Exercise3EscapeRoomAsyncio

Jump to bottom

sethnielson edited this page 10 days ago · 7 revisions

Part A: Adding a Hammer

Read through the sample code of the given escape room and modify the code such that you can add a hammer that you can then 'get' and thus add to your inventory. This is a relatively easier part of the assignment and will judge your ability to read through the code and modify it to fit a given specification

Part B: Using Aysncio

Part 1: Difference between Aynscio and Sockets

In the previous exercise we used sockets as the API to connect to server or have a client connect back to us. Sockets while powerful do have standout drawback; they block the code. Lets look at at a quick and small example to really understand this issue. Lets consider a client and server that are communicating with each other, if the server is waiting on the client to send data (as you did in the previous exercise), the socket blocks all other code execution from occurring until the server receives data from the client. This is where Asyncio comes into the picture.

Part 2: Event driven Asyncio

The Python3 documentation describes Asyncio as a library that allows you to write concurrent code. What this essentially boils down to, is the ability to do multiple things while you 'wait' for a certain even to trigger. Referring back to our previous example, unlike with sockets, code written with asyncio allows you to perform different tasks while 'awaiting' a certain input to be received (from something like a client or server). This is very similar to interrupts in microprocessor execution where the microprocessor continues to execute its current instruction but stops and handles an interrupt when it sees an Interrupt Request.

Part 3: What is a transport object

Those of you who read the assigned reading, (if you haven't already, please go read it) would have noticed something called a transport object. This transport object is the API through which you can send and receive data over a network. In the sample code that was given to you, you used to `stdin` and `stdout` to send and receive data. In the case of Ex2, you used the socket API to send and receive data. Here, things are a little different, with asyncio you can use the `transport.write` methods to send data and you use the `data_received` method to read data. But a question that may have risen by now is, 'Well, where does this transport object come from?' The answer to this question is Asyncio itself. When you connect to a server or client, Asyncio calls the connection_made method of your class and returns a transport object which allows you to send data over the network. This is why a lot of the code you've seen in the docs looks like:

```
def connection_made(self, transport):


    `    self.transport = transport       ` <- This is object through which you send data
```

Part 4: This is totally different from sockets, what is going on?!

Like it was stated, Asyncio is 'event driven', when you create an instance of your server or client, Asyncio generates an instance of your server class and first calls the `connection_made` method and returns the transport object with which you can read or write data over the network. The `data_received` method gets called when either you client or server instance "receives" data. Thus with this event driven you can program a server or client to be far more responsive and quicker in its execution. This is a very important concept that you must understand, if you do not, come talk to the TA or the professor and get this concept cleared.

Part 5: Echo Examples

There are two files under `src\samples` that are an echo client and echo server each. Lets start by looking at the echo server. The first thing you see is a class definition for an `EchoServer`, this is basically a template of an *instance* that the event loop will generate for us. For now, ignore the `__init__` method and look at the `connection_made` method. When you ask the `event_loop` to *create* an instance of this class, the first method it calls is the `connection_made` method of the EchoServer class. The transport object that is being returned to you in this method is all happening under the hood with asyncio and the event loop. This is very similar to automatic gear shift car, where its shifting gears for you, without any manual intervention from your part. (I don't know why I'm making so many car analogies, THEY JUST FIT). If you read the client file as well, you'd notice that it too has a `connection_made` method, this is because for a connection to be made and established, both a client and server need to connect to each other. However, you will also notice that the definitions for the methods are actually different. There is a reason behind this, depending on the nature of the connection, in certain cases, the client sends the server a message telling the server that it has a connection. In the case of this assignment however, you server actually sends a message to the client, when the client connects to it. This is an important part of the assignment and I will refer to it later. Server:

```
def connection_made(self, transport):
     self.transport = transport
```

Client:

```
def connection_made(self, transport):
     self.transport = transport
     self.transport.write("Isn't this the best class ever?!".encode())
```

So what is actually going on, to break it down, the event loop in the client code starts to establish a connection. The event loop accordingly, calls the `connection_made` method within the client class. Concurrently, the server notices a new inbound connection and calls its own `connection_made` method.

BUT WAIT THERES MORE!!!

Because the client sent data over the network using the transport object, the server event loop noticed inbound data. As a result it called the `data_received` method:

Server:

```
def data_received(self, data):
     self.transport.write(data)
```

Accordingly, the event loop executed the code defined in the `data_received` method, and sent the data it received back to the client that send it using the transport object it received using the `connection_made` method. The client event loop responds in a similar way, due to the new inbound data and calls its own `data_received` method.

As for the other code you see in the both files, most of it relates to acquiring an event loop and then creating instances of both your client and server class. You must note here, that this is not necessarily the way you must implement your code for the assignment. The code written at the end was for simplicity for the sake of this wiki. Feel free to talk to the TA or the professor on how this part must be written. (Don't worry, its not that different, the documentation may give you the answer too!)

Part 6: Switching to Asyncio

This is where the meat of the assignment lies, you have to create a Escape Room Server and Client using Asyncio.

Part C: The Autograding Protocol

You need to create your client and server to work with the autograder (or create a version that does). Here is how the protocol works.

1. Launch your server on a port of your choosing. I will call your server Student Server and your port "N"
2. Launch your client. I will call your client Student Client
3. Your client will connect to the autograde server on 192.168.200.52:19003.
4. The Autograde Server will transmit a human readable message asking you to submit a command.
5. For running your test, your response MUST be: `SUBMIT,<name>,<email>,<team number>,<N>` where N is your port from the first step.
6. The Autograde Server will respond with `SUBMIT: OK, <test_id>` or `SUBMIT: FAIL, <reason>`. You should only proceed if it responds `SUBMIT: OK, <test_id>`
7. Once you get the OK, you should submit steps that BOTH *get the hammer* and escape the room. You can submit your steps in any order so long as you successfully exit the room (and get the hammer).
8. Once you exit the room, the Autograde Server will send any final messages from the game followed by a `CLIENT TEST: OK` or if you died/had an error, `CLIENT TEST: FAIL, <reason>`. You should only proceed if it responds `CLIENT TEST: OK`
9. Once you get the OK, the Autograde Server will launch an Autograde Client that connects to your Student Server on port N.
10. Your student server does not need to do any special I/O. Upon connection from the client, it should simply redirect output from the EscapeRoom game object using the

`output=write_function` pattern from exercise 2. Any messages received should be sent to the game through the `game.command` pattern also from exercise 2.

11. Your student server can close the connection once the game is over, either because of escaping or dying.
12. Once the Autograde client finishes, the Autograde server will send a `SERVER TEST: OK` or a `SERVER TEST: FAIL, <reason>`. If you get the `SERVER TEST: OK`, you successfully completed the assignment.

Once you have completed the test, you can always find out if you passed or failed by performing the following protocol:

1. Connect your client to the autograde server
2. The autograde server will send a human readable string asking you to submit a command
3. Your client will transmit, `RESULT,<test_id>`. You must submit the test_id received from your original submission.
4. The server will respond `TEST: PASS` or `TEST: FAIL`. It will not have the reason you didn't pass, so you should pay attention to the output from when you run your test.

Grading

If you pass the test you will get 10 points.

Clone this wiki locally

```
https://github.com/CrimsonVista/20194NetworkSecurity.wiki.git
```