**Assignment 1 – Report**

**Intro:**

We have been tasked with cresting a python file sharing application with the aim of teaching us about basics of protocol design and socket programming for TCP connections. The brief provided to students specifies a client-server architecture and suggests features to add. This is a report summing up the functionality and protocol specification of said application.

**Description of system functionality and features:**

**Server and Client Directories**

Upon starting the client, the user will be prompted to where they would like their files to be stored. If they choose the default directory, all files the client downloads will be stored in the current working directory of the client. Otherwise, the client can specify which directory they would like for their downloaded files to be stored. On the command line interface, the user has the ability to list all the files in their own directory and also list the files in the server directory.

<pic + tkinter dialog box>

**Uploading and downloading**

Also using the command line interface, the user can both upload files to the server, and download files from the server. This functionality works just as you’d expect. However, as we know, waiting for files to download is never fun. To ease this burden, we have implemented a loading bar to show the progress of file uploads and downloads.

<pic upload + download>

Note: If you are new to using this program, or forgot certain features, you can type “h” to view the help menu which displays all client commands

**Encryption**

To protect the files as they are being sent between the client and server, this application has implemented end-to-end encryption. In *connection\_manager.py* we have wrapped the socket in an encryption layer, adding encrypt and decrypt functions that makes use of the built in cryptography library called Fernet. Now if someone intercepts messages being sent between the client and the server, the messages will be unreadable without knowing what the encryption key is, thus keeping users’ files safe from prying eyes. The key can be specified by a server administrator.

<pic>

**Reliable data transfer**

Despite TCP having its own integrated solutions to reliable data transfer, and we have used these

Data storage (server storage file and specifying client directory)

**Checksum**

This system includes a file validation system to check that the files sent are not altered in transit or corrupted.

This is achieved by using a md5 checksum generated in the send methods of both the server and the client. The checksum is then passed as an argument in the header and checked on the other side. In the event of the checksum not matching up, the corrupted or altered file is deleted.

<pic>

**File Authorisation**

When uploading a file, the user can specify whether they would like to protect their file behind a password….

<pic?>

**Specification (Protocol design & specification (sequence diagrams & message formats/structure))**

**Key phrases:**

defining the framework of communication

specifying requirements and constraints (i.e. reliability and authentication)

defining the types and structure of messages (three types: commands, data transfer, and control)

communication rules that specify the sequence of messages at every stage of communication

clearly specifying messages and reactions for every communication scenario. You will need to represent such rules with sequence diagrams (at lease two sequence diagrams will be required, one for upload process, and another for download process).

Gareth: talk about anything cool we’ve done with the protocol design… I’ve put some ideas below and there are helpful phrases above in grey

As per the description brief, this application uses Client-Server architecture. Here the server contains all the files, and the clients connect to the server to upload and download files.

Reliable Data transfer – The program ensures when files are uploaded or download, they are sent reliably. This means that if the client loses connection midway through transferring a file, they will be able to reconnect and then resume the download of the file. This is achieved by chunking the file. When the connection drops the latest confirmed chunk on the client is recorded and then when the client reconnects it specifies which chunk it needs to start the download from again and continues with the download.

Message Header format – The format of the header message is simple, which allows efficient and fool proof communication between the client and server. The header command are: GET, LIST, POST, DELETE. Here is an example of a header message that will request to download a file: **GET#file.txt#** The command is followed by any number of parameters that can be used for various functions.The **data transfer** has a short header containing the size of the file to be transferred and then the checksum of the file. After that the file is then sent through. Control messages are simply messages that can occur during errors such as 404 – File not Found

The basic rules or framework of communication between the client and server are as follows. Once connection has been established, the server waits for requests from the client. If the client sends a message through the server responds accordingly and then returns to waiting once complete.

Within the program the transport layer is handled by a single class file called **connectionManager**. It handles the sending of sending of socket messages as well as the conversion of bytes to string messages and encryption/decryption by wrapping the send and receive in another method. It also handles connection errors, such as the connection dropping or aborting. The class file neatly contains the

Server Sequence diagram

Client Sequence diagram

Gareth: talk about how the client and server take turns

|  |
| --- |
| **Robust protocol implementation (stress tests)**  This program aims to use effective error catching to make sure the user cannot crash the client or server, being built on the principle of GIGO (garbage in, garbage out), not allowing the server to process anything unless the input matches very specific server commands. If the user inputs an unknown server command, an appropriate error is displayed, then they are prompted to input a new command or input. Because of this (and our effective protocol controlling server and client messages), no input from a client can crash the server, thus the server service is very reliable.  Many filetypes have been tested and various sizes when uploading and downloading. We did encounter errors when trying to upload an image with hex values, but that was only when using a shift cypher to encrypt and decrypt messages  Gareth: talk about File types and sizes used to extensively stress test our program, also maybe many clients trying to connect to the server. |
| **Error Checking**  As part of our design, we have implemented an extensive error checking list to help give the user useful information if a problem occurs (or if they have done something wrong) and to avoid unwanted crashes.  This list includes, but is not restricted to: |
| |  |  | | --- | --- | | Error | Program action | | Incorrect password on protected file | Deny user access to protected file | | User inputs wrong file name when downloading | Discard input and ask for a new input | | Checksum Mismatch | Delete corrupt file | | User doesn’t specify a file directory. | Use current working directory of the client | | Unknown server command | Discard input and ask for a new input | | Client loses connection | Notify Server | |
|  |