

UDP File Transfer With DRTP

Faculty of Technology, Art and Design (TKD)

DATA2410 Spring of 2024

Individual Home Exam Report

Candidate: 124

University: OsloMet

Table of content

UDP File Transfer With DRTP.....	1
Table of content.....	2
Introduction.....	3
Implementation.....	3
UDP.....	3
Header format and flags.....	4
Connection Establishment Process.....	4
Sliding Window (Go-Back-N) Implementation:.....	5
File Transfer.....	6
Connection teardown.....	6
Throughput Calculation.....	7
Command Line Arguments and Error Handling.....	7
Discussion.....	8
Question 1: Window Sizes.....	8
Test result.....	9
Question 2: Round-Trip Time.....	9
Test result.....	10
Question 3: --discard.....	10
Test result.....	10
Question 4: Packet Loss.....	11
Test result.....	11
Conclusion.....	12
References.....	12

Introduction

This Python application allows a file transfer between a client and a server using User Datagram Protocol (UDP). Users can specify the operation mode, server's IP and port, filename, window size, and a sequence number for discard testing through command-line inputs. The file set for transfer is segmented into chunks at the client end. Each chunk is then encapsulated within a Data Reliable Transfer Protocol (DRTP) header before being dispatched to the server.

To guarantee data reliability, the application uses a sliding window protocol. This allows for the retransmission of any unacknowledged packets. Additionally, an out-of-order packet buffer is implemented on the server side to ensure proper data sequencing.

The application calculates and displays the file transfer's throughput to provide a measure of data transmission speed.

Implementation

The project is structured as follows:

- README.md: Contains instructions on project setup and usage.
- application.py: This is the main entry point of the application.
- client.py: This script contains the implementation of the client side of the application.
- server.py: This script contains the implementation of the server side of the application.
- utils.py: This script includes utility functions used by both the client and server scripts.
- img: This directory contains the jpg file that will be transferred.
- topologies: This directory contains the network topology.

The next section provides a breakdown of the various parts of the code:

UDP

User Datagram Protocol (UDP) is a connectionless protocol used for data transmission in networks. Unlike TCP, UDP doesn't require an established connection before communication begins, making it an unreliable data transfer service. Messages sent via UDP may not reach the receiver or could arrive out of order.

In the application, UDP is used for file transfer. Client and server are set up using Python's socket library, which creates a socket object for the IPv4 addressing:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # UDP
```

Here, `socket.AF_INET` refers to the IPv4 address family, and `socket.SOCK_DGRAM` signifies a UDP socket.

The server's UDP socket is created and bound to a specific IP address and port:

```
# Create a UDP socket and bind it to the IP and port  
sock = init_socket(UDP_IP, UDP_PORT)
```

This setup allows the server to receive packets sent to that IP and port.

Header format and flags

The DRTP is created with a !HHH format, implying three fields of 2-byte. The purpose of this header is to provide necessary information about the data packet being transmitted. These fields include:

- Sequence number: Used for ordering the packets correctly.
- Acknowledgement number: Used to confirm the receipt of packets.
- Flags: Indicate the state of the packet (whether it's an ACK, SYN, or FIN).

The Data Packet headers and flags are defined in the utils.py file:

```
header_format = '!HHH' # sequence number, acknowledgment number, and flags
ACK_FLAG = 1 << 2 # Flag for ACK signal
SYN_FLAG = 1 << 3 # Flag for SYN signal
FIN_FLAG = 1 << 1 # Flag for FIN signal
```

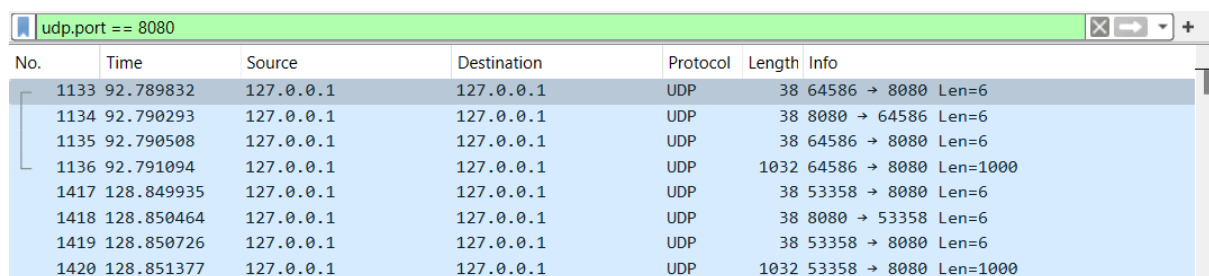
These flags, along with sequence and acknowledgement numbers in the header, play a crucial role in establishing and maintaining the reliability of data transfer over UDP.

ACK_FLAG allows the receiver to communicate to the sender that a packet has been received correctly, enabling the sender to keep track of packets that need to be retransmitted. SYN_FLAG and FIN_FLAG manage the connection state between the client and server, signalling the start and end of a data transfer session, respectively.

By using these flags and numbers, our application can handle packet reordering, acknowledge packet receipt, and manage data transfer sessions, thereby providing reliable data transfer over an inherently unreliable UDP protocol.

During the connection establishment phase, a SYN packet is created and sent to the server:

```
sock.sendto(syn_header, (UDP_IP, UDP_PORT))
```



The image shows a Wireshark packet capture window with a filter set to 'udp.port == 8080'. The packet list shows several packets. The first packet (No. 1133) is a SYN packet from 127.0.0.1 to 127.0.0.1 on port 8080. The subsequent packets (1134, 1135, 1136, 1417, 1418, 1419, 1420) are data packets of varying lengths (6, 1000, 6, 6, 6, 6, 1000 bytes) sent from 127.0.0.1 to 127.0.0.1 on port 8080.

No.	Time	Source	Destination	Protocol	Length	Info
1133	92.789832	127.0.0.1	127.0.0.1	UDP	38	64586 → 8080 Len=6
1134	92.790293	127.0.0.1	127.0.0.1	UDP	38	8080 → 64586 Len=6
1135	92.790508	127.0.0.1	127.0.0.1	UDP	38	64586 → 8080 Len=6
1136	92.791094	127.0.0.1	127.0.0.1	UDP	1032	64586 → 8080 Len=1000
1417	128.849935	127.0.0.1	127.0.0.1	UDP	38	53358 → 8080 Len=6
1418	128.850464	127.0.0.1	127.0.0.1	UDP	38	8080 → 53358 Len=6
1419	128.850726	127.0.0.1	127.0.0.1	UDP	38	53358 → 8080 Len=6
1420	128.851377	127.0.0.1	127.0.0.1	UDP	1032	53358 → 8080 Len=1000

Img: SYN, SYN_ACK and ACK in Wireshark (Anonymous, 2024)

Connection Establishment Process

Establishing a connection between the client and server is essential for ensuring the reliable transfer of data. This process is carried out through a three-way handshake, which involves the exchange of the flags mentioned above. The three-way handshake is a core concept borrowed from TCP to add reliability to connectionless protocols like UDP.

In this process:

1. The client initiates the connection by sending a SYN_FLAG packet.
2. The server acknowledges this by sending back a SYN_FLAG and ACK_FLAG packet.
3. The client confirms the server's response by sending an ACK_FLAG packet.

This is implemented in the `handle_connection` function in the client script:

```
def handle_connection(sock, buffer_size, server_ip, server_port):
    ...
    if flags == (SYN_FLAG | ACK_FLAG): # Check for SYN-ACK flag
        print("SYN-ACK packet is received")
    ...
    print("Connection established\n")
    return True
```

Upon receiving the `SYN_FLAG` from the client, the server acknowledges by sending an `SYN_FLAG` and `ACK_FLAG`, handled in the `handle_syn` function in the server script:

```
def handle_syn(sock, addr):
    print("SYN packet is received")
    syn_ack_header = struct.pack(header_format, 0, 0, SYN_FLAG | ACK_FLAG)
    sock.sendto(syn_ack_header, addr)
    print("SYN-ACK packet is sent\n")
```

Sliding Window (Go-Back-N) Implementation:

The Go-Back-N sliding window protocol enhances data transmission efficiency, by allowing multiple packets to be in-flight simultaneously. This approach reduces acknowledgement waiting times and boosts network resource utilisation. If a packet is lost, all subsequent packets are discarded and retransmitted, starting from the lost one.

The client function implements this protocol as follows:

```
# Begin the data transfer phase
base = 1
nextseqnum = 1
frame_buffer = [None] * WINDOW_SIZE
window_packets = []

# While there are chunks left to send
while base <= len(file_chunks):
    # Send all the chunks within the window
    while nextseqnum < base + WINDOW_SIZE and nextseqnum <= len(file_chunks):
        ...
        window_packets.append(nextseqnum)
        nextseqnum += 1
        ...
    try:
        data, _ = sock.recvfrom(BUFFER_SIZE)
        ack, _, _ = struct.unpack('!HHH', data)
        # If the ack is within the window, move the base of the window
        if ack >= base and ack < nextseqnum:
            window_packets = [seq for seq in window_packets if seq > ack]
            base = ack + 1
        ...
```

In this code:

- base is the sequence number of the oldest unacknowledged packet
- nextseqnum is the sequence number of the next packet to send
- frame_buffer and window_packets respectively hold the packets and their sequence numbers in the window

The client will continue to send packets until the window is full. Upon receiving an acknowledgement (ack), it checks if it's within the window. If so, all packets with sequence numbers less than or equal to ack are removed from the window, and the base is updated. In case of packet loss, all window packets are retransmitted.

File Transfer

The process of file transfer involves both the client and the server. On the client side, the file to be transferred is read and divided into chunks. Each chunk is then wrapped in a DRTP packet and sent to the server. On the server side, the incoming packets are processed, and the data chunks are extracted and written into a file, reconstructing the original file sent by the client.

Here is the code for the client and server:

```
# Client
file_data = read_file_data(args.file)
chunk_size = MAX_PACKET_SIZE - header_size
file_chunks = [file_data[i:i+chunk_size] for i in range(0, len(file_data),
chunk_size)]

# Server
if data_received:
    ...
    write_chunks_to_file(file_chunks)
```

Connection teardown

The application employs a two-way handshake protocol for the connection teardown process, slightly differing from the three-way handshake used in the connection establishment phase.

The client will send a FIN_FLAG packet to the server to signal the end of the data transfer when all packets are sent. The server responds with an ACK_FLAG packet to acknowledge the receipt of the FIN_FLAG. This process is implemented in the client script after all packets have been sent:

```
while base <= len(file_chunks):

# After all packets are sent, begin the connection teardown phase
fin_header = struct.pack(header_format, 0, 0, FIN_FLAG)
sock.sendto(fin_header, (UDP_IP, UDP_PORT))
print(f"{datetime.now().strftime('%H:%M:%S.%f')} -- FIN packet is sent")
# Wait for the final acknowledgement, then close the connection
data, _ = sock.recvfrom(BUFFER_SIZE)
if data == b'ACK':
    ...
    sock.close()
```

The server script listens for incoming FIN_FLAG packets from the client in the handle_fin function. Upon receiving a FIN_FLAG, it responds with an ACK_FLAG and closes the connection as well:

```
def handle_fin(sock, buffer_size, throughput_mbps):
    data, addr = sock.recvfrom(buffer_size)
    _, _, flags = struct.unpack(header_format, data)
    if flags == FIN_FLAG:
        print("\nFIN packet is received")
        ack_header = struct.pack(header_format, 0, 0, ACK_FLAG)
        sock.sendto(ack_header, addr)
        ...
        sock.close()
```

Future implementation of a three-way handshake can further increase the reliability of the teardown process, like with the connection establishment phase. This could prevent potential connection issues if the final ACK_FLAG packet gets lost.

Throughput Calculation

Throughput is a key performance metric that measures the speed and efficiency of data transfer. It's calculated by dividing the total amount of data transferred (in bits) by the time it took for the transfer. The result is expressed in Mbps.

The calculate_throughput function calculates the throughput at the end of the data transfer.

```
def calculate_throughput(elapsed_time, file_size_bits):
    throughput = file_size_bits / elapsed_time
    throughput_mbps = round(throughput / 1000000, 2)
    return throughput_mbps

if data_received:
    end_time = time.time()
    elapsed_time = end_time - start_time
    total_file_size = sum(len(chunk) for chunk in file_chunks)
    file_size_bits = total_file_size * 8
    throughput_mbps = calculate_throughput(elapsed_time, file_size_bits)
    write_chunks_to_file(file_chunks)
    handle_fin(sock, BUFFER_SIZE, throughput_mbps)
```

Command Line Arguments and Error Handling

The get_args and validate_args functions in the utility file manage the handling of command line arguments and some error handling. These functions use the argparse library to parse and validate the arguments provided through the command line when the program is executed.

The get_args function specifically handles the task of parsing these command line arguments:

```
def get_args():
    parser = argparse.ArgumentParser(description='UDP client and server')
    ...
    return parser.parse_args()
```

The command line arguments are as follows:

- -s/--server: Enables server mode
- -c/--client: Enables client mode.
- -i/--ip IP Address: Binds the IP address on the server-side and specifies the server's IP on the client-side (default: 10.0.1.2).
- -p/--port Port Number: Selects the server port number within the range [1024, 65535] (default: 8088).
- -f/--file File Name: Selects the jpg file for transmission.
- -w/--window Window Size: Sets the sliding window size (default: 3).
- -d/--discard [Seq Number]: Test case flag. The server discards the packet with the provided sequence number once to test retransmission.

Error handling is performed by the `validate_args` function, which checks for specific conditions and outputs an error message if these conditions are not met, halting the program.

```
def validate_args(args):
    if args.server and args.client:
        print('Error: Cannot run the application in both server and client mode.
Please choose one.')
        exit(1)
    ...
```

This function checks for the following conditions:

- The application is not run in both server and client mode at the same time.
- The port number is within the valid range of 1024-65535.
- The window size is within the valid range of 1-100.
- A file is specified when the application is run in client mode.
- A file is not specified when the application is run in server mode.

Although the current error handling covers several scenarios, it would be beneficial to cover more scenarios for a real-world application.

Discussion

The application's performance and reliability are evaluated through a set of tests conducted in Mininet, using the `'simple-topo.py'` script. The application's efficiency is measured by calculating the throughput value during these tests.

Question 1: Window Sizes

The initial set of tests involved executing the file transfer application with varying window sizes - 3, 5, and 10. The throughput value of these tests was recorded and analysed.

This modification is expected to enhance the efficiency of the data transfer process as the window size increases, due to the ability to transmit more packets simultaneously, reducing the overall wait time and making better use of network bandwidth.

The output from the client was as follows:

```
> python application.py -c -i 127.0.0.1 -f ./img/iceland_safiquel.jpg -w 5

Client started...
Connection Establishment Phase:

SYN packet is sent
SYN-ACK packet is received
ACK packet is sent
Connection established

Data Transfer:

12:50:36.049713 -- packet with seq = 1 is sent, sliding window = [1]
12:50:36.049713 -- packet with seq = 2 is sent, sliding window = [1, 2]
12:50:36.049713 -- packet with seq = 3 is sent, sliding window = [1, 2, 3]
12:50:36.049713 -- packet with seq = 4 is sent, sliding window = [1, 2, 3, 4]
12:50:36.049713 -- packet with seq = 5 is sent, sliding window = [1, 2, 3, 4, 5]
12:50:36.049713 -- ACK for packet 1 received
12:50:36.049713 -- packet with seq = 6 is sent, sliding window = [2, 3, 4, 5, 6]
12:50:36.049713 -- ACK for packet 2 received
```

Test result

Window Size	Throughput
3	0,24 Mbps
5	0,39 Mbps
10	0,77 Mbps

The test results confirm that an increase in window size from 3 to 5 to 10 leads to significant improvements in throughput. This aligns with the expected outcome, as a larger window size allows the application to send more packets without waiting for an acknowledgement from the receiver.

Question 2: Round-Trip Time

The next phase of testing involved manipulating the Round-Trip Time (RTT). The RTT was adjusted to 50ms, 100ms, and 200ms. This change affects the expected time for a signal to travel from its origin to its destination and back. This alteration is likely to result in a less efficient data transfer as the RTT value increases.

The test was conducted as follows, with 50ms, 100ms and 200ms in the file simple-topo.py.

```
# this adds a delay of 100ms - if you ping h2 from h1, the RTT will be 100ms
net["r"].cmd("tc qdisc add dev r-eth1 root netem delay 100ms")
```

Test result

Window Size	50ms	100ms	200ms
3	0,45 Mbps	0,24 Mbps	0,12 Mbps
5	0,75 Mbps	0,39 Mbps	0,2 Mbps
10	1,49 Mbps	0,77 Mbps	0,39 Mbps

The test results indicate that the Round-Trip Time (RTT) significantly affects network performance and influences the application's throughput. As the RTT increases, the throughput decreases due to longer delays for packet acknowledgements, which consequently slow down the packet transmission rate.

These results also highlight the significance of both window size and RTT. They confirm that a larger window size can increase throughput as in task one, and that a larger window size can be covalent to a faster RTT.

Question 3: --discard

The application's reliable transport protocol was tested by intentionally discarding a packet on the server side using the --discard or -d flag. This tested the protocol's ability to handle packet loss and retransmission.

The test was executed with a variety of window sizes and discard parameters. The test of focus here was executed using the following command:

```
python application.py -s -d 8
python application.py -c -f ./img/iceland_safiquel.jpg -w 5
```

Test result

```
-- RTO occurred
-- retransmitting packet with seq = 8
-- retransmitting packet with seq = 9
-- retransmitting packet with seq = 10
-- retransmitting packet with seq = 11
-- retransmitting packet with seq = 12
-- ACK for packet 8 received
-- packet with seq = 13 is sent, sliding window = [9, 10, 11, 12, 13]
-- ACK for packet 13 received
-- packet with seq = 14 is sent, sliding window = [14]
-- packet with seq = 15 is sent, sliding window = [14, 15]
-- packet with seq = 16 is sent, sliding window = [14, 15, 16]
-- packet with seq = 17 is sent, sliding window = [14, 15, 16, 17]
-- packet with seq = 18 is sent, sliding window = [14, 15, 16, 17, 18]
-- ACK for packet 14 received
-- packet with seq = 19 is sent, sliding window = [15, 16, 17, 18, 19]

# server
-- Discarding packet with sequence number 8
```

The output indicates that a Retransmission Timeout (RTO) took place, prompting the retransmission of packets bearing sequence numbers from 8 to 12. Once the Acknowledgment (ACK) for packet 8 was received, packet 13 was sent and the sliding window was implemented. This pattern continued up to packet 19. Meanwhile, on the server side, the discard of packet 8 was confirmed.

The test results highlight the reliability of the transport protocol. When a packet is discarded, the client side rightly identifies the loss of acknowledgement and triggers retransmission. This test underscores the application's efficient handling of packet loss by retransmitting lost packets, ensuring reliable data transfer.

Question 4: Packet Loss

The robustness of the application is tested against packet loss by using tc-netem to simulate different loss rates. This allows specifying the percentage of packets that are randomly dropped during transmission. The loss rates tested are 2%, 5%, 20% and an extreme case of 50%.

The loss rate determines the percentage of packets that will be randomly dropped during transmission. A higher loss rate means more packets are dropped and need to be resent, which is expected to significantly decrease the throughput. The test also evaluates the reliability of the system under various conditions.

The command utilized to simulate packet loss in the simple-topo.py file was as follows:

```
# this adds a delay of 100ms - if you ping h2 from h1, the RTT will be 100ms
#net["r"].cmd("tc qdisc add dev r-eth1 root netem delay 100ms")
net["r"].cmd("tc qdisc add dev r-eth1 root netem delay 100ms loss 2%")
```

Test result

PL Prosent	Throughput
2%	0.13 Mbps
5%	0.12 Mbps
20%	0.06 Mbps
50%	0.02 Mbps

The results confirm a reliable transfer protocol. Despite a 50% packet loss, the application maintained reliable, albeit slower due to frequent retransmissions, achieving a throughput of 0.01 Mbps over roughly 20 minutes.

The tests emphasised the application's stability in handling varying packet loss rates, with a decreased throughput as expected.

"Node: h2"	"Node: h1"
14:51:50.837849 -- sending ack for the received 1827	14:51:55.246207 -- packet with seq = 1836 is sent, sliding window = [1834, 1835, 1836]
14:51:52.940700 -- packet 1828 is received	14:51:56.246536 -- RTO occurred
14:51:52.940782 -- sending ack for the received 1828	14:51:56.246617 -- retransmitting packet with seq = 1834
14:51:52.941909 -- packet 1830 is received	14:51:56.246638 -- retransmitting packet with seq = 1835
14:51:52.941969 -- sending ack for the received 1830	14:51:56.246651 -- retransmitting packet with seq = 1836
14:51:53.041224 -- packet 1831 is received	14:51:57.247917 -- RTO occurred
14:51:53.041476 -- sending ack for the received 1831	14:51:57.247998 -- retransmitting packet with seq = 1834
14:51:53.141304 -- out-of-order packet 1834 is received	14:51:57.248018 -- retransmitting packet with seq = 1835
14:51:54.142930 -- packet 1832 is received	14:51:57.248030 -- retransmitting packet with seq = 1836
14:51:54.143191 -- sending ack for the received 1832	14:51:58.248476 -- RTO occurred
14:51:54.243836 -- out-of-order packet 1835 is received	14:51:58.248556 -- retransmitting packet with seq = 1834
14:51:55.245544 -- packet 1833 is received	14:51:58.248573 -- retransmitting packet with seq = 1835
14:51:55.245730 -- sending ack for the received 1833	14:51:58.248587 -- retransmitting packet with seq = 1836
14:51:58.348743 -- packet 1836 is received	14:51:58.349031 -- ACK for packet 1836 received
14:51:58.348988 -- sending ack for the received 1836	14:51:58.349251 -- packet with seq = 1837 is sent, sliding window = [1837]
14:51:59.450902 -- packet 1837 is received	14:51:59.350560 -- RTO occurred
14:51:59.451005 -- sending ack for the received 1837	14:51:59.350546 -- retransmitting packet with seq = 1837
FIN packet is received	14:51:59.463663 -- ACK for packet 1837 received
FIN ACK packet is sent	DATA Finished

50 percent packet loss (Anonymous, 2024)

Conclusion

The Python application used in this project is built around the User Datagram Protocol (UDP) to facilitate file transfers between a client and a server. It includes a sliding window protocol and a three-way handshake mechanism for reliable data transfer.

After a series of tests, the application proved reliable and efficient under different conditions. It demonstrated improved throughput with larger window sizes, effectively managed packet loss, and responded well to differences in Round-Trip Time (RTT).

There is also potential for enhancement as increasing the scope of error handling and incorporating a three-way handshake on the connection teardown could add more reliability as this could prevent connection problems if the final ACK_FLAG packet is lost.

In conclusion, this UDP-based Python application is a reliable solution for file transfers, with potential for further improvement and optimization.

References

To stay anonymous the GitHub repository for Oblig 1 and 2 are not directly linked in this document.

Anonymous. (Feb 29). *Oblig 1* [Python].

Anonymous. (April 4). *Oblig 2* [Python].

Argparse Tutorial. (n.d.). Python Documentation. Retrieved 28 April 2024, from

<https://docs.python.org/3/howto/argparse.html>

F. Kuros, J., & W. Ross, K. (2022). *Computer Networking—A Top-Down Approach: Vol. Eighth edition*.

Islam, S. (2023). *Header* [Python]. <https://github.com/safiqul/drtp-oppgave> (Original work published 2024)

Islam, S. (2024). *Safiqul/drtp-oppgave* [Python]. <https://github.com/safiqul/drtp-oppgave> (Original work published 2024)

UdpCommunication—Python Wiki. (n.d.). Retrieved 28 April 2024, from

<https://wiki.python.org/moin/UdpCommunication>