Emnekode: DATA2410_1 Kandidat nr: 343

2024

Reliable Transport Protocol (DRTP)

DATA2410 NETWORKING AND CLOUD COMPUTING

KANDIDATNUMMER: 343

Reliable Transport Protocol (DRTP)

This purpose of this assignment is to implement a file transfer application that uses a simple reliable transport protocol (DRTP). The protocol should be coded and provide reliable delivery on top of UDP which is unreliable transfer protocol (Islam, S., 2024). The protocol ensures that data is received in order, without duplicates and missing data without using exciting reliable transfer protocols like TCP. To achieve this a tree-way handshake is used to establish a connection and the file is divided into packets consisting of a header (6 bytes) and data (994 bytes) before it sends the packet to the server. The server only accepts packets in order and the client only sends new packets after receiving an acknowledgement from the server if not it will retransmit the packet.

Implementation:

The application is used to for image transfer, it can be run in client or server mode and invoked by command line arguments. The argpars module is used to create the arguments that the program can receive from the command line. The Server must be running for the client to be able to connect to the server. Figure 1 shows the arguments generated in the application.

```
# Adds arguments
parser.add_argument( *name_or_flags: '-s', '--server', action= 'store_true',
parser.add_argument( *name_or_flags: '-c', '--client', action= 'store_true',
parser.add_argument( *name_or_flags: '-p', '--port', type=InputValidation.che
parser.add_argument( *name_or_flags: '-i', '--ip', default='10.0.1.2', type=I
parser.add_argument( *name_or_flags: '-f', '--filename', type=InputValidation
parser.add_argument( *name_or_flags: '-w', '--windowsize', type=InputValidati
parser.add_argument( *name_or_flags: '-d', '--discard', type=InputValidation.
```

Figure 1 – Command line arguments

Server mode:

The server is invoked with -s argument. The IP address of the server should be specified and the port number the server should bind/listen to. If it is not specified or it will set a default IP address to 10.0.1.2 and port number 8088. If the user uses the -f command they can specify the filename they

prefer of the received jpg file. It must be a jpg file. If it is not set the application will give the file a default name received_image.jpg. When the server is invoked, it creates a socket/ communication endpoint and bind it to the specified port.

It is also possible to choose a packet that the server will drop when it receives it. The argument -d and then the sequence number for the packet will make the server drop it once before it accepts it when the client retransmits it. This makes it possible to manually test what happens if a packet is dropped.

Client mode:

The client mode is invoked with -c. The IP address and port number in which the client should connect has to be set by user and match the IP/port number the server is listening to for the application to successfully transfer a file. The -f argument is used to determine the file that the client want to transfer. The user can transfer a local file by provide the file path. Or if the file already is in the same directory as the program, it only needs the filename. The user can also determine the window size using the -w argument, default is 3. The window size determines how many packets are sent at once after a connection is established before the client waits on acknowledgment response from server. When the client is invoked, it creates a socket/ communication endpoint that it communicates through. A timer of the socket is also set (500ms) to make sure it exits if the server is not responding.

Establish a connection

When the server and client is invoked, they both create a socket/communication endpoint. To create a reliable connection, it is necessary to establish a connection between the sender and receiver. This does not happen in UDP since it is a connectionless protocol (GeeksForGeeks, 2024). To establish a connection a tree-way handshake is used. To implement this a 6 bytes header split in 3 sections, each consisting of 2 bytes is attached to the data sent. The header consists of sequence number, acknowledgement number and flags which will be referred to as seq, ack and flags. The sequence number tracks the order of the packets, and the acknowledgement number is an acknowledgment that a specific packet is received. The flags consist of 4 bits that is set. The SYN = synchronize flag, ACK = acknowledgment flag, FIN = Finnish flag and the R = reset flag. The reset flag will not be used. To

establish a connection the client sends an empty packet (just the header) with the SYN flag set. It waits on the receiver to send a packet with the flags in header set to SYN-ACK and then sends an ACK back. A reliable connection is established. The timer makes the client exit if it does not receive a SYN-ACK.

```
#Three-way handshake
print('Connection Establish Phase: \n')
try:
    SYN_packet = header.create_packet( seq: 0,  ack: 0,  flags: 8,  data: b'')
    clientSocket.sendto(SYN_packet, (server_ip, server_port))
    print('SYN packet is sent')

    packet_recived, serverAddress = clientSocket.recvfrom(2048)

    syn, ack, fin = header.parse_packet(packet_recived)

    if (syn, ack, fin) == (1, 1, 0):
        print('SYN-ACK packet is recived')

        ACK_packet = header.create_packet( seq: 0,  ack: 0,  flags: 4,  data: b'')
        clientSocket.sendto(ACK_packet, serverAddress)
        print('ACK packet is sent\nConnection established\n')

except socket.timeout:
    print('Connection failed')
    exit(1)
```

Figure 2 – Three-way handshake from client side.

When the server receives the last ACK it starts a timer that is used to calculate the throughput how much data is transferred and how long it took. Higher throughout makes a faster connection, more data in less time.

Transferring packets

The protocol makes sure that the server drops duplicate packets or any packet out of order. The client sets the sequence number (seq) of the packet and the server only accepts packet with expected seq which should be the same as the counter. See figure 3. If a discard packet is set the equivalent packet will be dropped one time. Then a Boolean variable will be set to false so the packet will not be

dropped again. If it is a in order packet, the header will be removed, and the data will be written to the file.

```
counter = 1
discard = True
total_data_recived =b''

with open(filename, 'wb') as f:

    while True:
        data_recived, clientAddress = serverSocket.recvfrom(2048)
        header_msg = data_recived[:6]
        seq, ack, flag = header.parse_header(header_msg)


        if seq == discard_packet and discard:
            discard = False

        elif seq == counter:
            f.write(data_recived[6:])
            print(datetime.now().time(),f' -- packet {seq} is received')
            counter += 1
            total_data_recived += data_recived
```

Figure 3 – Server side receiving packets: reads the header, drops out of order packet, discard packet or saves in order data.

On the client side a Go-back-N function (GBN) is used. It is not a stand-alone function in my code. The same number of packets as the window size will be sent without waiting for a ack form server, before it waits for an ack for all the packet sent. After that it will send one packet at a time, and only send a new one when it has received a ack. The packets have a timeout so that it will resend packets if it gets a timeout. See figure 4.

```
if(len(packets) >= windowsize):                                    # The first packets
    while seq_sent != ack_recived:                                 # Then the client w
        try:                                                       # As long as sequnc
            ack_packet_recived, serverAddress = clientSocket.recvfrom(2048) # the client resend
            header_msg = ack_packet_recived[:6]
            seq, ack, flag = header.parse_header(header_msg)       # If flags in reciv
            if flag == 4:
                print(datetime.now().time(),f' -- Ack for packet {ack} is recived')
                ack_recived.append(ack)

        except socket.timeout:
            print(datetime.now().time(),' -- RTO occured')         # If client does no
                                                                   # packets has been
            for i in seq_sent:                                     # wil be retransmit
                if i not in ack_recived:
                    packet = header.create_packet(i, ack: 0, flags: 0, packets[i])
                    clientSocket.sendto(packet, serverAddress)
                    print(datetime.now().time(), f' -- retransmitting packet with seq = {i}')
```

Figure 4 – Part of GBN function.

Connection tear down

When the client has reached the end of the file and don't have any more packets to send, it sends a FIN packet to the server. If the server receives a FIN it sends a FIN-ACK back to the client and the sockets closes. The server calculates throughput in Mega bits per second that is the amount of data that was transferred in the time it took to transfer, indicating the efficiency of the network.

Discussion:

1. Window size

The application was tested in Mininet using the simple-topo.py to create a realistic virtual network so that the transfer between to different hosts could be tested. First the window size was change to test if this effected the throughput. The transfer was tested with a window size of 3, 5 and 10, whit round trip time (RTT) set to 100 ms. See the results in figure 5, 6 and 7.

```
FIN packet is recived
FIN-ACK packet is sent
Connection closes
The througput is 0.07520718558337378 Mbps
root@UbuntuVM2:/home/solveig/Homeexam/src#
```

Figure 5 – Throughput when window size = 3 and RTT = 100 ms.

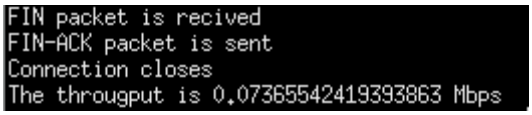

Figure 6 – Throughput when window size = 5 and RTT = 100 ms.

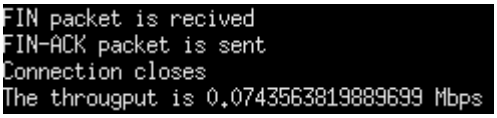

Figure 7 – Throughput when window size = 3 and RTT = 100 ms.

It was expected to get an increase in throughput as the window size was bigger because then the number of packets that is sent in the beginning without waiting for an ack would also increase. This is expected to increase the speed of the data transfer if the network can handle the increase in data. The results showed that the throughput decreased when the window was set from 3 to 5 but that it increases when the window was set to 10. Both 5 and 10 had a lower throughput then window size 3. The decrease of throughput could be a result of that the network capacity is full that can lead to network congestion or packet loss, but the server received the packets, and the client did not have to resend any packets. It would also be expected a decrease in throughput from window size 5 to 10 if that was the case, because then it would be even more packet loss or delay. So its not clear why the throughput is higher for window size = 3 than for any of the other window sizes.

2. Round trip time

To test how to RTT affected the throughput the RTT was changed to 50 ms and 200ms. This is the time it takes for a request to travel over the network and then the response to travel back. A shorter RRT makes an application more responsive (Amazon Web Services, u.å.). The window size was set to 3, 5 and 10. See results in figure 8, 9 and 10 for RTT = 50 ms, and figure 11, 12 and 13 for RTT = 200ms.

RTT set to 50 ms:

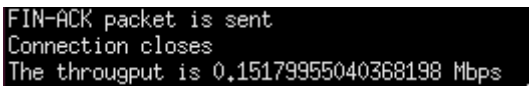

Figure 8 – Throughput when window size = 3 and RTT = 50 ms.

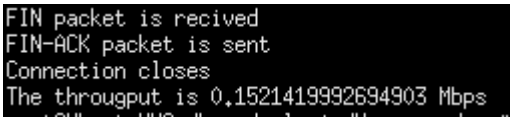

Figure 9 – Throughput when window size = 5 and RTT = 50 ms.

Figure 10 – Throughput when window size = 10 and RTT = 50 ms.

RTT set to 200 ms:

Figure 11 – Throughput when window size = 3 and RTT = 200 ms.

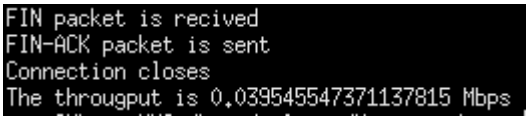

Figure 12 – Throughput when window size = 3 and RTT = 200 ms.

Figure 13 – Throughput when window size = 3 and RTT = 200 ms.

The results shows that a lower RTT (50 ms) makes the throughput increase a lot compare to the RTT set to 200 ms. The time it takes for the packet to be sent to the server and the server to respond with an ack back is much faster and therefore the whole transfer will take shorter time, making the throughput increase. It is also an increase based on the window size, where 10 packets has the highest throughput because it can send more packets before waiting for a respond which makes it faster, as mention in the last section, although there were some different results when just testing difference in window size.

3. Discard packet

To test what happens when a packet is lost the argument -d is used to drop a specific packet on the server side. When the packet is dropped from the server the client will not receive an ack and a timeout occurs, and the client will then retransmit the packet. This makes the transport protocol reliable because the client will make sure that any packet that is not received by the server, will be retransmitted. The client will retransmit until an ack for the specific packet is received before the client send the next packet. See figure 14. On the server side, it will also not accept any packet out of order, or duplicates making it a reliable protocol.

```
13:29:06.142138  -- packet with seq = 1835 is sent, sliding window = [1833, 1834, 1835]
13:29:06.643644  -- RTO occured
13:29:06.658606  -- retransmitting packet with seq = 1835
13:29:06.764684  -- Ack for packet 1835 is recived
13:29:06.764845  -- packet with seq = 1836 is sent, sliding window = [1834, 1835, 1836]
13:29:06.868728  -- Ack for packet 1836 is recived
13:29:06.868864  -- packet with seq = 1837 is sent, sliding window = [1835, 1836, 1837]
13:29:06.971171  -- Ack for packet 1837 is recived
DATA finished
```

Figure 14 – Printout from the client side, showing timeout for packet 1835 and retransmitting from client.

3698	191.059265521	10.0.0.1	10.0.1.2	UDP	1042	35874 → 8080 Len=1000
3699	191.163048275	10.0.1.2	10.0.0.1	UDP	48	8080 → 35874 Len=6
3700	191.163310237	10.0.0.1	10.0.1.2	UDP	1042	35874 → 8080 Len=1000
3701	191.670112991	10.0.0.1	10.0.1.2	UDP	1042	35874 → 8080 Len=1000
3702	191.771688175	10.0.1.2	10.0.0.1	UDP	48	8080 → 35874 Len=6
3703	191.771885721	10.0.0.1	10.0.1.2	UDP	1042	35874 → 8080 Len=1000
3704	191.875138224	10.0.1.2	10.0.0.1	UDP	48	8080 → 35874 Len=6

Figure 15 – Screenshot from Wireshark, that shows the retransmitted packet.

4. Packet loss

The application was also tested with a packet loss of 2 and 5% to test how effective the code was. The window size was set to 3 with RTT = 100 and the loss was set to 2% and 5% which means it drops random packets. The results shows that both a packet loss of 2 and 5 % has an impact on throughput with an increase in impact with higher packet loss. The 2 % loss had a small impact compared to the first test with no packet loss. This shows that increase in packet loss will affect the efficiency of the transfer because the more packets that must be retransmitted, the longer time must the client wait for an ack, and if a timeout occurs it must resend a packet. To have increase effectiveness it is important to reduce packet loss as much as possible.

```
FIN packet is recived
FIN-ACK packet is sent
Connection closes
The througput is 0.07199664313665323 Mbps
```

Figure 16 – Packet loss = 2%, window size = 3, RTT = 100 ms.

```
FIN packet is recived
FIN-ACK packet is sent
Connection closes
The througput is 0.059290283012556295 Mbps
```

Figure 17 – Packet loss = 5%, window size = 3, RTT = 100 ms.

5. Limitations of the project

When a connection is established, the server does not block any attempt to connect to the server. Which means another client can start sending packets. It might not interfere with the existing transfer since the packet number would be received as out of order, unless they connect at almost the same time which can corrupt the file. Therefore, it should block any connections when a connection is ongoing or adjust the code so that the server is able to handle multiple connections simultaneously. It should also be set more timeouts to make sure the connections do not stand and waits forever. For example, when the client sends a packet it will wait on an ack, and if the timeout occur it retransmit the packet, and it will continue to do so in a infinite loop. The server also waits for new packets infinitely until it receives a FIN. So if a connection is lost, it would still be listening for a packet.

References:

Amazon Web Services (u.å.). *What is RTT?* Retrieved May 19, 2024 from https://aws.amazon.com/what-is/rtt-in-networking/

GeeksForGeeks. (2024, 26. February). *User datagram Protocol(UDP).* https://www.geeksforgeeks.org/user-datagram-protocol-udp/

Islam, S. (2024, 02. February). *The transport Layer*. [PowerPoint presentasion]