Report on design of application

Fundamental design

For client side, user will first go through a login stage which requires the user to enter username and password respectively. For the information entered, the client side will wrap them up and send a LOGIN request to the server. Login process is a two-stage interactive validation procedure, after the account is validated or a new account is created, the client will transfer to the working mode.

There is a local command resolver to check the command format upon receiving each command. The format is specified in a local config file called command_format.json. If the message is valid, it will be packed up with the "feature" which is similar to sequencer and then sent to the server along with username. After receiving the feedback from server, the client will print the corresponding message to terminal.

For a server, it will run continuously after run() function is called. It will keep on listening from the UDP port and process the attached command. It will first examine the feature code of the command, if it's has already processed, the server will simply send the stored response again. If it's not seen before, the server will call handle() to process the command and add it to seen command list. While handling, the server parses the command first and call the appropriate functions according to the command name. Once a Error is caught, it will return the error message back to the client. If a file request is received, server will initiate a TCP connection to the client after the client acknowledges that it has received the feedback for file request. Once file transfer is completed, the connection is closed.

Message format

Encoded message is sent between client and server in the following format:

$$feature + command$$

Feature is a crc32 hash computed from the combination of timestamp and command. To decode the message, server just need to take the first ten bytes from the message received. The feature code is the unique identifier for a UDP request, this is similar to a seq number.

Reliable communication

After sending a message, the client will stop reading commands and keep waiting for the reply with correct feature code. The reply from a server can be regarded as acknowledgement for the command sent. Notice that, if packet loss is happened, server side will receive duplicated command, and it will just return the stored response to the client. And the client will keep sending command until desired response is received.

Multi-client support

Multi-client is naturally supported by this framework. Because the connection is stateless, the order of receiving requests doesn't matter. The server processes commands based on the attached username separately.

Trade-off

Multi-thread is not used for this application to support multi-client. Instead, we process commands from various clients sequentially. There are three main operations associated with an application: internet I/O, CPU computation and disk I/O. According to the spec, disk I/O should not support multi-threading. In addition, multi-threading cannot increase the clock efficiency of a CPU, the main purpose is to execute functions in parallel to minimize the effect of congestion. However, processing thread is not possible to cause a congestion. So, multi thread can only help to save time for internet I/O. As the data size is only few words, it is also not necessary to introduce multi-threading. Multi-thread may also introduce risks to the program. Because file transfer doesn't support multi-thread, the clients may keep establishing TCP connections at the same time to overload the traffic. And lots of state locks will be placed to protect forum info and to prevent establishment of new TCP connections. This will increase the complexity of the source code. In the case that many clients are interacting with the same thread, thread pool may also be exhausted. New mechanism for managing thread pool will also be required. In conclusion, the performance provided by multi-threading can be neglected.

Base Classes

There are several base classes defined: Server, Client, Thread Manager, Thread, Communicator, Sender and Receiver. They can be categorized into three types of which are communication, functionality and application. These classes will be introduced separately in the following section.

Communication classes

Communication classes provide interfaces for sending and receiving information via certain channels. There are three types of communication classes: Sender, Receiver and Communicator.

Sender

Senders are atomic classes for sending data to the different channels. They have a callable function *send(message)* that sends the *message* to the dedicated channel. Three types of senders are used in this assignment:

Receiver

Receivers are atomic classes for receiving data from different sources. A callable function read() is defined in each class which returns the message that the class has received. There are also three receivers used which are:

CommandLineReceiver, UDPReceiver and TCPReceiver:

Communicator

A **communicator** is a composite class consists of corresponding Sender and Receiver. Addition to send and read functions, there is also a function called fetch to send the message to destination and return expected response from there.

In this application, only UDP communicator is used by the client to send commands to the server and retrieve the feedback.

Application classes

Thread and **Thread manager** are defined to implement the functionality of the forum. Thread is an aggregate of Threads with methods to manipulate the threads such as adding posts, managing thread list and interact with files. Error recognition are defined mainly in a Thread Manager, and these raised errors will be used in the server to return appropriate message back to the client.

A class called **PassManager** is implemented to record the user information and online status of each of them. Interfaces for updating the registered accounts and maintain login status are defined in this class.

There is an application class called **CommandListener** for clients. Commands read in the client interface are passed to the Listener and the listener will handle the commands according to the rule defined in a local config file called command_format.json. If no error is detected, None will be return to the client.

Client Server classes

Client and Server are the top level classes for this assignment. To run a client or a server, user must run a method called run() after instantiating the classes .

Client consists of a command line receiver to read commands and then sends the command to a CommandListener, UDPcommunicator and a TCP Sender to.

Server class uses a UDPcommunicator and a TCP Receiver, PassManager and ThreadManager.