Chat Server Example

This lesson walks the reader through the design and implementation of a toy chat-server.

Chat Server

Problem

Let's embark on a more realist albeit toy-example of creating a server that enables instant messaging or chatting among various participants. You may consider this problem a dumbed-down version of full-blown real life chat room servers.

Solution

We can implement our chat server using either a multithreaded approach or a single-threaded asyncio-based approach. We'll explore and implement both the approaches and contrast them at the end.

Our chat server only allows three operations and receives them as commands.

- **register**: A client can register with the server by sending a comma separated command string **"register,<username>"**, where username is the name of the client intending to register.

- **list**: A client can retrieve the list of other clients already registered on the server by sending the command string **"list,friends"**. The client receives a comma-separated string of usernames including its own.

- **chat**: A client can request a chat message be sent to a friend using the command string **"chat,<username>"**, where username is an existing registered client on the server.

Important Note

The reader might find us making assumptions and taking shortcuts in the implementations we present. The reason is that our emphasis isn't on the networking aspects of the problem or its stability. Rather, the focus is on illuminating the differences between the two concurrency paradigms. We'll call out the simplifications and assumptions we make as we go along.

More generally, the implementations we present work for the happy paths we discuss. For example, we don't test for a client trying to erroneous input or attempting multiple register requests, etc. Also, the implementations run a simulation where the connected clients randomly chat with each other and we don't implement a graceful shutdown. We want the reader to be focused on the meat of the lesson here rather than be distracted by the niceties of a fully working solution.

Multithreaded Approach

In the multithreaded approach, we'll have a **main thread** that listens for incoming connections. Once a client connects, the **main thread** spawns a worker thread that maintains a persistent connection with the client and handles all future requests from the same client. This is in line with how web-servers generally handle incoming requests. The choices include forking the current process to deal with a new request or handing-off the client request to a pool of worker threads. It should be obvious to the reader that the **main thread** can't risk handling a client request as it can drop new incoming connections during the time it spends processing the current client's request. The **main thread**'s logic appears below:

Main thread Implementation

```
    def run_server(self):

        # networking stuff to setup the connection, that the
        # can ignore
        socket_connection = socket.socket()
        socket_connection.bind(('', self.port))
        socket_connection.listen(5)

        # perpetually listen for new connections
        while True:
            client_socket, addr = socket_connection.accept()

            # spawn a thread to deal with a new client and imm
ediately go back to
            # listening for new incoming connections
            Thread(target=self.handle_client, args=(client_soc
ket,), daemon=True).start()
```

The **main thread** spawns a **worker thread** that executes a method `handle_client()`. The **worker thread** is also passed-in all the necessary details to further the communication with the client. The

details to further the communication with the client. The **handle_client()** method is shown below:


```
    def handle_client(self, client_socket):
        user = "unknown"

        while True:
            data = client_socket.recv(4096).decode()

            command, param = data.split(",")

            # register handler
            if command == "register":
                print("\n{0} registered\n".format(param))
                with self.lock:
                    self.clients[param] = client_socket
                user = param
                client_socket.send("ack".encode())

            # list handler
            if command == "list":
                with self.lock:
                    names = self.clients.keys()

                names = ",".join(names)
                client_socket.send(names.encode())

            # chat handler
            if command == "chat":
                to_socket = None
                with self.lock:
                    if param in self.clients:
                        to_socket = self.clients[param]

                if to_socket is not None:
                    to_socket.send(("{0} says hi\n".format(use
r)).encode())
                else:
                    print("\nNo user by the name <{0}>\n".form
at(param))
```

Important aspects to note about the `handle_client()` method includes:

- The chat command simply sends a pre-defined hi message to the intended recipient in order to keep the implementation simple.
- The method makes use of a dictionary `self.clients` and `self.sockets`. The `self.clients` is a username-to-socket mapping. Say if Jane wants to send Zak a message then the **worker thread** should be able to find the socket on which to send Zak a hi message.
- We use a `Lock` object to guard the dictionary `self.clients` as multiple threads could potentially manipulate it.

The complete `ChatServer` code appears below:

Complete code for ChatServer

```
class ChatServer:

    def __init__(self, port):
        self.port = port
        self.lock = Lock()
        self.clients = {}

    def handle_client(self, client_socket):
        user = "unknown"

        while True:
            data = client_socket.recv(4096).decode()

            command, param = data.split(",")

            # register handler
            if command == "register":
                print("\n{0} registered\n".format(param))
                with self.lock:
                    self.clients[param] = client_socket
                user = param

            # list handler
            if command == "list":
                with self.lock:
                    names = self.clients.keys()

                names = ",".join(names)
                client_socket.send(names.encode())

            # chat handler
            if command == "chat":
                to_socket = None
                with self.lock:
                    if param in self.clients:
                        to_socket = self.clients[param]

                if to_socket is not None:
```

```
                    to_socket.send(("{0} says hi\n".format(
user)).encode())
                else:
                    print("\nNo user by the name <{0}>\n".f
ormat(param))

    def run_server(self):

        # networking stuff to setup the connection, that th
e
        # can ignore
        socket_connection = socket.socket()
        socket_connection.bind(('', self.port))
        socket_connection.listen(5)

        # perpetually listen for new connections
        while True:
            client_socket, addr = socket_connection.accept
()

            # spawn a thread to deal with a new client an
d immediately go back to
            # listening for new incoming connections
            Thread(target=self.handle_client, args=(client_
socket,), daemon=True).start()
```

Now let's look at how a client may be implemented.

Client Implementation

A client follows the below sequence when it starts:

- Connect and register with the chatserver.
- Add a slight delay to let other clients connect and register with the chat server.

- Request a list of friends from the chatserver.
- Spawn a perpetual thread to receive incoming chat messages from other users.
- Enter a perpetual loop to randomly select a friend from the retrieved list and send a chat message to.

The above flow is captured in the client code below:

User class

```
class User:

    def __init__(self, name, server_host, server_port):
        self.name = name
        self.server_port = server_port
        self.server_host = server_host

    def receive_messages(self, server_socket):
        while True:
            print("\n{0} received: {1}\n".format(self.name, se
rver_socket.recv(4096).decode()))

    def run_cleint(self):
        server_socket = socket.socket(socket.AF_INET, socket.S
OCK_STREAM)
        server_socket.connect((self.server_host, self.server_p
ort))

        # register and receive ack
        server_socket.send("register,{0}".format(self.name).en
code())
        server_socket.recv(4096).decode()

        # wait for friends to join
        time.sleep(3)

        # get list of friends
        server_socket.send("list,friends".encode())
        list_of_friends = server_socket.recv(4096).decode().sp
lit(",")
        num_friends = len(list_of_friends)

        # start listening for incoming messages
        Thread(target=self.receive_messages, args=(server_sock
et,), daemon=True).start()

        while True:
            # randomly select a friend and send a message
```

```
            friend = list_of_friends[random.randint(0, num_friends - 1)]

            server_socket.send("chat,{0}".format(friend).encode())

            time.sleep(random.randint(2, 6))
```

Critique

As noted earlier our chat server and its users are very limited in their functionality. For instance, a user thread never refreshes the friend list from the server. A user doesn't gracefully disconnect from the chat server. The **`self.clients`** dictionary on the server-side doesn't have any cleanup. However, the limited implementation effectively conveys the multithreaded design of such a system. The chat server is composed of multiple threads with one of them being a **main thread** that listens for new incoming connections and the others handle individual clients. Shared state if any needs to be appropriately guarded. The pitfalls of such a design are:

- As the load increases on the system so do the number of threads. Creating, managing and tearing down threads takes part of the system resources away from serving clients.
- Such a system can experience high contention for shared data-structures as multiple threads attempt to use them simultaneously.
- Potential for deadlocks in the system due to logical errors/bugs.

In the next part, we'll run a simulation of our multithreaded chat server.


Back

Next

Async Sleep Problem

... continued

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/chat-server-example__asyncio__python-concurrency-for-senior-engineering-interviews)