CS131 Project Report

Andrew Grove - 304785991

ABSTRACT

In this project, we attempted to find another way to manage an application server herd rather than using the LAMP platform. This was done to allow for rapid and numerous updates from client to server.

We created a prototype of this server herd in Python, using the asyncio. What this application does is when sent a message from a client, it propagates through the other four (maximum) servers through a flooding algorithm. This application also has the added functionality of retrieving the JSON data from the Google Maps API when asked for a radius around a location.

Introduction

This project utilized the asyncio library in order to implement its features. This served our purposes extremely well, for the asyncio library has extensive support for the creation of servers, as well as momentary streams to connect to other servers. It is also asynchronous, to allow for a large number of requests, something that this project requires.

When compared to languages such as Java and Node.js, we can see some of Python's advantages. This will be discussed in some of the later paragraphs.

The asyncio Framework

As said before, this project was implemented using the asyncio library. There was two main functionalities that we had to use asyncio for: the creation of the server that would handle the requests from the clients, and the creations of several streams that would allow for the server to talk to the other servers and to talk to the Google Maps API server.

Creating the Server

For the purposes of creating the server, Python and asyncio allowed to execute this extremely easily. Using an example found on the Python Tutorials, we were able to create a server trivially, using the function create_server. This function requires three arguments, a class from which to find its functions, an address, and port.

The first, is somewhat simple to implement, for all we have to do is create a class that inherits the asyncio.Protocol class, and create the functions connection_made and data_received. The connection_made function allows for clients to connect to the server by creating the necessary transport. The data_received also is simple to implement, for it just takes in bytecode as the message. All we have to do is parse through this bytecode, find whether it is a WHATSAT or IAMAT, and send it through the necessary channels (which will be discussed later). This class (due to the inheritance of protocol) also has the added benefit of being able to communicate back to its client extremely easily, using the method write within the transport that it created to send a bytecode back.

The address and port are also straightforward, for we are running this on our local computer. As such, the address is just the localhost, or 127.0.0.1, and the port is whatever we want (besides the reserved ports). In this project, we used the ports 8880, 8882, 8884, 8886, and 8888 for our five servers.

This all is created within the asyncio event loop, which is what allows for the asynchronicity. The event loop acts like a sort of handler, which tells what functions should run when. When all that is made is the server, it keeps this active constantly, where a client and message can be established instantly. When the server is busy doing other things however, it has the ability to pause these things (similar to an OS), and handle clients and their messages first. Additionally, this is all done within a single thread, allowing for much easier coding and debugging.

Creating the Streams

We used streams for two purposes in the making of this project. Firstly, we used streams for the server-to-server communication. Secondly, we used streams to talk to the Maps API. Both of these were done in very similar fashions however, due to the ease of asyncio.

To create the streams, all we had to do was create a coroutine function, that would operate in a way that would agree with the use of the loop. This function would call open_connection, taking in an address and port. For the server-to-server communication, the address would be the localhost, and the port would be

whatever servers that that server would talk to. For the Maps API, the address would be 'maps.google.com', the port would be 443, and we also have to make SSL true. We are unable to call this function directly however, instead having to encase it in ensure_future, allowing the loop to become aware of the pending task.

Doing this allows us to make use of an extremely important syntax, yield from. This truly allows us the become asynchronous, for the loop is able to do other things while waiting to send or receive messages to or from whatever server our server is communicating with. This ability to pause while a function is running and continue back later allows our program to become much more finely-grained, and allows it the ability to prioritize more important tasks. For example, if the Maps API is being overloaded, our program will not be able to read from it. Nevertheless, our server would not be blocked waiting for it, for the event loop is able to pause this processing and get back to it later.

In conclusion, when the server receives an IAMAT message, the server sends this message to its connected server through streams, then sends a message back to the client through the transport. When the server receives a WHATSAT message, it uses the data it previously received (through IAMAT), opens a stream to the API server, creates and sends a GET request, parses through the raw message that it sent back to find the required JSON, then sends this all back with an AT message.

Python (asyncio) vs. Java

In the most general sense, there are differences between Python and Java. One of the most prominent, especially when actually coding, is the fact that Python is dynamically typed. This has some pros and cons. It is easier to code in Python extremely quickly, and is suitable for smaller projects. It is however, much less organized, and it is difficult to know what variable is what at first glance, when reading it over once again. Also, we do not need to use the keyword new, as everything is created on the heap in Python, which helps with readability at the cost of clarity. Because this is a simple project however, most variables are straightforward, and in my opinion, the pros outweigh the cons.

Python is extremely straightforward too, especially in the use of a dictionary. It is extremely easy to create a hash table in Python, which we needed to do in this project for strings. Additionally, it is much easier to repeatedly test and debug. This is due to Python's advantage of not needing compilation. Once again, while this might be slower when having a large project, our project is small, so we have most of the advantages with very little of the drawbacks.

In terms of the application design as a whole, the usage of asyncio also had numerous upsides. If we wanted to create asynchronous IO in Java, we would most likely need to use multiple threads rather than an event loop. This has multiple downsides. Firstly, it makes the entire program much more difficult to read and comprehend (directly counteracting one of the aforementioned upsides of Java). Secondly, it will cause varying degrees of performance depending on the server it is running on. This is because the utilization of threads would make use of all the cores. This would cause a large number of issues with data races, and would require the use of locks and other synchronicity methods, further complicating the code.

Additionally, because of the future need for this program, i.e. constant GPS updates, this will generate way too many threads, each of them blocked and waiting for the other one to run. As such, Java will run extremely slow, and not being able to respond to user requests (and as such, will ruin the entire reason that we need asynchronous IO).

Python (asyncio) vs. Node.js

From a broad perspective, aysncio and Node.js are quite similar. Both are single threaded, event-driven, and asynchronous. This is what we need for the purposes of this program, and as such it appears that both would be able to be used to implement the server herd effectively. They are both also quite straightforward to code, as there Node.js and asyncio does not necessitate the use of factories or other such tools. Rather, it uses simple methods and classes to get the job done, which is made even easier because of the event-driven design.

Additionally, like asyncio, it makes extensive use of an event loop. This event loop is the baseline for the single-threaded asynchronicity, as we have discussed in the earlier paragraphs.

There are also some similarities between Python and Javascript. what Node.js is built on top of. Both make extensive use of dynamic typing. Both also are interpreted languages, which allows for easier debugging.

They are different in various ways however. For one, Javascript is more strongly typed. This would make the project more easily understandable, especially if there is plans for expansion in the future. Additionally, Javascript uses prototypal object oriented design, which

might be more difficult for those not used to Javascript to understand.

One extremely large advantage that Node.js has as opposed to asyncio is that it is built with http and web development in mind. As such, we would not have to go through the numerous hoops that we had to to create a GET request. If we are planning on developing a server herd with a large number of http requests, that project might be better suited to Node.js.

Conclusion

In conclusion, using Python and asyncio I feel is a good way to implement the project at hand. Python is a very simple and convenient language, much better than C and other languages that would require manual heap management. Not only that, but asyncio, which is built on top of Python, is extremely simple to use. Creating the server and streams are verys simple, requiring simple one line functions for the most part. The event-driven, asynchronous, and single-threaded nature of asyncio is very simple to code and understand, and I can see adding complexity to the program to be quite simple, as the library supports mostly everything.

The event-driven paradigm does serve our necessary GPS functionalities. Because the event will trigger the function that we need to be called, as many times as we need to, this suits our needs perfectly.

If we do end up choosing an alternative, I would most likely have to suggest Node.js. This is extremely similar to asyncio, and as such has many of the upsides that I have stated previous. It also uses a more popular language, and might be slightly clearer. It would also be able to send and receive http requests much more easily.

I would not however, suggest using Java. Using Java would seem to add unnecessary complexity for our purposes, as we would have to multithread it. And although Java has extensive library support, none of them would be as simple to implement as asyncio is. The need for rapid messages also will make Java much too slow.