Investigating Proxy Herding with Python's asyncio

Katherine Miao
UID 204970866
UCLA CS 131
Winter 2020

Abstract

In this report, we will investigate a particular architecture type called an "application server herd," in which multiple application servers communicate with caches, core databases, and each other. We assume we are building a news service in a style similar to that of Wikimedia; however, our application has much higher frequency article updates, various protocol accesses beyond HTTP, and majority mobile clients. We will look into Python's asyncio asynchronous networking library as a potential replacement for the Wikimedia platform.

I. Introduction

This project aims to utilize asyncio to implement an application server herd. We will focus on facilitating inter-server transmissions that can occur without eating time by going to a central database to get client location updates, greatly improving speed and performance. asyncio is an appropriate choice for this application, as it features an event-driven nature that allows an update from a client or another server to be processed and forwarded rapidly to the other servers in the herd.

asyncio provides an excellent infrastructure for concurrent single-threaded execution, making it suitable for this news application that must process updates concurrently, along with access protocols beyond HTTP and provide a straightforward client-server framework. We will proceed by discussing the implementation of our news application, featuring five interconnected servers that serve clients using TCP connections. This will be followed by an analysis of the suitability of asyncio by discussing factors such as performance, features, ease, and an overarching look at Python versus other languages.

II. Application Implementation

Prototype Overview:
Our prototype supports five servers (Hill, Jaquez, Smith, Campbell, and Singleton) that have pre-determined connections between one another to indicate who may communicate with

Client-Server Communication:
Each server accepts TCP connections from clients that simulate mobile users with IP addresses and DNS names. Clients send their location in latitude and longitude to one of their connected servers using an IAMAT message, formatted as so:

```
IAMAT <client ID> <coordinates>
<timestamp>
```

Servers then parse the message and update their client history if the timestamp proves it to be more recent. The server also floods its connected servers with a special AT message; see Server-Server Communication regarding AT commands below. Finally, the server responds to its client with an AT message including its own server ID and a copy of the received information:

```
AT <server ID> <time difference>
<client ID> <coordinates> <command
time>
```

Clients are also able to query servers for information about places near other clients' locations. This is done through a WHATSAT command, including a radius in kilometers and a maximum number of results to return:

```
WHATSAT <client ID> <radius>
<maximum results>
```

To implement this functionality, once the client sends a WHATSAT, the server constructs an API request string using the information it has stored in its client history, and sends it to Google Places using aiohttp. Google responds with a JSON that the server parses and sends back to the client. This response is accompanied by another AT message.

Server-Server Communication:
The final type of command a server can receive is an AT from other servers:

```
AT <server ID> <time difference>
<client ID> <coordinates> <command
time>
```

When a server receives an updated location through an IAMAT or AT, it must inform the other servers about it by flooding its connections. The challenge here was writing a flooding algorithm that ensured every server received the update (regardless of being connected to the client) and preventing infinite loops. My flooding algorithm essentially forwarded an AT message to all connected servers, but included a list of which servers to ignore. This list includes the server that originally sent the AT, therefore reducing redundant forwarding. When a server receives an outdated AT, it simply drops it and doesn't forward to its connections.

Miscellaneous Behavior:
In the case of an invalid command type, the server responds to the client with a ? and a copy of the invalid information:

```
? <invalid command>
```

In the case of invalid arguments, such as incorrectly formatted latitude/longitude coordinates, we don't respond with a ?, we simply print and log the error and drop the command. When we encounter a duplicate command, we reply with an IGNORE.

III. asyncio Evaluation

Features:
asyncio provides a set of high-level APIs to control subprocesses, perform network I/O, and run Python coroutines concurrently while having full control over their execution. Contrary to common misconception, Python does support threads; it merely switches between threads (i.e. concurrency) rather than running them at the same time (i.e. parallelism). Therefore asyncio is a single-threaded approach for concurrent programming. Its non-preemptive multitasking means that tasks run for as long as they want, and cannot be interrupted by another process.

This project focused on building an asynchronous server system that supports client-server and server-server interaction, as well as servers interacting with an outside source (Google Places). Therefore the main features we use in this project are the `async` and `await` keywords. `async` flags a function as a coroutine, meaning the function can suspend its execution and give control to another coroutine. The `await` keyword suspends execution of the current coroutine until the awaited function call is finished. These features were rather straightforward to understand and use after reading through some online documentation and the TA's discussion slides and helper code.

Ease:
Personally, I found both Python and asyncio's syntax to be fairly straightforward to install and learn. Since Python is such a popular language these days, there was a plethora of documentation, blogs, and forums where I could find clarity on fuzzy concepts. Python also supports data structures like dictionaries that proved extremely useful for this application, and made managing the data both simple and efficient. On top of that, I took advantage of Python's object-oriented-programming support to further organize my code.

However, I argue that with the goal of programming a server herd, the asynchronous nature of asyncio is more difficult to program than using synchronous execution and traditional routines most programmers are familiar with. On top of that, I found that the error messages outputted for asyncio errors were incredibly vague and unhelpful for debugging. Oftentimes it was much more straightforward and efficient to rely on adding my own print statements than attempt to decipher error messages.

Version Reliance:
One concern regarding version reliance came to my attention when running the TA's sample grading script. For two of the five test cases, a Deprecation

Warning popped up (The loop argument is deprecated since Python 3.8, and scheduled for removal in Python 3.10). The line of code the warning was referring to was: reader, writer = await asyncio.open_connection(self.host, port, loop=self.loop).

Python Type Checking, Memory Management, and Multithreading:
There were also questions as to whether Python's implementation of type checking, memory management, and multithreading might cause problems for larger applications.

Python utilizes an automatic memory management system, which differs greatly from languages in which developers explicitly deallocate dynamic memory like C++. This means garbage collection happens when the references count for an object decrements to zero, at which point we assume that no more variables rely on this item. Keeping track obviously requires some space and operations, and negatively affects performance.

Multithreading is supported by Python, but its efficiency is limited by its Global Interpreter Lock, making it a poor choice for multithreaded implementation when compared to languages with much greater support like Java.

Type checking comes into play because Python is an interpreted language, and therefore has dynamic checking. This means variable types are assigned based on their values at runtime, and are able to change when the values change. This led to a number of type errors while building the prototype for this news application, especially when logging and converting between types. Again, error messages were rather unhelpful, despite being small and easy fixes.

Python asyncio versus Node.JS:
Node.JS is known for its non-blocking event-driven architecture: this allows simultaneous requests, greatly speeding up code application. And because Node.JS enables single-module caching, app loading time is severely decreased and allows for even more responsive web applications.

On the other hand, Python's multithreading can never be parallel because of its GIL. GIL (Global Interpreter Lock) is a lock that can only be held by one thread at a time, allowing the holder to execute freely while other threads must wait. This is important for Python since its memory management depends on reference counting, which leads to potential race conditions. Since Python is single-flow, Node.JS is often preferred in terms of speed.

IV. Conclusion

An application server herd that needs to rapidly receive, process, and propagate a high volume of requests should be implemented with an asynchronous framework like Python's asyncio. Based on our prototype's performance, asyncio proved to be an excellent choice as it provides well-documented and straightforward features that allow for asynchronous event handling.

References

[1] Solanki, Paresh. "Python vs. Node.JS: Which One to Choose in 2020?" *MindInventory*, 6 Mar. 2020, www.mindinventory.com/blog/python-vs-node-js/.

[2] Ard, Sam. "Is Python's Asyncio Worth It?" *The DEV Community*, 11 Mar. 2018, www.dev.to/swizzard/is-pythons-asyncio-worth-it--24lk.

[3] Flaxman, Michael. "Python 3's Killer Feature: Asyncio." *PAXOS*, 21 June 2017, www.eng.paxos.com/python-3s-killer-feature-asyncio .
[4] "Coroutines and Tasks¶." *Coroutines and Tasks - Python 3.8.2 Documentation*, www.docs.python.org/3/library/asyncio-task.html.

[5] Solomon, Brad. "A Guide to Using Asyncio." *Faculty*, www.faculty.ai/blog/a-guide-to-using-asyncio/.