Asyncio proxy server herd in Python 3.9

Prathyush Sivakumar, *CS 131 Student*

Abstract

In this project I was required to implement a proxy server herd in Python 3.9 using the asyncio package. The project also involves issuing asynchronous http get requests to the Google Places API for a nearby search. The get request yields a certain number of items within a specific radius specified in the request.

Clients can connect to any of five servers named after the most recent UCLA basketball starting team. Commands that are accepted are:

- IAMAT: Client communicates its coordinates and timestamp to one of the servers.
- WHATSAT: Client asks whereabouts and nearby places with respect to another client already in the server's database.
- AT: Server responds to client requests and also flood the server network with the AT message so that every server knows where each client is even though it wasn't directly contacted.

1. Introduction

Wikipedia and related websites are based on the Wikimedia server platform, which is based on Debian GNU/Linux, the Apache web server behind an NGINX-based TLS termination proxy, the Memcached distributed memory object cache, the MariaDB relational database, the Elasticsearch search engine, the Swift distributed object store, and core application code written in PHP+JavaScript, all using multiple, redundant web servers behind the Linux Virtual Server load-balancing software, with two levels of caching proxy servers (Varnish and Apache Traffic Server) for reliability and performance.

However, such an infrastructure would result in bottlenecks for applications where clients are mobile, updates to databases happen more frequently and different protocols are used to request information.

Proxy server herds can help alleviate this bottleneck by allowing asynchronous communication between different servers. This makes it easier to retrieve information since the database doesn't need to be contacted for every request. Every server has information that any server receives and so doesn't need to communicate with the database directly. This provides a considerable boost in performance since network requests are generally slow. Moreover, the servers communicate with each other asynchronously so it can receive other requests while it is communicating.

2. Asyncio Source Code and Documentation

2.1 Event Loop

The event loop is an important feature in asyncio. It is responsible for receiving tasks and scheduling them to be run when ready. In the source code, the event loop is implemented as a multiplexer. This is the part that is responsible for notifying about I/O events. The event loop proper wraps a multiplexer with functionality for scheduling callbacks immediately or at a time in the future.

In my application, I initiate the server with asyncio.run(coro, *, debug=False) function. This function runs the passed coroutine and generates and manages the asyncio event loop. This is the main entry point for the program. Additionally, another event loop may be created using the asyncio.get_event_loop() if there is no current event loop set in the current OS thread or using asyncio.new_event_loop() to create a new event loop.

2.2 Coroutines and Tasks

Coroutines with async/await syntax is the correct way to write asyncio applications. To run the coroutine, one cannot simply call it. It has to be scheduled with asyncio.run(). asyncio.create_task(coro, *, name=None) wraps the coroutine coro into a Task and schedules its execution.

2.3 Awaitables

An awaitable is anything that can be used in an await expression. Awaitables are very commonly used in async functions to wait for some request to complete. Awaitable objects in asyncio include Coroutines, Tasks and Futures.

2.4 Applying to server herd

The APIs that Python's asyncio package provides make it relatively easy to implement a server herd. Using the asyncio.run() function to schedule a coroutine which in turn uses Server.serve_forever() to start accepting connection until it is cancelled. Cancellation of serve_forever() causes the server to be closed. The code snippet to implement a proxy herd would be something like this:

```
class Server:
        async def run(self):
                server = await asyncio.start_server(coro)
                async with server:
                        // Keeps executing coro as it
                        // receives requests
                        await server.serve_forever()
        async def coro(self, reader, writer):
                // Handle message
def main():
        server = Server()
        asyncio.run(server.run()) //creates event loop
```

The event loop is created with asyncio.run() and then server.run() uses serve_forever to keep scheduling tasks that are added to the event loop. Coro is a coroutine that handles messages and communicates with other servers in the herd as and when it updates its databases from a request.

2.5 Flooding protocol

A brief overview of the commands used in this program are specified in the introduction section. Servers are required to flood "AT" messages to every other server in the network. To implement this I use dictionaries to keep track of client's names, their coordinates and timestamps. When one server receives an IAMAT request from the client, it stores the information from the command in the dictionaries and then propagates the message to all of its neighbors. The neighbors in turn propagate that message to their neighbors and so on. To ensure that flooding doesn't go on infinitely, I propagate a message to the server's neighbors only if the server hasn't received the message yet. If it had already received the message, then information about the client would already be in its dictionaries. If the client is not present in the database, the server stores the information from the AT message and then sends it to its neighbors. If it already has information about the client, it doesn't send the message to its neighbors.

3. Ease of Use

The APIs provided by the asyncio package make it easy to implement a proxy server herd. One doesn't need to worry about threads to manage requests and the host of issues that arise due to that. Asynchronous execution is beneficial because the asyncio package's function do all of the scheduling for you so you only need to worry about how to implement the program's functionality. By offering a simple single entry point using asyncio.run() and providing coroutines and methods that allow to schedule tasks as and when they are received without having to manually schedule them is a great advantage.

4. Performance implications of asyncio

Implementing a proxy server herd using asyncio yields good performance benefits. Since time intensive processes like get requests and inter-server communication need to be processed, the event loop takes care of this in the background while running other tasks it receives. Ultimately, asyncio provides concurrency and not multi-threading. If the task is parallelizable, then it would make more sense to use a package that allows for multi-threading.

5. Reliance on older versions of Python

We cannot get by with older versions of Python since asyncio package was added to Python in version 3.4. Consequently, we wouldn't have convenient APIs like asyncio.run() and Server.serve_forever() to help with asynchronously responding and scheduling tasks. Hence, later versions of Python are required to implement the server proxy herd.

6. Analysis

6.1 Type Checking

Python is dynamically typed whereas Java is statically typed. The Python interpreter checks types at runtime whereas Java's compiler statically checks that there are no type violations in the code. A popular principle in Python is "duck-typing" where the program attempts to perform something according to how the object should behave and if not, handles it appropriately. In a large program, type errors in Python can be very hard to resolve since it is hard to track where exactly the code breaks. In large applications, a lot of duck typing is required and can quickly overwhelm the coder. Java is more reliable in this sense that it ensures that the program's types are all compatible. Basically, the machine does most of the work in Java but you have to do most of the work in Python. Java also offers better performance since type checking is done statically. To conclude, Java's type checking is more beneficial for large applications.

6.2 Memory Management

CPython, the default implementation of Python is written in the C Programming language and it interprets Python bytecode. PyObject is a struct which almost every object in Python uses. It contains two values: ob_refcnt – reference count; ob_type – pointer to another type. The reference count is used for garbage collection. Whenever the program sees a reference to the object ob_type, it increments the reference count and decrements it when the pointer is deleted. So the memory manager has to be careful when two different processes try to access the same shared resource. Python solves this with a global interpreter lock (GIL). This is a global lock on the interpreter when a thread is interacting with the shared resource. This implies that it is not possible for another thread to step on the current one. This locking and unlocking incur performance overhead. Garbage collection is performed when an object's reference count is 0. The interpreter must keep track of every object's reference count and clean up the memory if needed.

Java also keeps track of reference counts and frees up memory once an object is no longer needed. Java provides ways to give hints using System.gc() or Runtime.gc() to the garbage collector to optimize efficiency. One can also set flags on the JVM. They can be used to adjust which garbage collector to be used, ex. Serial, G1, etc., the initial and maximum size of the heap sections (e.g. Young Generation and Old Generation) and more.

6.3 Multithreading

Due to the global interpreter lock mentioned before, resources cannot be shared by more than one process at a time so multithreading is not possible. The lock only allows one thread to modify state at a time.

Java on the other hand, has the notion of multithreading built into its heart. It is a central concept in Java and there are ample resources and APIs to implement multithreaded programs. For a large program that is parallelizable, it would make sense to go with Java rather than Python.

7. Asyncio vs. Node.js

Node.js is an asynchronous event-driven JavaScript runtime. Similar to asyncio, it uses the event loop as a runtime construct. Like asyncio, Node.js uses asynchronous I/O primitives that prevent JavaScript code from blocking. For this application, when the program is performing a network request, Node.js could run some other tasks while keeping this in the background. This is precisely what asyncio does too. Both asyncio and Node.js provide concurrency to the code and not parallelism. If the task being performed is CPU intensive, then neither Node.js nor asyncio would be a good fit for the task.

8. Conclusion

In this report, we have explored the advantages and disadvantages of using asyncio to implement a proxy server herd. We have talked about the main concepts in asyncio and main APIs. Asyncio is definitely a viable method of performing asynchronous tasks efficiently, especially I/O intensive tasks like network communication. On comparison with Node.js, we observe some similarities but Node.js has been around for longer and hence has more support for such applications. We also analyzed the memory management, type checking and multithreading features in Python and Java and concluded that Java could be more helpful for large-scale applications especially when the tasks are parallelizable.

References

[1] Python documentation. *Asyncio – Asynchronous I/O*. https://docs.python.org/3/library/asyncio.html

[2] Olivia Smith. *Type checking in Python*. https://docs.python.org/3/library/asyncio.html

[3] Alexander Vantol. *Memory management in Python*. https://realpython.com/python-memory-management/

[4] Alexandra Altvater. *What is Java Garbage Collection? How it Works, Best Practices, Tutorials, and More*. https://stackify.com/what-is-java-garbage-collection/

[5] OpenJS Foundation. Node.js documentation. https://nodejs.dev/learn/introduction-to-nodejs