that will retransmit packets if this process takes too long to complete, or in case of an eventual failure to transmit packages. Packet loss and discarding of packets will be demonstrated later in this report.\

Discussion

The tests presented under were all done in a VirtualBox VM using Ubuntu OS with 2 CPUs enabled. The program used to simulate networks was Mininet as well as a provided simple-topo.py file from the assignments GitHub page. Unless stated otherwise, the image used for testing was Iceland_safiqul.jpg. Pictures of the terminal have been mostly left out as they take up a lot of space while also being very hard to read.

Question 1

Commands used to invoke server:

Python3 application.py -s -I 10.0.1.2 -p 8080

Commands used to invoke client:

Python3 application.py -c -i 10.0.1.2 -p 8080 -f <filepath>

Python3 application.py -c -i 10.0.1.2 -p 8080 -f <filepath> -w 5

Python3 application.py -c -i 10.0.1.2 -p 8080 -f <filepath> -w 10

Sliding window	Throughput (Mbps)
3	0.24
5	0.39
10	0.78

Since the sliding window is what specifies what data is to be sent out from client before necessarily having to receive acknowledgement packets back from server, increasing sliding window size increases the number of packets sent per second while the program is running. It reduces the amount of idle time between a packet being sent and receiving acknowledgements by constantly moving the sliding window and controlling what packet has gotten an acknowledgement back, what has been sent, and which are upcoming. Especially when using a network that has a high bandwidth, using smaller sliding windows means that the speed at which packets can be sent out and received is not used to its full potential and can cause a ”bottleneck” for data that is being transmitted. When increasing that window size, it will lead to better utilization of capacity, thus increasing the throughput of data being sent and received. That’s why we can see a steady increase in throughput as we increase the sliding window size (*Sliding Window Protocol | Set 1 (Sender Side)*, 2016).

Question 2:

To measure these throughput values, I went into the provided simple-topo.py file and changed the RTT value from default 100ms to 50ms and 200ms respectively for the needed test. To invoke the server and client I used the same commands as in question 1.

Sliding window	RTT 50 ms	RTT 100 ms	RTT 200 ms
3	0.46 Mbps	0.24 Mbps	0.12 Mbps
5	0.77 Mbps	0.39 Mbps	0.20 Mbps
10	1.54 Mbps	0.78 Mbps	0.39 Mbps

RTT, 'Round Trip Time', is the duration from when a browser sends a request to when it gets a response from the server ('What Is Round Trip Time | Imperva', n.d.). By changing the RTT of our topo script, we can simulate how long it would take for the client to get a response from server. Say for example with a net that can provide a RTT of 200ms, one would assume it was far away or had poor connection but using a sliding window of 10 from the table above shows that allowing a program to send packets in a window of 10 at a time makes up for the higher RTT. The efficiency of packet transmission is higher on a slower network than it is at RTT=100ms with less packets allowed in the sliding window, for example: RTT 200ms with -w 10 has the same throughput value of RTT 100ms with -w 5. This is a good way to compensate for a slower connection, but it would still only be useful for things like file transfers while not being as good for things where you want to get a faster response from the server, like gaming.

Question 3

Contrary to other questions, here I have chosen to include terminal prints to illustrate the function of my program. Here we have used -d to drop a specified packet in the server command line. The image used, iceland_safiqul.jpg, is a rather large file to transfer, so I chose to drop packet 1830 so it would be easier to capture the terminal output.

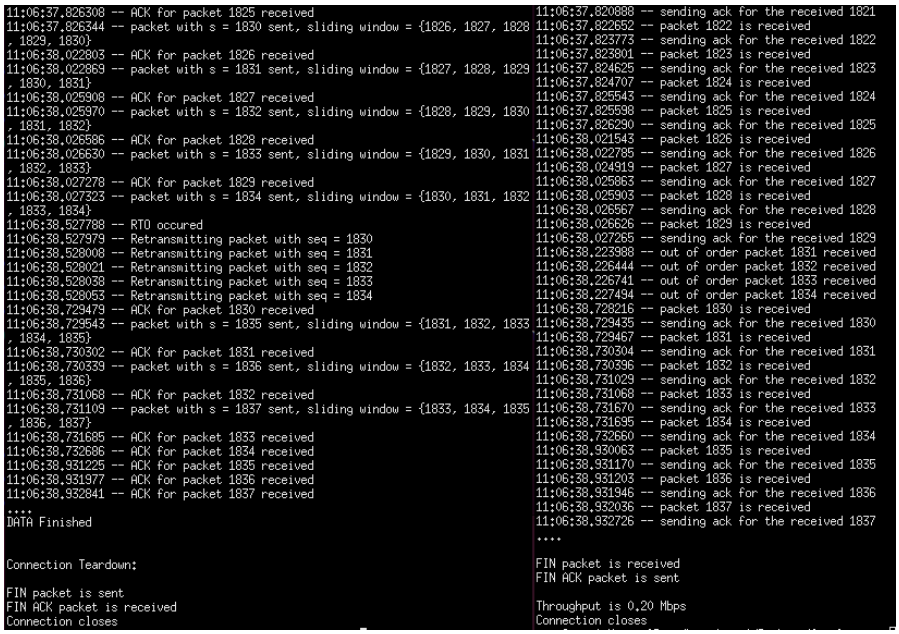
Command used to invoke server:

Python3 application.py -s -i 10.0.1.2 -p 8080 -d 1830

Command used to invoke client:

Python3 application.py -c -I 10.0.1.2 -p 8080 -f <file_path> -w 5

Sliding window set to 5 for faster transmission of the file.



On server side, we can see that the program discovers that it got delivered packet 29, set expected sequence number to 29+1, and then marked the next packet as out of order when it didn't match the expected sequence number. When this occurs, client side will get a message for RTO (as given in the assignment details) and retransmit the entire window before continuing on.

Question 4:

These file transfers were executed with an RTT = 100ms. I have included the packet loss while transferring the image given in the assignment details.

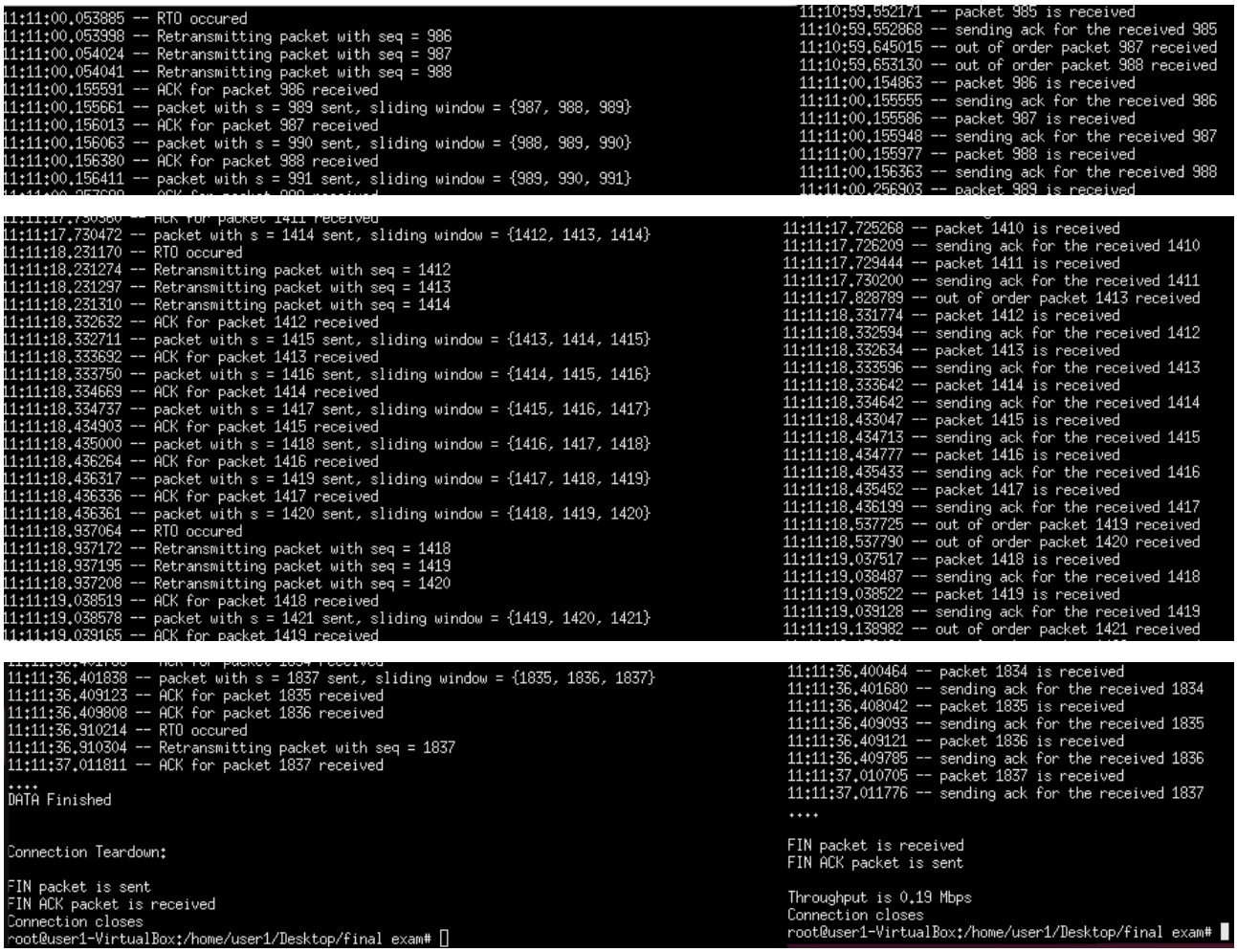
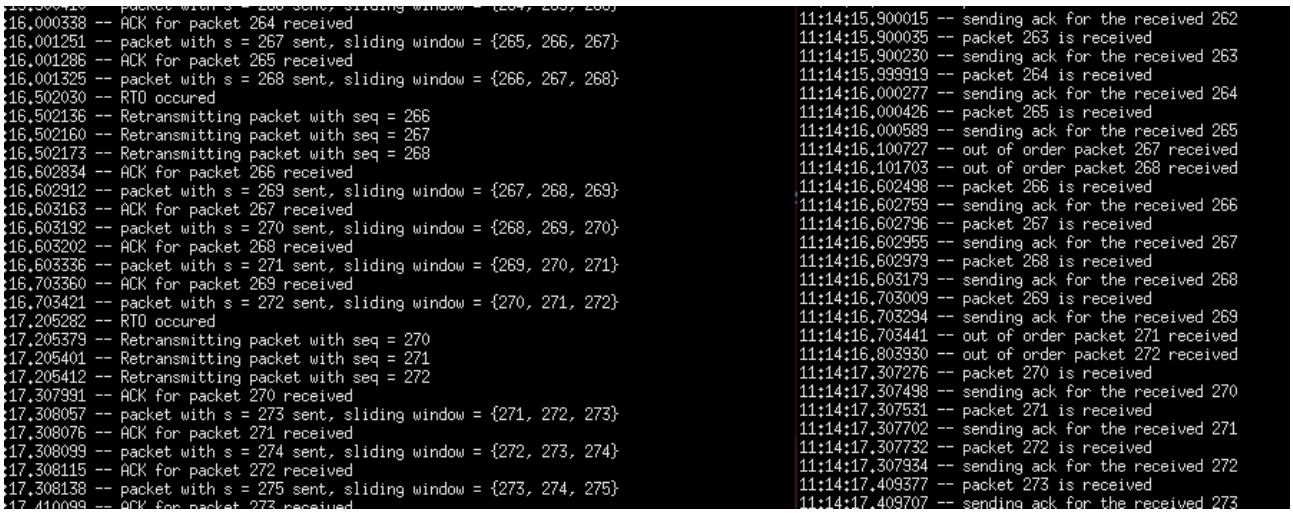


Figure 4: 2% packet loss with a large image file transfer

During the 1837 transferred packets, the packet loss was a lot more frequent. As the assignment descriptions didn't include a counter for number of packets lost during transmission, I don't have an exact number for how many were lost and retransmitted during this test. It's safe to assume that *approximately* 37 out of 1837 packets were lost and retransmitted.

The image below shows the same file being transmitted with the same RTT but 5% packet loss instead of 2%. During testing, packet loss would happen every second or so. With 1837 it would be safe to assume that approximately 92 packets were lost and retransmitted during the file transfer.



References:

argparse—Parser for command-line options, arguments and sub-commands. (n.d.). Python Documentation. Retrieved 21 May 2024, from https://docs.python.org/3/library/argparse.html

Sliding Window Protocol | Set 1 (Sender Side). (2016, December 13). GeeksforGeeks. https://www.geeksforgeeks.org/sliding-window-protocol-set-1/

What is Round Trip Time (RTT) | Behind the Ping | CDN Guide | Imperva. (n.d.). *Learning Center*. Retrieved 20 May 2024, from https://www.imperva.com/learn/performance/round-trip-time-rtt/