Asynchronous Chat Box: Request For Comments

Networking Group: Evan Deposit and Charles Bolton

June 9, 2019

Status of the Memo:

This memo defines an asynchronous pythonic protocol for an Internet Chat Relay application called the 'Chat Box'. Discussion, comments, improvements, and fixes are encouraged. Distribution of this memo is limited to Portland State University and its affiliates.

Abstract: Initially developed in 1998, the IRC application layer protocol was a popular chat protocol for many years, despite its steady decline in use over the last decade. Here we describe a specific experimental (yet derivative) implementation in an application called 'Chat Box.' The protocol uses the classic client-server architecture implemented in two python scripts or modules called server and client, respectively. It is a text-based protocol.

Table of Contents:

1. Introduction

As this RFC describes a specific application's protocol and implementation, it is particularly concerned with not only the protocol but the code that implements this protocol. Because it is an experiment in asynchronous application programming, we focus more on the programming aspects than is usual for an RFC. This protocol is implemented using Python's asyncio package. Messages sent to and from client to and from server are transferred over TCP using the familiar socket architecture. The application supports a general public chatroom as well as specific rooms created by users.

2. Python and the Asyncio Library:

For the following discussion, we have implemented the Chat Box using Python and the Asyncio (Asynchronous Input/Output) library. The choice to use the latter was an experiment in Python's handling of asynchronous coroutines. The Asyncio framework allows us to implement the web-server in an asynchronous, concurrent manner where execution is not delayed by IO. The program uses a single thread of execution, but splits the program into separate read and write sub-processes attached to a central event loop. When the program is waiting on IO from one subprocess, the event loop transfers control to other subprocesses, so that other messages can be received or sent, standard input can be processed, or other coroutines can be executed.

3. The Server:

3.1 The server must be started before clients can connect to each other via the server. The server is started by executing the server.py script.

server.py:

When this module is executed, the following happens:

3.2. An asynchronous event loop is created, and the server is instantiated as a coroutine. This coroutine takes the function *main* as a parameter. It also takes the IP address and port number for the server as arguments to be given to the main function. For simplification we use the localhost ('127.0.0.1') as the IP address and 8888 as the port number.

3.3. When a client attaches to the socket (8888), the asynchronous *main* function is called. Within the *main* function, a global list *addressList* is employed to keep track of the port numbers of the clients as they join. New client port numbers as well as usernames are passed to the *new_client* function, which creates an empty list in a global dictionary called *mailbox* for each new client.

3.4. The server then asynchronously listens for messages from clients and sends messages to clients. This is accomplished with the asyncio *gather* function, which will periodically alternate between listening (receiving messages) and sending messages. The functions passed to *gather* (*listen_to_client* and *send_to_client*) are given the port number, username and reader/writer objects for listening/sending respectively.

3.5. Within an infinite asynchronous loop in *listen_to_client*, the reader object awaits a message from each client asynchronously. If there is a message it is decoded and converted into either a command or a dictionary (for more on message passing, see below). Following message receipt, several things can happen in the server, depending on the type of message. Briefly, either the *broadcast* function unpacks the message from the dictionary and, for each client whose port number does not match the current client thread the server is waiting on and whose room number matches the message's room address, copies the message to the global 'mailbox,' or the server resolves the command, usually involving the creating of a dictionary 'gram' message which is echoed to the client.

3.6. Within an infinite asynchronous loop in *send_to_client* the server in turn (asynchronously) checks the 'mailbox' (*check_mail*) for each client it is currently serving. The mailbox is a dictionary that keys port numbers to message values. If a message for the current client thread's port number is collated to a message in the mailbox, this message is removed from the dictionary and written to the client thread, where the clients own listen method will print it.

3.7. As long as the server is running, it calls a number of functions and performs chatroom manipulation. More detail on these functions is provided below. The server may disconnect from a client at any time by typing the client's port number into standard input. The server then sends an exit message to the client which causes the client to close the connection on the client's side. If the client exits, either because it was sent a exit command from the server or because the client suddenly crashes without warning, an exception being raised in the server. The server prints to standard out that the client has exited and removes the client from the appropriate chat rooms. As long as the server is running, it calls a number of functions and performs chatroom manipulation. More detail on these functions is provided below.

4. The Client:

4.1 Once the server has been started, multiple instantiations of the client module can be executed and run simultaneously. New clients are created by executing the client.py script.

client.py:

When a new client is created and runs, the following happens:

4.2. Each client has a unique port number which attaches to the socket of the server (8888). When the client connects to the server, the server adds a new coroutine to the its event loop to handle the client's open stream. The clients port number is then added to the list of active clients.

4.3. Within the main function, the client provides a username and then, like the server, the client also asynchronously listens to and sends messages to the server using the asynchronous gather handler. The client is immediately added to a general chatroom (see 'Chatrooms' below

4.4. Within an infinite asynchronous loop in *listen_to_server*, the client, like the server, waits for messages to arrive. These messages can either be dictionaries which are generally the result of a command echo (the server is responding to a client command), or they can be broadcast messages from the other clients attached to the server.

4.5 Within an infinite asynchronous loop in *send_to_server*, the client can send simple broadcast/room-specific messages to the server, or it can send a set of commands, some of which are passed as message 'grams' object dictionaries.

4.6. In the event of a server crash, the client checks if the writer stream to the server is closing or if reader stream as reached an eof. If either of these conditions are true, the client initiates the shut down procedure, which includes canceling all coroutines that are attached to the event loop. A disconnect message is then displayed to inform the user that the client has lost its connection to the server

5. Chatrooms:

5.1. The main function of the Chat Box revolves around the ability to create, join, switch to, and leave chat rooms. When users initially login, they are placed in a 'General' chat room. Each

client is greeted by a message that tells them they are in the General chat room. They can then send a series of commands to the server (for the detailed protocol for these messages, please see the next section). The client can access the following command-menu by typing $m!$ followed by a carriage-return:

5.2. List all active users: any client may at any time see all of the users chatting in any room by typing $u!$. This function currently does not have any filters, but future versions could support displaying friend lists or users who have been active within a certain time frame.

5.3 Create a chat room: In the beginning, when there is just one client, there only exists one General chat room. Any new client who attaches to the server's socket will be able to chat with anyone else in the General room. In this way it is a forum for all who enter. To make things more interesting, users can create rooms by typing the c command followed by the $c!\{name\}$ syntax, where $name$ is the name given to the new room. If the room name is accepted (no longer than 25 characters and must have at least one character), then the room is created successfully. The client is prompted and told they are a member of the new room and how to switch over to begin chatting in this room.

5.4 Join a chat room: After a client creates a room, they are automatically a member of that room. However, if another client wishes to become a member they must send a command to the server by typing the j command followed by the $j!\{room\_number\}$ syntax, which allows them to select a room to join from a list of created rooms. Currently, the Chat Box does not support private rooms, so once a user joins a room, they automatically become a member of that room. However, they can only see messages created within a room from the time they become a member. This is a unique feature of the Chat Box, as other chat applications or open internet forums allow joined users to see the chat history from even before they were members. Future functionality of the application will allow users to print the rooms they've joined to the screen. Currently, this version supports printing all rooms and users; however, the switch command will display all joined rooms based on client.

5.5 Switch to a chat room: After a user has joined a room, they can send a switch command to the server by typing s followed by the $s!\{room\_number\}$ syntax, choosing a room from among a list of rooms they have joined. Switching rooms allows a user to chat in a separate space with only the other clients who have joined that room and are currently switched to this room. Therefore, if a user switches to 'Room A,' their messages will not be displayed in the General room, but only to the users who have joined and switched to 'Room A.'

5.6. Leave a chat room: A user can, at any time leave a chat room by sending the leave command $l!$ to the server. This will remove them from the list of clients that have joined the room. It also means they can no longer switch to this room unless they join again. Note that a client may not leave the General room without quitting, but they are prompted by the server with an 'are you sure' message if they wish to do so.

5.7. The user my exit the program by typing "exit()" at any time. This action will result in closing the stream to the server and canceling all coroutines that have been attached to the client's event loop, at which point an exit message will displayed to standard out and the the client program will exit.

6. Command Protocol:

6.1. The command protocol uses a very simple vim-esque command syntax that allows a client to quickly send commands to the server. These commands are:

1. $u!$: list all active users.
2. c : call create chat room prompt
3. $c!\{room\_name\}$: create and name the chat room.
4. j : call join chat room prompt
5. $j!\{room\_number\}$: join a room
6. s : call switch rooms prompt
7. $s!\{room\_number\}$ switch rooms
8. $l!$: leave a room
9. b : call broadcast prompt
10. $b!\{[rooms]\}$: broadcast to selected rooms
11: $q!$: quit the Chat Box

There is not much to say about the client side of these commands, as they are simply typed by the user and sent to the server. On the server side, the server waits for messages from the client (see section 3.5) and if it receives a message, it first checks from a list of commands if the message is a command. If it is a command, the server changes its normal behavior for message passing protocol (see below). In the command protocol, the server behaves in the following ways, depending on the command from the list above:

6.2. To list all users, the server creates a dictionary called an 'echo_roomGram.' The echo_roomGram has fields 'gram type' = 'echo rooms', 'rooms' = chatRooms, and 'tracker' = clientTracker. The 'gram type' field tells the client what kind of message the server is sending. The 'rooms' field tells the client that the object it is receiving is a dictionary of chatRooms. This allows the client to have access to the list of chatRooms. The 'tracker' field allows the client to see in which rooms each of the other clients is currently active. On the server side, the echo_roomGram is created and updated each time a list command is received. The server sends the echo_roomGram to the client, where the client parses the message based on the fields it expects to find. The client can anticipate these fields based on the 'gram type' field. Other gram-types contain different fields.

6.3. To call the create a chat room prompt, the server simply echoes a print statement to the client. This could be a completely client-side function, since the server doesn't need to know that the client is printing a prompt. However, we have used the command protocol here to ensure future functionality can be smoothly implemented.

6.4. To create a room and name it, the server actually receives a 'createGram' from the client. That is, the client actually creates and packs this dictionary on the client side and sends it to the server. On the client side, the createGram is created if the name of the room is both not blank and no longer than 25 characters. There is currently only one field in the createGram, 'create' whose value is the name of the room. Future implementations of this dictionary could include fields which provide other options such as private room, topic-specific, etc.

6.5. When the server receives a join chat room prompt, it does not just echo a menu, but dynamically creates the prompt menu for each client based on the current status of created rooms and joined. The server creates a 'room_chooseGram' with fields 'gram type', 'prompt', 'rooms',

'joined rooms,' and 'length.' The gram-type is 'room choose,' which tells the client to anticipate the other fields. The client parses the prompt message on their end and prints it. The client saves the 'rooms,' 'joined rooms,' and 'length' fields to create a joinGram (see 6.6).

6.6 When the client enters the join a room command (j! followed by the room number), they create and pack a joinGram on their end. This 'gram' includes the desired room to join (all rooms are open and public in the Chat Box). The client program verifies the room is valid and sends this gram over to the server. When the server receives a joinGram, it recognizes that it is a joinGram because the dictionary field is 'join'. It then adds the room name to the 'joined rooms' field in the clientTracker, but, importantly, the server does not switch the client to the joined room. For this, the switch feature is needed.

6.6. Similarly to the behavior of the server upon receipt of a join room prompt, a switch room prompt (when the client types s) commands the server to dynamically create a list of already-joined rooms (for the commanding client) along with a prompt for the client telling her how to switch over.

6.7. To switch rooms, the client must enter s! followed by the room number of the desired room switch. The switch rooms command creates a switchGram on the client side which is very similar to the join gram, with the only difference being once it receives a 'switch' dictionary, the server extracts the room number and changes the client's 'current' room in the clientTracker to the desired switch room.

6.8. To leave a room, the client simply sends the leave room command to the server, which in turn moves the client to the 'General' room, taking the left room off of their 'joined rooms' list.

6.9. The call broadcast room prompt is similar to the switch prompt, in that it dynamically sends back a list of all joined rooms to the client, along with a prompt.

6.10 The client enters b! followed by a list of rooms to receive the broadcast (they must be rooms the client has joined). The client creates and packs a 'broadcastGram' and sends it to the server, who extracts from it the list of rooms and the message to send to the rooms. The server then puts the message in the mailbox of all clients whose 'joined rooms' currently includes the room in the broadcast list for each room on that list.

6.11. The user may exit the program by typing $exit()$ or q! at any time. This action will result in closing the stream to the server and canceling all coroutines that have been attached to the client's event loop, at which point an exit message will displayed to standard out and the the client program will exit.

7. Message Passing Protocol:

7.1. When the user enters a string into standard input the client creates a dictionary called msgObj and places the message string in the msgObj. The dictionary, msgObj, is then converted to a string so that it can be passed through the stream to the server. When it arrives at the server, the server converts the string back into a dictionary, and adds additional fields, such as the client's port number and the user name. These fields allow the message to be delivered to the appropriate chat rooms and allow other users to view the sender's usernames when they receive the message.

7.2. Using a routine *checkMailBox* The mailBox is a virtual dictionary of port numbers and string messages that the server manages. The server gathers up all of the messages for each of its clients, sending the messages in turn if anything has been placed in the box. Note that if a message is to be broadcast to multiple client threads, this is made possible by each mailbox getting its own copy of the message. In this way, the message is duplicated to each client thread much like a flyer is stuffed indiscriminately into mailboxes (except here we have the discriminating factor of the room number and its members).

7.3 Passing dictionaries as strings and converting them right before they enter/right after they leave the sockets is made possible by abstract syntax trees. The ast module allows a string formatted as a dictionary or other valid data structure to be converted back into a string. There are other useful instances of abstract syntax trees, such as when the client broadcasts to multiple rooms. Here the client enters a comma-separated list of numbers which gets converted into a data structure list and then converted into a string which then gets placed into a dictionary which itself is converted to a string, passed to the server and converted back into a dictionary where the string list is extracted and converted into a list again.

References:

asyncio: https://docs.python.org/3/library/asyncio.html
https://stackoverflow.com/questions/31510190/aysncio-cannot-read-stdin-on-windows

Authors' Emails:

Evan Deposit: edeposit@pdx.edu
Charles Bolton: boltch@pdx.edu