CS 131 Python Project Report

University of California, Los Angeles

1 Abstract

There are many possible frameworks and architectures available to build application servers. LAMP is an excellent option, but for our Wikimedia-style service, it looks like it may be a bottleneck. Thus, we are proposing a different architecture called an "application server herd", where our application servers will communicate directly with each other as well as to our core database. In particular, this paper will focus on the analysis of a particular implementation of the server herd that uses Python's asyncio library. We will discuss its advantages and disadvantages, especially as compared to Java and Node.js.

2 Introduction

The prototype application server herd that we have implemented in this project consists of a small group of servers that propagate important information and updates to each other. The design of our server herd must prevent it from being a bottleneck for the application like LAMP appears to be. The new Wikimedia-style service is meant for news, which means that it must be prepared for frequent updates, mobile clients, and protocols other than just HTTP. We must look for options that will support these features.

In order to create such a web server, we will in this project consider the utility of Python, specifically using its asyncio module, focusing on its advantages to Java and Node.js in terms of type checking, memory management, and multithreading. The asyncio module in Python is a foundation for writing asynchronous, concurrent tasks, ideal for a web server such as the one we are implementing.

We will discuss the asyncio's utility in this project as well as any problems we ran into while implementing our server herd. Finally, we will make a final recommendation about whether or not asyncio is a good choice for our design.

3 Server Design

3.1 Server Herd Structure

Our prototype of the server herd is implemented with asyncio module of Python. It consists of a total of five different servers, with the following server ID's: 'Goloman', 'Hands', 'Holiday', 'Welsh' and 'Wilkes'. These five servers communicate with each other in the following way:

1. Goloman talks with Hands, Holiday and Wilkes.
2. Hands talks with Wilkes.
3. Holiday talks with Welsh and Wilkes.

These listed connections are bidirectional - that is, if Goloman talks to Hands, then Hands also talks to Goloman.

Although each of these server only communicates with some of the other servers, messages about clients are propogated to all servers through a flood-fill algorithm. Each server accepts TCP connections from clients and propagates commands from clients to the other servers, also via TCP connections.

Clients interact with servers by sending IAMAT and WHATSAT messages, while servers interact with each other by sending AT messages. Messages received by servers use asyncio to handle messages asynchronously, by adding them to an event loop with a queue of coroutines to process. This way, the server can continue to accept and process new messages asynchronously, even while it is waiting for the currently stored messages to be processed.

3.2 IAMAT Messages

Each server accepts TCP connection from clients. A client can notify a server of its location by sending an IAMAT command in the following format:

```
IAMAT <client> <location> <time>
```

where IAMAT is the name of the command, the location is given by latitude and longitude using the ISO 6709 notation, and time is when the client believes it sent the message expressed in POSIX time.

Upon receiving a valid IAMAT command from a client, the server processes the command and ensures that every other server in the server herd has the updated location of each client.

If the IAMAT message contains information about a brand new client, or new (updated) information about an old client, the server will store the updated information about the client, and propagate this information to the rest of the servers using a flood-fill algorithm. If the server already has information about the client, and the IAMAT message has a timestamp older than the server's current information about a specific client, this means that the information from the IAMAT message is outdated with old, no longer useful information. The server does not store this outdated information, nor does it propagate the message to other servers; it simply drops the message and moves on to its next task.

Regardless of whether the IAMAT message had new information or outdated information that wasn't stored, the server then responds to the IAMAT message with a message in the following format:

```
AT <server> <time difference>
<client> <location> <time>
```

where AT is the name of the response, the server is the name of the server that received the message, the time difference is the difference between the clocks of the server and client, and location and time are a copy of the data the client sent through the parameters of the original IAMAT message.

3.3 WHATSAT Messages

Each server accepts WHATSAT commands from clients. A WHATSAT command allows a client to query for information about places near other clients' locations using the Google Places API to perform a Nearby Search request. A WHATSAT command has the following format:

```
WHATSAT <client> <radius>
<upper bound>
```

where WHATSAT is the name of the command, client is the name of another client, radius specifies how far away from the given client to search (in kilometers), and upper bound is the upper bound on the amount of information to receive from Google Places within the radius of the client.

The radius must be at most 50km and the information bound must be at most 20, because those are the limits of the Google Places API. If a client attempts to send a WHATSAT command with parameters above these bounds, the server will consider the WHATSAT command invalid and respond accordingly.

Upon receiving a WHATSAT command, the server responds with the same AT response as it does to an IAMAT command. It then uses the Google Places API to perform a Nearby Search request, and outputs the returned JSON back to the client who made the WHATSAT request. The JSON is outputted in exactly the same format as the API returns, except that all instances of multiple consecutive newlines are replaced with a single newline, all trailing newlines are removed, and is followed by two newlines. The API key in my code can only be used for testing a few times a day, and thus if too many WHATSAT commands are send to the prototype servers too frequently, the API eventually will just return an error message in a JSON format, indicating that the API key has been used too many times for the day.

3.4 Invalid Commands

If a client attempts to send another type of command to the server, or otherwise formats a command incorrectly (e.g. not all the fields are correct or within the specified bounds), the server will respond to the client with the following format:

```
? <command>
```

which is a question mark, followed by a space, followed by a copy of the invalid command that was sent to the server. As soon as it sees that the command is invalid, the server will not execute any code or store any information from the invalid command.

A weakness in my prototype is that it does not robustly check for every edge case when doing input validation, so there are likely invalid commands that a client can send which may cause the servers to behave incorrectly.

3.5 Communication Between Servers

Servers communicated information to each other using AT messages. Although the specifics of these AT message are unimportant (as the client never interacts with them), they were sent in the following format:

```
AT <client> <server> <latitude>
<longitude> <time difference> <time>
```

where this AT message contains all the information sent from the client through the parameters of the original IAMAT message. The server uses this format to propagate the information to all other servers, which parse and store the data in their own internal data structures.

4 Asyncio

Asyncio is a Python library that I used in my implementation of this prototype. It allows us to write concurrent code using the async/await syntax.[1]

4.1 Advantages

Asyncio provides a wide variety of features that I used, including event loops and coroutines that made it easier to implement this project.[2] While working with asyncio, I found the API intuitive, well-documented, and straightforward to use. It provided a convenient way to implement TCP servers for our server herd structure and clients to help test those servers. It also provided a simple way to run the servers asynchronously, even though Python is traditionally not asynchronous. These details about asynchronicity were abstracted away, allowing a simple but effective way to implement a functional prototype.

All of our servers run the same code, so in order to run another server, all we have to do is run the following command:

```
python3 server.py <server name>
```

where server.py is the source code, and server name is the ID of the server to run. This is a very simple, quick command to run, and thus adding new servers to the herd is a simple task, which is yet another advantage of asyncio. This quality of our asyncio makes it more scalable, as increasing the number of servers in our herd server is easy to do.

Performance testing was difficult to do with a limited number of tests and no competing prototype to compare it with, but the asynchronous nature of asyncio makes it suitable to handle more requests than a synchronous approach.

4.2 Disadvantages

One of the disadvantages of the asyncio library was that although it supported TCP and SSL protocols, it was missing the functionality of making HTTP requests. This, however, didn't turn out to be a serious problem. Python has another module called aiohttp, also asynchronous, that allowed us to make an HTTP GET request to the Google Places API exactly as we needed to.

Additionally, each server in our application server herd design runs the same source code. If there are any changes or bug fixes in the code, a server must be restarted in order to run the new code. In the prototype implemented for this project, the servers don't store previous client information when they are shut down, so frequently shutting down servers could lead to problems of losing or having to resend data.

Another disadvantage of asyncio (and all asynchronous methods) is that execution does not always occur in the order that the programmer expects, making the servers more difficult and less intuitive for programmers to debug.

The asyncio module also lacks support for multithreading, so each server can only process one task at a time. A multithreaded server could create a new thread for each task, but asyncio cannot. For a small number of requests, asyncio works fine as we can see from testing. However, with a larger number of requests, the single-threaded aspect of this implementation with asyncio may hurt performance significantly as the server can only process one request at a time.

5 Python vs Java

Choosing the right language to implement the server in is an important part of selecting the right framework for the server herd model that we've implemented. We will take a look at Java, and compare it to our Python implementation, specifically looking at type checking, memory management, and multithreading.

5.1 Type Checking

Type checking differs in Python and Java. In Python, types are checked dynamically during runtime. On the other hand, in Java, types are checked statically during compile-time.

For this reason, Java will have much fewer runtime errors involving types, because all of the types will have been checked at compile time. Python, however, may crash in the middle of running due to the programmer not ensuring all the types are correct. The possibility of these crashes due to incorrect types is one disadvantage a Python server has in comparison to a Java server. In fact, this was one of the biggest challenges I faced while building the prototype of the server herd in Python: I would run into crashes when I didn't account for types and have to go back into the code and fix all these errors, whereas if I had implemented this prototype in Java, I would have had to fix these errors at compile time, before running the server at all.

Although using Java for the server will result in having fewer runtime errors involving types, Python's dynamic type checking allows the programmer creating the server to do so simply and easily, without having to know or concern themselves with what the type of reader.read() or writer.write() returns, and can just use the object without thinking about what exactly what type of object it is.

In Java, setting up a server involves knowing the types of every function and variable, requiring much more time and effort on the programmer's part just to get the code to compile. Java's type checking may result in a server that is less likely to crash, but may also intimidate new developers who want to create quick and simple prototypes. For this reason, making our server herd prototype in Python makes sense in comparison to Java.

5.2 Memory Management

Python and Java both have their own garbage collectors, which means the programmer doesn't have to worry about allocating and freeing objects themselves; the garbage collectors will handle it. However, Python and Java implement their garbage collectors in different ways.

Java primarily uses the mark and sweep method for garbage collection, called the Concurrent Mark Sweep collector that is designed for applications that prefer shorter garbage collection pauses, and that can share resources with the garbage collector while the application is running.[3] The mark and sweep operation is quite expensive and can severely affect performance while the garbage collector is running, even if it only happens occasionally.

Python primarily uses reference counts for its garbage collection; that is, along with each object, there is also a number stored that represents the number of pointers pointing to the object. When the object's reference count is zero, the object's memory can be reclaimed. This is advantageous in that much garbage can be removed immediately when it is not needed anymore. However, the reference count method alone fails to sweep up garbage with circular references. Additionally, each assignment also has to increment the count in addition to assigning the object. This adds some overhead to each assignment operation in the code. It also takes additional memory space to store a reference count in addition to each object.

For our appication, where memory is allocated and used briefly before it is no longer needed, Python's reference count method works well as it immediately frees unused memory. Python also has an additional garbage collector to handle garbage not collected by the reference count method. This additional garbage collector is generational and only periodically collects garbage.

5.3 Multithreading

While Python can support multithreaded programs, its multithreading capability is severely limited by the global interpreter lock (GIL).[4] In CPython, the GIL is a mutex that exists to prevent multiple threads from executing Python bytecodes at the same time, since CPython's memory management is not thread-safe. Naturally, this makes the GIL a bottleneck for many multithreaded programs in Python.[4]

Java, on the other hand, has much better multithreading support and can take full advantage of multiprocessor systems in many situations.

Because of Python's lack of multithreading support, a Java server with multithreading capability would likely be able to process more requests more quickly than the single-threaded Python sever application that we have implemented in this project.

6 Comparison of Asyncio and Node.js

Python's asyncio module and Node.js are vastly different. To begin with, JavaScript is asynchronous by nature; asynchronicity is built into JavaScript.[5] This makes Node.js seem like a more natural, intuitive framework to use, since a programmer working with Node is likely already in an asynchronous frame of mind when implementing a web server herd such as ours.

Python, on the other hand, is generally used in synchronous applications, with the asyncio module there to extend Python to be able to handle asynchronous applications like web servers. The asynchronous aspect of this module is less intuitive to a developer who is used to synchronous Python, leading to more difficulty debugging and using the module effectively. Asynchronous methods in Python must be explicitly declared asynchronous, whereas in node.js, the defined methods execute asynchronously by default, because that is how the language works.

It is worth mentioning that JavaScript is not statically typed; it uses a prototype model instead. This means that everything in JavaScript is an object that contains fields and values, and there is no essence of a type. This means that JavaScript, like Python, lacks type safety that can lead to bugs in the server code. For this reason, JavaScript may have many of the same issues with crashing and errors with types.

Node.js provides much of the same functionalities for JavaScript as the asyncio module does for Python, and both are suitable for creating web servers to asynchronously handle requests from a large number of clients. Both are capable for our usage, but it is hard to say which of the two would be the most suitable for our specific server herd application without making a prototype in Node.js and comparing the performance of the two. To make an accurate recommendation about Node.js vs Asyncio, it would be best to build a similar prototype of an application server herd in Node.js, and compare its performance with that of our prototype directly.

7 Conclusions

From our discussion of the various advantages and disadvantages of Python's asyncio module and the Python language itself, it is evident that asyncio is a suitable choice for the implementation of our server herd. The most important aspect of our server herd is that it can quickly process requests from many clients, and our prototype is able to withstand basic tests, indicating that asyncio would be able to do what we require it to do. However, after doing some basic research on Node.js, it seems that Node may perform better, and my recommendation to anyone trying to make a Wikimedia-style service (with an application server herd designed for news like we are) would be to investigate Node more deeply, create a prototype with Node, and compare its performance to asyncio before making any further decisions.

8 References

[1] *asyncio - Asynchronous I/O.* Python Standard Documentation. March 12, 2019. Available: https://docs.python.org/3/library/asyncio.html

[2] *Couroutines and Tasks.* The Python Standard. March 12, 2019. Available: https://docs.python.org/3/library/asyncio-task.html

[3] *Concurrent Mark Sweep (CMS) Collector.* Java Documentation. Available: https://docs.oracle.com /javase/8/docs/technotes/guides/vm/gctuning/cms.html

[4] *GlobalInterpreterLock.* Python Wiki. August 2, 2017. Available: https://wiki.python.org/moin/GlobalInterpreterLock

[5] *Asynchronous Programming in Node.* Code for Geek. Available: https://codeforgeek.com/2016/04/asynchronous-programming-in-node-js/