Group Based Peer-to-Peer Communication Network

Lucas Vincent Bowers

COMP1549: Advanced Programming
University of Greenwich
Old Royal Naval College
United Kingdom

***Abstract*—This report outlines the design and implementation of a peer-to-peer chat network, supporting multiple clients and implementing threading, accompanying the program written in Python. It includes a full run-through of every stage of the programs execution.**

Key words: *(peer-to-peer, chat, python, threads)*

I. Introduction

A Peer-to-Peer (P2P) network is one that operates with no central server, instead each connected peer operates as their own server, communicating directly with each peer. Networks like this are easier to setup than a Client-Server model and have much lower running costs due to the absence of the central server. Since each client maintains its own state, these networks are also simpler to keep communication running.

An effective way to learn how P2P networks are designed and how they operate, is to create a chat program since this can demonstrate all their fundamental features, such as state management between peers (e.g: ensuring all connected peers know of all other connected peers) and core communication techniques (e.g: byte streams).

The implementation of a P2P network accompanying this report is a chat program written in Python as it provides extensive support for the network services necessary, however it is not necessarily language independent. The fundamentals of P2P are portrayed throughout, allowing the report to be used to recreate a P2P network in any reasonable programming language.

This report breaks down a Python P2P chat program, explaining how it was designed, implemented and tested, as well as justification of decisions made throughout its development.

II. Design/Implementation

This P2P chat program is written in Python, which comes built-in with all the modules required to implement such a network, thus removing the need to find & import potentially unoptimised or insecure third-party modules (not to say Python's built in modules have no security flaws, this is of course still a possibility as with all programming languages).

Python's built in socket module is the first of two crucial modules for this program, a Low-level networking interface allowing communication over both TCP and UDP, perfect for a P2P chat program.

By default, python scripts operate in a single, main thread. Upon execution, they run from start to finish, working through a list of tasks, one at a time. Since a chat program has various blocking stages, whereby the program execution is paused (e.g: for user I/O), or a loop is iterating, it would not be possible to create without multiple threads. So the second crucial module is 'threading', which allows the flow of execution of a python script to be split into multiple threads, running simultaneously.

The program runs using 4 key threads…

1. Main thread: initiates the chat program for a peer, processing command line parameters, calls to start the peers server and client. Also handles reconnecting peer to a new coordinator server if existing coordinator disconnects.
2. Server thread: solely for the peers server to run on, handling connections and messages to it. This thread also has numerous sub-threads: each time a client connects to the server, a new thread is created, dedicated to receiving data from

the connecting client; a ping thread is used so that checking chat members are active does not block the rest of the chat from functioning.

3. Client thread: for the peers client to run on, has a sub-thread for listening to messages from server
4. Input thread: this continuously listens for input from client, using Pythons *input()* function, which is a blocking function so the program execution would only continue each time the user submits if this was not in its own thread. The input is passed on to the peers client for sending.

All threads in the program, apart from the input thread, run as daemon threads, meaning they run in the background and do not prevent the program from exiting. The program will "only exit when all non-daemonic threads exit". The exception to this is the input thread, which is non-daemonic, thus keeping the program open until it ends.

A technical requirement of a chat program is of course that members can connect together and communicate with one another. When the chat program is first run, the username, listening server and optionally, an existing member, are provided as arguments. Listening server is the address that this peer will start their server on so that any peer can connect via any existing peer, not just the coordinator. Existing member is an optional argument which, if given, allows the peer to connect to an existing chat, otherwise, the peer will start a new chat with just them.

The *Peer* class is then instantiated, used to keep track of the peers own data (their id, username, etc.), as well as storing information on all other peers connected to the chat at all times. This class is only instantiated once, the single instance being passed around throughout the program (e.g: to *Server* and *Client* class instances).

The other single instance class in the program is the *Server*, which is instantiated once, right after the *Peer*, but never again. The *Server* instance operates independently, only communicating with the rest of the program when mutating/accessing data in the *Peer* instance. The *Server*, given an ip address and port number to bind to, continuously listens for new connections to it. In Python, this is simply a *while True* loop which will always run until broken with a *break* statement. The while loop does not get out of control, running over and over again since the *socket.accept()* function within it, used to accept connections to the server, is blocking, meaning the server thread will pause completely until it returns.

Before client connections are explained, note how messaging works on this chat. Since the socket is running in *SOCK_STREAM* mode, both server and client can only send anonymous byte streams of data between each other, thus, without custom design, no message identification would be possible (telling who sent the message/who it was sent for). Furthermore, some messages are meant just to be processed by the program ('*SYSTEM*' messages), whereas others are messages sent by users, to all the other users in the chat ('*CHAT*' messages), something that could not be identified without custom design.

This chat program uses message headers to avoid this issue, prepending each sent message with a header containing data about the message, allowing the receiver to correctly parse it. The header is split into 3 parts…

1. Sender ID part: this is the id of the peer that sent the message (used to identify them in the list of chat peers, stored in the *Peer* instance, which can then be used, for example, to add their username to their message output)
2. Message Type part: the type of message being sent (either '*SYSTEM*' or '*CHAT*'). Chat messages are processed by the receiver as messages sent by a peer, to be output to all connected clients. System messages however are processed differently dependant on their body, they may contain data such as a list of peers connected to the chat, or a ping message to check a client is still active.
3. Message Length part: the length of the messages body. This is used to accept the correct amount of bytes from the server when the client receives a message.

Each of these header parts have a fixed length (could be anything reasonable, this particular chat program uses 10), meaning it cannot be tampered with by the user. For example, if the header parts were simply separated by colons, then you split the full message at each colon to get the header data, the user could manipulate this by adding extra

colons and cause errors or even worse, security flaws, within the chat.

When a client connects to the server, if the peer running the server is not the chat coordinator, the server sends them a system message containing the chat coordinators address so that the connecting client can connect to that address instead. Otherwise, if the peer running the server is the coordinator of the chat, they process and accept communication with the client by responding with a 'connected' system message. To clarify how messaging works in the chat program, this 'connected' message in its raw form would look as follows…

| *SYSTEM 9 connected* |

The blank space at the beginning is where the 'Sender ID part' would go, but since this message is sent by the server, it is blank. "SYSTEM" is the message type, since this is a system message, not meant for user chat output. 9 is the length of the body "connected", then at the end is that message body itself, "connected".

The server will also create a dedicated new thread for receiving data from this newly connected client, continuously looping, waiting for data with the *socket.recv()* function, which is parsed a single parameter, the number of bytes it will receive. This parameter is set to the message *header length* of the program, since every time a correctly encoded message is received by the server, it will always begin with a header of this length. So this header can be decoded, breaking it down to its 3 parts (sender ID, message type, message length). Now that the server has the length of the message body, it can run *socket.recv()* once more, giving it the message length as its parameter, to receive the rest of the bytes from the server, thus returned the message body. This received message can now be processed.

If the message is a system message, it will be ignored completely unless the servers peer is the chat coordinator, since system messages sent to the server are used for managing chat state, which only the coordinator is responsible for. If the servers peer is chat coordinator, there are various purposes the message could be fulfilling…

- "*pong*" message: this is the response from a client that has been previously pinged as part of the inactivity checking system of the program. When the server receives this, they can update the peers last successful ping time to now (stored in the chat peers list of the *Peer* instance).
- "auth" message: sent by a client to inform the server of their specific details (e.g: their username and server address). Data that can then be appended to the chat peers list of the *Peer* instance.

If, however, the message is a chat message, it is simply sent out to all clients connected to the server. This includes the client that sent the message, functionality that has little value at the programs current stage, however it means every message output to the client, whether sent by them or someone else, has been processed by the coordinator, ensuring consistency, as well as allowing for extra processing such as moderation.

If the *socket.recv()* function returns 0 bytes, this means the client has disconnected, so their connection is removed from the servers list of connected clients and every chat peer is 'pinged' to get an updated list of active peers. This ping is executed in its own thread so that the chat program can continue running normally, while the peers list updates in the background.

The ping functionality works as follows… A system message with body "ping" is sent to all connected clients, at which point, the current unix timestamp is saved (seconds since 00:00:00 1 January 1970). The ping function then sleeps for 1 second, giving the clients time to respond (functionality of client receiving/responding to ping message described alongside the *Client* later in this report). After the 1 second delay, the function loops through each currently saved chat peer (stored in the *Peer* instance), checking if they have been successfully pinged since this ping function started (the time saved when all clients were previously pinged). If they have not successfully responded to the server, they are deemed to be an inactive peer, and so are removed from the peers list of the *Peer* instance, and a "<username> disconnected" message is broadcast to all connected clients. Finally, the updated list of active peers is sent to as a system message to all clients, so that they can update their chat peers lists too.

During chat initiation, once the single *Peer* and *Server* instances have been initiated, the program starts the input thread. As previously described in the key threads outline, this continuously listens for input from the user using Python's *input()* function. When the peer submits data to this, if the peer has an active client, which is connected to a server, the clients *send()* method will be called to send the submitted chat message to all other clients. Therefore, the next step is creating an instance of the *Client* class to handle the peers client operations.

The client can either connect to an existing chat peer, as given by arguments on program execution, or it can connect to its own server as coordinator if none is given. On initiation, the instance of *Client* is returned to the main thread so that it can be used by the input function and messages can be sent using it.

First, the client initiates its own IPv4 socket, once again using byte streams (*SOCK_STREAM*) to match the servers configuration and attempts to connect to the given server address. As mentioned in the server functionality explanation above, when a client connects, they are sent one of two messages…

- "connected": the server accepted communication with the client
- "coordinator:<coordinator address>": the server informs the connecting client of the coordinators address.

If the client receives the "coordinator…" system message, it will extract the chat coordinators server address from the message body, and repeat its *start()* function, this time using the coordinators address. Otherwise, if they receive the "connected" message, the server has successfully agreed communication with the client, so the client can store the chat coordinator to their *Peer* instance for future reference.

On successful connect to the chat, the client needs to authenticate themselves with the server by telling it their username and server address. So the client will send an "auth" message to the server, containing this data, at which point, the client will now be able to successfully communicate within the chat and all other chat peers know of their existence since the server would have sent an update peers list too all.

A new thread is created, looping continuously, attempting to receive data from the server using *socket.recv()*. When data is received, identically to how the servers receiving functioned, first it gets the message header, parses it to get the message body length, before receiving the rest of the message. If the received message is a system message, just like with the server, there are various purposes it could have…

- "ping" message: this has been sent by the server to test if the client is currently active. The client will simply send a message back to the server with body "pong" to confirm they are active.
- "peers" message: this will contain a list of peers sent from the server so the client can keep their list in sync with the coordinator, saving it to their chat peers list in their *Peer* instance.
- "output" message: a string sent from the server to simply be output to the client, for example to print when someone connects to/disconnects from the chat.

III. Analysis and Critical Discussion

If at any point during the program, a clients connection to the chat coordinators server is interrupted, they should not be simply disconnected, but instead a new coordinator for the chat should be chosen, allowing all clients to effortlessly reconnect to the new coordinators server. In the main thread of the program is a *while True* loop, wrapped around functionality to do just that. On each iteration, it checks if the *Peer* instance is connected to a chat server. If it is not, the client is informed that the coordinator has disconnected, so a new one is being found. The program will loop through all chat peers, finding the one that joined the chat the earliest (based on a unix timestamp stored when the peer first connected). The new chat coordinator will then either be set to that earliest joined peer (which could be their-self), or if no other can be found, the client their self will become the new chat coordinator.

Once a new coordinator has been found, a new instance of the *Client* class will be initiated using the server address of this new coordinator. The program sleeps for a random

time between 0 and 5 seconds before attempting the new connection to reduce the chance of all the chat members attempting to join the new server at the same time, which risks raising a connection refused error. Upon success of the new connection, the chat can continue as before with no interruption and no need for the user to manually reconnect. This process continues until the last remaining chat member exits, at which point the chat no longer exists.

To exit the chat, a user can run *ctrl+c*, which raises a *KeyboardInterrupt* exception, something that usually prints to the console as a Python error. So the main thread uses a *try/except* statement to catch this exception, allowing the chat program to exit cleanly, without misinforming the user that an unexpected error has occurred.

When the user initially starts the program, a *try/catch* statement wraps around the instantiation of the *Client* class, thus, if the connection fails, it can be caught and the peer can simply be connected to their own server as coordinator instead.

The Singleton design pattern has been used for the *Peer* class as the single instance represents the user as a peer, throughout the running of the program from start to finish, without the need to instantiate again. In order to further improve the program, other design patterns such as Factory could be implemented more rigidly. Objects in the program do keep logic and data hidden, for example the *Message* instance keeps all data stored in private variables, allowing access only via accessor and mutator methods.

In order to sufficiently test the program to ensure this report compliments an accurate P2P chat, unit testing was carried out using the *PyUnit* module, imported as *unittest* in Python. This gives use of two primary testing techniques, Stubs and Mocks, to simulate methods in the program before the code for them has been written, allowing testing to be carried out on a unit level, during the development process, rather than trying to test every part of it upon completion.

In initial stages of development, starting with a simple client-server model for the chat simplifies things drastically as one only has to consider a single client communicating with a single server, with aim of receiving any sent data back from the server exactly as it was sent. Converting this into a P2P network is where PyUnit testing really benefitted the development process. To begin, the server from the client-server model based chat had its own *Client* stub, meaning the functionality for the client will be there in future, but for now it was essentially a placeholder. Allowing the focus of development to move to the core of the network itself.

IV. Conclusion

The program and report successfully outline and demonstrate an implementation of a peer-to-peer network in Python. It assigns a coordinator to keep track of peers currently active in the chat, the server for which new clients can connect too. If the coordinator leaves the chat, a new coordinator is automatically assigned, even if there is only one chat peer left, they simply become their own coordinator. A new client can also connect to any of the existing peers in the chat, allowing them to successfully connect without knowing who the coordinator is until they connect.

While connected, all clients are constantly kept up too date with the current state of the chat members, particularly when someone leaves or joins.

Future work to develop the project could include adding a GUI to make the chat program more interactive and user friendly. Particularly if it were to be targeted at a more general consumer, rather than the programming skill, even basic, that would be required to know how to run the chat program.

References

Beazley, DM., 2000. Advanced Python Programming. *Department of Computer Science, University of Chicago*