CS 131 Project Report

Abstract

Wikipedia, based on GNU/Linux, Apache, MariaDB, and PHP and JavaScript, uses multiple servers for reliability and performance using multiple layers of caching proxy servers and load-balancing virtual routers. However, this project implements a different architecture known as "application server herd" so that multiple application servers and directly communicate with each other to better support the constantly changing data from client's broadcasted GPS locations. Using the asyncio asynchronous networking library will allow the data to be processed faster to other servers in the herd. This project implements a parallelizable proxy for the Google Places API based on asyncio for Python 3.8.2. The purpose is to not only increase response time to update the servers from the client but also allow clients to be more mobile and with increased portability, allow for access to be required via various protocols. We analyze this server-client program to compare the performance, complexity, and reliability of this Python program of synchronizing data versus Java and Node.js.

1. Test Implementation

The program implementation consists of two files: server.py and client.py. The server.py acts as a proxy server. In the client.py, the clients can send their current location to any server using TCP connection and the server will then respond. Between server and server communication, upon receiving a location, that server will send that location to other servers. This is implemented using the flooding algorithm so that every server will receive the same data, careful not to create infinite loops in the flooding algorithm implementation. HTTP requests are implemented using the aiohttp library in order to make requests to the Google Places API, implemented in server.py.

1.1 IAMAT Messages

The server will save the location that the client send and propagate that data to all the other servers. IAMAT request is formatted in the following format

```
IAMAT <clientID> <coords> <timestamp>
```

A sample input is `IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997`. The first parameter, besides the request command, is the client ID. The client ID can be formatted as any string of non-white-space characters, and cannot include andy spaces, tabs, newlines, and more. The coordinates are the latitude and longitude coordinates in decimal degrees using ISO 6709 notation. The timestamp represents the client's idea of when the message was sent, formatted in POSIX time. POSIX time will consist of seconds and nanoseconds since 1970-01-01 00:000:00 UTC and will also ignore leap seconds. If the IAMAT request is in a valid form, then the server will remember this data. In this project's implementation, this data is saved in a dictionary in order to propagate this information in the form of AT messages, which is discussed in the next section.

1.2 AT Messages

The server will send an AT message once the server receives a valid IAMAT message. Then, it will propagate this AT message to all other servers, using a flooding algorithm. This interserver communication is to ensure that all the servers have the appropriate data. AT requests should be formatted in the following way:

```
AT <server> <timeDiff> <clientID> <coords>
<timestamp>
```

A sample input is the following: `AT Hill +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997`. The <server> denotes the ID of the server that received the message from the client. The time difference is the calculated difference between when the server first received the message from the client and the client's timestamp. Notice that this time stamp could be positive or negative because we cannot assume that the server and the client have the same time sync, or if there is a clock skew. Generally, however, the time stamp should be positive because the server's time stamp should be greater than the client's. The remaining fields, clientID, coordinates, and timestamp, are identical to that from the IMAT message.

1.3 WHATSAT Messages

WHATSAT is a query asking what is near one of the clients. The command is formatted as such

```
WHATSAT <ClientName> <Radius>
<MaxNumResults>
```

A sample message is: `WHATSAT kiwi.cs.ucla.edu 10 5`. The radius is measured in kilometers from the client. We put an upper bound of the radius to be at most fifty kilometers and the maximum number of results can be at most twenty items. The server will use the Google Places API in order to find nearby locations. That API will return results in JSON format to the client, after removing duplicate newlines. We use a Python library called aiohttp to handle a

HTTP request made by the API and will then obtain the data through Python's json library.

1.4 Invalid Inputs

Messages can also be invalid. For example, "WHATSAT kiwi" query is missing information about the input radius and the input maximum number of results. In response, the invalid message will respond in the following format::

```
? <received message>
```

Using the previous example, it will similarly respond as "`? WHATSAT kiwi`" Such errors would also be reported in the program's log file with the description of why the message request was invalid.

2. Asyncio Pros and Cons

Asyncio is an infrastructure that allows for single-threaded concurrent code using coroutines. We will explore the functionality of asyncio and its advantages and disadvantages.

2.1 Simplicity

Python is dynamically typed and because of duck typing, Python is relatively simple to use. Therefore, using Python's library asyncio to implement asynchronous programming is also relatively simple. We are able to use event loops to run asynchronous threads. However, one of the disadvantages of asyncio is that it cannot support HTTP protocols and it can only support TCP and SSL protocols. The solution to one of asyncio's shortcomings is that we can use another one of Python's libraries known as aiohttp to handle HTTP requests.

2.2 Performance

As mentioned in a previous section, Python is able to implement multithreading and concurrency through event loops and coroutines. Because of asynchronous code, the programmer is able to keep track of when the program will relinquish control to another task. Therefore, race conditions from asyncio are extremely rare. Asyncio will also require less memory than threads for example because all the data will be shared on the same stack as opposed to having each thread being on its thread. [1] With the rarity of race conditions as well as side effects and deadlocks, asyncio guarantees a reliable performance for a multithreading application and also guarantees that eventually all tasks will be finished as soon as possible.

2.3 Compatibility

In this project, we implemented asyncio using Python 3.8.2. In this version of Python 3.8, we are able to execute a coroutine and automatically manage the event loop upon returning the result. Another new improvement to this version of asyncio is that instead of having to run asyncio.run() directly, we use the following command `python -m asyncio` to launch a natively async REPL. This simplification allows us to no longer need to directly call asyncio.run() in order to spawn a new event loop for every innovation [2]

In terms of scalability, if we were to add more servers to this project, then we could effectively and efficiently add more servers to the proxy server. The implementation would be simple and we could add any server, regardless of how that server was implemented, to our proxy herd. Therefore, asyncio is compatible with other servers and is scalable. However, the issue of scalability arises where if there are multiple servers created and are used as individual threads, then there's the potential risk that there would be a bottleneck, causing the server to crash. As the number of servers increases, the number of requests will also increase, which would result in drastically decreased performance because asyncio prioritizes reliability over performance. Such an issue occurred while testing the project where if the traffic increased or if the flooding algorithm was not efficient enough, then the application would time out.

3. Python vs Java-based Approaches

3.1 Type checking

Python variables are dynamically typed while Java variables are statically typed. [3] Because Python type checking is done at runtime, variable names and types don't need to be declared before assignment. In Python, we don't need typecasting because types of objects are retrieved from the container objects. However, a disadvantage of this approach is that the memory consumption will also increase.

On the other hand, because Java variables are type checked at compile time, Java variable names and types need to be declared before assignment. A type exception will be thrown if an object is assigned to a variable with a different type. Typecasting is needed because Java container objects are needed for generic type objects but the type of the object from a container is not remembered.

Because Python is interpreted and determines the variable type at runtime, this will increase the workload of the interpreter, thus causing Python to be slower than Java-based approaches. [3]

3.2 Memory Management

Python is an interpreted language so Python will be compiled down to bytecode and then interpreted by a virtual machine whenever the program is run. The default Python implementation is known as CPython and is implemented in C. The reference count is used for garbage collection where there's a pointer to the object type and this information is stored on the heap. [4, 5] Each object in Python has its own object-specific memory allocation to know how to access memory to store than object and each object also has its own deallocator to free the memory once it's no longer used. [6] The reference count will increase upon assigning another variable or passing the object as an argument or adding an object to a list for example. In other words, reference count keeps track of the number of objects that are referencing that data. Once the reference count is equal to zero, then the object's deallocation function will free the memory so that the memory is available to be used by other objects. [6] Because there may be memory leaks if there are cyclic objects such that the reference count will never reach zero so the memory will never be freed, Python utilizes an algorithm known as mark and sweep to handle these unreachable objects.

Java also uses a heap for memory management and the heap will store all class instances and arrays. However, one of the differences in memory management between Java and Python is that in Java, whenever a new object needs to be allocated in memory, the programmer must use the keyword new(). For each running JVM process, there will only be one heap memory. Whenever a thread is created, a stack is created at the same time to store data and any intermediate values.For garbage collection, Java will pause all threads on the application and it's up to Java when to invoke the garbage collector, which results in being an expensive process [6]. Because the garbage collector is automatic, the programmer does not have to worry about when to remove any unused objects.

The Java Memory Model, also known as JVM, will divide the heap memory into the two categories known as the young and old generation. As the name suggests, the young generation is where all new objects are created while the old generation memory contains objects that have not yet been cleaned by the garbage collector after multiple cycles. [7] Similar to Python, Java also uses a mark and sweep algorithm to analyze variables stored on the stack and mark the objects that are currently in use and cleans up any unused objects. However, this approach is not efficient because the JVM model is generational so even though newly created objects are unused will still remain on the heap in the young generation while objects on the old generation would likely to be continued to be used yet the garbage collector will eventually free such objects.

Both Python and Java's approach to memory management is sufficient in ensuring that any unused objects will be cleaned up and frees up the memory

3.3 Multithreading

Multithreading is where there are multiple processes, each with its own process ID or PID and each have their own memory space. While there is interprocess communication, different processes are unable to read the same variables. In Python, multithreading is supported through libraries. In this project, we implemented multithreading using asyncio. As mentioned in an earlier section, CPython utilizes Global INterpreter Lock or GIL in order to ensure that each thread needs to acquire GIL before running any bytecode. Asyncio is a single thread single process cooperative multitasking, meaning that asyncio tasks have singular access to the CPU and can determine when to release control to the event loop. [CITE NUMBER ABOVE] Therefore, we can control when each coroutine would be executed.

On the other hand, in regards to Python, Java has a stronger implementation support for multithreading. In Python, threads would have to wait for GIL, impeding true concurrency within a single process. Java's concurrency depends on the number of processors and cores. Therefore, because of Java's reliance on architecture, Java will have a better performance than Python with varying results.

4. Asyncio vs Node.js

Node.js is a runtime environment from Javascript. We will compare the differences between asyncio from Python and Node.js, comparing its ease of use, performance, and compatibility. Similarities between Python and JavaScript include that they are both dynamically typed languages, thus limited to only single-threaded behaviors and can be used in asynchronous programming.

4.1 Simplicity

Node.js, because of JavaScript's dynamic typing, makes the environment easy for programmers and is thus a popular implementation for web development. Web application developers already familiar with JavaScript should easily learn the Node.js framework. Python is also known for its syntactic simplicity and is a well-documented language.

4.2 Performance

Node.js is interpreted with a V8 engineer and applies the code outside a website browser so application performance is much more resource-efficient. Node.js is a non-blocking event-driven architecture [8] that allows for many requests at the same time. With Node.js's single module caching, it eliminates application loading time and thus increasing the performance. Python, on the other hand, is a single-flow language so requests will be processed much slower and asyncio should not be used for applications that are prioritizing performance.

4.3 Compatibility

Python doesn't support asynchronous programming by default [8]. Therefore, while we can use asyncio from Python to implement asynchronous programming, the architecture of Python is not as scalable as Node.js. Node.js framework is designed with scalability in mind and builds asynchronous architecture in one thread. Therefore, Node.js is more compatible for websites and applications where no procedure can hinder the thread [8] while asyncio and Python is more compatible for writing applications that anticipate interacting with other frameworks.

5. Conclusion

In conclusion, we explored the advantages as well as disadvantages of our implementation of this project. We can conclude that our approach of asynchronous programming via the Python library of asyncio was able to successfully build a new Wikipedia-styled service to allow for information to be updated more often across different protocols. Because of asyncio's simplicity of use, implementing this project using Python instead of Java or Node.js proved to be favorable for the programmer. Because of the versatility of Python's libraries, we are able to easily scale this project to include more servers into our proxy herd. Such flexibility with our Python implementation ensures the maintainability and reliability of these applications.

6. References

1. Python Concurrency: Making sense of asyncio: https://www.educative.io/blog/python-concurrency-making-sense-of-asyncio
2. What's New in Python 3.8: https://docs.python.org/3/whatsnew/3.8.html
3. Python vs Java Performance: https://www.snaplogic.com/glossary/python-vs-java-performance
4. Memory Management in Python: https://realpython.com/python-memory-management/
5. Python vs Java: https://www.snaplogic.com/glossary/python-vs-java
6. Java Memory Management: https://www.geeksforgeeks.org/java-memory-management/
7. Java (JVM) Memory Model - Memory Management in Java: https://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java
8. AsyncIO, Threading, and Multiprocessing in Python: https://medium.com/analytics-vidhya/asyncio-threading-and-multiprocessing-in-python-4f5ff6ca75e8