Server Herd for Google Places API using Asyncio

Kevin Chuang, University of California, Los Angeles

Abstract

When designing a web server, we must consider several types of architectures with their own pros and cons. In addition to the architecture we must consider which language to write the architecture in. In this project, we explore implementing a server herd with Python and Asyncio, and compare it to Java and Node.js respectively.

Introduction

Proxy herd architecture consist of a decentralized group of servers where all servers can individually service a client without referring to a central database. With this architecture, we avoid a bottleneck of a single server processing every event but face the problem of updating every server using a flooding algorithm to ensure a client can connect to any different server for the information.
We consider this architecture in terms of Python's type checking, memory management, and multithreading in comparison to Java. Finally, we compare Python's asyncio library to that of Node.js and make a decision.

Implementation

There are five servers within the herd that all communicate with each other according to the following chart.

```
'Goloman': ['Hands', 'Holiday',
'Wilkes'],
'Hands': ['Goloman', 'Wilkes'],
'Holiday': ['Goloman', 'Welsh',
'Wilkes'],
'Welsh': ['Holiday'],
'Wilkes': ['Goloman', 'Hands',
'Holiday']
```

The left hand side is the server and the right hand side is an array of all the servers that the left hand server talks to. The client can connect to any server using ports (that differ between the students, my particular ports are 12080-12084) and communicate with the server with two different messages, IAMAT and WHATSAT. In addition, the servers can communicate with each other by sending AT messages with a flooding algorithm. Each server accepts TCP connections from the client and asynchronously handles messages using a co-routine. Each co-routine is added to an event loop and this way a server can accept many connections and messages at the same time.

IAMAT messages

The client sends this message in the format:

IAMAT <clientId> <location> <time_sent>

In this case, <clientID> is the name of the client sending the message, the <location> is the latitude and longitude, and <time_sent> is the reported time in POSIX time.
When a client sends this message to a server, the server updates its client dictionary and sends a response to the client in the form of:

AT <server> <time_difference> <clientID> <location> <time_sent>

Where the new arguments are <server> which is the server that received the IAMAT message, and <time_difference> is the difference in time between when the server received the message and when the client sent it. After, the server floods the servers it talks to with this AT message so other servers know where this client is.

WHATSAT messages

Whatsat messages are in the format:

WHATSAT <clientId> <radius> <num_entries>

The new <radius> argument tells the server to look for places within the radius of the clientId, and the <num_entries> tells the server to only display up to <num_entries> places. The client can ask any server for this information. The clientID must be a client that has already told a server its location with a IAMAT message. In addition, the radius is bounded from 0 to 50 km, and the number of entries is bounded from 0 to a max of 20 entries.

The server then uses the Google Places API to get a JSON response using my personal API key. It then appends it to an AT message of the same format, and then includes the JSON response after the AT message.

AT messages

An AT message has the same format as the response to the client in the form of:

AT <server> <time_difference> <clientID> <location> <time_sent>

If a server is receiving an AT message, then it is getting an update from a server who either got an IAMAT message or is flooding their own AT message. The server will first check to see if its client dictionary contains the clientID. If it doesn't then it adds it to the client dictionary. If it contains the clientID in the dictionary, then it checks the timestamp to see if it comes after the timestamp it has in the client dictionary. If it does, it is an update of a previous AT message, and it updates the dictionary. If not, then it already has the most updated message in its dictionary and does nothing.

Python vs Java

To figure out the best framework, we must determine which language we will use for it. To decide between Python and Java we consider the differences between the two in terms of type checking, memory management, and multithreading.

Type Checking

Java is statically typed meaning that types are declared and checked for consistency before runtime. In comparison, Python is duck-typed which means that programmers do not need to know what type variables are. The interpreter will guess the type of the variable at run time based on the functions and methods called on them and will instead return a run time error if it is inconsistent. But for a new developer, they would need to look for the functions to figure out what type each variable is, but with Java's statically typed variables, they can figure it out right away. In this case, Python is easier in terms of starting development, but Java is safer, and easier to debug.

Memory Management

How memory is managed differs between each language. Python uses the reference count method while Java primarily uses a mark and sweep method. With reference count, each object has a reference count which is increased when a variable is assigned and decreased when the object is deleted. When the number of references is 0,
the object is therefore safe for deletion. In the case of reference count, objects are immediately destroyed when no longer needed but has a performance overhead. Since our application creates a lot of variables that are only used once or twice, Python is advantageous because they immediately give up memory when freed. The con of Python's reference count method is that occasionally, circular references of data can occur, leading to memory never being freed, but Python has an additional garbage collector that frees memory occasionally to ensure that circular references do not occur.

Multithreading

In terms of multithreading, Python is at a big disadvantage in comparison to Java. Python uses a global interpreter lock and can only execute one thread at a time. In comparison, Java can utilize multi-core processors, and which increase throughput of the server. But since our framework is relatively light and doesn't have to be scaled up to service more clients, Python works fine for our framework.

Advantages of Asyncio

Asyncio is a very powerful framework that makes code both simple, and effective at handling multiple requests at the same time. By having an event loop with asynchronous events, we can process many requests at the same time and add new connections. In addition, writing in Asyncio is incredibly easy because each server uses the same code and we can easily scale and add more servers to the herd. The aiohttp package also allows us to make asynchronous HTTP requests, which can perform the GET requests to the Google Places API.

Disadvantages of Asyncio

Asyncio can make debugging difficult and introduces complexity into our architecture that a synchronous server would not have. Asyncio can make debugging harder because processes and requests are not guaranteed to occur in order of which it happened. This can cause problems in the case that a WHATSAT message and an IAMAT message occur at the same time. With a synchronous model, you can ensure that a

WHATSAT message that is after an IAMAT message will always return a valid request to the Google Places API. With an asynchronous model, it is possible that the IAMAT message is processed after the WHATSAT message and return an error instead. In the case of our asynchronous model, our client would have to ask the server again, causing more traffic, or wait for a bit to ensure the message was received before asking, increasing latency.

Asyncio vs Node.js

Node.js and Asyncio have many similarities. Both use an event loop and are non-blocking. Both are asynchronous and are processed once the event loop reaches them. Both are also single-threaded and have similar functions, for example a Future is analogous to a promise in Node.js. The differences are that Node.js is a much newer language compared to Python's Asyncio which comes with both advantages and drawbacks. Because it is new, it is more lightweight and has constant updates. On the other hand, it lacks an established library and a developer might encounter more problems than if they worked with Python. But an advantage that Node.js has over Python is that it is written in JavaScript, which is the same language that most front-end clients are written in. This means fewer mistakes because you could hire a JavaScript developer that can deal with both front-end and back-end, instead of either two developers or one developer who is inexperienced in one of the languages.

Conclusion

After analyzing Python and Java for the language, and asyncio versus Node.js, we can conclude that the asyncio framework is suitable for our server herd. In terms of choosing Python and Java, both have their advantages and disadvantages, but choosing Python is still a reasonable choice because of its ease of prototyping and its solid garbage collector. In terms of Node.js versus asyncio, both are incredibly similar and have very small differences between the two. In our case, asyncio is still robust enough and a tried and true method of implementing server side code and I don't foresee many problems using asyncio.

6. References

[1] *Memory Management*. Python Software Foundation. Available: https://docs.python.org/3/c-api/memory.html

[2] Golubin, Artem. *Garbage Collection in Python: things you need to know*. Mar 11, 2019. Available: https://rushter.com/blog/python-garbage-collector/

[3] *Python vs Node.js: Which is better for your project*. Mar 11, 2019. Available: https://da-14.com/blog/python-vs-nodejs-which-better-your-project