CS131 Project Report

Jeremy Cristobal
Discussion 1C

Abstract

Exploring using the asyncio asynchronous networking library in Python for the implementation of a server herd. Additionally, the asyncio method is evaluated for its suitability for an application server herd, as well as the differences it brings as compared to Java based implementations.

1 Introduction

Wikipedia and other similar websites have traditionally been based on the Wikimedia server platform. This server platform, itself, uses a sort of modified LAMP platform with: GNU/Linux, Apache, MariaDB, and PHP+Javascript. This works well for the aforementioned services, and though we were tasked with designing the architecture for a Wikipedia-style news service, there are some significant differences to keep in mind: more frequent article updates, requirements for access being extended to other protocols in addition to HTTP, and an increased mobility in clients.

Given these particular specifications, it becomes clear that the current Wikimedia platform would be less than ideal for the proposed news service because the Wikimedia bottleneck will significantly slow the response time down. As such, we instead investigate the idea of an application server herd, in which multiple application servers communicate with each other and can thus propagate the received data to connected servers and avoid having to go through a central server.

To accomplish this goal, we turn to Python's asyncio asynchronous library in order to implement this idea of an application server herd. Asyncio is an ideal library to explore because of its ability to quickly flood updates throughout the network of servers. To put this idea into practice, we created a prototype of sorts consisting of a network of five servers that use asyncio. With the prototype, we can evaluate the potential of asyncio for this kind of application and compare it against a Java-based approach.

2 Python and Java

Before discussing the implementation of the prototype and asyncio, it is important to understand the viability of Python as a whole for this kind of project. As such, we will dissect Python's implementation of type checking, memory management, and multithreading and furthermore compare them against these same aspects of Java.

2.1 Type Checking

Python is a dynamically typed language, which ultimately means that its type checking happens during runtime, not during compile time. Java, on the other hand, is a statically typed language, which ultimately means that its type checking happens during compile time, before the program ever starts running.

Amongst the reasons to use a dynamically typed language like Python are the following three benefits: first, one does not have to specify the type when declaring a variable or a function. This allows for simpler, more concise code that is easier to read. Second, being dynamically typed allows the potential for the same code to be usable for different types. And third, a variable or function type is permitted to change across the course of its lifetime.

There are, however, drawbacks as well that manifest themselves in comparison to statically typed languages like Java. Because type checking happens during compile time, statically typed languages are able to catch type issues before the program runs, unlike the dynamically typed counterparts. As such, one is less likely to encounter errors during runtime, which ultimately means that debugging in statically typed languages is easier. Thus, for larger programs Java may be superior in this regard.

Ultimately, the superior speed of Python's type checking as compared to Java's makes it the better choice for our application server herd, especially considering our implementation is relatively short and thus the increased difficulty presented by debugging will be less important.

2.2 Memory Management

Both Python and Java use automatic memory management and garbage collectors, unlike languages such as C++. This ultimately means that memory is deallocated without the need for the programmer to do so manually. However, the ways in which Python and Java go about automatic memory management is where they differ.

Python accomplishes this through reference counting, meaning it tracks the number of times an object is referenced. This count decrements, and when the reference count falls to zero, it means the object is no longer referenced and can thus be claimed by the garbage collector.

In contrast, Java accomplishes automatic memory management through the mark-and-sweep algorithm. This two-step process works by first marking all the objects that are referenced to, then freeing all memory that was not marked by the first step.

When comparing the two different approaches to automatic memory management, it is clear that Java's approach will have superior speed because it will not have to constantly access and update reference counts. Furthermore, Java's approach is able to successfully handle circular references. Python, alternatively, offers more consistent memory management due to the sporadic nature of Java's garbage collection and the relative simplicity of reference counting. Both clearly have their benefits and drawbacks, but for this project Python's memory consistency may prove more useful for a server than Java's improved speed.

2.3 Multithreading

Python has a Global Interpreter Lock (GIL) that ensures only one thread can be executed at a time. This is necessary because, as previously mentioned, Python's memory management relies on reference counting for garbage collection. Race conditions could lead to inaccurate counts which furthermore could lead to memory leaks.

Alternatively, Java is able to have both automatic memory management and the capability of multithreading. This, of course, allows for the existence of race conditions, but Java is able to circumvent this issue through a variety of different synchronization methods.

Though Python is incapable of multithreading in the same manner and ease as Java, this ultimately should not prove to be much of an issue for our application server herds. The benefits of multithreading will not be fully realized in this project because of each server's rather simple implementation and processes. Thus, for this project, a quick, single threaded program that has the added benefit of not having to worry about synchronization will be able to accomplish the goal nicely.

3 Prototype Implementation

To create this prototype of an application server herd, we used five different servers with the names: Hill, Jaquez, Smith, Campbell, and Singleton. Each server communicates with at least two other servers and at most three other servers. Additionally, clients are able to send requests to any of the given servers.

The server herd acts as a simple Proxy for the Google Places API. As such, a client is represented by their latitude and longitude in and by a form of identification. Clients are able to send one of two requests: 'IAMAT' requests or 'WHATSAT' requests. Any other received request will be simply considered to be invalid, and will return: '? <message>' to the client, where message is the original request that was received by the server.

3.1 IAMAT Requests

IAMAT requests require exactly four fields of non-whitespace characters in exactly the following order: the string 'IAMAT', the client's identification, the client's location, and a timestamp. The client's identification can be any string of non-whitespace characters, the location must be expressed through a latitude and longitude using ISO 6709 notation, and the time must be expressed through a POSIX time expression.

Once the request is properly processed and flooded throughout the other servers, the client is responded to in the following manner: the string 'AT', the difference between when the client sent the request and when the server received it, the client's identification, the client's location, and the timestamp at which the client originally made the request. Again, all time is expressed via a POSIX time expression and the location is expressed using ISO 6709 notation.

In my implementation, I kept a global dictionary of clients that retained information received from a client. When a client whose identification is not yet in the dictionary makes a request, it implies that they are a new client, and as such a new entry is made in the dictionary for said client. When a client whose identification could be found was in the dictionary, however, it meant that this client had previously made an IAMAT request, so their location and timestamp are updated accordingly.

3.2 WHATSAT Requests

WHATSAT requests require exactly four fields of non-whitespace characters in exactly the following order: the string 'WHATSAT', the client's identification, the desired range of search, and the number of desired results to be returned. The desired range of search is taken in kilometers with a maximum range of 50, and the number of desired results has a maximum of 20. Additionally, a WHATSAT request will be considered invalid if the client identification received is not found within the aforementioned client dictionary.

WHATSAT uses the client identification to search through the clients dictionary in order to find the most recent location of the requesting client. Then, through the Google Places API, locations within the desired range of the last known location are found. WHATSAT returns the 'AT' statement that corresponds to the IAMAT request that is being used to determine the client's location. Additionally, WHATSAT returns a list of the found locations in JSON format, with any locations beyond the requested maximum removed from the list. Thanks to flooding, WHATSAT requests are able to be made by a client regardless of the server that the corresponding IAMAT request was made to.

3.3 Flooding

Once a server has received and processed an IAMAT request, it needs to flood the information to its connected servers. In order to accomplish this, I have the server that received the IAMAT request send the resulting 'AT' response to its connected servers. Once these servers receive this, they dissect the information received, update accordingly, and send the same 'AT' response to their respective connected servers.

In order to prevent inevitable infinite looping that this would cause, each server checks its client dictionary upon receiving an 'AT' message. If the information in the dictionary corresponds to the information received in the 'AT' message, then the server has already encountered this message and so it stops the flooding.

4 Asyncio

4.1 Suitability for Project

Python and the Asyncio library allow us to explore asynchronous programming and its uses in the implementation of a server herd. With these newfound tools of event loops, tasks, and coroutines, we are able to avoid obstacles presented in the prompt regarding the issues that using the Wikimedia application servers would pose.

A major issue that using these servers would present is, again, the central bottleneck that would significantly slow down the servers. However, the asyncio library can be used by servers to establish

TCP connections with both clients and other servers. This is crucial because each server can then propagate its received information directly to other servers without having to first go through a central server. Because we now have a way to avoid this central bottleneck that is found in the Wikimedia server herds, efficiency will be greatly improved.

Furthermore, this ability of cross-server TCP communication allows for easy scaling. In other words, though our prototype proxy herd used only five servers, it would not be a challenge to extend it and increase the number of servers within the herd.

Additionally, coroutines give us the ability to suspend and resume the execution of tasks. As such, while waiting for a time-intensive operation, we can allow for other tasks to be executed. This avoids issues that are posed by time-intensive operations and once again yields improved efficiency.

Furthermore, by using asyncio for the framework of this implementation, any new tasks are immediately treated as a coroutine. As such, each new task is added to the event loop to be asynchronously executed. Because we are expecting frequent updates to this service, the ability to process multiple tasks concurrently is a great benefit.

Asyncio is not, however, without its drawbacks. As discussed in the comparison between Python and Java, Python does not have multithreading capabilities and as such relies on concurrency. Multithreading is a more powerful tool than concurrency because it utilizes multiple cores or multiple CPUs to its advantage in order to improve efficiency.

Another downside of asyncio is the nature of asynchronicity itself: it is very likely that the code will execute out of order. Though this was not an issue during the testing of our prototype, with an increased number of servers this may occur more often. If, for example, an IAMAT request is sent before a WHATSAT request but they were executed out of order, the client would receive the WHATSAT response for an invalid or outdated location.

4.2 Asyncio vs. Node.js

Overall, asyncio and Node.js share many similarities. Because Node.js is written in JavaScript, both languages are dynamically typed, whereas Java's static typing gave Python a slight edge for this implementation. Similar to Python, JavaScript lacks the capability of multithreading, which was an edge that Java held over Python. However, Node.js was designed to be asynchronous from the start, whereas Python requires asyncio to do so. Consequently, Node.js holds better overall efficiency.

However, because asyncio is better suited for handling and processing large amounts of I/O than Node.js, and because the service expects frequent updates, we can conclude that asyncio is the superior choice for this project.to process large

5 Conclusion

After implementing an application server herd using asyncio and researching some other methods of which one can implement this, the best choice seems clear. Though Java has multithreading capabilities that are unattainable through the other methods, the slowed efficiency from being statically typed and the sporadic nature of its garbage collector keep it from being superior. And though Node.js is efficient and similar in many respects, it is ill-suited to handle large amounts of I/O, which is crucial for this service. Therefore, I recommend the use of asyncio for the application server herd.

References

About Node.js
https://nodejs.org/en/about/

asyncio — Asynchronous I/O
https://docs.python.org/3/library/asyncio.html

asyncio features of Python 3.9 or later
https://docs.python.org/3/whatsnew/3.9.html#asyncio

Introducing JSON
https://www.json.org/json-en.html

Node.js vs Python: What's the Difference?
https://www.guru99.com/node-js-vs-python.html#:~:text=Node%20is%20better%20for%20web,uses%20PyPy%20as%20an%20interpreter.

Project. Proxy herd with asyncio
https://web.cs.ucla.edu/classes/fall20/cs131/hw/pr.html

TA Kimmo Karkkainen's Slides
https://ccle.ucla.edu/mod/folder/view.php?id=3430933

Understanding Java Garbage Collection
https://medium.com/platform-engineer/understanding-java-garbage-collection-54fc9230659a