Lab: Different patterns of the server

There are different patterns for the server application. In this lab, you'll be looking at a few of them.

We use a sphinx-like server as our running example for learning the different server patterns. The client sends questions to the server, the server acts as an oracle and send back one of the five replies: yes, very likely, maybe, not likely, no.

Simple single-threaded server

We begin with creating some helper functions that will be useful whichever server pattern will be adopted. The file `helpers.py` contains these functions. Take a look at this file. The five stock replies are stored here in the `replies` list. There is also a list of three example questions that will be used by the client. From the functions, you can see the progressing stages of server-client communication.

Server side

1. Server gets the address (network interface, port number) at which it will create a listening socket.
2. Server creates the listening socket, in `get_server_socket()`.
3. Server accepts incoming connections received at the listening socket, for each incoming connection, a new socket dedicated to that connection is created, in `setup_connections()`.
4. In `process_connection()`, server does error checking. If no errors, then hand off to another function to process the request from client.
5. Request from client is dealt with in `process_request()`. Bytes are received until the delimiter '?' which means a single question is received from client. The function `make_reply()` takes the question from client, randomly picks one of the five replies, and concatenates the question with the reply, to be sent back to the client. The actual receiving the bytes is done in the `recv_upto()` function.

6. Receiving the bytes up to a provided delimiter is done in `recv_upto(socket, delimiter)`. This is the same as what we have done in earlier labs. You cannot assume that the entire request will be received in one shot, since the receiving buffer might have been too full to store all of the bytes. So we use a loop to keep receiving more bytes until the delimiter is encountered. Notice that the delimiter is passed into the function as a parameter. This way, the client program can also call this function to receive replies from the server but with a different delimiter.

Client side

The client code is very simple, in `client_questions.py`. It takes two or three command-line arguments, the first two are the host IP address and the port number of the server. The third one is optional, if the argument '-e' is provided as the third argument, then it tells the program to create an error. We will go into this in more detail later. For now, do not provide the third argument on command line. So the client connects to the server (in error-free scenario) and sends all three example questions to server and prints the replies.

Run simple server

The simple single-threaded server program is even simpler, in `server_simple.py`.

Assume server IP address is 10.123.234.55. Obviously, when you run these programs, you should substitute in your own IP address.

Run the server then the client:

```
python server_simple.py 0.0.0.0 1234
python client_questions.py 10.123.234.55 1234
```

Remark. This server handles one connection at a time. Look at the while loop in the helper function `setup_connections()`, when a connection is accepted, it's handed off to the function that processes this connection, and only after that is completed, the server can continue and accept another connection. So if another client tries to connect to the server while it's busy with an earlier client, the new client will have to wait until the other client is finished.

This server pattern is not only vulnerable to denial-of-service attacks, but is also terribly

inefficient — there is a lot of idle time when the server is waiting to receive data and that time could be used to process other clients.

Multi-threaded server

We can make the server multi-threaded to increase efficiency. The main server process creates and starts a number of threads. It passed to each thread the listener socket. When a client connection request comes in, an available thread will be assigned by the OS to handle that connection through the function `setup_connection()`. This is in the program `server_threads.py`.

You can run it to try it out with multiple clients (use the same client program from above).

Remark. This server dedicates a thread to each client connection. Usually it's not a good idea to have too many threads, because the OS has to do context switching every time it switches from one thread to another, and this overhead adds up. It boils down to that the number of clients this server can service at a time is limited by the number of threads that the OS can handle without slowing down. And for the average OS, this is not a very large number (e.g., a few thousands).

Asynchronous servers

To see where we can improve efficiency and parallelism even more, consider a single thread in the above multi-threaded server. When it's waiting to receive data from client, the receive is blocking, so it's idle and this time is wasted and could be used to serve another client.

A server can instead be ***asynchronous*** — it serves multiple clients concurrently, that is, it simultaneously looks after sockets connecting to multiple clients, and whenever a socket has data from one of the clients, it processes that data. The receive is **not blocking**.

The server is asynchronous in the sense that it is not waiting on any single client, it is not synchronized with one client at a time, instead it is switching among the clients, serving them at the same time.

We look at one way of constructing an asynchronous server using the `asyncio` module

provided by Python.

Asynchronous versus synchronous

We contrast an asynchronous server with a synchronous one. Consider the example program in the file `async_sync.py`. This program takes a command-line argument, if it's 'sync', then it runs the synchronous server; otherwise if it's 'async', it runs the asynchronous server.

```
if (sys.argv[1] == 'sync'):
    # run clients synchronously
    # this will take 6 seconds in total
    sync_main()

elif (sys.argv[1] == 'async'):
    # now make them run asynchronously, i.e., concurrently
    # will only take 3 seconds
    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(async_main())
```

First, look at the synchronous server. It has a dummy function called `sync_netio()` that does network IO with a remote client, i.e., sends and receives data with a remote client. The `delay` parameter passed to the function is used to simulate network communication delay. So this server communicates to two clients, each communication will take 3 seconds.

```
# regular functions to do network IO with two clients
def sync_netio(delay, addr):
    time.sleep(delay)
    print("talked to {}".format(addr))

def sync_main():
    print("begin at time {}".format(time.strftime('%X')))
    sync_netio(3, 'client 1')
    sync_netio(3, 'client 2')
    print("end at time {}".format(time.strftime('%X')))
```

If you run this synchronous server, it will take a total of 6 seconds to finish, as expected.

```
λ python async_sync.py sync
begin at time 17:08:22
talked to client 1
talked to client 2
end at time 17:08:28
```

To make this server asynchronous, i.e., handle the communication with the two clients concurrently, we use the `asyncio` module. An asynchronous `coroutine` is defined, called `async_netio()`, in which communication to a client is simulated with a `delay` value that's passed in. An asynchronous server coroutine is defined, called `async_main()` which assigns each client to a separate `task` by calling `async.ensure_future()` and passing to it the coroutine defined to handle the communication (`async_netio()`). Then the client coroutines are run **asynchronously** or **concurrently**, by the `await client` calls.

```
# Define asynchronous coroutines
# to run the network IO to two clients asynchronously (concurrently)
async def async_netio(delay, addr):
    await asyncio.sleep(delay)
    print("talked to {}".format(addr))

async def async_main():
    client1 = asyncio.ensure_future(async_netio(3, 'client 1'))
    client2 = asyncio.ensure_future(async_netio(3, 'client 2'))
    print("begin at time {}".format(time.strftime('%X')))
    await client1
    await client2
    print("end at time {}".format(time.strftime('%X')))
```

In the main function, we create an event loop that runs the asynchronous server and takes care of all the actions under the hood involving switching among the client sockets whenever there is a receive/read or send/write event.

```
elif (sys.argv[1] == 'async'):
    # now make them run asynchronously, i.e., concurrently
    # will only take 3 seconds
    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(async_main())
```

If you run this asynchronous (dummy) server, it will only take 3 seconds in total, because the clients were running asynchronously/concurrently.

```
λ python async_sync.py async
begin at time 17:19:43
talked to client 1
talked to client 2
end at time 17:19:46
```

The entire program is listed below.

```python
import asyncio
import time, sys

# regular functions to do network IO with two clients
def sync_netio(delay, addr):
    time.sleep(delay)
    print("talked to {}".format(addr))

def sync_main():
    print("begin at time {}".format(time.strftime('%X')))
    sync_netio(3, 'client 1')
    sync_netio(3, 'client 2')
    print("end at time {}".format(time.strftime('%X')))

# Define asynchronous coroutines
# to run the network IO to two clients asynchronously (concurrently)
async def async_netio(delay, addr):
    await asyncio.sleep(delay)
    print("talked to {}".format(addr))

async def async_main():
    client1 = asyncio.ensure_future(async_netio(3, 'client 1'))
    client2 = asyncio.ensure_future(async_netio(3, 'client 2'))
    print("begin at time {}".format(time.strftime('%X')))
    await client1
    await client2
    print("end at time {}".format(time.strftime('%X')))

if (sys.argv[1] == 'sync'):
    # run clients synchronously
    # this will take 6 seconds in total
    sync_main()

elif (sys.argv[1] == 'async'):
    # now make them run asynchronously, i.e., concurrently
    # will only take 3 seconds
    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(async_main())
```

Figure 1: async_sync.py

Asynchronous TCP server

Now we can write our TCP server from before as an asynchronous one. We need to define an asynchronous coroutine to handle each client conversation.

The program is in `server_asyncio.py`.

```
import asyncio, helpers

# Define asynchronous coroutines
# to talk to multiple clients asynchronously (concurrently)
async def handle_client(reader, writer):
    # get address of client on the other end of this connection
    addr = writer.get_extra_info('peername')
    print('Connected to {}'.format(addr))
    while True:
        data = await reader.read(4096)
        if not data:
            break
        elif data.endswith(b'?'):
            reply = helpers.make_reply(data)
            writer.write(reply)
            await writer.drain()
        else:
            print("Got incomplete question {} from {}".format(data, addr))
            break
    writer.close()

if __name__ == '__main__':
    addr = helpers.get_address()
    loop = asyncio.get_event_loop()
    coroutine = asyncio.start_server(handle_client, *addr)
    result = loop.run_until_complete(coroutine)
    print('Listening for incoming connections at {}'.format(addr))
    try:
        loop.run_forever()
    finally:
        server.close()
        loop.close()
```

The asynchronous server is created by calling the `start_server()` function of `asyncio` module. The coroutine that handles a client is passed to this function as a parameter, as well as the server address and port. This allows *callback* of the passed-in coroutine, in this case `handle_client`. Whenever a new client connection is made, this function is called and receives a `reader` and a `writer`, for receiving and sending data, respectively. Because a coroutine (a function defined with the keyword `async`) is passed in for callback, a task will be automatically created for each new client connection.

Same as above, an event loop is created and is asked to run the server coroutine.

You can try it out with the same client program from the previous sections, `client_questions.py` Run the server first.

```
λ python tcp_async.py 0.0.0.0 1234
Listening for incoming connections at ('0.0.0.0', 1234)
```

Then run the client (assuming the server IP address is 10.123.135.72).

```
λ python client_questions.py 10.123.135.72 1234
Will I be happy tomorrow?  Very likely.
Is general artificial intelligence possible?  Not likely.
Is there true free will?  Not likely.
```

For this server, when multiple clients connect to it, it will not start multiple threads to deal with each client. Instead, in a single thread, it will simply have tasks assigned to clients and run those concurrently. The underlying mechanisms are **hidden** by the API of the `asyncio` module. But what is being done underneath is roughly a constant polling of the multiple client sockets, and whenever one client socket has an event — either it has data to receive or ready for data to be sent — it will be processed by the server. You can think of there being a dictionary of file handles to sockets, and the OS is constantly monitoring these sockets, and whenever a socket has an I/O event, it passes this event up to the server. The server then looks up the dictionary to find the socket and handles that receive or send.

The coroutine passed to the `start_server()` function is always defined to have two parameters passed in: `reader` and `writer`. The `reader` object is used to receive data from the socket by calling `reader.read()`, the `writer` object is used for sending data, `writer.write(data)`.

Remark. The advantage of this asynchronous server pattern is that a single thread can handle thousands of clients. The reason is that **switching between clients no longer results in an expensive context switch between threads, instead it can be thought of as just a simple key lookup in a dictionary data structure**.

Task

Modify the client program in `client_questions.py` to

- continuously cycle through the questions and send to server, in an infinite loop;
- before sending each question, sleep for 2 seconds to simulate network delay.

Modify the server program in `server_asyncio.py` to reply to a maximum of 10 questions from each client, then close the connection.

To Submit

Submit the task.