COMP9331 Assignment Report
Chencheng Xie z5237028

Program design

The program is implemented in Python 3.7.4. The connection between client and server are based on TCP connection, and for P2P connection we used UDP to simulate.

For how program is designed, I pretty much followed section 4 “Tips on getting started” in the specification.

1. To implement single client login, first ask user for username, then a while loop for client to input password until server indicate login successfully. Each time client receives a password attempt from keyboard input, it sends this password to server. Server will fetch the username, password pair in “credentials.txt” and compare with received pairs. Server responses according to compare outcome.
2. Add 3 fail attempts block function. Server has a dictionary stores the number of attempts left for that username and block timestamp (timestamp when it’s blocked not when it will release). When client trying to login, server will first check its block timestamp, only timestamp that plus block_duration are earlier than “now” (time.time()) will be allowed to proceed. Each fail login costs one attempt. Zero attempt resets the account blocks timestamp to now (it’s now blocked).
3. Extend the system to multiple clients. Multithreading is used. The main thread is a while loop that always looking for new connection, and as long as it founds a connection, a sub-thread is created to deal with this, and the main thread loop back to wait for new connection.
4. Let client takes commands. Once client successfully login, a while loop infinitely waits for keyboard input until the command is “logout” which break the loop and finish. How commands are formatted will be dicussed in Message Format section below.
5. “Download_tempID” function. If the command is “Download_tempID”, client sends a packet with ‘DT’ as commandType to server. Once server receives this command, it randomly generate 20-bytes string of tempID, check if it already exists in the “tempIDs.txt”. If it’s duplicated, regenerate. Otherwise, send this tempID back along with timestamp of start and end of validation. Client will receive the tempID and timestamps, so client and server agree on the validation timeframe of current tempID.
6. “Upload_contact_log” function. Once received command is “Upload_contact_log”, client sends a packet with ‘UC’ as commandType and number of Log as numOfSend to server. Server replies with numOfSend in numOfRecv field as indication at server expects these many logs. Client will then send each log, and wait for server’s ack. This stop-and-wait will be discussed more in the Trade-offs section.
7. “logout” command is just simple, sent ‘LO’ command to tell server that client wants to logout, close the socket and break any loop for that client to terminate. The server will close the socket that is currently with this client (not all sockets).
8. P2P functions. Clients now need one extra thread to listening to the potential beacon from other clients. This thread will listen to each beacon, and response to that beacon (check if the beacon message is valid by comparing current timestamp with beaconed timestamp, if valid, add this message to contactLog, and record the current timestamp which will late be used for 3min privacy deletion). Clients also need another command “Beacon xx.xx.xx.xx xxxx” which just sends message to “xx.xx.xx.xx xxxx” using UDP protocol.
9. Since we need to delete any log that are 3 mins old, we need another thread that constantly check if any timestamp of logs expires (with some time interval, for 0.1s in

my system). Since the logs are stored from oldest to youngest, we stop check until we first meet a valid timestamp and delete all expired logs.

Application layer message format

The Application layer message format during login phase are simply strings. When TCP connection established, server will expect username, 3 * password if needed in order. So, client will send username in string directly, follow by password, until server told client to stop.

The Application layer message format between client and server after successfully login are stored in a data-structure associated with python dictionary. It includes "commandType", "numOfSend", "numOfRecv" and "data". "commandType" can be one of "DT", "UC" or "LO" which corresponding to "Download_tempID", "Upload_contact_log" and "logout". Client and server will use same commandType if they are communicating on the same task. "numOfSend" indicates how many more packets, or message, will be sent next. "numOfRecv" indicates how many more packets is expected to be received. And "data" stores the actual payload of each packet.

The Application layer message format between peer to peer are also simple. Since this beacon message is based on UDP, single direction communication is sufficient. So, when client tries to beacon neighbour clients, it can just send beacon message in string in the format of "tempID, start timestamp, end timestamp, version#". Once other clients received this message, they will parse the string by themselves.

How system works

1. Server need to be online before any client tries to login.
2. Client tries to login and acquires their tempID.
3. When two clients are close enough, they can beacon each other (simulated by UDP), tempID of their account and timestamp will be exchanged.
4. If any user is tested positive for the COVID19, the client will upload their contactLog file, so the server knows who this user has encountered during the last 3mins, and access to their phone number (have a way to contact them, warnings, medical suggestions etc.)

Design trade-offs

When implementing "Upload_contact_log", a pipelining would be more efficient, but my implementation cause one problem: if you sending two consecutive "Upload_contact_log" commands really quick, or the contactLog file is really long, without expecting the "ACK" from server, one of the actual data from first contactLog file might be treated as the message of number of Logs of second "Upload_contact_log" command which causes error. My solution is always wait for an "ACK" so not until first file transmission is finish, you don't send the second file.

Possible improvement and extension

1. Network quality variation is normal, and connection might loss for short period but back to normal very some. In my implementation, once the connection is break, the channel become brokenpipe, and the communication between server and client need to start all over again (authentications, contactLog files). This can be solved by more complex re-authtication mechanism which involves saving all unfinished file, their states and auto-reconnection etc.

2. The client cannot beacon before knowing their own tempID. Because the "Download_tempID" command is manually given by the user under this assignment's context, it's possible for user to just beaconing others without downloading their tempID. This can be solved by automate "Download_tempID" whenever user logs in, then the beacon will always happen knowing their tempID.

Cases might cause error

1. Assuming no contactLog when first sign in, otherwise we don't when these logs were beaconed and when to delete these logs.
2. Two clients cannot be assigned with same UDP port
3. Logically clients cannot beacon before download tempID (in my system, only print error message and ignore the command)
4. Assuming no external file modification (deletion, overwrite etc.)
5. Once server is disconnected, and reconnected, the whole communication needs to start all over (no pick-up mechanism).