CS 131 Project Report

University of California, Los Angeles

Abstract

In this project, we implement an application architecture called an "application server herd" for a service similar to Wikimedia server platform, originally based on GNU/Linux, Apache, MariaDB, and PHP+JavaScript. We explore the use of the asyncio library in Python to implement our more event-driven application. We will explore the pros and cons to the asynchronous, single-threaded nature of asyncio and compare it to Node.js.

1. Introduction

Wikipedia and the Wikimedia server platform uses GNU/Linux, Apache, MariaDB, and PHP+JavaScript with multiple, redundant web servers behind a load-blancing virtual router and caching proxy servers [spec]. This works well for systems that are not updated as often and have fewer and less frequent client connections. We want to build a new Wikimedia-style service that is designed for news. This means that we will have many more updates to the articles from many different devices and we will experience more mobile accesses [1]. Access will now require more protocols than just HTTP, so we hypothesize that the server can be a huge bottleneck. To combat this, we look into a different architecture called an "application server herd" which will take advantage of multiple servers that communicate with each other and flood messages amongst the network. With this architecture, servers can receive information quickly without connecting to one server or database, eliminating the bottleneck. This interserver communication is tailored for rapidly-evolving data to reduce the load on the central database.

To explore this architecture, we look into the asyncio asynchronous networking library to implement the server. Using asyncio, we create a small application using application server herd that performs HTTP requests from the Google Places API and communicates between servers and between servers and clients using TCP protocol. The event-driven nature of asyncio allows us to rapidly update and flood the other servers in the herd. We will explore through this exercise the practicality and the ease with which applications can be written using this tool. We will also explore its reliability, maintainability, and flexibility to additions and expansions. In addition, we'll evaluate Python and asyncio's type checking, memory management, and multi-threading when it comes to scaling to larger scale applications vs. other languages like Java.

2. Implementation of Application Server Herd with asyncio

Our prototype is written in Python 3.8.2 and consists of five servers with server IDs of 'Hill', 'Jaquez', 'Smith', 'Campbell', and 'Singleton' (named after the objectively five best players on the 2019-2020 UCLA Men's Basketball team). These servers communicate with each other bidirectionally, but each server only talks to two or three other servers. The servers are connected in a way which forms a connected graph, so there is a path from any one server to another either directly or indirectly. They talk to each other using TCP connections and send encoded messages from server to server. The servers communicate with clients via TCP connections as well. The clients represent mobile phones with client IDs, location coordinates, and a time keeper.

Clients can send two types of messages to any server, IAMAT messages and WHATSAT messages. An IAMAT message from a client to a server contains the keyword IAMAT followed by the client's ID, its ISO 6709 coordinates, and the client's idea of when it sent the message in POSIX time. This is essentially a check-in and the server records the check-in and floods the rest of the servers with the information on the client's ID, location, and time. It responds back to the client with a confirmation AT message consisting of the keyword AT followed by the server's name that received the message, the time difference from when the server received the message back to the client's idea of when it sent the message, and lastly a copy of the information for the original IAMAT message. Clients can also send a WHATSAT message to any of the five servers with the keyword WHATSAT and a ClientID to search for followed by a radius of search in km and a max number of items to query for.

2.1. Server Implementation

To implement the server, I created an object to represent the server with member variables containing the server's name, IP address, port number, and dictionary of clients to their most recent "AT" statements. Upon running the server.py file, a command line argument is taken to indicate which of the five servers this instantiation of server.py is creating and running. When server.py is ran, a server is created, started, and ran forever until interupted by a KeyboardInterrupt.

The server runs a function upon startup that awaits a reader to read an incoming message. The coroutine of the server gives up the CPU here until the read finishes (when a message is incoming and is read). This allows other threads to run while we wait for an incoming message since there is no

parallelism in Python. Upon receiving the message, the server decodes it and records the time received. The message is passed along to another member function which interprets the message, handles it in the proper way, and returns the output string we are to return back to the client.

In terms of handling specific inputs. If the server receives an IAMAT message, it extracts the ClientID, coordinates, and time and records the information as a value to that ClientID key in a dictionary containing all most recent information on all clients that have sent an IAMAT to one of the five servers. It then floods an UPDATE message to all servers that it can talk to in order to start the flooding of information about this client so every respective server's dictionary contains this check-in.

2.2. Flooding Information

As a check-in comes in to one server in the form of an IAMAT message, it needs to propagate this information to the rest of the servers in the herd. The network of servers is a fully connected graph, so by sending an UPDATE message with the information from each IAMAT check in to every server connected, and each server receiving the UPDATE message passing it on to its connections, every server will receive an update. The servers continue to forward the updates on until the update they have received is already stored in their dictionary. If the ClientID doesn't exist in the current dictionary, the server adds it with the AT message as the value. If the ClientID already exists, the server checks to see if the received update matches the value of that ClientID key. If it does, it does not update or forward anything. If it doesn't it will update the value and forward the UPDATE message to its connections. This process effectively floods each server with any IAMAT information in an efficient matter.

3. Is asyncio a Suitable Framework for this type of application?

Asyncio allows our server's coroutines to give up the CPU while they wait for awaited function to finish such as waiting for a message to read or writing a message back to the client [2]. These are expensive processes that would halt for a long time if run in a fully sequential matter. If we ran the server, we would be stuck waiting for an incoming message and the server would continue to block the only thread with the read method. With asyncio, we can allow the server to give up the CPU and allow other areas of the program to run on the CPU while we wait for the time-intensive I/O to finish. In an I/O heavy program, multithreading is essential to avoid long stalls that block the rest of the program. Since python does not support parallelism, our only option is to switch which threads have control of the CPU. Therefore, we turn to asyncio to implement asynchronous code.

Asynchio depends heavily on cooperative multitasking and threads voluntarily giving up their control of the sole processor. Asynchio is heavily used to implement high-performance networks, webservers, and database connections.

The library asynchio provides two keywords, async and await. The async keyword denotes a function as a coroutine and one that can voluntarily halt its execution and give up control of the CPU to allow another coroutine to execute. The await keyword stops the current coroutine from running until the awaited function finishes. The processor then chooses the next task from the event loop to start executing while it waits for the awaited function. In the case of our application, we await the read function and the write function which are time consuming I/O functions [3].

3.1 Pros

Asynchronous programming is very beneficial for this particular type of application dealing with a herd of servers. Instead of each server processing one message at a time, they are allowed to operate much more efficiently and faster. The event loop with asyncio allows for handling of new connections to servers from various devices at the same time as processing messages. This is essential if we have a mobile based app with many frequent mobile connections. The asynchronous nature of the code allows for handling of this volume of connections. Servers can work on multiple tasks since they are all coroutines within the one thread. The bottleneck of time consuming I/O processes or HTTP requests to the database is eliminated by the ability to give up CPU control and await their termination. We expect many more connections and edits than before, so asynchronous code is a very essential upgrade.

In terms of HTTP request bottlenecks to the database, the new proxy herd architecture allows us to propagate messages via TCP across the five servers instead of continuously needing to talk to a single database. This way, we can utilize the asynchronous nature to its full extent and not put stress on the single HTTP connection.

In general, asyncio's syntax is very easy to use and very programmer friendly. There are only two keywords to be dealt with so it is easy to learn, execute, and interpret. Asyncio also flawlessly supports compatibility with TCP which is ideal for this particular application.

3.2 Cons

Asyncio supports TCP well as well as SSL, but not HTTP. We needed an additional library, aiohttp, to send HTTP requests to the Google Places API server to get location data. Luckily we are able to halt execution while the HTTP request awaits its finish.

While Python does not support parallel processing, it only supports concurrency. Multiple threads can run at once but only one can execute at a time. Concurrency prevents the blocking of one slow process and starving the rest of the

program, but does not take full advantage of multi-core computers. Asynchio inherits Python's disadvantages of lack of parallel processing which causes asynchio to be slower and less efficient than another possible implementation in a different language that takes full advantage of parallelism.

Because asyncio is asynchronous, coroutines can execute out of order and finish in different orders from when they started. This causes a small reliability issue in terms of servers receiving up to date data.

4. Python vs. Java

On a larger scale than asyncio, the choice of language to program our server application is an extremely essential choice to explore. Python and Java are commonly used to program applications but are far different from each other. We'll explore the main concerns of our boss of type checking, memory management, and multithreading. We evaluate whether Python is a suitable language to scale to larger applications.

4.1 Type Checking

Python is a strongly, dynamically typed language which checks types at runtime instead of compile time. Types must be consistent and function properly in order to not run into a runtime error. Python uses duck typing which allows for much more flexibility and less verbose code as we do not need to declare a type when declaring a variable. Python is flexible to various types of data being used in the same functions and variables, as long as all operations on those variables are valid for its type. Type errors typically are not caught at compile time, so programmers will not find out about their error(s) until they run into the situation at runtime. If we have a very large program, we could never find out about a type error of our program in a small corner case that would only be revealed through very robust testing. Java is statically typed, so all type checking occurs at compile time. If we have a large application, the compiler will be able to notify us of type errors without having to reach that point in the program in runtime to find out about the error. The code is more verbose because type annotations must be used every time we declare a variable or declare and argument of a function. The type checking is much less flexible, which cuts down on our freedom as a programmer but also makes the program more reliable and easier to debug since you will know what type each variable is at each point in the program.

For our small application, Python takes the advantage for type checking as it will allow us more flexibility and ease of programming for small applications. If we are to scale to larger applications, a switch to Java or another language may be of use.

4.2 Memory Management

Python utilizes a garbage collector that keeps track of memory automatically through reference counting. Python keeps track of how many references are made to each object on the heap and will clear the memory when that reference count reaches zero. The mark and sweep algorithm is used by Python's garbage collector as well to ensure no memory leaks. Since we have the overhead of constantly keeping tack and updating reference counts and occasionally running the mark and sweep algorithm, Python's memory management is slow and relatively inefficient.

Java uses the mark and sweep algorithm for garbage collection as its main mechanism. It marks all objects on the heap pointed to by something from the stack or by other marked objects and sweeps the unmarked objects and free's their memory to be used on the heap. Constant updates to reference counts are not needed with this implementation, so it is more efficient than Python's. However, this approach is more memory intensive.

For our project specifically, Python is sufficient since it is not large enough for the inefficiencies to cause a significant effect. We are able to save memory and take advantage of the reliability in a smaller application.

4.3 Multithreading

Python only supports concurrency and not parallelism. Race conditions are fatal for python that could cost catastrophic memory leaks or losses of information. To ensure there are none, Python does not allow for parallel processing with multiple threads on different CPUs at one time. In Python, threads pass around control over a central CPU and rely on cooperative multitasking to make programs efficient.

Java supports multi threading and parallel processing with the Java Memory Model defining behavior of multithreaded applications. The JMM allows for a lot of flexibility in terms of the amount of restraint or freedom you give parallel threads, so it can be adjusted to the application of your choosing. Code has the potential to be much faster with Python as long as it is safe as it can take advantage of multi-core computers instead of relying on cooperative multitasking.

For our small project, Python's multithreading is suffice because our program is small, so the differences in speed are not as big of a deal. Python allows us to more efficiently deal with memory at the expense of performance. However, if we were to scale to a larger application, we would want to switch to Java to take full advantage of the parallel potential and speed potential of our larger program.

5. asyncio vs. Node.js

Both asyncio and Node.js are very similar in the fact that they are both asynchronously event driven. Node.js is designed to build scalable network applications in JavaScript. They are both single threaded and run code concurrently and not

parallelly. They are both also event driven to asynchronously organize and execute tasks. [4]

Node.js is faster than asyncio by nature and are known to perform better than Python applications when we are dealing with large programs with a lot of code and a lot of memory and data. Front end code can also be written in JavaScript so the front end to back end connection can be a lot more compatible and smooth using Node.js. [5]

Asyncio is said to handle I/O tasks much better than Node.js but front end code is not developed in Python, so some more effort will have to be made to integrate front end to back end.

For our small application, we do not have that much data but our interface is primarily I/O. We also do not have a front end so the obvious choice for this particular application is Python and asyncio.

6. Conclusion

After exploring the pros and cons of asyncio and Node.js as well as Python and Java, it is clear that Python is the simpler, more programmer friendly language that allows for ease of development. However, Java is the stronger language in terms of speed, scaling, and robustness with larger programs. Python also holds the edge for I/O performance and efficiency when dealing with I/O heavy programs. For our particular application, we have a small application with little memory, but we rely heavily on I/O processes to implement our server herd with a multitude of TCP communication between various servers and to clients. The best choice for this application is Python with asyncio because of these reasons. However, if we are to expand and scale the application to be larger, it is worth switching to Node.js and Java/JavaScript to take advantage of the speed and efficiency that they offer.

7. References

[1] *CS 131 Project.Proxy Herd with asyncio Spec:* https://web.cs.ucla.edu/classes/spring20/cs131/hw/pr.html

[2] *Python asyncio Documentation: https://docs.python.org/3/library/asyncio.html*

[3] *How the heck does async/await work in Python 3.5?: https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/*

[4] *Intro to Aync Concurrency in Python vs*. Node.js: https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36

[5] *Understanding Asynchronous IO with Python's Asyncio and Node.js: https://sahandsaba.com/understanding-asyncio-node-js-python-3-4.html*

[6] *Kimmo Karkkainen CS131-Week8 UCLA Spring 2020 Slides: https://ccle.ucla.edu/plugin-file.php/3495994/mod_resource/content/0/CS131%20-%20Week%208.pdf*