Project: Proxy herd with `asyncio`

Anh Mac (905-111-606)
University of California, Los Angeles

Abstract

This paper discusses the implementation of an application server herd that replicated the Wikimedia Architecture. It discusses the strengths and weaknesses of the `asyncio` library used and offer recommendations by comparing with the JavaScript `Node.js` library.

1. Introduction

The Wikimedia Architecture uses a LAMP (Linux/Apache/MySQL/PHP) platform to implement a virtual router with several web servers. The Linux Virtual Server (LVS) is used to load balance incoming requests on commodity servers or as an internal load balancer to distribute MediaWiki requests. The proxy web servers that handle web requests uses the Apache HTTP server. The main web application is written using MySQL, PHP, and JavaScript to query data stored in MariaDB.

For this project, I built a Wikimedia-type server using an "application server herd", where multiple application servers communicate directly with one another via the core databases and caches. The server utilized the Python's `asyncio` library to receive location requests from the client, then it proceeds to processing the request by querying from the Google Places API for GPS locations given coordinates and radius. While our interserver communications were designed for rapidly-evolving data as the server is expected to constantly receive requests from the client, the database server using Google Places API required less transactional semantics as it is used for more stable-data and the need to access the database is far much less often.

Using Python's `asyncio` asynchronous networking library, I was able to replace part of the Wikimedia platform and implemented the semantics of our servers using a simpler solution. As `asyncio` is event-driven, it allows us to process and quickly forward updates from one server to the next using a simple network flooding algorithm.

This paper will delve into the strengths and weaknesses of the `asyncio` library, assess problems regarding type checking, memory management, and multithreading of Python, and further discuss our implementation of the parallelizable proxy for the Google Places API.

2. `asyncio` Python library

The library is utilized to write concurrent code and is a foundation for multiple Python asynchronous frameworks that is commonly for high-performance network and database connection libraries. The library provides a high-level API to run Python coroutines concurrently, perform network IO and IPC, synchronize concurrent code, etc. Additionally, it also provides a low-level API where developers can create and manage event loops for networking, controlling subprocesses, and handling OS signals. The library relies on cooperative multitasking, where tasks voluntarily gives up the CPU for other tasks.

Asynchronous programs allow us to improve the performance of our sequential programs. It allows us to parallelize tasks to prevent any potential bottlenecks if smaller tasks having to wait idly when a large task is run in the beginning. In our server implementation, the I/O process is extremely time-intensive. By utilizing asynchronous operations, we are able to accomplish other tasks such as processing information from previous messages and propagating to the next server without having to wait for the I/O operation to complete.

Using `async` and `await` in our program, we can specify whether a function is a coroutine and how it should behave in response to other coroutines. An object is an awaitable object if it can be used in an await expression with three main types of awaitable objects, including coroutines, Tasks, and Futures. A coroutine is able to stop executing to allow another coroutine to run with this syntax. Awaitables stops the other coroutine from operating until it finishes executive. Lastly the `asyncio` library allows programs to establish reliable TCP connections to set up a network between clients and networks easily.

2.1. Strengths

This library allows me to implement the proxy herd server very efficiently from its detailed documentation and available server and client predefined functions in its High-level API to establish my TCP connections and accomplish I/O operations between streams. As `asyncio` focuses on coroutines and asynchronous operations, it is very suitable to use for this project.

With the available async/await syntax, I can specify functions as coroutines that can voluntarily halt execution for another task. Since this is similar to multithreading and parallel programming, our server implementation was able to prevent certain bottlenecks such as time-intensive I/O operations that many other tasks would have to wait upon.

Most importantly, the asynchronous implementation of the proxy herd server solved the main bottleneck of the traditional Wikimedia architecture where it relies on the main application server. By not having rely on a central server to distribute information received from the client, each server can operate independently and propagate the message themselves to neighboring servers using our flooding algorithm more efficiently.

With its excellent support for TCP connections and simple API for asynchronous operations, `asyncio` is a perfect library suitable for our implementation of the application server herd.

2.2. Weaknesses

The two main weaknesses of using `asyncio` is the incompatibility with HTTP requests and filesystem I/O operations and the potential introduction of race conditions that can be introduced in asynchronous programs.

`asyncio` only supports TCP and SSL protocols, which posed a problem as we have to send HTTP GET requests to query information from the Google Places API. Furthermore, filesystem I/O operations are blocked since most operating systems do not provide support for it. Due to such incompatibilities, we were required to utilize other libraries like `aiohttp` and interact with the system directly to accomplish the desired behavior for our servers.

Another weakness is within the nature of asynchronous programming itself. Different from parallelism which uses multiple cores and CPUs to perform multithreading, concurrency does not actually execute multiple processes at the same time. Since Python only supports concurrent code and not parallel code, race conditions can occur when trying to update referenced memory. Lastly, a problem could be introduced in our programs where our tasks is completed out of order. This can caused a server to access stale data if it has not received the new message from another server that communicated with the client previously.

2.4. Comparison with Node.js

Node.js is an asynchronous event-driven JavaScript runtime that is also highly scalable. Traditional concurrency model uses OS threads which is usually inefficient and difficult to use. Since there are no locks and no function in Node.js directly performs I/O, the process is never blocked. This would solve our previous problem of having blocked processes like filesystem I/O when used with `asyncio`.

Similar to `asyncio`, Node.js also runs on the single thread, so it is concurrent and not parallel. However, Node.js does provide support for users to take advantages of multi-core operations in its API. It is different in that Node.js utilizes callbacks in an event loop, and exits when there are no more callbacks to perform. On the other hand, asyncio utilizes coroutines in an event loop that would voluntarily stop execution for another task. Thus, both frameworks are event-driven.

It is important to note that Node.js support HTTP requests, and it is a "first-class citizen in Node.js". This solves the other weakness found in `asyncio` with its lack of compatibility with HTTP, so performance would also increase since Node.js provides support for HTTP operations with lower latency.

Since Node.js features and compatibility would allow us to solve both weaknesses found in `asyncio` and offer a similar proposition with its support for concurrent programming, Node.js might be a more preferable solution to implement our server herd application.

2.4. Python 3.9+ features

With the introduction of `asyncio.run` and `python -m asyncio` in Python 3.9+, implementation of our program becomes more convenient in setting up the server. Using older versions of Python that does not provide these features can make our implementation more complicated as we would have to rely on the `asyncio` library's Low-level API to manually set up tasks and run event loops.

3. Implementation

Our proxy herd network consists of fiver servers with ID's: Riley, Jaquez, Juzang, Campbell, and Bernard. They communicate bidirectionally with each other with the specified patterns where Riley talks with Jaquez and Juzang, Bernard talks with everyone else but Riley, and Juzang talks with Campbell. To emulate mobile devices with IP addresses and DNS names, the servers allow TCP connections with client servers to process requests and send responses.

With each request received from the client, a server must propagate the information to the rest of the network using a simple flooding algorithm and constrained by the relationships between servers stated above. The servers are able to receive two types of requests, `IAMAT` and `WHATSAT`, from the client. The `IAMAT` request contains information about the client including the client ID, location coordinates in longitude and latitude, and the time the client sent the request. The `WHATSAT` request asks the server to query nearby places information from the database using the Google Places API providing a radius and an upper bound of information to receive.

Using AT messages, servers are able to propagate information received from the client to other servers in the network and provide a response to the client's request. Each AT messages consist information regarding the server that the client communicated with, the time difference of when the server is thought to receive the message from the client, and a copy of the information given from the client. In between servers, these AT messages contain additional information such as the sender server ID and a count to implement our flooding algorithm.

Each server operates independently of the rest of the servers in the network, meaning a dropped connection from one server does not affect other running servers. To query the Google Places API, we sent an HTTP request to the Google servers using the aiohttp library to perform a GET operation.

Invalid messages are not propagated and returned to the client with a '?' prepended to the original message. Lastly, the servers log all of their input and output to a log file by redirecting the server's stdout streams to a file created when the server is first created.

3.1. IAMAT requests and AT responses

The IAMAT request consists of 3 operands: the client ID as a string of any non-whitespace characters, the latitude and longitude in the ISO 6709 notation, and the client sent time in POSIX time, consisting of seconds and nanoseconds since 1970-01-01 00:00:00 UTC.

To process client's information and provide a response, I implemented a global dictionary named CLIENTS with Client ID as key and a dictionary with fields 'server', 'coords', 'time', 'diff' to store and update the information of each client as requests are received. I computed the time difference between the server's idea of when it received the message and the client's original time using the getTimeDiff function to convert the string messages to floats for math operations.

Once the request is processed and the AT message is formed, the server sent the response back to the client while also propagating it to its neighboring servers by appending its own server ID as the sender name to the message and a count of 2. To implement our flooding algorithm, this count is decremented with each subsequent server that receives the message, and the message stops propagating among servers once the count hits 0. Since our proxy herd network only consists of 5 servers, having a count as 2 is sufficient to propagate the information to the rest of the network.

Noted that the AT messages sent to the client are different from those sent to neighboring servers, they are only processed by servers if the sender and count is available as operands since client cannot send AT messages to servers. After comparing the client's time in the message and the time previously stored in the server's own CLIENTS dictionary, the information is only updated if the new time is greater than the old time to correctly keep track of the information that the client most recently talked to. Once the message is processed and the count is non-zero, this packet is once again propagated to the server's neighboring servers using the same mechanism stated above after decrementing the count by 1.

3.2. WHATSAT requests and AT responses

With every WHATSAT request received, the server first checked if information about that client is available in its global CLIENTS dictionary, then it queries the Google Places API if data is available to form the AT response by opening an aiohttp.ClientSession(). The AT response contains the information as stored in the dictionary in addition to a json object of the results received from the database query.

4. Problems Encountered

The only major problem that I encountered when implementing this solution is logging the server's input and output to the logging file. In particular, the synchronous operations using the asyncio library does not support the standard Python file I/O or file operations. This is due to the fact that most operating systems do not support asynchronous file operations. This means that read and write to files are blocked within asynchronous operations.

To provide a workaround for this problem, I simply redirected the standard output stdout stream of my server to a file by setting sys.stdout to a new file opened with write permissions. By doing this, all normal print statements occurred within the program is then directed to a log file as desired.

Downsides of this solution includes that the log file is opened or created if not exists in the starting up of the server, but logs are not registered to the file until the server connection is terminated. This means that live logging is not happening while the server is up and running. Another downside in my program is that the log file is completely overwritten when the connection to the server is dropped.

Another workaround that I considered using is utilizing the os.system() operation to send commands to the Bash kernel for writing operations. The Linux kernel also provides asynchronous operations on the filesystem (aio), but it requires a different library that doesn't scale with many concurrent operations. Thus, because I wanted to utilize only Python standard libraries to accomplish this task, so I went with my first approach.

5. Comparing Java and Python

Python is a strongly but dynamically typed language, and type checks occur at runtime. On the other hand, Java is a static typed language, and type checking occurs at compile time instead of runtime. This makes Python programs much simpler to write as we do not have to specify type annotations when declaring variables and functions. However, this would introduce runtime errors for incompatible types when the program is run, so Python programs are less safe compared to Java, which is more reliable for its static type checking.

Secondly, Python implements memory management through reference counting where it checks the amount of times that an object in memory is referenced. This allows for a quick garbage collection process but might also introduces memory leaks if links form cycles and object's reference counts do not reach zero. Java uses a mark-sweep algorithm that allows for more efficient garbage collection to keep track of claimed and unclaimed memory using pointers.

Considering multithreading support, Java would allow for a safer implementation of concurrent programs since it supports multithreading and parallel code with its primitive `Thread` class and the Java Memory Model that defines the behavior of Java multithreaded application. Conversely, Python does not support multithreading and relies on reference counts, which can potentially introduce race conditions and inaccurate memory access.

6. Conclusion

The `asyncio` library has allowed us to implement our application herd server efficiently through its simple API and support for TCP connections. However, after assessing the strengths and weaknesses of the Python library in comparison with the Node.js library, using Node.js and Java might be a more suitable solution to our problem to create a faster, safer, and more reliable program by taking into accounts the program language as well.

References

Asyncio Documentation

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

List of third-party libraries for AsyncIO

https://github.com/python/asyncio/wiki/ThirdParty#filesystem

Node.js Documentation

https://nodejs.org/en/about/