Event-Loop Driven Server Network with Python's `asyncio`

Computer Science 131, Project Report

Michael Wu
UID: 404751542

June 4th, 2018

1 Abstract

In order to give a recommendation about whether Python is a good language for server applications, I have created a prototype server architecture that implements a flooding algorithm using the `asyncio` library. My prototype consists of five servers that talk to each other in order to keep track of clients' locations. The servers also uses the Google Places API to respond to queries about what is near a client. As a point of comparison, I examine Python's type checking, memory management, and multithreading support against Java's. I have concluded that Python's dynamic type checking is a major drawback of using `asyncio`, and would recommend to use Java for our server instead. Java tends to use less memory and execute faster. We probably don't want to use multithreading in our server, but Java and can support the same event-driven execution that `asyncio` supports. Overall, Java is the best language for our needs.

2 Introduction

In this project I investigated the possibility of implementing a server architecture using Python's `asyncio` library. This library supports event-loop driven concurrency and allows for servers to handle asynchronous events. The server architecture consists of multiple nodes that are connected as a network. Clients connect to a particular node and can either broadcast their location or query for locations near other clients. The servers handle these client requests by propagating clients' location information throughout the server network.

I implemented a prototype server network with `asyncio` to evaluate how easy it is to work with the library, how maintainable Python is compared to other languages such as Java or Javascript, and whether or not Python can fit in with existing software.

3 Implementation with `asyncio`

The core of my server application, whose source code can be found in the attached `project.tgz`, is a single file named `server.py`. This application starts a server to accept incoming TCP connections, then runs an event loop until it receives an interrupt. The server contains a callback method on each new connection. The callback is passed a reader and a writer that allows for the server to communicate with each new client.

The reader and writer are a part of the `asyncio` library, so they must be used inside a function that has the `async def` syntax. Inside a function declared with `async`, we can use the `await` keyword on a statement to indicate that it is asynchronous and can block. If execution blocks on a statement declared with `await`, the program can switch to another task. This allows for concurrency because different connections can be served while the application waits for a client's input [1].

3.1 Server Commands

The default behaviour of the server on input is to echo back the input of the client with a question

mark prepended. This indicates that the client input was invalid. But when the client inputs an `IAMAT` or `WHATSAT` command, the server processes it and responds appropriately.

The `IAMAT` command is used to inform the server of the client's location, along with a timestamp. Upon receiving a valid `IAMAT` command, the server must update the stored location of the client if the previous location's timestamp is older than the input timestamp. After updating the client location locally, a server must propagate the update to its neighbors. Each server stores the ports associated with its neighbors in the server network. In the prototype, the ports begin at 12125 and go up to 12129. These are mapped to the five named servers that can be started. Their names are `Goloman`, `Hands`, `Holiday`, `Welsh`, and `Wilkes`.

The `WHATSAT` command is used to query a server for places around a given client's coordinates. Processing a `WHATSAT` command is much simpler than a `IAMAT` command, as the server simply checks to see if it knows the given client's coordinates and then issues a HTTP request to the Google Places web API to retrieve the places around the client. This does not require any propagation. The server returns a response in a JSON format.

3.2 Propagation

A server propagates updates to its neighbors by connecting to a neighbor through TCP. It then issues a `PROPAGATE` command to the neighbor which contains the update information, and then closes the connection. The neighbor server will treat the incoming connection like any other client connection, and does not know that a `PROPAGATE` will occur until the command has been sent. In this way, no server knows about the status of any other server in the network until a propagation occurs. If no connection can be made, the source server will simply assume that the receiving server is offline and will continue execution as normal. If a `PROPAGATE` can be sent successfully and the receiving server has not already seen a newer update, the receiving server will try to `PROPAGATE` to its own neighbors, excluding the source server. This allows the update to travel through the entire server network.

The servers also log their inputs and outputs into a log file. A sample execution with propagation is shown in figure 1. The server first receives input from a client, then it propagates the update its neighbors. It then replies to the client with a `AT` message. The old `PROPAGATE` that `Goloman` receives is ignored because it already has been recorded on the server. Note that the server treats the incoming propagate from `Wilkes` as just another client connection, and closes the connection to `Wilkes` after the `PROPAGATE` occurs. It keeps the original client connection at `('149.142.26.148', 59509)` alive.

More detailed documentation about the format of the `IAMAT`, `WHATSAT`, and `PROPAGATE` commands can be found in the attached `readme.txt` located in `project.tgz`.

4 Design Concerns

4.1 Type Checking

When implementing this server, I had to validate the arguments of the `IAMAT` and `WHATSAT` commands. This was troublesome in Python because the language uses dynamic type checking. To get around this, I used duck typing and assumed that an input would be in the valid format when dealing with integers and floats. If the inputs were correct, my program would execute as intended. If an input was in the wrong format, I would catch the exception that occurred and instead execute the code that handles invalid inputs. In the end this programming paradigm can work, but it leads to harder to catch errors during development. Type mismatches in the code can only be checked at runtime rather than at compile time, which adds overhead to debugging. However, an advantage of this dynamic type checking is that it leads to faster code writing. I can focus on the logic of my code rather than worrying about verifying the type of data that is being passed around. Dynamic type checking leads to less verbosity in the code.

One situation where the Python's dynamic type checking did not help was when I had to validate

```
Goloman - Received ‘IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1528158363.1649215\n’ from (‘149.142.26.148’, 59509)
Goloman to Hands - Send: ‘PROPAGATE Goloman Goloman +0.657886266708374 kiwi.cs.ucla.edu +34.068930-118.445127
1528158363.1649215’
Goloman to Hands - Close the socket
Goloman to Holiday - Send: ‘PROPAGATE Goloman Goloman +0.657886266708374 kiwi.cs.ucla.edu +34.068930-118.445127
1528158363.1649215’
Goloman to Holiday - Close the socket
Goloman to Wilkes - Send: ‘PROPAGATE Goloman Goloman +0.657886266708374 kiwi.cs.ucla.edu +34.068930-118.445127
1528158363.1649215’
Goloman to Wilkes - Close the socket
Goloman - Send: ‘AT Goloman +0.657886266708374 kiwi.cs.ucla.edu +34.068930-118.445127 1528158363.1649215\n\n’
Goloman - Received ‘PROPAGATE Wilkes Goloman +0.657886266708374 kiwi.cs.ucla.edu +34.068930-118.445127 1528158363.1649215’
from (‘127.0.0.1’, 33056)
Goloman - Old information: Goloman +0.657886266708374 kiwi.cs.ucla.edu +34.068930-118.445127 1528158363.1649215
Goloman - Close the client socket
```

Figure 1: The log for `Goloman`.

the coordinate inputs. In this case, I had to write my own function to verify if the argument was in a valid format. I used regex to extract the longitude and latitude, then cast them to floats in order to check their ranges. Because this forced me to think about the types that I was dealing with, Python offers no advantage over a statically typed language in this case.

With regards to type checking, I would strongly prefer to use Java instead of Python. Since I had to rely on documentation to check the types of the expressions that I wrote, it would make coding much easier if I was working in Java. Everything in Java has a well defined type, making new APIs easier to read through and understand.

4.2 Memory Management

Python's memory management is taken care of in the background of the interpreter, and the user has no direct control over how the memory is allocated. Objects are allocated on a heap containing all of a program's data [2]. Thus the concept of memory is abstracted away from the language. While this allows Python to be simpler and easier to use, it comes at a performance cost. Compared to languages such as C, where the programmer must manage memory directly, Python performs considerably worse on some benchmarks and takes as much as four times more memory [3].

However, this problem will also occur if we choose to use Java to implement our server architecture. Java also employs a garbage collector that cleans up objects on the heap. Although objects on the heap in Java must be declared with the `new` keyword, allowing for limited control over memory management, Java still aims to abstract away the concept of memory. So the amount of memory Java uses can vary. In one benchmark that performed matrix computations, a Java program took over five times as much memory as a Python program. But in another benchmark that performed string alignments, a Java program outperformed its Python equivalent by using less than 20% of the memory that the Python program used [3].

4.3 Multithreading

Most of the `asyncio` library is not thread safe, and is intended to be run in a single threaded environment [1]. Instead of using threads as the primary form of concurrency, `asyncio` uses the event loop. This has the advantage of performing better in situations where a many connections to our server are necessary. If a separate thread were create for each connection, we could have thousands and thousands of threads. Then the overhead of managing the threads may be too costly, and an event loop would perform better [4].

In our server network design, we do not need multithreading because each server can be run as a separate process. They will propagate information to each

other, allowing for work to be distributed over many machines. This approach can use multiple CPU cores to run different processes, but it also maintains data integrity by only allowing communication through TCP. So this approach takes advantage of the primary benefit of multithreading, which is parallel execution, but avoids problems that arise from race conditions and data races. Instead of the work being split across threads in a single process, the server load is split across multiple processes.

In comparison, the classic Java approach to running a server is by using threads, as seen in the Java Tutorials [5]. A typical multithreaded Java server will first listen on a socket for incoming connections. When it detects a connection, it will create a new thread that runs on its own and handles the communication with a client. This approach could potentially spawn many threads that cause performance to degrade. We could implement our flooding algorithm using threads, but it seems that an event-driven approach is the most natural way to do what we want.

Note that Java libraries such as Netty also exist that allow for event-driven programming [6]. So this point of comparison does not solely depend on which language we choose, since we can choose a multithreaded or event-driven approach in either language. It mainly comes down to familiarity with a given API and the language's ease of use.

4.4 Comparison to Node.js

Node.js is a framework developed specifically for event-loop driven servers, and it provides plenty of support for HTTP requests [7]. The language that Node.js uses is Javascript, which has dynamic type checking like Python. If we chose to use Node.js instead of `asyncio`, we would probably have an easier time with our application as the API is intended for developing servers.

Briefly looking at the code for Node.js, it appears much more readable than Python since it uses a C-like syntax and has better support for the types of operations that we want to do with our server. Accessing the Google Places API would be easier over HTTP, as well as parsing the JSON response. When working with the `asyncio` library I constantly found myself confused about what a particular variable was doing due to the documentation being hard to read. In comparison, the Node.js documentation looks promising.

A downside of choosing to use Node.js is the difference in library support compared to Python. Python has great libraries for number processing and machine learning, such as `NumPy` and `scikit-learn` while the Node.js libraries are more web-focused. This may not be relevant for our intended application, but is a point to consider if we would want to do some data processing on our server.

5 Recommendations

Personally, I would recommend developing our application in Java. Languages using dynamic type checking are too hard to debug, especially when multiple developers are working on a project. It is worth it to use Java just to have well-defined function calls and good documentation. Although some may say that Java is too verbose and developing with it is too slow, Java can be written very fast with the help of a good IDE. Since the IDE can check types as you are writing code and it can auto-complete boilerplate code for you, the mental overhead of keeping track of types is negated. For many companies, nothing can beat the reliability that static type checking guarantees. That's the reason why Java remains one of the most widely used languages in the software development industry today [8].

In addition to this, a Java server application will most likely perform better in terms of memory and speed than a comparable Python application. Although it abstracts away memory management similar to Python, it still tends to perform better than Python in terms of execution time [3].

Lastly, Java has support for a vast array of libraries given its age and widespread use. Compared to something like Node.js which is newer, Java has lots of community support and helpful code examples. Although `asyncio` has its advantages due to its event-driven nature, this type of event loop is not exclusive to Python. By using Netty, we can implement a similar server architecture in Java. Overall, we should use

Java because it is simply the most practical general-purpose language for most business applications.

6 Conclusion

I have implemented a prototype server network that relays information across multiple nodes using a flooding algorithm in order to evaluate how good Python's `asyncio` library is. In the process, I have found that the dynamic type checking and hard to understand documentation is a major drawback of using Python. I would definitely recommend using Java instead, as it will most likely be faster and easier to develop in. In Java, you can know exactly what you're doing and catch trivial bugs easily with the type check at compile time. A simple solution to overcome Java's major drawback, its verbosity, is to use a good IDE.

Many languages can be used to develop our application, but the advantages of static type checking cannot be overlooked. It greatly aids in the ease of development as a project grows, which is why I would not want to use Python for larger projects. Python is elegant for smaller programs like my prototype, but it would not be a good choice for the real server design.

References

[1] Python 3.6.5 asyncio Documentation. docs.python.org/3/library/asyncio.html.

[2] Python 3.6.5 Memory Management Documentation. docs.python.org/3/c-api/memory.html.

[3] Fourment, Mathieu, and Michael R. Gillings. "A Comparison of Common Programming Languages Used in Bioinformatics." BMC Bioinformatics, vol. 9, no. 1, 2008, p. 82., doi:10.1186/1471-2105-9-82.

[4] Beazley, David. "Python Gets an Event Loop (Again)." Usenix, vol 39., no. 3, 2014. www.usenix.org/system/files/login/articles/09_beazley.pdf.

[5] Java Socket Tutorials. docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html.

[6] Netty 4.x User Guide. netty.io/wiki/user-guide-for-4.x.html.

[7] Node.js About Page. nodejs.org/en/about/.

[8] TIOBE Programming Community Index. May 2018. www.tiobe.com/tiobe-index/.