Proxy Herd with Python's asyncio

Abstract

We are looking into alternatives to Wikimedia's server platform that uses GNU/Linux, Apache, MariaDB, Swift, and more technologies linked together. This multifaceted approach works for Wikipedia, but we need something better for a news service with rapid updates, various access points (not just HTTPS), and heavy mobile phone usage. We are looking into the server herd application architecture, where many servers talk to clients, each other, and the core databases directly. We are assessing the viability of Python's asyncio networking library for this task, and comparing it to Java multithreading, as well as briefly mentioning Node.js.

1 Introduction

The current architecture that Wikipedia uses is not well suited for the task at hand. New servers will need to be added, and the application server will create a bottleneck in our system, increasing latencies for users. On the contrary, having many servers communicating with each other and the necessary databases for timely access and sharing of information is an intriguing alternative. This approach would dodge the bottleneck of Wikipedia's current approach, as having a herd of servers to spread out the workload on the system would improve performance. Clients could connect to the closest server that is taking on more clients (not fully busy at the moment), leading to reduced transmission and propagation delays worldwide. By setting up a network between the servers, data sent between, say, client A and server A could effectively be flooded to all other servers in the system (as well as any other important information, such as server shutdowns and other time-critical errors/updates).

Python (version 3.11.2) is the primary language being considered for this approach. Using its asyncio library, servers could asynchronously network with other clients and servers in a safe and simple manner. For our investigation, we created a class Server that can be instantiated in different running programs to simulate our server herd. In typical concurrent applications like this, parallelism is exploited using multiple cores to achieve high performance. Locking mechanisms are needed to ensure correct ordering of execution, an important problem in a news application with timely information being passed through the network constantly. In today's world, consumers on the internet expect fast, accurate results, so delivering updates slowly (or inaccurately) is something our application will try to avoid. Unlike many concurrent and time sensitive paradigms, asyncio is single-threaded, meaning that we will rely on keywords like 'await' to ensure correctness. This may come at a performance cost, which is something we will consider when comparing it to two other possible implementations of a proxy server herd using Java or Node.js. Java specifically has a very well developed multithreading interface, with very popular paradigms associated with its thread object. Multithreading in Java is very popular, as it has been optimized for safety and performance over decades of language development. Python's asyncio has tough competition going against languages like Java, but its unique approach to asynchronous network draws benefits of its own.

2 Python's Memory Management

Firstly, Python's memory model is largely hidden to developers, providing an ease of use that C or C++ does not. Therefore, Python runs a garbage collector, so programmers do not have to worry about allocating or freeing memory. The mentality of Python is that programmers should spend their time focusing on the problem at hand and not have to worry about low level details like memory allocation. Garbage collection allows for this, at the cost of performance. Python's garbage collector is implemented fairly simply and is based on the idea of reference counts. Every time an object is created (or referenced by another point of code), its reference count gets incremented. Essentially, Python keeps track of how many things in a codebase are still "looking" at a certain object. When one of those things falls out of scope, returns, etc. Python decrements the reference count for the objects it looks at. Once an object's count reaches 0, nobody cares about it anymore, and the garbage collector can safely free its memory for someone else to use. For simple lines of execution in a Python program, like creating a new variable, this overhead is minimal as we simply have to maintain and increment/decrement a count, a blindingly fast operation. However, for large, possibly circular data structures in a codebase, maintaining counts and especially freeing memory can cause large amounts of latency and inconsistent and seemingly random times. Unpredictable performance halts are core to Python's nature, and this is one key example of how they occur. For the simple server herd simulation we created in our investigation, these "performance dangerous" data structures were not present, so Python fared well in terms of memory management and performance. However, in a real-world application, we could certainly run into delays depending on the data being sent across the network.

3 Python's Type Checking

Python is a strongly, dynamically typed language. By being strongly typed, we are guaranteed that the type of a variable does not change unexpectedly. During execution, only very simple implicit conversions are allowed, and the original objects remain their expected type. This is a sort of safety procedure that attempts to balance or limit the unreliability that comes with the intense dynamic nature of the language.

By being dynamically typed, programmers can change the type of a variable on a whim with explicit casting. A string x can be set to 5, and the interpreter sees no issue whatsoever. Therefore, type checks occur at runtime, leading to even more overhead in even the simplest of lines. This performance cost is largely seen as a good deal, however, as the simplicity and ease of use associated with Python's dynamic typing allows for more concise, and easy to digest programs. Python code is very readable, and writing it takes significantly less effort due to the language's typing mechanisms. Using API's become more user-friendly, as we do not need to know the exact specifications and types of the arguments or return values. This lets programmers use the asyncio interface in a more hassle-free manner.

4 Python's asyncio

The asyncio library is Python's flagship product for asynchronous networking. This is exactly what we need to develop a server herd, as our servers will have to wait for other processes to cooperate with their needs. Due to Python's multithreading limitations, this is our best bet at getting correct behavior and solid performance. By allowing cooperative multitasking, our servers can effectively work together, eliminating the bottlenecks that would be present in Wikipedia's approach. A server will wait regularly, either for a certain response or to confirm that an event has occurred across the network. To achieve this, asyncio uses the 'await' and 'async' keywords. We use 'async' to define a function that is capable of suspending its execution, likely waiting for an event to occur before waking up again. The 'await' keyword is how this is done, as our program will await a specific condition before executing after the line this is declared in. This will be essential in our networking program, as we will await a response before processing/replying to said response.

5 Java's Memory Management

Java takes a similar approach to Python in regard to memory, but with an innovative and unique implementation. In Java, developers do not need to worry much about allocating and freeing memory, as they simply use the 'new' keyword when creating objects. Like Python, this lightens the workload of the programmer and allows them to better focus on the task at hand. Java also has a garbage collector, but it works much differently than Python's. Java periodically starts at the roots of object trees and traverses through everything it can find in the heap. Any object not found once completely done traversing is no longer in use, and ist memory can be freed. As this is a periodic sweep, we will see a large, seemingly random drop in performance during runtime when the garbage collector is in use. If the heap is very large, with many objects still in use, this could lead to real-time issues for our application. However, Java's garbage collector fares much better against circular data structures, so if these are common in our database Java may perform better.

6 Java's Type Checking

Java is a statically typed language, resulting in much different behavior and coding practices than Python. Developers must specify the type of each variable in their program, which allows type checks to be done at compile time. This leads to better error detection, and decreases overhead during runtime, improving general performance. However, this is annoying and requires extra effort from a development perspective for every single variable used within the codebase. For large, real-time applications where smooth running is essential, the safety Java's static typing provides is worth the extra development hassle. Java is also strongly typed, drawing the same characteristics from Python's strong typing.

7 Java's Multithreading

Unlike asyncio's single-threaded approach to concurrency and parallelism, Java has very popular paradigms to execute multithreaded code. Python's Global Interpreter Lock severely limits the parallel threading capabilities in Python, whereas parallelism in Java is unbounded. With great power comes great responsibility, so Java must also be equipped with locking mechanisms to protect critical sections and guarantee correctness. Thread objects can be implemented to safely execute code on multiple cores or servers at once, leading to potential performance increases. In our case, we would use Java's preemptive threads, allowing us to switch threads at runtime to the most important task or server based on a priority algorithm. This could be useful for our real-time application, as responses arriving, updates occurring, etc. could change the priority of who should run. However, there is also significant overhead involved with setting up our threads, so parallelism may not be worth it if it cannot be exploited enough. In the small-scale simulation done in our investigation, parallelism would not provide enough benefit to be worth the overhead or safety concerns that come with parallel code. However, in a large-scale operation, where very large amounts of data must be processed efficiently, multithreading could provide a potentially massive impact, outperforming any single-threaded approaches (while increasing code complexity significantly.

8 Proxy Herd Details/Implementation

Our proxy herd simulation consists of a relatively small set of message interactions. Clients will choose which server to connect to and then ask about places near a specific location. If a specific location is in the database, the server will return answers using the new Google Places API. Thus, if one server learns about a location, it must flood the information to the collective database that all servers pull data from. Our interface is as follows:

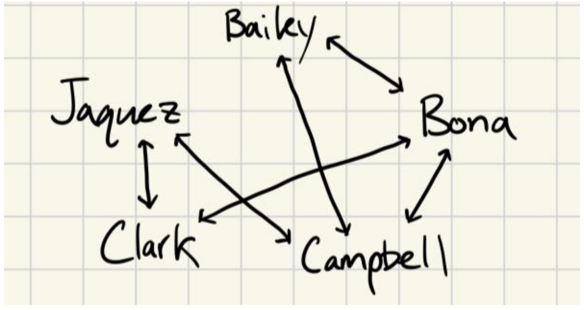
Client sends: [invalid message]
Server responds: ? [invalid message]

Client sends: IAMAT [clientID] [latlong] [timesent]
Where latlong is the latitude and longitude of the Client, this tells the server where Client is.
Server responds: AT [servername] [deltaT] [copy]
Where deltaT is the time difference between when the server received the message and when the client sent it, and copy is a copy of Client's message with "IAMAT" omitted.

Client sends: WHATSAT [oclient] [radius] [bound]
Where oclient is the ID of another client, radius is the distance we will search around the client's location, and bound is an upper bound on the number of places we will return. The server tells Client what is within radius km of oclient with the response:
AT [servername] [deltaT] [copy]
[JSON object]
Where JSON object contains information about all of the places the Google Places API found.

This simple interface provides the foundation for what our news service will be required to do. For our simulation, we had 5 servers and a sample client that could pick which server to connect with. When our connected server learned something from the client it had to flood the information to all of their connected neighbors. We had servers 'Bailey', 'Bona', 'Campbell', 'Clark', and 'Jaquez', with bidirectional communication patterns:

Figure 1: Network communication diagram



When reliably flooding, servers would simply send information to their neighbors, who would then send to their neighbors, etc. until everyone had the most updated information. Timestamps would tell servers what updates and propagations were necessary, and what information should be discarded. Using asyncio's 'start_server', 'open_connection', and 'run' API calls, we could effectively model this behavior in our asynchronous simulation. Simple argument parsing (argparse library), timing mechanisms (time library), and data formatting conversions (json library) allowed our servers to effectively distinguish between message types and behave/respond correctly using time checks and the proper JSON objects returned by the Google Places API. The use of these libraries and Python's flexibility allowed for a clean design process when implementing our proxy server herd.

Although Python provided a great deal of ease during development, this project still had certain challenges. Firstly, the old Google Places API was deprecated, so code we had been given as an example was no longer relevant. I initially tried to get a working solution using the formatting of the old API as others had, but quickly learned that I would need to refactor. This involved changing the format of the parameters passed to the API and obtaining an API key from Google. As well as this, I was inexperienced in the logging, asyncio, and json interfaces in Python, so I had to read documentation to learn which function calls I should be using.

9 Python and Java: Key Differences

Python and Java offer very different strengths and weaknesses as languages, as they attempt to solve different problems. For concurrency, Python's execution model largely relies on cooperative multitasking in the form of asynchronous event handling through libraries like asyncio. On the other hand, Java is widely used for parallelism and includes all the safety mechanisms required to build fast, reliable parallel programs. Java's preemptive threading mechanism would be useful for real-time applications like our news service, as threads could adapt on the fly to allow the highest priority job to run.

For CPU-bound tasks, Java's true parallelism is able to be exploited, resulting in great performance. However, in I/O-bound tasks, Java may have more threading overhead and spin-waiting, whereas Python's asynchronous event handling may be more natural, less resource intensive and quicker.

Python is widely regarded as the easiest programming language to code in, as it is extremely flexible, the syntax is elegant and easy to understand, and there are many amazing libraries to make our lives easier. However, everything comes at a cost, and Python's flexibility can cause runtime nightmares that could lead to constant debugging and maintenance. Java takes the opposite approach, with much of the effort being done before runtime. Characteristics such as static typing cause Java to be more of a pain while developing but can catch way more errors at compile time. Once finished, Java programs are more likely to run safely than Python programs, requiring less debugging and maintenance for a finished product.

Python's deployment issues can be exacerbated in large codebases and systems where dependencies can lead to a larger probability of runtime errors. As such, Python is best in small-scale environments like simple scripts. On the contrary, Java is built for large-scale systems design due to its safety features and emphasis on catching issues at compile time. As well as this, as a project scales up, Java's parallelism paradigms can likely be exploited more with larger datasets, leading to greater efficiency.

10 Asyncio or Multithreading: The Pros and Cons of Each

Using asyncio's asynchronous event handler for networking relieves us of dealing with the very complex issue of locking, critical sections, and other problems with writing parallel code while still allowing our servers to interact in a concurrent manner. The asyncio library contains many simple and useful interface calls, providing a simple to understand control flow within our network. The use of Reader and Writer Streams allows messages to be elegantly processed and sent. Overall, the asyncio API provides a hassle-free mechanism for guaranteeing robust networking code free of the dangers of parallelism.

Asyncio, while providing a great interface, does so at the cost of performance. By avoiding the complexity of true parallelism, we also avoid the benefits of true parallelism: rapid performance. Asynchronous event handlers can only do one thing at a time, so a very intensive real-time application built under asyncio may have delays when there is heavy traffic. Unlike preemptive threads, asyncio has predetermined ‘await’ times, so it cannot be dynamic during runtime in deciding who gets to do what. If priorities of tasks are quickly shifting in an application, and high priority tasks must be done ASAP, asyncio has no way to adjust to these needs.

Multithreading solves many of asyncio’s issues while introducing its own. In a large-scale application that handles lengthy data, true parallelism offers great performance (but requires careful safety measures for reliability). Preemptive threading allows for more intelligent and dynamic decisions at runtime, which would be very useful depending on the application. The primary drawbacks of multithreading are difficulty to develop reliably as well as overhead costs in small-scale systems.

11 A Brief Look at Node.js

JavaScript is a dynamically, weakly typed language. Already, this tells us that programs in JavaScript are prone to reliability issues at runtime, even more so than Python. Node.js is JavaScript’s asynchronous, single-threaded framework, as JavaScript is also primarily single-threaded. Asyncio and Node.js are similar in many fashions, as both rely on event loops and waiting to achieve the desired result. As well as this, both are web development friendly with the use of Python’s aiohttp library and Node.js’ built-in HTTP server. Node.js is generally faster than Python due to its V8 engine optimized for asynchronous I/O, but asyncio generally provides a simpler design and easier to understand code.

12 Importance of Python 3.9 for asyncio

In Python version 3.9 or later, developers have access to features of asyncio such as asyncio.run() and python3 -m asyncio that are unavailable on older versions. Although asyncio.run() is an easy and useful tool, methods like new_event_loop or run_until_complete would not be too difficult to implement instead. For our simulation, python3 -m asyncio was never needed. Overall, Python 3.9 or later provides a nicer interface, but isn’t necessary.

13 Conclusions

Between Python’s asyncio, Java’s multithreading, and JavaScript’s Node.js, I believe Python was best suited for our simulation of the proxy herd. Our 5-server herd was a small-scale application with small datasets and large amounts of I/O waiting (due to the nature of computer networking) which best fits into Python/asyncio’s use case.

In our case, we did not have to deal with large, circular data structures (mainly just simply strings and JSON objects), so Python’s garbage collector will perform well and Java’s will not see a large advantage.

For a small system, Python’s type checking paradigms make it very easy to program in, and its flexibility leads to understandable code. For our purposes, this is worth the tradeoff of safety/reliability that we would get with Java.

Asyncio’s asynchronous event handling does perfectly well at commanding behavior throughout the network while dodging the pains of true parallelism. Java’s concurrency models only get to shine under large systems with tons of data, which was not the case here. I/O bounds are the primary concern in our application, so Python trumps Java, a language that would benefit from CPU-bound tasks.

Node.js may be too unreliable for our job, and weak typing could lead to hard to debug issues when using multiple interfaces/APIs.

In conclusion, for our small-scale simulation of the proxy herd, Python’s asyncio is preferred over Java’s multithreading or JavaScript’s Node.js. However, if we were dealing with a large-scale version of this application, with substantial datasets and real-time dynamic threading requirements, Java would likely have been the better choice.

References

[1] asyncio Documentation
https://docs.python.org/3/library/asyncio.html

[2] Google Places API
https://developers.google.com/maps/documentation/places/web-service/overview

[3] Node.js Documentation
https://nodejs.org/docs/latest/api/

[4] GeeksforGeeks Java Multithreading
https://www.geeksforgeeks.org/multithreading-in-java/

[5] GeeksforGeeks Python asyncio
https://www.geeksforgeeks.org/asyncio-in-python/

Appendix

Sample Use Case:
Start server: `python3 server.py Clark`
Start client: `python3 sampleclient.py Clark`
Below is happening in the client terminal:

```
Enter input to send to Clark: test
Sending message to Clark
Received message back from Clark
Enter input to send to Clark: IAMAT
kiwi.cs.ucla.edu +34.068930-
118.445127 1621464827.959498503
Sending message to Clark
Received message back from Clark
Enter input to send to Clark:
WHATSAT kiwi.cs.ucla.edu 50 1
Sending message to Clark
Received message back from Clark
```

And below is what Clark.log looks like:

```
INFO:root:I, Clark, am starting
INFO:root:I received: test

INFO:root:Replying with: ? test

INFO:root:I received: IAMAT
kiwi.cs.ucla.edu +34.068930-
118.445127 1621464827.959498503
INFO:root:Error with TCP connection
to Jaquez
INFO:root:Error with TCP connection
to Bona
INFO:root:Replying with: AT Clark
+120360031.58941054
kiwi.cs.ucla.edu +34.068930-
118.445127 1621464827.959498503
INFO:root:I received: WHATSAT
kiwi.cs.ucla.edu 50 1

INFO:root:Replying with: AT Clark
+120360031.58941054
kiwi.cs.ucla.edu +34.068930-
118.445127 1621464827.959498503
{
    "places": [
        {
            "name": "places/ChIJ-
VmDrIi8woARlPzcc6OkHmU",
            "id": "ChIJ-
VmDrIi8woARlPzcc6OkHmU",
            "types": [
                "health",

"point_of_interest",
                "establishment"
            ],
            "nationalPhoneNumber":
"(310) 206-1915",

"internationalPhoneNumber": "+1
310-206-1915",
            "formattedAddress":
            "Strathmore Building
            North Entrance, 2nd &
            3rd Floors, 501
            Westwood Plaza, Los
            Angeles, CA 90095,
            USA",
```

This JSON output continues for many more lines…