CS131 Project Report: Proxy Herd with asyncio

Ethan Wong
Discussion 1C
305319001

Abstract

There are many different ways to structure an online web server. For this project, I investigated the use of an architecture known as an "application server herd". This specialized architecture has multiple application servers that communicate not only with the client, but also with each other via a central database and caches. The application server herd is particularly useful in projects where the servers can handle rapidly changing data, and the central database is primarily used to store information that is long-lasting and less likely to change. Combining the asyncio library from Python along with the Google Places API, this project aims to analyze the reliability of the architecture while also comparing it to a Java application that utilizes Node.js.

1. Introduction

The Wikimedia platform is most well-known for the fact that it hosts Wikipedia and other related websites. The platform relies on technologies such as Debian GNU/Linux, the Apache web server, the Elasticsearch search engine, the MariaDB database, and much more. This approach is tried and true for websites such as Wikipedia, where articles are updated rather infrequently. However, this approach would not work well when designing a Wikimedia-style service designed for news that has frequent updates to articles, has access that will be required via various protocols, not just HTTP or HTTPS, and has clients will most likely be more mobile. Using the aforementioned approach for this news service would be painful to implement and would lead to noticeable bottlenecks that would limit the performance of the service. A better approach for such a service would be the application server herd that uses multiple application servers that communicate among one another through a centralized database and caches to help solve client requests. Having multiple servers allows for a large number of mobile users and solves the bottleneck problem from the previous approach as clients can easily connect to one or more servers that will help them with their queries.

This project aims to implement the application server herd by using Python and its "asyncio" library. This library contains two very important functions – "async" and "await". Asyncio is a fantastic library for implementing this service because uses a single-threaded approach known as cooperative multitasking that tackles the problem of concurrent programming. This approach is much easier to work with in comparison to parallel programming as there is no need to worry about race conditions, locks, or any of the other tools necessary for implementing parallelism. The asyncio library will be used in this project to set up five servers that will help a client with their requests. The nature of asyncio ensures that if one of the servers is busy, the communication between servers will allow for a feeling of concurrency when in reality there are only single-threaded programs.

The main purpose of this application is more than just a news service. The application created in this project allows a client to tell a server where they are located and at what time using an "IAMAT" message. The application server herd will work together with the Google Places API to understand where the user is. The client can then ask the server about interesting places in a specific radius around their current location using a "WHATSAT" message. The application will then respond by utilizing the Google Places API and provide the user with a JSON-format message of nearby destinations, such as universities or other notable landmarks.

2. Server and Client Behavior

The service that is to be implemented in this project will utilize a simplified version of the application server herd that is seen in the real world. There will only be five servers for this application, and they will each have a different name and port number so that they can be seen as distinct from one another. While there are five servers, they do not necessarily all communicate with one another via their TCP connections. The five servers are named 'Riley', 'Jaquez', 'Juzang', 'Campbell', and 'Bernard'. Riley only communicates with Jaquez and Juzang Bernard communicates with all the other servers except for Riley. And finally, Juzang communicates with Campbell. The application will still work despite the fact that not every server can communicate with every other server. This is because they share information among each other either by direct or indirect pathways of information, in a way resembling the flooding algorithm. This is important because it means that no matter which server the client connects to when using the application, they will receive a response as their information propagates among the servers and they work to fulfill the request.

2.1. IAMAT Messages

The first message that a client will want to input to this application is to let the servers know of their current location. This is done via an "IAMAT" message with a very specific format (each field is separated by a space):

IAMAT{client ID}{location}{time}

The client ID is just a name for the client. It can be any arbitrary string provided that there are no whitespaces in the name. This will be important later on when the client wants to make more requests. The current location is specified using the latitude and longitude in decimal degrees using ISO 6709 notation. An example of this would be '+34.068930-118.445127', which represents the University of California, Los Angeles. The time is expressed in POSIX time, which consists of seconds and nanoseconds since 1970-01-01 00:00:00 UTC, ignoring leap seconds. A relevant example of what this would look like in a message is '1621464827.959498503'.This is a representation of 2021-05-19 22:53:47.959498503 UTC.

Upon receiving an IAMAT message from the client, the server that received the message will reply with its own message of a specific format (each field is separated by a space):

AT{server name}{time spent}{client ID}{location}{time}

The server name is just the name of whichever server received the message first. The time spent is the difference between the server's idea of when it got the message from the client and the client's time stamp. The client ID, location, and time are the same as in the IAMAT message – the server just repeats them to confirm with the client that they have the correct information about the client's whereabouts. In the case that the IAMAT message that the client provided was invalid (it did not follow the aforementioned format correctly), then the server will respond with a question mark (?), a space, and then a copy of the invalid command.

2.2. WHATSAT Messages

The second type of message that a client can send to the servers is a "WHATSAT" message. As the name suggests, this is a message that is used to determine what is at or in the vicinity of the client's current location that they shared with the server via the IAMAT message. Of course, this means that the server has to know the client's location before this command can be successfully executed – the IAMAT message must precede the WHATSAT message to work as expected. Just like the IAMAT, this type of message has its own specific format (each field is separated by a space):

WHATSAT{client ID}{radius}{number of results}

The client ID refers to the name of the client, which should have been provided to the servers via the IAMAT message. The radius explicitly marks a range of kilometers around the current location in which the server will look for interesting locations. For example, a radius of '5' would mean the server would search the area in a five-kilometer radius. The maximum radius that can be used is 50. The final field is the number of results. As the name implies, this is the number of locations that the client wants to receive from the server. The largest number that can be inputted for this value is 20, as this is the largest number that the Google Places API can handle well. In the case that the WHATSAT message that the client provided was invalid (it did not follow the aforementioned format correctly), then the server will respond with a question mark (?), a space, and then a copy of the invalid command.

This is the client message that truly uses the Google Places API. The IAMAT message does not really rely on the API, but it must be used here as it is responsible for scouting out the surrounding area and looking for notable locations. The servers will communicate with the API issuing a simple HTTP GET request to the Google Places API. The arguments of the GET request are the API key, the location (in ISO 6709 latitude and longitude notation), and the radius. The API then understands that it should output a JSON-format message that details the client-specified number of results for locations around their location. The JSON-format message will be preceded by an AT message, which was detailed in the "IAMAT Messages" section. Of course, the length of the server's response message will vary based on how many locations that the client requests.

2.3 Communications between Servers

As mentioned earlier, the servers must communicate with each other in order to propagate information among themselves. This typically happens after the client inputs an IAMAT message, because the server that receives the message first must share the information with the other four through direct or indirect means of communication. This is done via the flooding algorithm. The initial server, after confirming the IAMAT message, will share the AT message to the servers it is directly connected to, and those servers will do the same until every server has received the message. This can be done quite simply by taking advantage of the asyncio method called "open_connection". It is important that the servers also keep track of the respective time at which they received the message, because they do not want to have an infinite loop where messages are being spread constantly for no reason. This would be a waste of resources and also server no benefit for the servers or for the client. Thankfully, this feature is handled automatically by the use of "open_connection" as the connection to a server will close after that server receives the message.

3. Python vs. Java

There are quite a few differences between Python and Java. In particular, there are notable comparisons to be made between their methods of type checking, memory management, and multithreading. This section will detail some of these differences and analyze which features are better with regard to the project.

3.1 Type Checking

There is quite an immediate difference between type checking between Python and Java. First, Java is an example of a statically typed language. This means that the compiler will automatically check the types of all the objects within a program before it is even run. This approach is generally quite safe as it will help to catch errors before your program executes, which allows the programmer to easily fix them and prevent errors down the road. This approach is beneficial particularly when writing large, complicated programs as it can be extremely difficult to comb through hundreds or thousands of lines of code to search for where the type error occurred. However, it can be quite annoying to the programmer because they constantly have to declare the types of their variables. The tradeoff for this is that when the programmer manually declares the type of a variable, it saves some time for the compiler and therefore generates more efficient code. In a sense, the variables feel a lot more restricted with this method. This approach is utilized in other popular languages such as C and C++.

Python, on the other hand, is a dynamically typed language. This means that the types are not checked before the program is run like they are in Java. The types are checked while the program is being run. Errors will occur during the execution of the program and will most likely result in a type error that crashes the said program. This can be a bit more dangerous in comparison to static checking as it can cause some unseen errors that may be difficult to debug in the future. It may also seem unclear what type a specific variable is because there will be no clear identifier when the variable is declared. Dynamic typing can also be harmful when programs use many instructions, as the interpreter will have to check all the types while the program is executing. However, this approach is quite beneficial in that it is simpler for programmers to understand. There are no clumsy, wordy type identifiers before each variable. This makes Python code very concise and easy to read, even for beginners. There is also no need to worry about the strictness of types as much because variables are very fluid since their type will be automatically interpreted based on the value assigned to them. It also helps a lot when using built-in or library functions because the types are more flexible, and programmers are therefore less prone to run into errors with parameter types. In regard to this project, using dynamic type checking is definitely easier because the code is not extremely long and because it simplifies the calling of library functions, which are used quite often. This approach is utilized in other popular languages such as JavaScript and PHP.

3.2 Memory Management

In a sense, memory management for these languages follow a similar trend to the trend seen in type checking. Python memory management is simple and intuitive, but a bit less efficient than Java's efficient but somewhat confusing method. Java's garbage collector utilizes something known as a mark-and-sweep algorithm. This complex algorithm investigates every object addressed by a root and marks them. It then uses depth-first search to investigate the children of that particular object to find any other objects that are addressed by a root. After all objects have been investigated, then all objects that are unmarked are swept away onto the free list. As seen in the above description, this algorithm is quite confusing. However, it is great for preventing circular references, so it is more thorough and safer overall.

Python uses a much simpler method of garbage collection. It uses something known as reference counting. Every object in the program will be given a counter that keeps track of how many other objects are referencing it. As long as the value of the counter is greater than zero, the object will not be freed. However, when the value of the counter is zero, that means that the object no longer serves any purpose in the program and will be freed. Some versions of Python will occasionally use the mark-and-sweep algorithm to be extra thorough as well. Once again, the Python method of garbage collection is much more intuitive than Java's version. However, it is quite expensive to maintain, especially in large programs that generate many objects. It is also not very good at detecting circular references, which is the main reason that the mark-and-sweep algorithm must be used periodically to clean them up. The program developed for this project is unlikely to generate enough objects so that the efficiency would be drastically affected. It is also improbable that any circular references would ever appear. Thus, for the sake of this project, the simpler memory management approach that Python provides is more than sufficient.

3.3 Multithreading

Java is much better suited to multithreading than Python is. This is due to the fact that it has a more complex and thorough garbage collection system and because of the Java Memory Model (JMM). The JMM details what behaviors are legal within Java multithreaded code, which is already a telling sign that Java has put a lot of thought into multithreading. This allows for complex multithreading which can be used to drastically increase the performance of programs that require large amounts of computations or instructions. However, multithreading is far from perfect in Java. There is the ever-looming problem of race conditions that must be addressed. Race conditions are easy to create and difficult to debug, which is the main reason that parallel programming can be so difficult. The maintenance of locks and semaphores can also be quite a pain for programmers inexperienced with parallel programming as well.

Once again, Python is very straightforward when it comes to multithreading. It allows for multiple threads to be created, but it does not allow true parallelism because the threads can not be run simultaneously by multiple processors. This is enforced by Python's Global Interpreter Lock, or GIL. The GIL makes prevents a program from truly becoming concurrent as it allows only a single thread to run at a time. In a sense, the threads must take turns using resources. Of course, this means that Python programs can not achieve that efficiency boost that Java programs can. Python programs will just have to be slower overall. It is very helpful that programmers will not need to worry about locks, semaphores, or race conditions with this very simple approach though. For the sake of this project, the Python approach is sufficient. There should not be any operation that is complex enough so that it would benefit heavily from parallel processing. Implementing multithreading in this project would be far more trouble than what it is worth for this reason. Overall, Python works better than Java for this project when looking at type checking, memory management, and multithreading because the program is decently simple.

4. The Asyncio Library

Python's asyncio library plays a huge role in allowing this project to succeed. This library is designed to give Python programs a way to run concurrently using its core async/await syntax. It is a popular foundation for Python frameworks that handle database connection libraries, task queues, and much more. This makes it perfect for the purposes of this project.

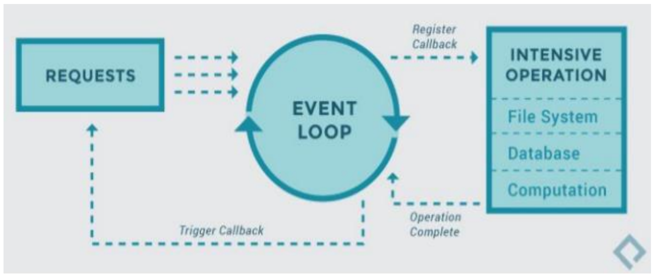
4.1 Asyncio Overview

The main reason why asyncio is beneficial for this project is because it provides a simple, single-threaded approach to multitasking. As mentioned before, there is no true parallelism in Python so asyncio provides a way for the servers to take turns using the resources. For instance, one server may be working on fulfilling a request and it can be asked to stop and yield control to another server so that it can perform some work. This allows for efficient sharing of computation resources that is organized and concise, which is great because there are five servers that need to be well-maintained.

A brief explanation of the purpose of async and await is as follows:

- *async* – This keyword should serve as a label for a function. It means that the function is coroutine that can be suspended mid-execution to allow another async function to run.
- *await* – This command suspends execution of the routine or coroutine that is currently running. This is typically used to stop one routine so that another one can run.

The core reason that asyncio succeeds is because of its event loop. The event loop schedules the coroutines with a task queue. The general idea of the event loop is that when one coroutine is suspended, the next coroutine in the task queue gets its turn to use resources and make computations or do whatever it needs to do. This is quite similar to a factory service line, or a scheduling algorithm seen in operating systems. A high-level visual model of the event loop is shown below:



4.2 Asyncio Analysis

While asyncio is definitely helpful in the implementation of this project, it is by no means perfect. This section will give an overview of the pros and cons of asyncio.

Asyncio is a library that almost seems like it was built for the purposes of this project. It is quite intuitive and simple to learn how the basics of the library. It is great as it provides a way to create coroutines so that the servers can work with multiple client requests while also being able to cede control to different parts of the program. In particular, the built-in functions are a huge bonus to this library since they provide simple ways to maintain the five servers. This saves a lot of effort because it means that the programmer is not forced to come up with their own implementations for many of these more common functions. Asyncio also tremendously simplifies the problem of communication over a network. This is quite a difficult feature to implement without the built-in functions. Examples of extremely useful built-in functions that would be difficult to implement are "open_connection" and "start_server".

There are a few downsides to asyncio. Perhaps the most obvious one is that asyncio is single threaded. While this is partially because of Python's core implementation and the GIL, it is definitely a con worth mentioning. This limitation means that programs that are sufficiently complex and require a lot of computation may experience subpar performance when implemented with asyncio in Python versus a multithreading library in another language. Another downside is that the programmer does not have total control over the execution of coroutines. This could prove to be problematic in situations where there are dozens of servers and many more types of client messages. Thankfully, neither of these issues really pose a problem for this project because it is quite small-scale and simple. However, asyncio is probably not the best option for larger, more complex programs.

4.3 Asyncio vs. Node.js

Asyncio and Node.js are both similar frameworks that serve similar purposes. Both are asynchronous and use an event loop that allows for better performance. Node.js is a bit different in that it uses JavaScript and runs on browsers, but the general idea is the same. For example, both of those frameworks have the await and async keywords integrated into them. One of the major differences is that asyncio uses coroutines while Node.js uses callbacks. While asyncio clearly works better since this project is meant to be coded in Python, Node.js would also likely be able to achieve similar results. Node.js is quite new in comparison to Python, but its user base is growing rapidly, and it may exceed the performance of asyncio in the near future.

4.4 Importance of New asyncio features

The asyncio library is updated regularly and recently introduced many new features along with the released of Python 3.9 in late 2020. There were both improvements to old features and the addition of many unique ones. These additions are all very handy, but none of them were really necessary in the implementation of this project. The only feature that was used in my implementation was asyncio.run, but this probably could have been circumvented with a little extra code. There was little need to use the new features as the older ones worked just fine.

5. Conclusion

The implementation of this project covered many interesting aspects of Python, such as TCP connections, interserver communications, and most importantly, the asyncio library. These tools were all very helpful in creating an application that utilized the Google Places API to provide clients with meaningful information about locations around them. This seemed daunting at first, but the use of the flooding algorithm and the application server herd led to an educational experience on how the simplicity of Python is one of its greatest characteristics.

6. Works Cited

https://web.cs.ucla.edu/classes/spring21/cs131/hw/pr.html

https://docs.python.org/3/library/asyncio.html

https://docs.python.org/3/whatsnew/3.9.html#asyncio

https://medium.com/python-features/pythons-gil-a-hurdle-to-multithreaded-program-d04ad9c1a63

https://wiki.python.org/moin/GlobalInterpreterLock

http://tutorials.jenkov.com/java-concurrency/java-memory-model.html

https://en.wikipedia.org/wiki/ISO_6709

https://www.wikimedia.org/

Discussion 1B Slides: Week 9, and Week 10

Professor Eggert's Lecture Notes and Recordings