... continued

Continues the discussion on implementing a chat server.

> The code widget below runs a simulation of the chat server with four users that message each other (including themselves) randomly.

```
from threading import Thread
from threading import Lock

import socket, time, random


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
                client_socket.send("ack".encode())
```

Note that our implementation makes generous use of daemon threads so that when the main program thread exits, all the threads are killed and the code-widget doesn't time out. The output will show various users receiving hi messages from other users.

Next, we'll pivot our implementation to an asynchronous one.

Asynchronous Implementation

The asynchronous paradigm to implement a chat server involves using a single thread, usually called the event loop. In the case of Python, we'll not implement an event loop from scratch rather leverage **`asyncio`** module for the job.

Similar to the multithreaded approach, we need an entity to continuously listen for new incoming connections. We'll use the asyncio's **`start_server()`** coroutine. The coroutine takes in the server's hostname and a port to listen for incoming connections on. Once a client connects with the server, the coroutine invokes a user-specified callback to invoke. The callback includes the details to read and write from the connected client. The code snippet to start the server would look like as follows:

```
    server_port = random.randint(10000, 65000)
    server_host = "127.0.0.1"
    chat_server = ChatServer(server_port)
    server = await asyncio.start_server(chat_server.run_server
, server_host, server_port)
    await server.serve_forever()
```

Note we are awaiting the **`start_server()`** coroutine. As a reminder, in order to make a coroutine run, we'll need to **`await`** it. The **`run_server()`** is the callback that gets invoked whenever a new client

connects. At this point the name *run_server* is misnomer since the server is really being run by asyncio's event loop but in order to remain congruent with the multithreaded approach we'll let the name be so. When we await the coroutine **`run_server()`** we are returned a **`Server`** object called **`server`**. Finally, in order to run the server perpetually, we invoke the **`serve_forever()`** coroutine defined on the **`server`** object.

Now let's examine the callback we pass into the **`start_server()`** coroutine:

```
    async def run_server(self, reader, writer):

        while True:
            data = await reader.read(4096)
            message = data.decode()
            print("\nserver received: {0} -- {1}\n".format(mes
sage, current_thread().getName()))

            await self.handle_client(message, writer)
```

The **`run_server()`** is a coroutine too. It is passed the **`reader`** and **`writer`** objects that can be used to either read from a connected client or send to it. The coroutine accepts the command from the client, decodes and passes it off to another coroutine **`handle_client()`** defined as follows:

```
    async def handle_client(self, message, writer):

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

        if command == "register":
            print("\n{0} registered -- {1}\n".format(param, cu
rrent_thread().getName()))
            self.clients[param] = writer
            self.writers[writer] = param

            # send ack
            writer.write("ack".encode())
            await writer.drain()

        if command == "chat":
            to_writer = None
            if param in self.clients:
                to_writer = self.clients[param]

            if to_writer is not None:
                to_writer.write(("{0} says hi".format(self.wri
ters[writer])).encode())
                await to_writer.drain()
            else:
                print("\nNo user by the name |{0}|\n".format(p
aram))

        if command == "list":
            names = self.clients.keys()
            names = ",".join(names)
            writer.write(names.encode())
            await writer.drain()
```

Note the following:

- We aren't using any locks. All the execution takes place within the same thread.

- We can combine the two coroutines `run_server()` and `handle_clients` into one. We kept them separate for comparison with the multithreaded approach.

- Note that all blocking operations involving receives and sends to a client are replaced with non-blocking coroutines which we `await`. It is very important to understand that if we have a blocking operation at any point, the entire application will come to a screeching halt!

- We introduce another dictionary to keep a mapping of readers-to-names so that we know who is trying to send a chat message to a user. Earlier, in the multithreaded version, we were able to store that information in the variable `user` which acted as a thread-local variable. However, since there's only a single thread, we need additional data-structures to store state.

Let's turn our attention to the `User`. Although the client code will run on machines different than the server machine, we'll make our clients run on in the same event loop as the one in which the server runs. Similar to the `ChatServer`, the `User` class will replace all blocking methods with coroutines that can be awaited. The `run_client()` method becomes a coroutine and is refactored as follows:

```
    async def run_client(self):
        reader, writer = await asyncio.open_connection(self.se
rver_host, self.server_port)

        # register
        writer.write("register,{0}".format(self.name).encode
())
        await writer.drain()
        await reader.read(4096)

        # get list of friends
        writer.write("list,friends".encode())
        await writer.drain()
        friends = (await reader.read(4096)).decode()
        print("Received {0}".format(friends))

        # launch coroutine to receive messages
        asyncio.create_task(self.chat(reader))

        friends = friends.split(",")
        num_friends = len(friends)

        while 1:
            friend = friends[random.randint(0, num_friends - 1
)]
            print("{0} is sending msg to {1} -- {2}".format(se
lf.name, friend, current_thread().getName()))
            writer.write("chat,{0}".format(friend).encode())
            await writer.drain()
            await asyncio.sleep(3)
```

Contrast the above coroutine with the multithreaded approach, where we launched a separate thread to receive chat messages, whereas, in the async version, we launched a coroutine `receive_messages()` to run on the same event loop as all the other coroutines to listen for incoming chat messages. The `receive_messages()` coroutine is as follows:

```
    async def receive_messages(self, reader):

        while 1:
            message = (await reader.read(4096)).decode()
            print("\n{0} received: {1} -- {2}\n".format(self.n
ame, message, current_thread().getName()))
```

In the next section, we run a similar simulation with async version of the chat server as we did for the multithreading one.

Back

Chat Server Example

Next

... continued

Mark as Completed

Report an Issue

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