BlueTrace Contact Tracing App

# Program Design

The program is comprised of two major modules, the server, and the client. Information is sent between the server and each client via a TCP connection. Information is sent between clients in a P2P manner over UDP. In both instances, a lightweight messaging protocol is utilized to minimize the information transferred between parties.

## Server Design

The server utilizes separate threads for each connected client. Each client passes through a strict control flow to authenticate the user. Once the user has been authenticated, the server waits for the client to either request a tempID, upload a contact log or log out.

**User Authentication**

The server manages user authentication by waiting for client’s username/password combination upon connection. The client is compared against the list of blocked users and is rejected if the client has been blocked more recently than the blocked duration (which is set on server start-up). The client provided pair is compared against the credentials database which is stored on the server as dictionary. If the pair is in the database, the client is successfully passed through to the ‘logged on’ phase. If the password does not match the database, a counter is incremented. A prompt is sent to request the user to retry password input. Once the counter reaches 3, the user is added to the database of blocked users (implemented as blocked.txt) along with the time of blocking. Limitations of this design are described in the Limitations Section.

**TempID Generation**

TempID Generation is done upon (logged-on) client request. The server checks the tempIDs.txt first to see if the user has a valid tempID. If the client has a valid tempID, the server sends this to the client, otherwise the server generates a new tempID, writes this tempID to the database and then sends it to the client. The tempID is generated by joining 20 random digits together.

**Contact Log Checking**

When a client requests to upload a contact log, the server awaits receipt of the log. Upon receipt of the contact log, the client extracts the unique tempID’s and uses the tempID’s database to determine the associated users. The users who need to be contacted are printed to server terminal, along with their tempID and time of contact.

## Client Design

The user is initiated as an object (of type User) with attributes of id, UDP port, tempID and window of validity for the tempID. These attributes are initialized as default values, except for UDP port which is known at start-up. The other attributes are populated and updated as they become known

**Login Design Flow**

The client program prompts the user to input username and password on start-up and sends these details to the client, as well as populating client object ID field. The password is not saved by the user object for security reasons. If the server informs the client that the user/password combination is incorrect, the user is prompted to retry.

**Logged in Control Flow**

* Doesn’t request tempID if tempID is still valid (Basic version of caching)

# Application Layer Message

As the information sent between the server and the client is quite simple and of fixed length (with the exception of the contact log) the intention was to keep the messaging as lightweight as possible. During the login phase, where the user is sending username and password, the following is the agreed semantics between the client and server.

|  |  |
| --- | --- |
| **Message from Server** | **Meaning** |
| 0 | "Invalid Password. Please try again" |
| 1 | “Successful Login” (Welcome message is sent) |
| 2 | "Invalid Password. Your account has been blocked. Please try again later" |
| 3 | "Your account is blocked due to multiple login failures. Please try again later" |

During the ‘logged in’ phase, the agreed semantics are:

|  |  |
| --- | --- |
| **Message from Client** | **Meaning** |
| 0 | "Please log me out" |
| 1 | “Please send me my tempID” |
| 2 | “Please standby to accept my contact log” |

During the P2P phase, a client in the ‘peripheral mode’ will broadcast it’s beacon, which is a triple of tempID, tempID start time and tempID expiry time, which is the minimum information it must convey. A client in ‘central mode’ will listen for beacons, check if the current time falls within the beacon start, expiry time and if so, add the beacon to its contact log.

Knowing exact size of possible messages where possible

Handshake to determine object size where not

# Design Trade-offs

considered and made.

Threaded implementation of peripheral/central

# Possible Improvements

Describe possible improvements and extensions to your program and indicate how you could realise them.

* Don’t send strings
* Server doesn’t crash, but prints ugly errors when a client unexpectedly drops
* Credentials stored in an encrypted format
* User/Pass sent encrypted
* Multiple threads accessing blocked.txt (maybe other files at the same time), should use locking.

# Limitations

If your program does not work under any particular circumstances, please report this here.

* Wrong password count per connection. Malicious user could try one or two passwords, then make new correction to get around 3 password block

# Code from other sources

The initial design for the multi-threaded TCP server was from the following link suggested by the lecturer on the forums:

<https://www.geeksforgeeks.org/socket-programming-multi-threading-python/>

The following two links served as inspiration for the code to modify, clear and append to .txt files user in the program.

<https://thispointer.com/how-to-append-text-or-lines-to-a-file-in-python/>

<https://stackoverflow.com/questions/48829584/python-program-to-delete-a-specific-line-in-a-text-file>