CS131 Project Report: Proxy Herd with asyncio

Belle Lerdworatawee
Discussion 1C

Abstract

There exist numerous frameworks for implementing server-client platforms. This paper presents one such framework, asyncio, as a candidate for building the new Wikimedia-style application. The asyncio library is used to prototype a herd of five servers that connect to clients and other servers via TCP. This paper also discusses the advantages and disadvantages of the Python language and asyncio with respect to Java and Node.js. In short, asyncio is well suited for the application.

1. Introduction

Wikipedia and similar websites are based on the Wikimedia server platform which relies on many different softwares. In the new Wikimedia-style service designed for news, the previous softwares are not adequate for handling the demands of the new site. The new Wikimedia-styled service must adapt to handle frequent article updates, access through multiple protocols beyond just HTTP or HTTPS, and support a larger population of mobile users. Adding new servers is difficult on the previous application, and the response time of the servers is also slow due to bottlenecks in the core database of the Wikimedia application server.

Taking these concerns into consideration, we explored Python's asyncio module as a possible solution. At a glance, asyncio appears to fit the needs of the news application well. Python's builtin data structures can update data quickly, and asyncio provides convenient functions for an application server herd model. Asyncio is based on an event loop that runs asynchronous tasks and subprocesses, and performs network IO operations [2]. This enables it to handle multiple requests concurrently despite running on a single thread and CPU. It is expected to update news articles more efficiently than a purely synchronous runtime environment.

2. Server Herd Prototype and Design

The prototype can receive, store, and send client location and nearby places. It supports five servers—Riley, Jaquez, Juzang, Campbell, Bernard. Each server has specific servers whom they forward data to. They also receive specially formatted client messages which will be described in detail later.

Upon receiving a request, the message is first parsed for validity. If valid, the coroutine that handles that specific command updates or adds the client's data to the server's personal database, and responds accordingly. Each server has their own database.

Intuitively, an asynchronous framework is a good option to handle high message loads and avoid network congestion. We will further explore this idea in the following sections.

2.1 IAMAT Messages

IAMAT messages indicate to the server where a client is currently located. Clients send IAMAT commands in the following format:

```
IAMAT <client> <location>
<timestamp>
```

where *client* is the ID of the client sending the request, *location* is the coordinates of the client in ISO 6709 format, and *timestamp* is the time (in seconds and nanoseconds) since 1970-01-01 00:00:00 UTC from when the request was sent. Upon receiving this command, the server checks if it presents new coordinates or timestamps. If not, the server ignores it. Otherwise, the server updates or adds to its client database and sends an AT message to the same client. The AT command is formatted as follows:

```
AT <server> <time-diff> <client>
<location> <timestamp>
```

where *server* is the name of the server who received the command, *time-diff* is the difference between the time the message was received and the time that it was sent, and the same parameters as earlier.

2.2 WHATSAT Messages

Clients can request information about places near other clients through WHATSAT messages. Clients send WHATSAT commands to servers in the following way:

WHATSAT <client> <radius> <max-results>

where *client* is the ID of the requested client, *radius* is the radius (in km) around the requested client from which places are returned, and *max-results* is the maximum number of results to return.

To handle these requests, the server sends an HTTP GET request to the Google Places API. Asyncio does not provide native support for HTTP; however aiohttp, another Python library, can do so asynchronously and was used in the project. The server supplies the Google Places API with the coordinates of the requested client, radius, and an API key. Google Places returns a JSON result of all places within the specified radius of the client. Afterwards, the server ensures that the number of results returned does not exceed the specified limit.

Lastly, the server replies to the client with a different AT message:

```
AT <server> <time-diff>
<client-name> <location> <time-sent>
<JSON-data>
```

where all the parameters are identical to the original AT message. It is followed by JSON data, formatted for readability. The data is not stored so servers must send a query to the Google Places API everytime a WHATSAT command is issued.

2.3 Server-to-Server Messages

Upon receiving new client information, servers propagate the information to other servers. They do so using a specially formatted AT message:

```
AT <server> <time-diff> <client>
<location> <timestamp>
<server-names>
```

where the first five parameters are identical to the original AT message. Appended to this string is the server's name. If a server receives an AT message with their name appended to it, that means this message is redundant and they ignore it. Otherwise, they update their database and forward the message to other servers. If a server is not available, a message is logged and execution continues. Old messages are not resent to servers that were down.

2.4 Invalid Messages

All other commands are considered invalid. The server responds to such as follows:

? <message>

where *message* is the message that it received. The server checks messages for basic syntax like if the location is in ISO 6709 format, if numbers were provided in the numerical parameters, if the length of the string is correct, etc. In cases where the format of the message is correct but the parameters are incorrect, the server responds with the format above.

2.5 Difficulties Encountered

Some difficulties I encountered while building the prototype was figuring out how to use the JSON library. I was confused on how to handle the requested JSON result, but I overcame this by reading code examples in online documentation and print debugging. Initially, I also had difficulty grasping the concept of event loops and was unsure which functions to await. With the help of the hint code supplied by the TAs and discussion slides, I figured out when to use `await` and `async def`.

3. Asyncio

Asyncio is a Python library that is commonly used to write concurrent code. It provides cooperative, multitasking functions in which tasks can voluntarily yield to let other tasks run. The two main keywords are summarized below:

async — labels a function as a coroutine which can suspend execution and yield to another coroutine
await — suspends execution of the current coroutine until the awaited task is completed

As mentioned earlier, the event loop is fundamental to asyncio as it schedules and runs asynchronous tasks. In brief, Python implements the event loops with generators, which allow a function to be partially executed, and halted at a specific point to yield to another task before resuming again [1]. This non-blocking feature makes it an excellent choice to solve I/O network performance, as operations can be performed while waiting for input.

3.1 Advantages

Asyncio provides a variety of useful features like coroutines and awaitables. While working with asyncio, I found it relatively easy to use as the module is well documented. Compared to previous TCP projects I've done, it provided a more straightforward way to write server-client programs. To start a socket server, I simply passed an IP address, port number, and coroutine function to the function `start_server`. Other languages and frameworks required more complicated means to accomplish the same task.

Even though Python isn't traditionally asynchronous, asyncio abstracted away the low level details and provided intuitive methods. Since coroutines are scheduled concurrently, servers don't waste time blocking and waiting for input before completing other tasks. Thus server databases are kept up to date, which fulfills one of the requirements by the Wikimedia-styled service.

Starting a server is easy and fast via the command:

```
python3 server.py <server>
```

where server.py is the source code and *server* is the name of the server that we wish to run. So adding new servers is convenient and makes the project scalable.

Finally, since Python is a popular language, many programmers are already familiar with it. And for those with less experience, its syntax is relatively easy to learn. Python also enjoys extensive community support so the available documentation on both the language and asyncio is readily available, which makes it more convenient to debug. The builtin dictionary data structure also provides an efficient way to store, retrieve, or update client data.

Performance testing was limited by the number of tests I could run and no competing prototype to use as a benchmark. While I lack concrete performance metrics, we can trust that asyncio's asynchronous runtime environment handles requests better than its synchronous counterparts.

3.2 Disadvantages

One disadvantage of the asyncio module is that it lacks HTTP functionality. However this is not a huge setback as Python has other modules that support HTTP protocols. In this project, we utilized aiohttp since it is also asynchronous and lets us easily send HTTP GET requests to the Google Places API. Although there is good documentation for aiohttp as well, it still requires extra time and effort to figure out how to integrate it as opposed to if asyncio had built in abstractions like it did with TCP.

Moreover, asyncio is single-threaded and has limitations. Asyncio improves performance by scheduling tasks concurrently and keeping the CPU busy at all times, but the program is still ultimately restricted by the CPU. When a server has too many tasks to process, a bottleneck inevitably occurs. Since Python lacks support for multithreaded programs, the servers' performance may lag significantly as servers become inundated with requests. Other programming languages that support parallelism can handle this better. With that in mind, this is only a concern if the network becomes flooded with incoming messages and servers become backlogged.

Lastly, asyncio (and all other synchronous methods) doesn't necessarily execute steps in the order that the programmer expects and it is a cooperative multitasking library. There is a possibility that a task chooses not to yield and starves other tasks of the CPU. A preemptive multitask solution could be taken, but that has other downsides such as figuring out how to save and return to the original point of execution correctly. As for the first point, with more servers in the herd, IAMAT commands might come in before WHATSAT commands but be scheduled after. Thus, the WHATSAT command would incorrectly result in an invalid command.

4. Python vs Java

This section assesses Python's type checking, memory management, multi-threading approach with respect to Java.

4.1 Type Checking

Python is a dynamically typed language, meaning that the Python interpreter type checks code as it is run. Additionally, Python variables can change types throughout execution. An upside of dynamic typing is that the language is more readable and concise as all declarations are shorter. Due to the flexibility of variable types, variables and objects can be passed between different functions. This made using the asyncio library easier to use as I didn't need to focus on the parameters of asyncio functions.

In contrast, Java is a statically typed language, meaning that the code is type checked during compilation. Variables are more strictly typed and thus can change during execution. The benefit of this is that errors are caught early on, requiring less debugging later. For larger, more complex programs, Java would most likely be a safer choice to avoid runtime errors. Additionally, it may also be helpful in large applications to have types explicitly declared to avoid confusion [3].

4.2 Memory Management

Both Python and Java use Garbage Collectors, however, they implement it very differently.

Python makes use of reference counts. Each object keeps track of how many places reference it. When the reference count decreases to 0, the object's memory is automatically freed [4]. This garbage collection method is efficient and easy to implement. However, cyclic references pose an issue as their reference count never reaches 0 and they will never be deleted. However, the prototype doesn't make use of circular references so Python is still a valid candidate with respect to memory management.

Java, on the other hand, has a garbage collector that runs periodically. It performs the Mark and Sweep algorithm to find unused objects. The algorithm starts at the roots of an object tree and makes note of all reachable objects as it traverses. These objects are kept, and the rest of the heap memory that is unreachable is freed [5]. Java's advantage over Python is that it is effective against cyclic references. However, this garbage collection method takes a longer time to run and is more complicated to implement.

4.3 Multithreading

Python does not allow parallelism as the Global Interpreter Lock (GIL) guarantees that only one thread at a time can hold the lock and execute tasks. Although this compromises performance, Python's garbage collection method requires it to be this way. Multithreading introduces the risk of memory leaks as updating the reference count can lead to data races and objects might be deleted when they shouldn't have been or vice-versa.

In contrast to Java's Memory Model, multithreading is supported. This feature does, however, increase the chance of race conditions, and relies on more complicated synchronization methods and locks to avoid data races. Additionally, a higher knowledge of the Java Memory Model is required and naturally, there's a steeper learning curve at the cost of better performance. For large applications where tasks can be executed in parallel, this provides a significant benefit over Python. But for smaller applications, Python makes up for the lack of parallelism by providing asynchronous programming to keep the

CPU busy at all times. For the purposes of this project, a single-threaded program is enough to accomplish the server-client tasks since they don't require intensive computations.

5. Asyncio vs Node.js

Both asyncio and node.js are popular frameworks for implementing web servers. At a high level, both Python and Javascript are dynamically typed languages and share the benefits of readability and type flexibility. Additionally, these two languages have lots of community support and documentation. As a result, their libraries are well maintained and users don't need to implement many functions from scratch. Both support single-threaded designs as well. Event loops are the driving concept behind asyncio and node.js, and their asynchronous natures are similar. Node.js has the concept of callbacks which are similar to coroutines in Python [6]. async and await exist in both libraries and more or less, function the same way.

While they share many features, one key difference between asyncio and node.js is that node.js frameworks are inherently asynchronous [7]. Any methods defined will execute asynchronously unless otherwise specified. In contrast, asyncio methods are synchronous unless specified as coroutines (with "async def"). So there are implementation differences in that asyncio requires the programmer to explicitly specify where the event loop occurs and what functions are awaited.

Without explicit testing, it is difficult to determine if node.js is more suitable than asyncio. Both serve the same purposes and provide similar functionalities. There may be differences in performance, but without a comparison of programs, it is difficult to accurately conclude which performs better.

6. Conclusion

In summary, this project explored Python as a programming language and asyncio as a framework for building an application server herd. The prototype consisted of five different servers who communicated with each other and with clients via TCP connections. Overall, Python and the asyncio module are well suited for the Wikimedia-styled news platform as we reap the performance benefits of asynchronous programming while taking advantage of the simplicity and widespread support of the Python language. This, in addition to the existing aiohttp and json modules, allows developers to focus purely on building the network. While Python is lacking in areas such as garbage collection and multithreading as compared to Java, these features are not essential to implementing a server herd application at our desired scale. Therefore, I would recommend using Python and the asyncio library to implement the new Wikimedia-style platform.

7. References

[1] Mark McDonnell. Guide to Concurrency in Python with Asyncio. https://www.integralist.co.u/posts/python-asyncio/#why-focus-on-asyncio, 2019. Accessed: June 3, 2021.

[2] Python 3.9.5 documentation. Event Loop. https://docs.python.org/3/library/asyncio-eventloop.html. Accessed: June 3, 2021.

[3] Oracle. Dynamic typing vs. static typing. https://docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html, 2015. Accessed: June 3, 2021.

[4] Pablo Galindo Salgado. Python Developer's Guide. 26. Design of CPython's Garbage Collector. https://devguide.python.org/garbage_collector/. Accessed: June 3, 2021.

[5] Baeldung. JVM Garbage Collectors. https://www.baeldung.com/jvm-garbage-collectors, 2021. Accessed: June 3, 2021.

[6] OpenJS Foundation. About Node.js®. https://nodejs.org/en/about/. Accessed: June 3, 2021.

[7] OpenJS Foundation. Modern Asynchronous JavaScript with Async and Await. https://nodejs.dev/learn/modern-asynchronous-javascript-with-async-and-await. Accessed: June 3, 2021.