CS131 Project: Proxy Herd with Asyncio

Dorian Jimenez
205-376-959
Discussion 1B

Abstract

In this project, I implemented a Proxy Herd in Python using Asyncio. This library, Asyncio, allows us to execute code concurrently, which is a big help when making programs like this. I also looked into the various aspects of Asyncio, as well as Python as a whole. I then looked into other languages, how these other languages implement asynchronous behavior, and compared the languages and libraries. I do a brief comparison of asyncio to Node.js, and compare their asynchronous abilities. I do a deeper comparison between Java and Python, and I go over the various differences between the languages. I discuss the various pros and cons of each language, and the tradeoffs they make. I believe using Python and Asyncio are both good ways to implement a Proxy Herd (or anything similar), as it is easy to read, more or less easy to write, and overall a good implementation / abstraction to use to write concurrent code efficiently.

1. Introduction

The Proxy Herd I implemented needed to meet the specifications given in the CS131 Project webpage. We were given the task to make servers (exactly 5 servers, called "Juzang", "Bernard", "Jaquez", "Johnson", and "Clark"), and we were told to use asyncio to implement the functionality. Using code that can run concurrently or asynchronously helps in a task like this.

For example, in this project we were given the task that the servers must have server-to-server communication. The way that I implemented this was by creating a connection between servers every time I wanted to send server-to-server, and then killed the connection afterwards. Because of this, I must start multiple connections throughout my program's lifetime. Unfortunately, creating a connection is something that could take a while, as you are waiting for the other server to accept the connection, handle the connection, etc. Using asyncio can help us execute code concurrently which helps speed up our code execution in cases like these by allowing us to execute other code instead of just waiting.

2. Implementation

I will give a brief overview of how I implemented the servers, such as how I implemented the server-to-server communication, how the client requests were handled, and also how I used asyncio in my code.

When you start the server, it uses asyncio to start the server.

```
# Start Server
server = await asyncio.start_server(...)
await server.server_forever
```

It is important to know what the keywords "async" and "await" mean. The keyword "async" is used when declaring an asynchronous function. When you declare a function with the "async" keyword, it means that this function is a coroutine, meaning that this function can suspend its execution and give control to another coroutine if necessary. The "await" keyword can be used in an asynchronous function, and it suspends the execution of the current coroutine until the awaited function call is finished.

One question I had when I was doing this project was where the control goes to after you await a function. For example, if you suspend the execution of a function because you are waiting on a time-intensive function such as an I/O operation, where does the control go to? What code executes when this function is suspended? Well, you need to define a task in order to make the code you write actually asynchronous.

```
task = asyncio.create_task(function_call)
```

By creating a task, you can execute this code when you are awaiting for another function to finish. This is what makes your code asynchronous and actually makes your code run concurrently.

After the server is up and running, it waits for incoming connection requests. It handles requests by first reading in the data that is in the message, decoding it, and then parsing the message to extract the information. Parsing the message is a bit of a lengthy process, which involves a lot of string parsing as well as error checking. I use a dictionary which holds the various client connection IDs and the last known location of the client (it saves the entire AT message response as the value of the dictionary). This is the main data structure of the server, and it is what gets updated to store the up-to-date information of each client.

2.1 Messages

Disregarding invalid messages, there are three possible messages that can be received. If the message is an IAMAT message, then an AT request is created, the client information is stored in the dictionary, and then the message is propagated to all the servers (via legal server-to-server communications).

The way the propagation works is by sharing AT messages along servers and comparing timestamps. If a server receives an AT message, it checks to see if the client ID is in their dictionary. If not, then that means that a new client has connected to one of the servers, so add its location to the dictionary. If the client ID is in the dictionary, then the server checks the timestamp it has on file (from the previous request of the client) and sees if the AT request it received was at a later time than the AT request it has on file. If so, then that means the location of the client was updated, so update the dictionary. Else, do not propagate. By doing this, there is no propagation looping, and the propagation ends after sending the AT message a few times.

You can also receive a WHATSAT request. The way I handle this message is by making a Google API request, and formatting the request with the information the client gave. This process involves lots of string processing, as well as lots of error checking. The last type of message you can receive is an AT message. If a server receives this message, it is because of server-to-server communication. This is a result of the flooding algorithm, and to check timestamps, as explained above.

3. Problems

In this project, I ran into a multitude of issues, some regarding asyncio, some regarding Python, and others just due to typical software engineering hiccups.

One of the "issues" I ran into was due to Python's dynamic typing. Having used C++ mainly for the past few years, it took a while to get used to Python's dynamic typing. There were quite a few errors just because I was unable to infer the types of certain objects when writing my code, which led to errors later on in my code. For example, when passing parameters from the IAMAT message, I forgot that although the message parameters would be numbers (i.e. the timestamp is a number), it is the type of string. Therefore, you cannot do basic arithmetic on it until you cast it into an int.

```
# Example of Type/Error Checking
try:
    time_diff = time.time() - float(timestamp)
except:
    return "? " + message
```

Of course, casting arguments can create issues as well. According to the project spec, clients can make various mistakes in their requests to the server, therefore I needed to implement various error checks around my code. For example, it is dangerous to simply cast one of the message parameters from a string to an int, as there could be an error in the message from the client. For example, the timestamp could have a letter in it instead of it just being a number. Therefore, when you cast from a string to an int, it is important to do a try/except or an if statement that does type checking to avoid ValueErrors and other casting issues. This is one of the "issues" with Python's dynamic typing. Due to its flexibility, it can sometimes cause issues that are hard to debug or cause unexpected behavior from the program.

Another issue I ran into was a software engineering hiccup with the flooding algorithm. When I implemented the logic for the flooding algorithm, I made an error such that if the timestamps were equal, the server would propagate the message to the other servers. Because of this, the servers would constantly propagate the message they receive from other servers infinitely. Although this seems like a very basic bug to fix, it took me quite a while to figure out what the issue was.

3.1 Memory Management and Multithreading

I did not experience any issues with memory management. I think this is due to Python having automatic garbage collection. Because of this, I did not have to use any keywords such as free(), malloc(), etc. Python's reference count mechanism is able to detect when an object is no longer being used and is able to garbage collect it automatically. I also did not have any multithreading issues either. This is most likely because asyncio uses only one thread, not multiple threads. Also, due to Python's Global Interpreter Lock, there are typically not many issues due to multithreading.

4. Comparisons

4.1 Python and Java

Comparing Python to Java, there are many clear differences with the two languages. For one, Java is statically typed while Python is dynamically typed. Another immediate difference is that Python is interpreted, while Java is compiled and interpreted. Java is compiled into bytecode, which is then interpreted by the Java Virtual Machine. These differences make Python slower than Java, which is a very important performance comparison to note. Python and Java are also similar in some ways, as they both have automatic garbage collection. The way they do it is different however. Python uses reference counting (as well as other garbage collection algorithms) to see if garbage collection of an object is necessary, while Java uses Mark-and-Sweep (as well as other garbage collection algorithms) to do garbage collection.

Another quick thing to discuss is that Java allows multithreading while Python typically doesn't. Java has many functionalities such as locks, volatile, semaphores, ThreadPools, etc. to implement multithreading, while Python doesn't have this. Due to Python's Global Interpreter Lock, Python doesn't typically have multithreading unless you remove the Global Interpreter Lock or are using some other language such as C/C++ for a library (many Python libraries such as numpy were written in C/C++ and have multithreading capabilities).

4.2 Asyncio and Node.js

In my opinion, Node.js has a bit of a nicer API to think about when using asynchronous functionalities. Also, Node.js itself runs asynchronously, while asyncio uses Python, which does not itself need to be asynchronous. This changes the mindset of the programmer a bit, to think more asynchronously automatically when using Node.js as opposed to Python. I think that overall both are quite similar, and they both have similar functionalities. I believe that Node.js code looks a bit nicer, however that is more based on opinion.

5. More on Asyncio

Asyncio allows the programmer to handle more tasks in a concurrent manner. The performance implications of this means that for expensive timely I/O operations, you can await for the operation to be done and do other tasks instead. This means that for certain programs, the asynchronous version will execute much faster than the synchronous counterpart.

It is quite easy to write a server using asyncio as there are many functions that are built-in that speed up development. For example, asyncio.start_server(), serve_forever(), and asyncio.open_connection() were all functions I used in my program that were very high-level and didn't need lots of time to use. The API was very simple and easy to use, and overall I'm very happy with the development time of this part of the program.

In Python 3.9, asyncio got a few features that were added.

```
# New features
loop.shutdown_default_executor()
asyncio.to_thread(...)
```

One of the features is loop.shutdown_default_executor(), which "schedules the closure of the default executor and waits for it to join all of the threads in the ThreadPoolExecutor". Asyncio also got the to_thread() feature added, which asynchronously runs the function passed in in a separate thread. These new features that were added are important, and since this functionality was added to the previous version, it makes writing concurrent code using asynchronous easier to write with later versions of Python than earlier.

Although these features weren't added in Python3.9 or later, "asyncio.run()" runs a coroutine taking care of the asyncio event loop, and "python3 -m asyncio" allows the programmer to call async functions using await in the Python3 REPL (Python3 Interactive Shell). Again, these changes I think are beneficial and make development of concurrent code using asyncio faster and easier.

6. Conclusion

Python's asyncio is very useful for writing asynchronous programs: the function APIs are easy to use, it's very high level which makes development easier, and overall the development process is pretty fast. I recommend any programmer to consider the pros and cons of using asyncio and Python when wanting to write asynchronous code. It is important for programmers to consider the pros and cons of using the asyncio library compared to something like Node.js, as well as the pros and cons of using Python over something like Java.

References

[1] "Event Loop — Python 3.10.4 Documentation." Python, docs.python.org/3/library/asyncio-eventloop.html. Accessed 31 May 2022.