Project Report

COEN 366

Date

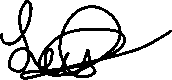
17 - 12 – 2023

By

Michael Hong & Luis Ramirez

4020 9068 & 4021 3391

**“*We certify that this submission is my original work and meets the Gina Cody School’s Expectations of Originality.”***



**Signatures:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Table of Contents

[Introduction 3](#_Toc153547811)

[Organization 3](#_Toc153547812)

[Basic functionalities 3](#_Toc153547813)

[Server 3](#_Toc153547814)

[Client 4](#_Toc153547815)

[Flow of the program 5](#_Toc153547816)

[TCP and UDP Transfer implementation 8](#_Toc153547817)

[TCP 8](#_Toc153547818)

[UDP 9](#_Toc153547819)

[Wireshark-related 11](#_Toc153547820)

[Questions for TCP: 11](#_Toc153547821)

[Questions for UDP: 15](#_Toc153547822)

[Conclusion 18](#_Toc153547823)

# Introduction

This report details the implementation and overview of our project in which we implement a simple file transfer protocol in which one can send files of different types between a client and a server, using either TCP or UDP sockets through Python. TCP ensures reliable, in-order and error-checked delivery of files, which UDP provides a faster but connectionless approach which is less reliable, however.

# Organization

In order to test the communication between the server and the client, we create two different directories. The first one in a “client” directory in which the script for the client is, and the second one another folder “server” in which the script for the server is. This ensure that the client and the server are “independent”, in the sense that they will not share files, and will each have their own “storage”.

## Basic functionalities

The basic functionalities that are supported by the file transfer service are the following:

|  |  |
| --- | --- |
| put <filename> | Uploads a file from the client to the server |
| get <filename> | Downloads a file from the server to the client |
| summary <filename> | Downloads a file with the max, min, and average of the numbers in a file in the server. |
| change <oldFilename> <newFilename> | Renames a file on the server side. |
| help | Gets a list of supported command from the server |
| bye | Stop client connection with the server and client exits program. |

## Server

The server is responsible for handling client requests. Depending on the type of request, it will either store a file into its own directory (“storage”) or take a file from its own directory and send it to the client. Below is an outline of the different responses that the server can send.

|  |  |  |
| --- | --- | --- |
| **Code** | **Purpose** | **Contains** |
| 000 | Indicates that the file was successfully uploaded to the server or the filename change was successful. | * Response code |
| 001 | Indicates that the file was successfully fetched from the server. | * Response code * Filename * Filename Length * File size * File Data |
| 010 | Indicates that the client will receive a file with a summary containing the max, min and average of the number file in the server. | * Response code * Filename * Filename Length * File size * File Data |
| 011 | Indicates to the client that the file requested was not found in the server | * Response Code |
| 100 | Indicates to the client that the request is not known or understood by the server | * Response code |
| 101 | Indicates to the client that the change request to modify the file name was not successful | * Response code |
| 110 | Sends to the client the list of available instructions | * Response code * Length * Data containing the available instructions |

## Client

The client is responsible for handling user input from the user, parse that user input and create a request that can be then interpreted by the server. It is also responsible for receiving the response from the server, parsing it, and creating the required files in the case of a get request.

|  |  |  |
| --- | --- | --- |
| **Code** | **Purpose** | **Contains** |
| 000 | Indicates that the file was successfully uploaded to the server or the filename change was successful. | * Response code |
| 001 | Indicates that the file was successfully fetched from the server. | * Response code * Filename * Filename Length * File size * File Data |
| 010 | Indicates that the client will receive a file with a summary containing the max, min and average of the number file in the server. | * Response code * Filename * Filename Length * File size * File Data |
| 011 | Indicates to the client that the file requested was not found in the server | * Response Code |
| 100 | Indicates to the client that the request is not known or understood by the server | * Response code |
| 101 | Indicates to the client that the change request to modify the file name was not successful | * Response code |
| 110 | Sends to the client the list of available instructions | * Response code * Length * Data containing the available instructions |

# Flow of the program

The flowchart below demonstrates a top-level overview of the program, for both server-side and client-side operation.

A diagram of a computer program

Description automatically generated

Figure 1: Client-side program flow

A diagram of a process

Description automatically generated

Figure 2: server-side program flow

## TCP and UDP Transfer implementation

In order to transfer information through TCP or UDP, we either sent information as plain text in the case of simple instructions such as the change or help command. However, for more data-heavy operations, such as the put, get and summary commands, we encoded the files that we wanted to send/receive into base64, which is a common encoding style that allows us to send any type of file between the server and the client. However, the TCP and UDP implementations were a little bit different between each other.

## TCP

The TCP socket implementation was one of the easier ones, as it takes care of the packet integrity checking, order-checking, datagram sizes, etc. for us, so we did not have to implement anything additional in regard to that. A flow chart of this process is outlined below.

A diagram of a process

Description automatically generated

Figure 3: Summary of TCP data transfer

## UDP

The UDP Socket implementation posed some more complications in comparison to the TCP one. One of the main problems is that there was a limit on the maximum datagram size that could be sent. Therefore, we had to perform fragmentation in the case where larger files such as images had to be sent. More precisely, the files which were split into chunks of 1024 bytes. This resulted in another problem: the server and the client would not communicate at all on the order in which the datagrams would be received. Therefore, we had to implement some kind of communication between the server and the client. This process is elaborated upon in the chart below.

A diagram of a flowchart

Description automatically generated

Figure 4: Summary of UDP data transfer

# Wireshark-related

## Questions for TCP:

Note: these are the results obtained after doing a “put image.jpg” command on the client, which uploads an image to the server.

A screenshot of a computer

Description automatically generated

**1. TCP SYN Segment:**

**- What is the sequence number of the TCP SYN segment used to initiate the TCP connection between the client and server? What identifies the segment as a SYN segment?**

The number of the TCP SYN segment used to initiate the TCP connection is sequence number 0. This can be seen by looking at the flags of the segment:

**2. Initial TCP Connection:**

**- What are the sequence numbers of the first two segments in the TCP connection?**

The sequences numbers are 0 and 1

**- At what time was each segment sent? When was the ACK for each segment received? Given the time difference between sending each TCP segment and receiving its acknowledgment, what is the RTT value for each of the two segments?**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No. | Seq. No. | Time when Sent | Time ACK was received | RTT | Sample RTT |
| 1 | 1 | 3.254764 | 3.256532 | 0.001768 | 0.001768 |
| 2 | 65496 | 3.256566 | 3.257669 | 0.001685 | 0.001103 |
| 3 | 130991 | 3.256977 | 3.257669 | 0.001561 | 0.000692 |

**- Build the round-trip time graph.**

A diagram of a diagram

Description automatically generated

**3. Length of TCP Segments:**

**- What is the length of each of the first six TCP segments?**

The length of the first six TCP segments are 56 44, 65539, 44, 65539 and 28320 respectively.

**4. Receiver Buffer Space:**

**- What is the minimum amount of available buffer space advertised at the receiver for the entire trace?**

The minimum amount of available buffer space advertised by the server is 65539.

**- Does the lack of receiver buffer space ever throttle the sender?**

In order to determine whether or not we reach the throttle limit, we can see the buffer space at the last reply sent by the server, and we see that the window size of 2619648 is still intact, meaning that there was no throttle

**5. Retransmitted Segments:**

**- Are there any retransmitted segments in the trace file? What criteria were checked in the trace to determine this?**

After a quick look, there are no retransmitted segments in the trace file. Looking at the SEQ numbers as well as the ACK numbers, we see that they increase consistently, without returning to a lower value. A lower value would show up in the case where a segment needed to be re-transmitted.

**6. Receiver ACK Behavior:**

**How much data does the receiver typically acknowledge in an ACK? Can you identify cases where the receiver is ACKing every other received segment?**

Looking at the image above, we can determine the amount of data in an ACK by looking at the “len” attribute. It seems to usually revolve around 65495 bytes, Additionally, We can see cases where it is acknowledging every other received segment for the packet # 282, where it acks the packet with sequence number 65496 and 130991 by sending an ACK 159267.

**7. Throughput Calculation:**

**- What is the throughput (bytes transferred per unit of time) for the TCP connection?**

(159267-1)/(3.257669-3.254764) = 159266/0.002905 = 54 824 784.85 bytes per second = 54.8 MBps

**- Explain the method used to calculate this value.**

The method used to calculate this value was to subtract the last acknowledgment number which is 159267 with the first sequence number which is 1 and you divide it by the time difference of the packets which will give you in bytes per seconds (Bps).

**8. Time-Sequence-Graph:**

**- Use the Time-Sequence-Graph (Stevens) plotting tool to view the sequence number versus time plot of segments being sent from the client to the server.**

A graph with numbers and lines

Description automatically generated with medium confidence

## Questions for UDP:

A screenshot of a computer

Description automatically generated

**1. Select one UDP packet from your trace. From this packet, determine how many fields there are in the UDP header. Name these fields.**

There are 4 fields in the User Datagram Protocol. The 4 are Source Port, Destination Port, Length and Checksum.

**2. By consulting the displayed information in Wireshark’s packet content field for this packet, determine the length (in bytes) of each of the UDP header fields.**

The length in bytes of each of the UDP header fields is 8 bytes because each of the fields in the header are 2 bytes long.

**3. The value in the Length field is the length of what? Verify your claim with your captured UDP packet.**

The value in the length field is the length of the packet (in bytes) that indicates the number of bytes in the packet. For example, looking at packet number 138, we see the number of bytes:

A screenshot of a computer

Description automatically generated

Which is 1024 bytes, which is the amount that we specified in our implementation per packet.

**4. What is the maximum number of bytes that can be included in a UDP payload?**

The maximum number of bytes is equal to (2^16 -1) – 8, which is 65527 bytes. The “-8” comes from the fact that there are 8 bytes reserved for the header.

**5. What is the protocol number for UDP? Give your answer in both hexadecimal and decimal notation.**

The protocol number for UDP is 17 in decimal notation and 0x11 in hexadecimal.

A white background with black text

Description automatically generated

**6. Examine a pair of UDP packets in which your host sends the first UDP packet and the second UDP packet is a reply to this first UDP packet. (Hint: for a second packet to be sent in response to a first packet, the sender of the first packet should be the destination of the second packet). Describe the relationship between the port numbers in the two packets.**

Looking at the packet sent from the host, such as packet number 138, we see that the destination and the source are as follows:

A screenshot of a computer

Description automatically generated

We also see in the reply the following information:

A close up of a number

Description automatically generated

We see that the destination port from the request is 12000, and the source port of the reply is 12000. Additionally, we see that the source port of the request is 51865, and the destination port of the reply is 51865.

# Conclusion

In conclusion, we implemented a file transfer protocol Python TCP and UDP sockets and simulated a client sending and receiving files from a server, as well as performing different actions. The implementation was also elaborated upon with diagrams and workflow. While TCP was a straightforward solution for reliable and ordered data transfer, UDP was more challenging to work with, as it had datagram size limitations and the need for fragmentation. Finally, Wireshark-related analysis was conducted for both TCP and UDP, addressing key questions related to packet sequences, round-trip times, etc. Overall, this project was a valuable insight as to how internet data transfers function, and all of the engineering that is required to implement a stable communication protocol.