Application Server Herd with Python3's *asyncio*

Abstract

When analyzing Python 3's *asyncio* library to implement an application server herd, Python's type-checking, memory management, and multithreading are examined under the libraries offered by *asyncio*, alongside a comparison of the same implementation in Java. The server herd serves as a proxy for the Google Places API, using *asyncio* features such as the Streams API for network I/O to handle lower-level asynchronous event loop interactions. It is recommended that lightweight applications use Python for its ease-of-learning and readability, while larger, more intense applications should use Java for its multithreading and higher memory consuming yet faster garbage collection.

1. Introduction

Wikipedia and its related sites use the Wikimedia server platform, which is based on the GNU/Linux, Apache, MariaDB, and PHP+Javascript stack, using multiple redundant web servers, load-balancing virtual routing, and cached proxy servers. While this works well for Wikipedia, a service that attempts to use this architecture which has more frequent writes, various protocols, and mobile clients will face difficulties in the PHP+Javascript portion of the stack when adding new, potentially mobile servers to the network.[1]

This report looks into implementing an **application server herd**, an architecture in which multiple application servers communicate directly with each other as well as the centralized database and caches. A benefit to this architecture is the ability to query quickly-expiring data from nearby servers, bypassing the effect of increasing the load on the centralized database. The database can still be used for stable data that is less-often accessed or requires more complex query language.

Python 3's *asyncio* library is used for implementing this architecture, as *asyncio*'s event-driven nature allows for updates to be processed and forwarded rapidly to other servers across the herd. Additionally, *asyncio* allows these requests to be handled asynchronously, "pausing" and giving control to other functions while a response is awaiting. Through implementing a proxy of the Google Places API using a small five-server herd, a sufficient analysis of *asyncio* can be derived. Specific details, such as writability of programs, type-checking safety, memory management, and multithreading capabilities are examined in this report, with respect to using *asyncio* for large, parallelizable applications.

2. Asyncio Details

The *asyncio* library is widely used as the networking infrastructure for several Python asynchronous frameworks which provide high-performance networking and web-servers, database connection libraries, and distributed task queues. It is a library to write concurrent code using the async/await syntax, and is often a perfect fit for IO-bound and high-level structured network code. *Asyncio* includes a set of high-level APIs to run Python coroutines, perform network IO and IPC, and synchronize concurrent code. There are also low-level APIs for creating and managing event loops, implementing custom network-layer protocols using transports, and bridging callback-based libraries with async/await syntax.

The event loop is the core concept behind every asyncio application; the event loop is a single loop which runs synchronous tasks and callbacks, is responsible for network IO operations, and runs subprocesses to dispatch these and other functionalities. These tasks are executed on the event loop until it encounters an operation which needs to "await" for its completion, to which it then gives control back to the event loop to schedule other coroutines to run while awaiting the initial result. Coroutines are processes scheduled on the event loop that allow for the pausing and resuming of execution while other asynchronous processes await and execute.[2]

Python 3.9 introduced small improvements from 3.8 to the *asyncio* library which make it easier for implementing asynchronous programs, although these mainly focus on dictionary operations, string methods, and a new parser with better performance, all of which are not necessary for implementing large-scale parallelizable applications.[3]

2.1 Implementation in Server Herd

To implement *asyncio* in the server herd, I used the Streams API in *asyncio*. Streams are high-level primitives which do not use callbacks or low-level protocols or transports, allowing us to focus on pure implementation details. The two main functions used in the API are open_connection, used to open a socket at a specified port, and start_server, used to serve incoming requests through a callback function. The open_connection function is called anytime a TCP packet is sent between the servers, which is handled by start_server's callback. These two functions handle all of the application's transport functionality.

One drawback of the Streams API is that it does not include callbacks for when a connection is lost between servers, thus enforcing liveness to be checked when a packet is sent out through open_connection. When flooding an update between

the servers, an open_connection had to be made for each instance of propagation on the network.[4]

3. Server Herd Implementation

The application server herd consists of five servers: Hill, Jaquez, Smith, Singleton, and Campbell, each at their own TCP port. The herd communicates with each other bi-directionally, and accepts connections from clients using start_server. Specifically, Hill communicates with Jaquez and Smith, Smith communicates with Hill, Singleton, and Campbell, Campbell communicates with Smith and Singleton, Jaquez communicates with Hill and Singleton, and Singleton communicates with Jaquez, Smith, and Campbell.

Clients are able to send two types of commands: IAMAT, which updates the server's cache about the client's geographic information, and WHATSAT, which uses the location information stored in the cache of the client's location to query the Google Places API and return the information to the client in the same JSON format from the API. In each case, the server responds with an AT message, which includes the ID of the server which received the message from the client and the time difference between when the server received the message and when the client created it, along with the information inside the client's IAMAT information: the location, POSIX time, and ID of the client.

3.1 Client Request Callback

A callback function is implemented to handle IAMAT requests from the client, which calls the flooding algorithm, Google Places API query, and returns an AT message to the client. The callback uses the (reader, writer) pair returned from the Streams API for network I/O, and does not have to worry about implementing the network transport details itself. The strings that are communicated across the network are byte-encoded and decoded using the stream's built-in encode() and decode() functions.

3.2 Flooding Algorithm

As a client is able to connect to any of the servers on the network, each server must have an up-to-date copy of the client's most recent IAMAT message. This is implemented by having a server which receives an IAMAT message from a client propagate the information to all of the servers it is connected to using an AT message after storing the information in its cache. Each receiving server then checks if the newly received IAMAT data is not already stored within its cache: if not, then it proceeds to update its cache and propagate it. As we have cycles in our network graph, the check for uniqueness before propagation prevents an infinite broadcast storm of propagating servers never ceasing to propagate IAMAT data.

3.3 Google Places API Query

When a server receives a properly-formatted WHATSAT message with a client ID, radius of the client to search around, and maximum number of items to return, the server queries the Google Places API using an API key and returns the corresponding information in a JSON to the client. The server uses the *aiohttp* library to create an HTTP GET request and formats the API call to a URL endpoint on Google's servers, encoding the query data on the URL itself by appending a "?" to the end of the API endpoint to start the encoding of the query. Key-value pairs of the query are then encoded using key=value notation, and the "&" symbol is used for appending additional query parameters such as the API key itself, e.g. key1=value1&key2=value2.

4. Python vs. Java Implementation

An alternative to the python implementation explored in this report is one using Java, with its own native asynchronous and network functionality, and details specific to the JVM and language compilation and syntax.

4.1 Type Checking

Type checking is one of the main differences between the languages, as Python is dynamically type-checked, whereas Java is statically type-checked. As dynamic type-checking happens at run-time, errors may appear that are not readily apparent at compile time, while in a statically-typed language, successful compilation usually entails that a significant degree of errors have been cleared. One benefit to dynamically type-checked code is that it is usually much more readable for simple applications, but can quickly become less readable than statically type-checked languages as variables become numerous and functionalities and types unclear.

4.2 Memory Management

Both Python and Java provide a built-in garbage collector, although they are implemented very differently. Python uses the reference count method to handle object lifetime, so an object that has no more use will be immediately destroyed. However, in Java, the garbage collector marks and destroys objects which are no longer used at a specific time rather than instantly.

One drawback of reference counting is circular references: if A references B which references C which references A, if A were to drop its reference to B, both B and C will have a reference count of 1 and won't be deleted with a traditional reference counting garbage collector. On the other hand, this "eager" method of garbage collection usually uses minimal memory, as it does not need to bring execution back to the garbage collector when objects need to be dereferenced.[5]

Java's implementation of its garbage collector, through marking and sweeping, uses much more memory, and as a result, may become clunky on older systems or those running without much memory available. However, this implementation optimizes for speed when it has enough memory, as objects do not need to be re-created though persistency. When memory is scarce on the device, for example on lightweight systems, Python's implementation is better in terms of garbage collection.

4.3 Multithreading

Java supports multithreading natively through the JVM, allowing programs to run on multiple cores in parallel. This multithreading feature is best used on I/O applications, as context-switching will occur frequently when using a high number of threads in an application.

The popular implementation of Python, CPython, is not thread-safe and therefore does not allow for multithreading in execution. This is due to the fact that the interpreter doesn't support fine-grained locking mechanisms like the JVM does.[6]

This is a significant tradeoff, to the point that it merits using Java to implement any large-scale application server herds as the parallelizability of the JVM would allow for the servers to run multithreaded, a critical performance boost when used for a server's network I/O, which may handle many different requests at the same time.

4.4 Asyncio vs. Node.js

Both Python's *asyncio* and the Node.js frameworks are very similar asynchronous I/O models which use an underlying event loop to schedule the operations, and both support only a single thread.[7] The main difference between the two languages is mostly left up to programming style, as some may be more comfortable with how Node.js handles callbacks or asynchronicity, using promises as objects.

5. Implementation Difficulties

The application server herd was overall quite enjoyable to implement in Python as I am usually used to languages with a steep learning curve. The streams API encapsulated much of the information that was needed for this implementation, and allowed me to focus on issues such as debugging, which proved to be problematic at times when using Python's dynamically type-checked interpreter. Since the dynamic typing caused the interpreter to not catch issues that are normally caught at compile-time, I ran into many more trivial bugs during the development process than I would have when developing with a statically-typed language.

6. Conclusion

Based on my research, I would not recommend using Python over Java for large, parallelizable asynchronous applications. Since there is no multithreading in Python, there is a distinct lack of performance when scaling to higher network loads. Although the ease of implementation on Python is much better, from a software architecture point of view, I believe multithreading is much more valued due to the performance it can bring, although Python's ease of use and maintainability offer much to be desired from Java. However, if a program is designed to be lightweight in either usage or devices operating the servers, I would recommend Python and *asyncio* due to its shallow learning curve and ease of understanding the syntax.

7. References

1. Wikimedia Network Architecture Information: http://web.cs.ucla.edu/classes/fall20/cs131/homework.html
2. Asyncio Event Loop: https://docs.python.org/3/library/asyncio-eventloop.html#asyncio-event-loop
3. Python 3.9 Asyncio updates: https://docs.python.org/3/whatsnew/3.9.html
4. Asyncio Streams API: https://docs.python.org/3/library/asyncio-stream.html
5. Python and Java Garbage Collection: https://stackify.com/python-garbage-collection/
6. Python and Java Multithreading: https://liyanxu.blog/2019/04/19/summary-of-multithreading-and-multiprocessing-in-java-python/
7. Asyncio vs. Node.js: https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36