PROXY HERD WITH ASYNCIO

Mert Dusunceli

Abstract

In this project, we will try to see if Python's asyncio library is a viable framework for our application, which supposedly receives a lot updates, is accessible via various protocols and is mobile. We will make a simpler application, an application server herd, to decide if we can get rid of the bottleneck of our application, which is the application server.

1 Introduction

In this project, we will use Python's asyncio library in order to implement a single-threaded asynchronous application. Being asynchronous means switching tasks in order to save time compared to synchronous functions where each and every task has to be done in order. If implemented correctly, asynchronous programs with many functions should be faster than their synchronous counterparts, since the thread running through it spends less time waiting and more time doing work.

2 Asyncio Library

Asyncio is a library that provides infrastructure for writing single-threaded concurrent code using coroutines and running network clients and servers. [3] Using asyncio, we can implement an asynchronous server and have these servers communicate with each other. Before Python 3.5, this was done using "yield from," but I guess "await" is a better word choice. [4] (And probably has better implementation) The library is called with "import asyncio" in Python 3.5 and higher.

Coroutines:

Previously, they were called generators, which were introduced in Python 2.2 [4]. Coroutines are defined with the keywords "async def" and they are basically functions where the execution can be stopped. Coroutines can "await" other coroutines, or they can return values to other coroutines. More importantly, all of the functions in a program need to be defined with "async def", meaning they all need to be coroutines, in order for your program to be asynchronous.

Event Loops:

Event loops, as their name suggests, have a queue of events and the loop brings one task after another so that the running thread can work on that task. When a coroutine is awaited, the event loop stops executing the task and it gets added to the queue. [2]

The following is a simplified snippet from my code, partly taken from asyncio documentation.

```
async def handle_input(reader, writer):
   data = await reader.read(1000)
   message = data.decode()
   answer = await self.handle_message(message)
   encoded = answer.encode()
   writer.write(encoded)
   await writer.drain()
   writer.close()

def main():
loop = asyncio.get_event_loop()
coro = asyncio.start_server(handle_input,
'127.0.0.1', 8888, loop=loop)
server = loop.run_until_complete(coro)

try:
   loop.run_forever()
except KeyboardInterrupt:
   pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

Now, we will closely look at the functions used above.

We need an event loop in order to start processing messages, and get_event_loop() function creates a

new event loop when it's called from the main. Since we would need to read/write both to clients and the servers, I chose start_server() function, which is automatically called with StreamReader and StreamWriter objects, over create_server(). [5] The loop.run_until_complete(arg1) runs the loop until arg1 is done. In the implementation, the main loop waits for the server to be closed and the server waits until coro is done, which starts the server and handles the input given. Also, it can be seen from the handle_input() coroutine that the functions that it calls are actually coroutines, and both of them are awaited.
We also have to note that aiohttp framework is one of the few frameworks that work nicely with asyncio. The main reason for this is that asyncio is a still relatively new framework.

3 Implementation

My implementation principle for this project was to start simple and build on top of it. Basically, Prof Eggert suggested that we should build one server that can process IAMAT and WHATSAT messages that come from the client and answer back to the client. Therefore, I first created a server object and had clients as a dictionary in the server. Then, I added more dictionaries that use clientID as the key to store more information such as the time difference and the server name that client is stored the most recently.

4 Problems

The first problem I faced was I did not know where to start in asyncio documentation. Luckily, the TA slides were helpful because they at least pointed at the correct place. By looking at the documentation, I figured out how to implement the server and client sides. I did most of my testing with a client.py file, so when I tried using telnet for testing for the first time, I couldn't input more than one line (such as first inputting IAMAT and then WHATSAT commands). The reason for this was that my client.py was sending one message at every loop, so it was internally running the loop, sending a message and closing the loop. The tricky part was sending more than one message in the same loop. I tried to fix this by reading until EOF, which was, in telnet, the CTRL+] key. However, I faced another problem. My servers were trying to read input in a loop until the client (either client.py or telnet) was closed by the user. This caused some very wrong empty line inputs to the server, which responded to these requests by sending "? ", complying to the spec. I fixed this by not accepting empty lines for parsing, which solved my problem. Also, since my server was open until manually closed, its reader was continuously waiting for input and sometimes it threw a ConnectionRefusedError. I fixed this by putting a try/except statement around my reader, where I dealt with this exception. However, my code is still buggy. Sometimes, while propagating the AT message to the other servers, I get ConnectionRefusedError from writer.drain(). However, as far as I can tell, the code is working perfectly. Therefore, I do not know how to solve this problem since trying to solve it like I solved the problem with reader caused the servers go into infinite loop.

5 Python vs Java

Even though Python's asyncio is a strong contender, we should still compare it with another popular implementation, which is Java.
Type Checking:
Python uses dynamic type checking while Java uses static type checking. This means that Python uses duck typing where the philosophy is "If it walks like a duck and talks like a duck, it is a duck." This ultimately means that Python does not care about the object type, it just assigns the object to the name. Moreover, one can reassign another object to the same name later in the program. On the other hand, Java is a statically typed language. This means that, at compile time, every name must have a type. If it is bound to an object, that object must also be the same type. Python may be easier to write but it feels ambiguous to the reader at some points, whereas Java is more clear and easier to follow because every variable has a type attached to it and therefore, there's no confusion.
Memory Management:
When Java stores objects and it runs out of space, its garbage collector stops every other thread and then frees the storage. The storage it frees is actually the objects which are on the heap but without any references. These objects are called 'garbage'. Garbage collection is actually a very expensive process, because, especially if the heap is large, we do not want to wait for a full second or two for garbage collector to free up some space. Python also uses a garbage collector, but for the purpose of our application, it is better because it is more efficient. Basically, Python uses a reference counter and frees up the object as soon as the reference counter reaches zero. This means that unlike Java, Python will not hold onto unused

objects for an unknown time. Moreover, Python will definitely not need to wait for the garbage collector to do its job while Java has to wait for it and it is obvious that Java's garbage collector will cause a problem if our program is large and it has to delete a lot of objects in order to free up some space.
Multithreading:
In Python asyncio, the event loop runs the tasks one at a time. When it encounters an 'await,' the event loop switches to the next task. This is all happening in a single thread. In order to support multithreading in Python, we need Global Interpreter Lock (GIL). [1] Without the lock, we would have the general concurrency problems such as incrementing a counter more than once, etc.
We can compare our implementation of asyncio in Python to multithreading in Java. Java's multithreading, too, has race conditions and deadlocks. Running a large, multithreaded Java application will definitely cause more problems than running a single threaded Python implementation using asyncio.

6 Python's Asyncio vs Node.js

Like Python's Asyncio library, Node.js is event driven and asynchronous. Node.js issues callbacks in a queue in the event loop, while Asyncio queues coroutines in the event loop. [6] They can stop processing after completing a task and move onto a completely different task. Where they differ is the implementation. Python's asyncio is single threaded, while Node.js is seen to be single threaded to the developers, but it implements its event pool using multithreading. [7] Interestingly, Node.js is known to be very faster compared other languages. Also, it works very efficiently in web development, mainly because its powerful integration with JavaScript. On the other hand, I would still choose Python, and therefore asyncio, on an application that relies on server-based communication.

7 Recommendation

Overall, I believe asyncio is a suitable framework for this kind of application, because it is single-threaded and concurrent at the same time. This is ultimately better than concurrent multithreaded frameworks such as Node.js, because its efficiency does not depend on individual machines that run the servers or the code. Python's garbage collector suits this type of application the best, because we don't know how big our database will be, so we cannot afford stopping threads in order for garbage collector, like Java's, to do its job of freeing up the heap. It is easy to write asyncio-based programs, as we have done in a week, so I don't think it would be hard to write a program that exploits server herds. Maybe having an asynchronous function to wait for a long time may break it, since the previous task doesn't return anything. Performance-wise asyncio is a lot better than any other library currently available in Python that can do this job, but it can still break when a lot of requests are given. Still, I believe asyncio approach is the best framework to implement an application server herd.

8 References

[1] https://wiki.python.org/moin/GlobalInterpreterLock
[2] https://medium.freecodecamp.org/a-guide-to-asynchronous-programming-in-python-with-asyncio-232e2afa44f6
[3] https://docs.python.org/3/library/asyncio.html
[4] https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/
[5] https://docs.python.org/3/library/asyncio-stream.html
[6] http://sahandsaba.com/understanding-asyncio-node-js-python-3-4.html
[7] https://hashnode.com/post/can-nodejs-be-considered-multi-threaded-cizz90ap6002jwp533jq97tl9