COMP3331 – Assignment Report

James Davidson, z5257605

1. Program design

The client and server are both written in Python 3.7.x, chosen for the language's simplicity and my own familiarity with Python development. I elected to develop the implementations using object-oriented programming, since both the server and client have their own unique state which must be managed.

The client implementation was composed of the following top-level classes:

- `BlueTraceClient`, the main instance of the client;
- `BlueTraceClientPeripheralThread`, a thread of the client to use for sending beacons;
- `BlueTraceClientCentralThread`, the client's primary receiver thread for beacons;
- `BlueTraceClientCentralSubthread`, an individual thread to handle one beacon.

The server implementation was split across two top-level classes:

- `BlueTraceServer`, which represents the main instance of the server;
- `BlueTraceServerThread`, which represents connection to that main server as a thread.

These classes are in the `bluetrace.py` file. The `bluetrace_protocol.py` file is used to specify all of the protocol messages exchanged between the client and server, as well as defining constants in the protocol – e.g. temp ID size.

2. Application layer message format

The protocol for communications I designed were inspired by the ideas of TCP handshaking covered in the course prior. Many interactions borrow the idea of a client or server broadcasting their intention to do something and waiting for the other party to acknowledge this first, and only after this performing the actual task. All protocol messages by convention are prefixed with `BT_`, indicating that they are part of the BlueTrace protocol.

2.1. Authentication

The authentication process is a good example of the inspiration of TCP handshaking, having an establishment phase before usernames and passwords are sent. The usernames and passwords themselves are sent simply in plain text.

Message	From	Meaning
`BT_AUTH_INIT`	Server	The server is initiating authentication with the client
`BT_AUTH_READY`	Client	The client is ready to begin authentication
`BT_AUTH_UN`	Server	The client should send their username now
`BT_AUTH_PW`	Server	The client should send their password now
`BT_AUTH_LOGOUT`	Client	The client is logging out

Figure 1: Protocol messages sent by the client and server during authentication

The protocol also defines some human-readable messages that the server sends to the client when authentication finishes, the exact text of which is similar to the examples given in the spec, so we omit them here to keep this report concise.

2.2. Downloading temp IDs

A single protocol message is used on the client's end to send to the server when a temp ID is requested from the user.

Message	From	Meaning
`BT_DOWN_TEMP_ID`	Client	The client is requesting a new temp ID

Figure 2: The protocol message sent by the client during temp ID downloading

The protocol also defines the size in bytes and lifetime in minutes of each temp ID as per the requirements of the assignment. Each temp ID is a 20 character string of randomly-chosen digits.

2.3. Uploading contact logs

The following protocol messages are used when a client uploads a contact log:

Message	From	Meaning
`BT_UPLOAD_CONTACT_LOG`	Client	The client wants to upload a contact log
`BT_READY_FOR_CONTACT_LOG_UPLOAD`	Server	The server is ready to receive the log
`BT_FINISHED_CONTACT_LOG_UPLOAD`	Client	The contact log has been fully uploaded

Figure 3: The protocols message sent by the client and server during contact log uploads

The client sends each line of the log one at a time as plain text, with spaces separating each field of the log for easy processing on the server's end. The size of each line in this log is fixed and defined in the protocol.

2.4. Peer-to-peer beaconing

Similar to the above, a peripheral client will first send a protocol message expressing their intention to send a beacon to a central client, and the central client will respond when they are ready to receive the beacon.

Message	From	Meaning
`BT_SENDING_P2P_BEACON`	Peripheral client	The peripheral client wants to send a beacon
`BT_READY_FOR_P2P_BEACON`	Central client	The central client is ready for the beacon

Figure 4: The protocols message sent by the peripheral and central clients during beaconing

The protocol version, size and lifetime of each beacon is defined in the protocol. Each beacon itself is sent in plain text with space-separated fields.

3. How the system works, at a glance

The programs are started using the commands

```
python3 client.py [server IP] [server port] [client port]
python3 server.py [server port] [block duration]
```

The server will simply print the output of events to the terminal it is run in. For the majority of the client program's runtime, it will be in a state of accepting user commands typed in the terminal it is run in by providing the user a prompt.

When the client program is started, it immediately prompts the user to login using their credentials. If the particular credentials entered are those of a blocked user, the program will stop here, and the same will happen if the user fails to login more than 3 times. After authenticating, the client program will then enter an infinite loop of accepting and processing commands until the `logout` command is issued, or the client program is forcefully killed by the user. Each valid command is handled by a separate helper function in the client class, which will then communicate with the server or other clients as required. Interactions between the client and the server happen on a single thread on a TCP socket, but the client's program will also be open to receiving beacons from other clients on a separate central thread on a UDP socket, which has its own separate thread system for handling multiple beacons.

When the server program is started, it immediately enters a loop of accepting incoming client connections on a TCP socket. Each connection is handled in a separate thread object, and persists until the connection is closed by the client. Each client's authentication process is initiated by this server thread. When authentication has been completed, the server processes any commands sent to it from the user in helper functions. To maintain thread-safe access and modification of resources, all common server resources have an associated mutex which these threads must acquire.

The credentials for the users are stored on the server side in a file named `credentials.txt`. The server also maintains a list of temporary IDs in the `tempIDs.txt` file. Contact logs on the client side are stored in the file `[username]-contactlog.txt` with the client's username as appropriate.

4. Reflection

I think that choosing Python as the language was a good choice, although if I had more free time this term (and, honestly, if I had started earlier) I would have liked to use C for this assignment. Using C would have increased the volume of the code considerably though, since so much of what is necessary for the assignment that is already part of languages like Python would be non-trivial in C. If C++ was allowed, this would've been a fair compromise.

I believe the design of the program itself is solid, and given the chance to perhaps do things differently, I don't think I would. The main design tradeoff I made was opting for an object-oriented approach rather than just functions and global state variables. The reason I chose this was to avoid the need to constantly pass parameters back and forth between functions which cleans up the return logic of the code. The drawback is that in order to properly adhere to best practices when doing OOP, getter methods are needed so that encapsulation isn't broken – these methods can then potentially bury the actual methods we are interested in. I think the design choice to also have the constants and protocol messages defined in a separate file was also a good idea, and I wouldn't do it any other way.

One minor thing which slightly bothered me in the end about the program was that the output of peer-to-peer beacons interrupts the prompt of the receiving client. This is inevitable when the receiving of beacons can happen at any time, however I wonder if it was possible to design the concurrent nature of the program such that these interruptions wouldn't occur – if I had allowed myself more time, this would be one thing I'd try and investigate.

Overall I am pleased with the result.