System Design Report

Program Design + How it Works

The system consists of 2 files: `Client.py` and `Server.py`, with these files containing the corresponding code for each.

The client works by initialising 2 threads: one to act as a server to listen for UDP messages from other clients, and one to allow user interaction via the CLI. Both threads are killed when a user uses the `OUT` command.

The server when started listens for connections from clients. It then creates a new thread for that client and prompts the client to authenticate themselves first. Once authentication is successful, the server then requests a command from the client.

The server keeps track of all clients via a dictionary called `devicesInfo`. Each dictionary entry uses the client's username as a key with another dictionary as the value. That client-specific dictionary contains the information such as `timestamp`, `deviceSeqNum`, `deviceIPAddr`, and `UDPPortNum`, which are all initialised after the client is authenticated. This allows the edge device log to be reconstructed in order of device sequence number everytime a device is removed or added.

The server maintains a blacklist of blocked accounts via a set called `blockedAccounts`. Everytime an account reaches the maximum attempts limit, the username is added to the set. A 10 second `sleep` function is then called, before the username is removed from the set again. This set is checked everytime a user attempts to login.

The `blockedAccounts` struct as well as a `nDevices` counter utilise a `lock` to prevent concurrency related issues.

Server output is clearly associated with a specific client by listing the IP address and port of the client being interacted with, and whether it was a request or response by the server (by listing `send` or `recv`). The full and exact response sent is also output. eg.

```
[('127.0.0.1', 57920):send] RC0;username authentication request
```

The sending client will display ongoing progress during a `UVF` command for improved usability, as it can take a while depending on file size.

Application Layer Message Format

The application layer has a very simple message format. All requests sent by the server to the client begin with a **Request Command** (RC) header, with `RC1;` indicating a command from user is required and `RC0;` otherwise. A `RC0;` header is used for other special responses such as during authentication. The reason this header was introduced is outlined in the **Design Tradeoffs** section.

The RC header is then followed by the actual response header, separated by a semicolon. These response headers are very human readable messages that a hard-coded on both the server and client side. The client then outputs a message or changes behvaiour depending on the response received. An example complete header is `RC0;password authentication request\n`. Any actual response data is then sent after the new line.

All responses from both client and server end with a `\r` character, to mark end of message.

UDP files are sent by first sending the header with `UVF {fileName} {nPackets} {username}`. This confirms that it is a `UVF` request. A `nPackets` value is also sent to allow the client to know how many packets to expect and listen for. The byte data is then sent in 4096 size packets, one after another until all are delivered.

Design Tradeoffs

It was decided that the server would request authentication before allowing commands, instead of simply trusting the client to send credentials for security reasons. This makes it easier to prevent users sending commands without being authenticated.

A RC header was introducted when it was found that sending multiple responses was occasionally leading them to strange behaviour on the client side thinking they are part of the one response. This header prevents the need to send a response, then another to request the next command. An alterative to this was to use `sleep()` to delay responses being sent, however this is much less performant. The `\r` character was also added to the end of messages to make it clear that the message is ended.

The UDP messages between client and server utilise a much larger packet size of 4096 instead of 1024. This is because UDP is used with the UVF command which involves transfer of large amounts of data such as video files. The UVF command utilises a `sleep()` of 0.25s to prevent responses overlapping. Since a larger packet size is used, a lower number of packets are sent and hence a lower number of sleeps leading to increased performance.

The response headers were chosen to be human readable instead of using a code (such as that used by HTTP). This is because although it is not very scalable and could lead to many issues if the response set was increased, for the purposes of this assignment it was decided that a more decipherable server output would be of greater value.

Known Issues

The TCP functions fail when a packet size greater than 1024 is required. This is clearly seen when attempting to run the `UED` command. An attempt to fix this was made by using a similar method to that used in `UVF` by splitting the data into specific sizes chunks to send. However, this for some reason was causing an infinite loop on the server side and data being continuously written to a file until storage and memory were filled. The cause of this was not found, and since this command was only worth 1 mark it was decided that this was not worth investing the time fixing. The packet sizes could have been increased to a value greater than 1024, however this would not have removed the problem but simply slightly mitigated it.

Potential Improvements + Extensions

- Fixing the issue with TCP packet sizes needing to be > 1024.

- Expand the computations available via `SCS` with others such as `MULTIPLY`.
- Allow for the creation of more complex files via the `EDG` command.
- Add reliabilty and flow control to `UVF` as it was assumed no issues would arise through `localhost` transmission.
- A more scalable application layer format.

Borrowed code

- Both the server and client utilise the supplied multithreading TCP code as starting templates.
- Breaking a file into packets to send over UDP adapted from: Stack Overflow