DRTP

a reliable transport over UDP

Project report for portfolio exam in DATA2410

Candidate number: 139

1. Introduction

In this project, my aim is to design and develop the DATA2410 Reliable Transport Protocol (DRTP) - a simple reliable data transfer protocol, built upon UDP. It is specifically engineered to guarantee reliable data delivery in the correct sequence, eliminating any instances of missing or duplicate data.

My approach to the assignment was methodical, I concentrated on implementing one function at a time before moving on to the next. This strategy allowed me to isolate any potential issues and address them promptly, maintaining the reliability of the overall code.

The foundation for my code is based on previous assignments, lab exercises, as well as sample code provided on our Canvas learning platform, and within the eighth edition of Jim Kurose and Keith W. Ross book, "Computer Networking: A Top-Down Approach." Starting with familiar code enabled a more efficient development process. I could capitalize on existing, reliable solutions and modify them to meet the distinct requirements of the DRTP, incorporating my own understanding and insights into the process. Sikt KI-chat was used as a tool for identifying and resolving bugs in the code.

I utilized libraries available in Python, like the socket library, which offers the crucial functions needed for the UDP-based DRTP implementation. The argparse library facilitated the application of command-line arguments, while the struct library was used for packing and unpacking data into bytes.

The result is an executable file transfer application, application.py. This application is programmed to use command-line arguments to trigger a client and server. The client's role is to read a jpeg file from the system and dispatch it via DRTP/UDP. The filename, server address, and port number are all provided as command-line arguments. Additionally, you have the option to define a preferred window size for file transfer.

The server receives the file from the client and transfers it to the file system. The server accepts filename, server IP address and the listening port number as command-line arguments. It also includes a - - discard argument to activate a custom test case, which omits a sequence to verify retransmission. For instance, if -d 11 is entered on the server side, the server will disregard the packet with sequence number 11, but only once. The server also monitors the data volume received from a connected client and the duration of the data transfer. This feature enables the server to calculate the throughput, offering a look into the performance. It is calculated using this formula:

$$Duration = end\_time - start\_time$$

$$throughput = \frac{total\_data * 8}{1000 * 1000 * duration}$$

`Total_data` represents the total amount of Megabytes transferred. Multiplying `total_data` by 8 converts the total data from bytes to bits. The multiplication by 1000 * 1000 converts the bits to megabits. Duration measures the total time taken to transfer the data in seconds. So, dividing the `total_data` by the duration gives us the throughput in Mbps.

2. DRTP – a reliable transport over UDP

In this section, I'll go more into details on the implementation of my code, demonstrating how the application operates using code snippets for clarity. To enhance the manageability of the code, I have organized it into five separate files:
`application.py, server.py, client.py, header.py and utilities.py`

`application.py` is the command-line application that can function as either a server or a client for transferring a jpeg file over a network. The application uses the argparse library to interpret command-line arguments, guiding the execution of the script. The available command-line arguments are as follows

```
    parser.add_argument("-c", "--client", action="store_true", help="Run as client.")
    parser.add_argument("-s", "--server", action="store_true", help="Run as server.")
    parser.add_argument('-i', '--ip', type=str, default="10.0.1.2", help='Server IP address.')
    parser.add_argument("-f", "--file", type=str, help="File to transfer.")
    parser.add_argument('-p', '--port', default=8080, type=int, help='Server port number.')
    parser.add_argument('-w', '--window', default=3, type=int, help="The window size")
    parser.add_argument('-d', '--discard', type=int, help="Value of seq to skip")
```

Figure 1. Available command-line arguments

If the command-line arguments are valid, the script runs either the `server()` function or the `client()` function, depending on whether it should run as a server or a client.

`header.py` and `utilities.py` provides functions for creating, parsing, sending, and receiving packets over the socket connection. The protocol uses a custom 6-byte header that includes a sequence number, an acknowledgment number, and flags that can signify SYN, ACK, and FIN flags. The custom header format is defined in `header.py`

The `server.py` file is a basic UDP server application that receives data from a client and `client.py` is the client-side application, enabling file transfer via UDP. Both incorporate the DRTP protocol to ensure reliable data transmission. The server only handles one client at the time. To prevents other clients from connecting to the server while it's already serving a client it uses `serving_client` and `client_addr_serving` variables. When a client first connects to the server and sends a SYN packet, the `serving_client` variable is set to True and `client_addr_serving` is updated with the address of the connected client.

```
while True:
# Calls receive_packet function to receive a packet from the client.
    msg, client_addr, seq, ack, syn_flag, ack_flag, fin_flag = receive_packet(server_socket)

    #Test to see if the server already has a connection with a client
    if serving_client and client_addr != client_addr_serving:
        continue  # skip this loop iteration if another client is trying to connect

    if syn_flag:  # If the SYN flag of the received packet is set
        start_time = time.time()  # initialize start time
        serving_client = True  # set serving_client to True when a client connects
        client_addr_serving = client_addr  # keep track of the client being served
```

Figure 2. This code snippet illustrates the server's method of managing a single client at a time

For every incoming packet, the server checks if it's already serving a client and if the address of the packet sender matches the address of the client it's currently serving. If the server is indeed serving a client and the incoming packet is from a different client, the server skips the current iteration of the loop and doesn't process the packet. When the server receives a FIN packet from the connected client, indicating the client wants to close the connection, it sets `serving_client` back to False. And both the client and server close the connection.

DRTP start the reliable connection by using a method known as a three-way handshake, a process that resembles the one used in TCP. The client starts the process by sending an empty package with the SYN flag activated in the header. In response, the server sends a package with both SYN and ACK flags set, thereby setting up the connection. After receiving the SYN-ACK packet, the client dispatches an ACK to the server.

```
    syn_packet = create_packet(0, 0, 8, b'')# Packet with SYN flag.
    send_packet(client_socket, syn_packet, (args.ip, args.port))# Sending the packet with to the server.
    print(f'SYN packet it sent')
# Start of the three way handshake
    while True:
        try:
            msg, server_addr, seq, ack, syn, ack_flag, fin = receive_packet(client_socket)
            if syn and ack_flag:
                print("SYN-ACK packet is received")  # Print a message indicating a SYN-ACK packet is received
                ack_packet = create_packet(0, 0, 4, b'')  # Create a packet with ACK flag set
                send_packet(client_socket, ack_packet, server_addr)  # Send the ACK packet to complete the three-way handshake
                print("ACK packet is sent.")  # Print a message indicating an ACK packet is sent
                break
        except socket.timeout:
            print("\nConnection failed")
            exit(1)
        #End of the three way handshake
```

Figure 3. Three way handshake from client side

```
    if syn:  # If the SYN flag of the received packet is set
        start_time = time.time()  # initialize start time
        print("SYN packet is received.")  # Print a message indicating a SYN packet is received
        syn_ack_packet = create_packet(0, 0, 12, b'')  # Create a packet with both SYN and ACK flags set (8 | 4)
        send_packet(server_socket, syn_ack_packet, client_addr)  # Send the SYN-ACK packet back to the client
        print(f'SYN-ACK packet is sent')  # Print a message indicating a SYN-ACK packet is sent
```

```
    elif ack_flag:  # If the ACK flag of the received packet is set
        print(f'ACK packet is received')
        print('Connection established')
        continue  # Continue to the next iteration of the loop without executing the rest of the code in the loop
```

Figure 4. Three way handshake from client side

DRTP utilizes the Go-Back-N (GBN) reliability function to guarantee reliable data transmission. In a GBN protocol, the sender is allowed to transmit multiple packets (when available) without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, N, of unacknowledged packets in the pipeline (Kurose & Ross, 2022, p. 245).

The receiver monitors the sequence number of the next expected package, discarding any that don't match this number and acknowledges the last correctly received package. As long as the packet with the correct sequence number is received and acknowledged, the sender will slide the window forward and send the next packets. If a packet in the window is lost the receiver will not ACK this package. If the sender does not receive an ACK within a predetermined timeout, defaulted at 500 ms for DRTP, all packets in the window are assumed lost and subsequently retransmitted, as shown in figure 5.

Each packet in the data transfer phase is sequentially numbered, commencing from one. The sender reads data in 994-byte chunks and appends the 6-byte header prior to sending the data. The header contains a sequence number, an acknowledgment number, and flags, with only 4 bits of the flags being used for connection establishment, teardown, and acknowledgment.

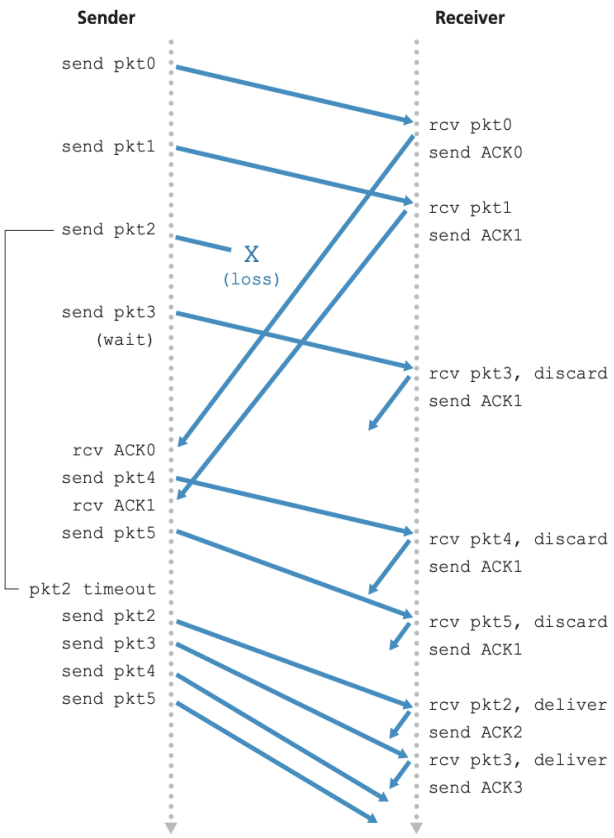

Figure 5. GBN in operation
(Kurose & Ross, 2022, p. 250)

Upon completing the data transmission, the sender initiates a connection teardown to terminate the connection. It dispatches a FIN flag to the server indicating the intent to close the connection. The server acknowledges this request by sending back a FIN-ACK flag. Once the server sends this acknowledgment, it then proceeds to close the connection.Even if the client does not receive a FIN-ACK flag, it will proceed to close the connection as well.

```
# Start of teardown
print('\nConnection Teardown:\n')
# Create a packet with the FIN flag set
fin_packet = create_packet(0, 0, 2, b'')
print('FIN packet is sent')
send_packet(client_socket, fin_packet, (args.ip, args.port))
while True:
    try:
        _, server_addr, seq, ack, syn, ack_flag, _ = receive_packet(client_socket)
        if ack_flag:
            print("FIN-ACK packet is received.")
            print('Connection closes')
            client_socket.close()
            break
    except socket.timeout:
        print("Timeout waiting for FIN-ACK packet. Closing the connection.")
        client_socket.close()
        break
#Teardown complete
```

Figur 6. Teardown from client side

3. Discussion

3.1 Execute the file transfer application with window sizes of 3, 5, and 10. Calculate the throughput values for each of these configurations and provide an explanation for your results. For instance, discuss why you observe an increase in throughput as you increase the window size

The file transfer application was executed in mininet with window sizes of 3, 5, and 10. The throughput values were as follows:

Window size 3: Throughput = 0.23 Mbps

Window size 5: Throughput = 0.38 Mbps

Window size 10: Throughput = 0.73 Mbps

From these results, it's clear that as the window size gets bigger, the throughput also increases. This is because of how the Go-Back-N protocol works. In GBN, the window size tells us how many packets we can have in flight before we need to stop and wait for an acknowledgment. If we have a bigger window, we can send more packets at the same time. This means we're making better use of our network, because we've always got packets on the go, rather than having to stop and wait for acknowledgments all the time. The Stop and Wait Protocol is a simpler method of flow control where each packet waits for an acknowledgement before the next packet is sent. Therefore, GBN can potentially achieve much higher throughput than the Stop and Wait Protocol, especially for larger window sizes. And because we're using our network more efficiently, we can send data faster, resulting in an increased throughput.

It's important to note that while a larger window size can improve throughput, it may also lead to a higher probability of packet loss in an unstable network. A single packet error can thus cause GBN to retransmit a large number of packets, many unnecessarily (Kurose & Ross, 2022, p. 250). The optimal window size may depend on the specific network conditions. If the network bandwidth is high enough to accommodate the increased number of packets put into transit by a larger window size, then the throughput will likely increase. But, if the network bandwidth is limited, increasing the window size might decrease the throughput. This is because the network could become congested due to the high number of packets in transit, leading to increased packet loss and retransmissions. In conditions of

limited bandwidth, an optimal balance between window size and bandwidth must be achieved to maximize throughput.

I experimented with the window size in mininet to try to identify the optimal window size by increasing it incrementally. I found that at a window size of approximately 75, I was able to consistently achieve a throughput of ca. 5.0 Mbps. When I increased the window size beyond this, the throughput showed significant fluctuation. When the window size was set to 200, the highest throughput achieved was 12.85 Mbps, although it sometimes dropped to as low as 2 Mbps. When I increased the window size to 400, the throughput showed high variability, ranging from 19 Mbps to under 2 Mbps. And in some instances, the delivery couldn't be completed at this window size, and seemed to hang after several minutes, necessitating an abortion of the process.

3.2 Modify the RTT to 50ms and 200ms. Run the file transfer application with window sizes of 3, 5, and 10. Calculate throughput values for all these scenarios and provide an explanation for your results.

The tests were executed in mininet with varying Round-Trip Times of 50ms and 200ms, and window sizes of 3, 5, and 10. The corresponding throughput values for each were as follows:

RTT 50ms:
Window size 3: Throughput = 0.45 Mbps

Window size 5: Throughput = 0.75 Mbps

Window size 10: Throughput = 1.35 Mbps

RTT 200ms:
Window size 3: Throughput = 0.12 Mbps

Window size 5: Throughput = 0.20 Mbps

```
FIN packet is received.
FIN ACK packet is sent

The throughput is 0.20 Mbps
Connection Closes
```

Window size 10: Throughput = 0.39 Mbps

```
FIN packet is received.
FIN ACK packet is sent

The throughput is 0.39 Mbps
Connection Closes
```

These results clearly show that for a given window size, the throughput decreases as the RTT increases. A longer RTT means the sender has to wait longer for acks from the receiver before it can advance the window and send more packets. This waiting time reduces the rate at which data can be sent, thereby reducing the throughput.

Additionally, the results also confirm the earlier observation that larger window sizes result in higher throughput. But, if a packet is lost and the window needs to be retransmitted, a high RTT can cause a significant delay. These results underline the importance of tuning the window size according to the network conditions. For a network link with a high bandwidth-delay product, a larger window size is needed to fully utilize the link's capacity.

3.3 Use the --discard or -d flag on the server side to drop a packet, which will make the client resend it. Show how the reliable transport protocol works and how it deals with resent packets.

The --discard flag can be used during the server's initialization to set a specific packet sequence number to be discarded. During the packet receiving loop, when a packet arrives with the sequence number that matches discard_seq, the packet is discarded and not processed by the server.

```
elif seq == expected_seq:  # If the sequence number is the expected one
    if seq == discard_seq:
        discard_seq = None  # Reset the discard value so the packet isn't skipped again
        continue  # Skip the rest of the loop for this packet
```

Figure 7. Codesnippet showing what happenes if discard_seq is set

When the server receives an out of order packet it discards the packet and resends an ACK for the most recently received in-order packet.((Kurose & Ross, 2022, p. 248). The client will assume the packet was lost during transmission and a timeout will eventually occur. When the client resends the discarded packet, the server can process it correctly since `discard_seq` has been reset to None. An ACK will be sent back to the client, and the server can continue receiving the rest of the packets.

In the following server output examples, the packet set to be discarded has sequence number 1823 and the window size is set to five.

Figure 8. Output from the client side when discarding packet 1823

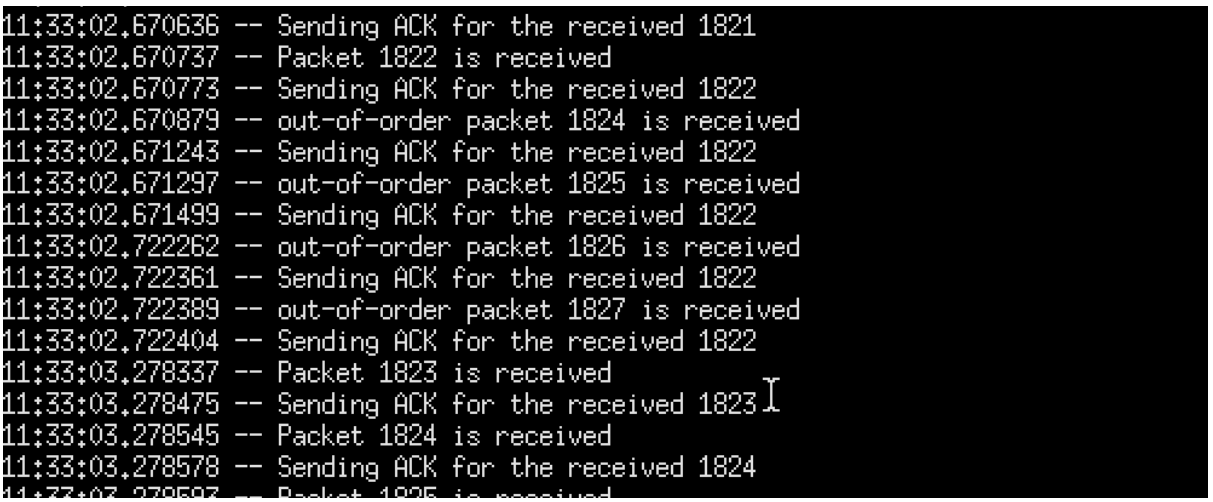

Figure 9. Output from the server side when discarding packet 1823

The provided examples illustrate that all packets within the window are retransmitted, following the principles of the Go-Back-N protocol.

4. To demonstrate how effective your code is, use tc-netem (refer to the lab manual) to simulate packet loss. Test with a loss rate of 5% as well.

With a delay of 100 ms and a 2% loss rate:

- Window size 3: Throughput = 0.17 Mbps
- Window size 5: Throughput = 0.27 Mbps
- Window size 10: Throughput = 0.37 Mbps

With a delay of 100 ms and a 5% loss rate:

- Window size 3: Throughput = 0.12 Mbps
- Window size 5: Throughput = 0.16 Mbps
- Window size 10: Throughput = 0.20 Mbps

From the results, we can see that increasing the window size improves throughput, even when some packets are lost. But it also shows that the higher the packet loss, the lower the throughput becomes. This is because lost packets need to be resent, consuming both time and network resources, which in turn slows down the overall data transfer. High packet loss rates can significantly degrade performance, posing a potential limitation in environments with unstable network conditions.

If we were to look at how we could improve performance for DRTP an adaptive window size could be more beneficial. If the network becomes congested, the window size can be reduced to decrease the number of packets in transit and hence the likelihood of packet loss. If the network conditions improve, the window size can be increased to take advantage of the improved conditions and increase throughput. But unlike TCP, the UDP protocol lacks built-in congestion control and applications using UDP must implement their own congestion control strategies.

DRTP manages timeouts and retransmissions by retransmitting all the packets in the window, complying with GBN guidelines. This may not always be the most effective approach, and employing a more sophisticated retransmission strategy, like Selective Repeat(SR), could potentially enhance performance. Selective-repeat protocols avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver. (Kurose & Ross, 2022, p. 251). SR can be more efficient than GBN in situations where the network has a high packet loss rate. By only retransmitting the lost packets instead of the entire window, it can potentially save network bandwidth and improve overall performance.

For DRTP, the primary focus is on the verification of whether the expected sequence number is being sent. This is a crucial aspect of ensuring the correct order and completeness of data transmission. But, this does not include functionality to inspect for corrupted data. While the code ensures data is received in the correct order, it doesn't validate the integrity of the data content itself. If a packet becomes corrupted during transmission, this might go unnoticed in the current setup.

Lastly, while the code inherently includes some error checking, incorporating more comprehensive error checking could further enhance the reliability of the application.

4. References

Kurose, J. F., & Ross, K. W. (2022). Computer networking: A top-down approach (Eighth edition, global edition). Pearson.