CS131 Project Report

Devyan Biswas
804-988-161

Abstract

Nowadays, with the speed and reliability dominated network requirements, the architecture of web server(s) plays a big part in our day to day lives, wether we actively realize it or not. For example, WikiMedia utilizes a LAMP platform for their own servers. However, with mainly mobile users and frequent updates, LAMP's highly centralized system can cause issues. Thus, a different approach is required to support this need. In this project, we will implement a proxy server herd architecture using Python's asyncio module and analyze the benefits and downsides to such an approach.

1. Introduction

First of all, what is a server herd? Like the name suggests, it is a group of servers that communicate openly with each other. This removes the common bottleneck problem you get with a centralized server architecture. The servers each share the information (or at least a copy). With this model, the lookup time per request made to each server is minimized, since they all have the same information. The main point at which this can create bottlenecks, conversely, is with increasing server numbers. However with a multiple update requirement, the server herd approach is much more efficient in reducing the information lookup time and situations in which one device connects to multiple different servers. This particular example creates a scenario in which one device may update its own location, and can also request any server for other known client(s) location(s).

Besides purely implementing a prototype for this system, the analysis will explore the benefits of python and python's asynchronous event-driven API's (asyncio) vs. JavaScript's (Node.js).

2. Prototype

Now, onto a little bit about the prototype itself and some details of implementation. There are 5 servers in the server herd, named Goloman, Hands, Holiday, Welsh, and Wilkes. These servers are assigned port numbers 11490-11494 respectively. Each server must be able to communicate with client devices attempting a TCP connection via the web socket allocated art this port. Additionally, any information passed to one server via IAMAT message must be propogated to the other servers by way of a flooding algorithm and AT message. The bi-directional connections between servers is in the chart[1]. Clients can also send WHATSAT messages, requesting information about clients/users that have communicated with the server herd and ask about nearby locations using an HTTP request to Google Cloud API's Google Places Platform. Incorrect commands are handled by returning:

? [invalid_command]

A typical event loop of a server will go through the following: accept clients and input, determine the command's validity, if so, write/update the information, send the correct response message to clients, and continue until a keyboard interrupt.

SERVER	COMMUNICATES WITH
Goloman	Hands, Holiday, Wilkes
Hands	Goloman, WIlkes
Holdiday	Goloman, Welsh, Wilkes
Welsh	Holiday
Wilkes	Goloman, Hands, Holiday

[1] Server-Server comm.

2.1 IAMAT Messages

IAMAT messages are sent only from client to server, acting as a way to either add themselves to the server's list of known client devices or to update their data (like location and the timestamp of the command). The messages are of the form:

IAMAT [client_name] [latlon_string] [time_sent]

Client_name is the name of the device sending the message, latlon_string is a string of the latitude and longitude of the location of the device in ISO 6709 notation, and time_sent is the time at which the client says it sent the message (there is often a time delay between this and the time the server gets the message, so I chose to record this as well). Time is in seconds and nanoseconds since 1970-01-01 00:00:00 UTC. The TA's mentioned only up to date commands would be sent, therefore, I do not check for time in the IAMAT message processing section, and I do not also have a robust check for each message (meaning so long as a newer IAMAT is sent, there will be an AT response sent to the client).

2.2 AT Messages

A valid IAMAT message with updated data will return this to the client device:

AT [server_name] [time_diff] [client_name] [latlon_string] [time_sent]

This has the same information as the IAMAT, but with the addition of time_diff info as mentioned above and the client's name and a change in ordering. However, AT messages can also be sent in between servers (via the flooding algorithm and bi-directional connections above). The format of those messages is:

AT [client_name] [server_name] [latitude] [longitude] [time_diff] [command_time]

With all the same parameters with a split longitude and latitude for ease of parsing. The reason I chose to do it like this is so that there is a structural difference between the two types of AT messages. Additionally, within server.py, the manage_commands function actually uses the same value for time_diff and time_sent. For inter-server communication, this is fine, as any propagation and update will either overwrite this (if sent an IAMAT directly), or will ignore if not the most relevant. This doesn't affect what gets sent to the client, so it maintains its structure and keeps to spec.

2.3 WHATSAT Messages

These are messages clients send to get information about nearby locations to other clients or themselves. The WHATSAT message format is:

WHATSAT [client_name] [radius] [number_of_entries]

To understand the radius and number of entries, we have to talk a little bit about the overall function of WHATSAT. The WHATSAT command queries the Google Places platform via an HTTP request to the API's endpoint. The format of the query is:

'https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=%s,%s&radius=%d&key=%s'

What this command does is that is looks at nearby locations to the latitude and longitude passed to it, within a certain radius. The radius WHATSAT gets passed is the radius we put into here, the latitude and longitude from the client devices list we have in each server, and the key is hardcoded from the Google Places API. Additionally, the number_of_entries allows us to limit how much information is displayed.
The response to a WHATSAT command is the exact same as the response that is sent to the client after a successful IAMAT command, but with the addition of a formatted JSON message:

AT [server_name] [time_diff] [client_name] [latlon_string] [time_sent] [JSON_msg]

The information from the Google Places API, in addition to being sent to the client in this format, is also logged to the log file for the server that was requested.

3. Asyncio Review

Since there were no real benchmarks that could be done with the way I implemented the servers (in a way of comparing with, say, a Node.js approach), it will be a bit hard to discuss the base speed of the servers, but there are other significant topics to be discussed with regard to its usability and its processing ability.

3.1 Pros of Asyncio

Asyncio, as a framework, makes it very easy to start up and run servers that can handle asynchronous events, and within each server, the event loop allows for adding coroutines for every new message. Due to the way the message is processed, it allows us to add multiple requests for processing at once, and the only thing that we need to do is create a coroutine and add to the event loop! We can also add new connections at the same time as processing messages, accommodating for new readers and writers, which is perfect for the mobile-centric server-herd architecture that we are attempting to create.

Code writing is also very simple. Since each server runs with the same code, there is very little need to create blocks and functions to copy over information. All we needed to do was add an inter-server processing function, and we had all the infrastructure after that already in place. Running servers is a breeze as well, with only a simple command needed to start up each one. For the actual server communications, asyncio comes with TCP and SSL support, which makes it reliable and compatible with web-sockets.

Additionally, what asyncio lacks in HTTP requests, Python covers for. Python's aiohttp module works well in parallel with asyncio, allowing for a client session within a task/coroutine of a server's event loop (within the whatsat message processing). And, using the JSON library, we can process the results from the Google Places API with ease.

Asyncio, at a fundamental level, is very much abstractly-centered. The ease of defining tasks and coroutines, coupled with the ease of setting up servers on top of the existing Python libraries, makes this a very valid approach to our current task.

3.2 Cons of Asyncio

However, there are a few problems with the asyncio library.

One of the biggest issues I had was figuring out the ordering of messages. The problem with asynchronous models in general is the lack of order in processing (ironically, one of the benefits) and the resulting messages. This made it hard to debug, and also means its is harder to deal with bugs. For example, before I make an asynchronous writing function that could be awaited, the logger would report errors with logging, making the claim that the writer had written and end of file and, thus, could not continue to write. This was a direct result of asynchronous calls backfiring and closing writers when not intended. This problem is exacerbated with the server-herd architecture, which creates even more confusion with logging. This creates a reliability concern for asynchronous models in general, including asyncio.

Another big problem is with maintenance of the servers themselves. Since each server runs the same code, any bugs in one, of course, affect the other. However, this also means that when working on fixing one server's big, each server must be shut down to make the changes stick. And this creates a problem with scalability, as mentioned earlier.

Lastly, there is the issue of multithreading. There is no support for multithreading, and this creates a bottleneck for performance. Granted, not as much as the lookup overhead from a synchronous, centralized server architecture, but definitely a significant amount. With increasing requests and more servers, the bottleneck for asynchronous servers may exceed the bottleneck caused by a centralized synchronous server system. This is because, even with asynchronous processing, an overload of commands without multithreading creates a linear bottleneck, whereas a multithreaded solution could simply execute in parallel (though this won't solve the out of order problem as mentioned before).

4. Python vs. Java

Another big concern overall is determining which language you want to use as as framework for the server herd. Python, while it is easier to write in and very commonly used, does have some problems with more robust and scalable applications. And for overall concerns, we have to weigh the ability of Java vs. Python in memory management, type checking, and multithreading.

4.1 Memory Management

Python's garbage collection primarily uses the link/reference count method, while Java uses mark and sweep. Each allocated object in Python has a link count, that increases or decreases when variables are assigned or copied and when objects are deleted. When an object hits a count of 0, no other variables are referencing it, and so it can be safely deleted and/or handled by the garbage collector. As mentioned in lecture, this technique fails with circular references, and does allow for overhead during references, since the reference count will have to be updated every time. However, for our purposes, where variables are used at most one or twice and are allocated frequently, this memory model works to our benefit by giving back memory quickly.

In contrast, Java's mark and sweep model, every time you wanted to create a new variable, or during a round of garbage collection, you would have to traverse all reachable objects to mark them. This creates a level of overhead at a higher rate and with higher impact to the server, since variables would be created and removed quite frequently. The same problem applies for the sweep aspect.

4.2 Type Checking

Python's type checking is dynamic, while Java's is static. Dynamic means that it will check at runtime, and static means that they must be checked (and in fact declared) at compile time. The nice thing about Python's "loose" type checking policy is the ease at which you can get started with setting up a server. Without knowing exactly the types needed (as mentioned before, the abstraction), you can get a running server with a quick review of the documentation. You need not know the type of a writer, a socket, or asyncio.open_connection().

For Java, you have to be aware of all types for all writers, readers, functions, variables, etc…. This requires a lot more knowledge and time to understand. Of course, as is with many things, as you learn more and more about the types, it becomes a very useful skill for debugging and, ultimately, makes the code more understandable and documentable. With the aid of statically checked types, you can also allow for optimizations and find points of interest simply because the abstraction is not there anymore, allowing more robust solutions.

4.3 Multithreading

Python's lack of multithreading serves as a major disadvantage, as mentioned earlier. Python uses a global interpreter lock, meaning that the synchronization of threads forces a one at a time execution. Java, on the other had, can fully utilize multiple threads and processors, taking the same amount of time to process more commands and messages. This also means a python server is less scalable than a Java server.

5. Asyncio vs Node.js

Both are frameworks for server-side asynchronous event driven code. Essentially, it's a debate in Python versus JavaScript. Javascript and Python are both dynamically typed, and so they both utilize duck typing. This means the ease of usage is the same between the two. OF course, JavaScript is one of the most common browser/web-based languages, so compatibility in that regard makes Node.js a bit more attractive, but Python boasts many useful libraries that make it more robust.

5.1 Performance

Node.js is based on the V8 engine of chrome, which is a ridiculously fast and powerful engine. By contrast, Python's performacs suffers with memory intensive programs. Of course, this is not a big issue with server herds of our scale, but for larger and perhaps more centralized systems, this might be an issue.

5.2 Multithreading / Concurrency

Since both are single threaded, event driven, and asynchronous, they share a lot in common. The concepts of the event loop, for example, still has the parallels of Futures/Tasks, but they have Promises. The promise functions like the Future, stating that it will evaluate it sometime in the future. Node.js does not have coroutines, though, which makes asyncio more dynamic and flexible.

6. Conclusion

Python's asyncio is a very robust and suitable framework for the application at hand, as shown by the testing, and is a reasonable choice for practical use. Python is also a common and "easy" language to use, with its heavy abstraction, duck typing, and quick memory management, not to mention its code-reusability. The real benefit of asyncio comes from its asynchronous event-driven architecture, which allows for tasks and Futures to be stored and executed in reasonable time. This also allows for a decentralized server-herd architecture. Though Node.js wasn't looked at in depth, it would also be a reasonable framework for this task.

References

https://aiohttp.readthedocs.io/en/stable/

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

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

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

https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/

https://cloud.google.com/maps-platform/places/

https://realpython.com/async-io-python/

https://asyncio.readthedocs.io/en/latest/tcp_echo.html