Asyncio Framework for Application Server Herds

CS 131 Project

Stephanie Doan

I. Abstract

Web servers are structured in many different architectures that offer specific advantages and features. Here we are presented with the existing Wikimedia server platform, groups of full web servers behind a load balancer. In our study, we seek to explore the use of Python and its asyncio asynchronous networking library to implement a server herd. The proposed architecture features frequent communication between servers instead of a central database, suitable for dynamic data updates across servers. By writing a simple application that queries and stores data from the Google Places API, we examine the usability and reliability of this language and framework as a more efficient replacement for the Wikimedia system.

II. Introduction

A. Wikimedia Server Platform

Currently, Wikipedia and other Wikimedia projects are hosted on groups of servers located at datacenter around the world. The LAMP architecture is used, consisting of Apache web servers, PHP, Linux, and MySQL database. In front of these servers is a virtual load-balancing router and caching layer that all requests go through. This singular point of entry for a large amount of requests, as well as the direct and frequent communication with the database, is a bottleneck that hinders horizontal scaling by adding more servers. To adapt to highly mobile users and possibly increased traffic, we find the need to explore an alternative solution that is faster and more scalable.

B. Server Herd Architecture

In this study, we examine the application server herd architecture as a possible solution. It consists of servers that communicate with each other bidirectionally, as nodes in an undirected graph. Although each server is not directed to every other server in the group, each update from a client to the server will be propagated to other servers, which will pass on the information to its connected servers. Several degrees of communication can propagate an update across the graph, making this architecture good for frequent and small updates such as user location, which can come from many areas through different servers. Larger requests and queries still require communication with the database, but the decentralized system removes bottleneck caused by a single load balancer as an entry point.

C. Python Asyncio Library

To build this architecture, we decide to use Python's asyncio library, used to write concurrent code programs with "async" and "await". As opposed to multithreading and multiprocessing as previously seen in Java and C, Python's async IO is single-threaded and single-processed, mimicking concurrent behavior in a design called cooperative multitasking. Asynchronous routines together take on multiple tasks by pausing to wait for results of specified operations, such as IO and network requests which are often sources of bottleneck. While one routine is paused, other routines that are not waiting can run in the meantime. This design gives the feel of concurrency without multithreading or multiprocessing. The event-driven nature is a good fit for our use case, in which the servers act in response to user requests (or other servers).

III. Methods

Our study consists of building a simple application server herd with this async library, that can synchronize data and communicate with client applications. This herd is made up of five distinctly named servers with assigned port numbers, each of which is a separate process. Each server can communicate bidirectionally with a subset of the servers using TCP. The connections are as follows:

- Hill — Jaquez, Smith
- Smith — Hill, Singleton, Campbell
- Campbell — Smith, Singleton

- Singleton — Jaquez, Campbell, Smith
- Jaquez — Hill, Singleton

When a client communicates to one of the servers, either the client's location is updated in that server and then further propagated to that server's immediate connections, who further propagate this information to their immediate connections, forming a chain until all connected servers in the herd are updated. Therefore, all servers are quickly updated when discovering client information, distributing the knowledge across multiple nodes. Clients can also request more extensive location information that will trigger an HTTP GET request to the Google Places API. More detail on these specific requests are provided:

A. IAMAT Messages

The first type of message is the IAMAT message coming from the client, which updates the receiving server about its own location. This message follows the format:

IAMAT <client_name> <location> <posix_time>

Specifically, the location field is a string that specifies the positive or negative latitude and longitude in ISO 6709 notation, with "+" or "-" preceding each number (e.g. +34.068930-118.445127). The final field is the POSIX time, consisting of seconds and nanoseconds since 1970-01-01 00:00:00 UTC.

The server responds immediately to this client with a message acknowledging this new information, following the format:

AT <server_name> <time_interval> <client_name> <location> <posix_time>

Most fields are the same as those in the incoming message, except the server name and time interval. The server name is that server's own name, which communicates that this information lives with that server. The time interval is the time elapsed from the IAMAT message was received to when the server responds with the AT message. The server also propagates information to other servers in response to an AT message, but more detail is provided in Part C: Interserver Updates.

B. WHATSAT Messages

A client can also send a WHATSAT message, which asks the server for information about nearby places to a specified client. An error will be returned if that client's location has not yet been communicated to the servers through a previous AT message. Following a similar format, these messages are as such:

WHATSAT <client_name> <radius> <max_results>

Here, the radius is in kilometers and specifies an upper bound of the distance to search for nearby places. At most max_results locations will be returned to the client, limiting the scope of search so the application does not get overwhelmed by a single query.

The server acquires this information by sending an HTTP GET request to the Google Places API, sending an API key, location latitude and longitude, and radius. The received JSON response is then limited in count based on the maximum number of results and returned to the client in JSON format, following the same AT message sent in response to the IAMAT message.

C. Interserver Updates

In response to every IAMAT message, the server sends a message to every other server it is connected to, with the same format as the AT messages previously used. This type of message is easily recognized by every server as updates from other servers, and it will update its internal knowledge base of clients' locations. The next server will then send the same message to its connections, until essentially every server in our network is updated. Before updating the client location and propagating information, each server checks the timestamp of the received message to see if that update has already occurred. This prevents infinite loops of updates and is called the flooding algorithm.

We follow a specific protocol to respond to invalid messages from a client: ? <message>. For example, this can occur when a client's request has an invalid number of arguments or the location is poorly formatted and cannot be understood. Checks are put throughout each server's program, and every error is also logged along with other events.

IV. Analysis of Asyncio

Python's asyncio offers both high and low level libraries that provide various levels of control. High level APIs allow control over coroutines, subprocesses, task queues, network IO, and interprocess communication. Low level APIs allow management of event loops (that run tasks in a queue with control over the thread pool) for networking, subprocesses, and OS signals as well as protocols via transports. This framework mimics concurrency with a single thread on a single process, alleviating users from the need to deal with multiprocessing and multithreading and worries about race conditions and thread safety.

Our application server benefits from async's strength in handling multiple requests at once. Specifically, the server runs inside the *Asyncio.run()* method, which runs the top-level entry point in an event loop. Coroutines are the interleaved events that run inside this loop. Each coroutine is created to execute a specified task, useful for handling client requests, which are events unable to be anticipated. Each coroutine is guaranteed to run uninterrupted by another coroutine until it explicitly gives away control, similar to the remedy to race conditions when using threads. For each individual server, we use the coroutine *serve_forever()* which lets it accept connections until the coroutine is shut off.

Key to the async framework are the keywords *async* and *await*. A function declared with the async keyword is defined as such a coroutine, which has the ability to suspend its execution to give control to another coroutine. It yields control with the *await* keyword, which stops that coroutine until that awaited operation (network request, IO, etc.) is finished. These words are used often used in our program to maximize concurrency.

A. Usability of Asyncio

The production of our application server is an experiment with the usability of Python's asyncio and the ease of writing such an application herd program. The previously mentioned principles and APIs used to provide asynchronous functionality is simple to learn and use. The *async* and *await* keywords and common methods do not require drastic code changes to utilize, as well as minimal management of the event loop. In addition to the asyncio methods, the programmer is only required to declare functions with async and use await for critical operations, leaving the event loop management to the program. In addition, it was easy to add other functionalities that are compatible with Python, such as the aiohttp library, which we use to make HTTP requests to the Google Maps API. The flexibility and rich support of Python makes asyncio powerful for many use cases. Starting many servers is also simple, allowing rapid testing and expansion if needed.

B. Advantages of Python

We also examine Python as a language choice. Although it offers high readability, support, and flexibility, it is also necessary to look at other aspects. Firstly, the dynamic typing means that types are not checked until runtime, as opposed to static typing in Java, which checks type at compile time. Java requires every object and function to be declared by type for the program to compile, providing reliability during execution time, when type is assumed to be correct and does not have to be checked again. Therefore, Java code is less efficient to write but more efficient to run. Python's dynamic typing allows faster workflow since we do not have to exactly know every variable's type, but unexpected type errors can occur during runtime without thorough testing. Python is also slower because it is an interpreted language requiring each line to be processed every time in a loop during runtime, adding overhead that makes it up to 10 times slower than compiled code. This tradeoff is significant for our use case, which requires high performance.

Memory management of Python uses a heap that is allocated to objects and internal buffers via the Python memory manager, controlled by the interpreter itself to abstract this management from the user. This offers high usability but little control if specific operations and required. On the other hand, Java's memory is allocated to objects when they are created, but memory is not explicitly allocated like in C. The Java Virtual Machine (JVM) provides a garbage collector that frees space no longer referencing an object. On the other hand, Python uses a reference counter to free these objects, doing most of the work before Python's own garbage collection mechanism is used. The reference counter is slow because the interpreter must check if reference count of an object is zero whenever an object is referenced or dereferenced. This bookkeeping, as well as locking by circular references, incur extra overhead that should be considered.

Very different from Java, multithreading in Python is limited by the Global Interpreter Lock (GIL), making true concurrency impossible in a single process. This inability to multithread and perform parallel tasks across threads makes Python much slower than Java, which offers well developed and popularly used concurrent libraries. Parallel tasks and be split across threads and achieve higher throughput.

C. Performance of Asyncio

With typing, garbage collection, and inabilities to multithread in consideration, Python many performance drawbacks as opposed to Java for example, tradeoff for usability and ease to understand. Since the asyncio module is single-threaded, the coroutines execute serially, with IO operations run synchronously. But overall, since the event loop executes only one coroutine at a time, it would not run any faster than serially executed code that is never blocked by critical operations. We do not observe significant lag in our application server because the load is generally light during testing, but we expect significantly lower performance with increased traffic. The server herd architecture, as opposed to a single entry point, alleviates potential bottleneck, but a multithreaded solution still yields better performance.

D. Comparison to NodeJS

NodeJS is a popular runtime environment that runs JavaScript code outside of the browser. Like Python's async, Node follows an asynchronous, nonblocking IO model that uses an event loop to manage a queue of operations. Callbacks, the queued events in JavaScript, are analogous to Python's coroutines. The interleaving of these events gives the feel of concurrency.

Node is built on top of Chrome's V8 Engine and is by nature asynchronous, so it offers better performance in comparison to asyncio, which is just a library on top of the Python language, which is synchronous by nature. Although both Python and JavaScript are interpreted languages and are naturally slower than compiled languages, they are better for fast development. Although Python has a richer history and diverse modules that are compatible with asyncio, Node has a quickly growing developer base and also offers versatile libraries. In the case of our application server, Python suffices because any efficiency possibly gained would come from multithreading, which NodeJS is also unable to do. The ability to scale horizontally can be used to address traffic demands.

V. Conclusion

By building our own application server, we have gained first hand experience using Python's asyncio library, which we find to be very usable. Although tests performed on the application do not measure its performance in high traffic, research shows that Python's asyncio is slower than Java due to inability to multithread and its nature as an interpreted language. A library made for a synchronous language is also slower than Node, a framework built on a naturally asynchronous language. However, for the purposes of our Wikimedia server-herd structured application, the async library works well as it highly usable for the developer and useful for this case.

VI. Works Cited

[1] "Asyncio - Asynchronous I/O¶." Asyncio - Asynchronous I/O - Python 3.8.3 Documentation, docs.python.org/3/library/asyncio.html.

[2] Cannon, Brett. "How the Heck Does Async/Await Work in Python 3.5?" *Tall, Snarky Canadian*, Tall, Snarky Canadian, 13 Dec. 2018, snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/.

[3] Oct 12, 2017, et al. "Paxos Engineering Blog." *Michael Flaxman, Principal Engineer*, eng.paxos.com/author/michael-flaxman-principal-engineer.

[4] projects, Contributors to Wikimedia. "Wikimedia Servers." *Meta*, Wikimedia Foundation, Inc., 14 May 2020, meta.wikimedia.org/wiki/Wikimedia_servers.

[5] *Python Programming Tutorials*, pythonprogramming.net/asyncio-basics-intermediate-python-tutorial/.

[6] Romanyuk, Oleg. "NodeJS vs Python: How to Choose the Best Technology to Develop Your Web App's Back End." *FreeCodeCamp.org*, FreeCodeCamp.org, 21 Mar. 2020, www.freecodecamp.org/news/nodejs-vs-python-choosing-the-best-technology-to-develop-back-end-of-your-web-app/.

[7] Sridharan, Mohanesh. "Garbage Collection vs Automatic Reference Counting." *Medium*, Computed Comparisons, 21 Aug. 2018, medium.com/computed-comparisons/garbage-collection-vs-automatic-reference-counting-a420bd4c7c81.