Analysis of Server Herd Architecture with asyncio

Kevin Zhang

104939334

Abstract

In our study of the Python programming language, one of the biggest applications in the modern technological climate is its application towards server-side development. In this project, we investigate the performance and efficiency of Python, specifically the asyncio asynchronous networking library as a means to configure a multi-node application server herd to model Wikimedia's news systems architecture that require highly mobile, efficient fast-updating functionality. In our specific case, where servers serve various protocol requests and client as mobile, Wikimedia's architecture may cause certain bottleneck problems; our proposed server herd architecture handles these requests better. We will also consider the benefits and drawbacks of using this asyncio module.

1 Introduction

In the current state of modern computing technology, there exists a variety of web stack architectures including Wikimedia's LAMP-stack architecture which features 4 main components: GNU/Linux, Apache, MySQL, and PHP. However, this, albeit popular architecture, faces a very important bottleneck that is caused by the load-balancing router. Moreover, it loses flexibility because of the difficulties of adding new servers. With a centralized bottleneck, response times become too slow to be practical. As a result, we introduce a server herd architecture that consists of a group of servers, where each server can stand-alone and serve client requests without needing to query a central database. This design avoids the main bottleneck that may paralyze the Wikipedia architecture. In this project, we implement a server herd that stores client locations across all server nodes and gives clients the flexibility to query any server node for nearby places.

2 Implementation Design

In this project, we design the application server herd using the asyncio python library and leverage the powerful coroutine and event loop mechanisms to process connections asynchronously without hitting any blocking bottlenecks from a high number of simultaneous requests.

2.1 Server Herd Structure

There are 5 main servers that are implemented: Goloman, Hands, Holiday, Wilkes, and Welsh. In our study, we restrict each of the servers to communicate with only a subset of other servers in order to model server isolation in the real environment. However, server restrictions do not prevent required information about clients to be successfully updated to all servers in the architecture. Client locations are still propagated throughout the herd. Here is the server connection grid.

Server Name	Accessible Nodes
Goloman	Hands, Wilkes, Holiday
Hands	Goloman, Wilkes
Holiday	Goloman, Welsh, Wilkes
Welsh	Holiday
Wilkes	Goloman, Hands, Holiday

Each server receives connections from clients asynchronously using TCP and the messages are processed at the server that received the query using an asynchronous event loop. The server uses asynchronous coroutines to accept and handle requests. After receiving requests, the server processes them by either making a request to the Google API or propagating information to other servers using a flooding algorithm [7].

2.2 IAMAT Requests

The first request the server handles is IAMAT requests. This is effectively a POST request from the user in a regular full-stack setting that provides the server with certain new updated information about a specific client. This information is formatted in this way:

IAMAT <Client name> <Location> <Time>

where client name is the client hostname, the location is formatted so that the first number is latitude and the second number is latitude separated by + or – denoted top/left or bottom/right hemispheres respectively following the ISO 6709 standard, and the time follows the POSIX standard.

2.3 WHATSAT Requests

The WhatsAT request is the provided functionality for the client to make the server useful that models a GET request. It provides functionality to query any server in the server herd to get the needed location information from the client. This information is formatted in this way:

WHATSAT <Client name> <Radius> <Max Number of Elements>

where client name is the client hostname, the radius is the search radius in kilometers to search around the client's stored coordinates, and the max number of elements denotes the number of places to return to the client.

2.4 FLOOD Requests

The first request the server handles is IAMAT requests. This is effectively a POST request from the user in a regular full-stack setting that provides the server with certain new updated information about a specific client. This information is formatted in this way:

FLOOD <ClientName> <Location> <Time Sent> <Time Received>

where client name is the client hostname, the location is formatted using the ISO 6709 standard, and the time stamps are used to prevent the flooding algorithm from flooding infinitely from node to node. In other words, my algorithm implements a flavor of the flood-fill algorithm where new updates are differentiated by the time stamps in which they were sent [7].

2.5 Incorrect Requests

Incorrect requests are handled right after the decoded message is processed. The message is decoded out of the StreamReader object and immediately sent into a validation method that validates the input using regex comparison in the Python package re. Any messages that does not fit the spec are returned immediately and labeled as incorrect. The server handles these incorrect messages by sending them back to the client that sent the message and prepends a '? '. The response is formatted in this way:

? <Invalid Message>

Further, a message is logged to the log file to show that the server processed an incorrectly formatted message.

3. Asyncio

3.1 Advantages of Asyncio in the Server Herd Architecture

The asyncio framework implements a Python version of a single-threaded asynchronous event loop. A server is set up with an event loop that can process multiple requests at once by interleaving requests using a construct called a coroutine. [1] A coroutine is created every time a new method is called in the server, especially when new requests from clients come in. Coroutines are implemented cooperatively without preemptive rights so coroutines decide when to yield and when to continue execution. Because of this asynchronous functionality, a server herd architecture is implemented very well using the asyncio framework. Asyncio is also bolstered by the large variety of supporting libraries in Python. Although Asyncio does not have its own dedicated HTTP web API, it is compatible with libraries such as aiohttp which allows Asyncio to keep its core functionality intact and lightweight without giving up any other functionality. The core asyncio framework provides the essential asynchronous functionality required by virtually all asynchronous servers and it is compatible with a variety of libraries to add on the specific niche requirements of each individual developer's server.

Asyncio's cross-server compatibility also makes scalability not an issue. Since we can initialize multiple different servers and communicate with each server using open_connection, server herds are implemented very easily. Asyncio has built-in functionality to support rapidly increasing numbers of servers in a server-herd architecture.

3.2 Disadvantages of Asyncio

One of the main problems for the asyncio framework is its incompatibility with most multithreading functionality. Since asyncio, along with Python as a whole, is single threaded, certain costs that come with single-threaded programming can occur. Multithreaded packages can create multiple threads to handle high intensity traffic to serve up the same fast response times as with low traffic. However, single-threaded packages become exponentially less responsive as the traffic load increases. This is a scalability issue but the workaround is the server herd architecture where multiple servers are created that are used as individual threads. However, there is still risk for traffic to bottleneck at a single server and cause that server to crash or stall because of high traffic. [1]

Another problem with asyncio is a general problem with asynchronous models. Asynchronous servers generally execute their tasks based on performance which may cause tasks to be executed out of original order. This presents certain problems with the developer, since seemingly out-of-order execution can be hard to debug and introduce bugs that are out of the developer's control. Furthermore, with the introduction of a server herd, the potential of a race condition is a very likely scenario. As the number of servers increase, the number of requests that are flooded amongst the servers along with singular requests to individual servers might conflict and cause issues with reliability. For example, a WHATSAT request may come in after an IAMAT request but be processed after the IAMAT request resulting in an error even though the user expected a successful queried response. Asyncio trades reliability for performance when it chose asynchronous execution and because of this asynchronous feature, models can be stalled or become unreliable when traffic increases.

4 Python vs Java

One of the main concerns that affect this project is the decision to choose Python, specifically the asyncio module, for implementation of the server-herd architecture. One of the main concerns is Python's suitability because of the language's implementation of type checking, memory management, and multithreading in comparison to Java's implementations. We consider this problem by examining each language's implementation of these specific topics in more detail.

4.1 Type Checking

Python uses dynamic typing which is essentially type-checking during run time. It also implements duck typing such that any method that is defined by an object can be invoked by that object [6]. Because of the relatively lax restrictions on typing during compile time for Python, it is easier to set up a prototype server in Python without needing to figure out all of the miscellaneous types of each variable in that prototype server. Python's typing lets users write executable code that is easily read and understood without needing to understand the nitty gritty details of what type each specific function returns. On the other hand, setting up a server in Java requires every function and variable to have concretely defined types and return types. This requires a lot more research which disrupts the overall work flow [6].

However, Java's implementation provides reliability measures as there is no typing conflicts during runtime. Since Python is interpreted, an error may not surface until the program is executed to that given line which may cause unforeseen consequences if the server requires certain guarantees of reliability. Furthermore, an interpreted language like Python requires extra instructions during run time to interpret the language before executing it. Java compiles the program down to low level byte code so it is more efficient during runtime and runs faster. In this vein, Python is less efficient than Java during a server's runtime.

4.2 Garbage Collection

Python uses a reference counter scheme when it performs garbage collection. Objects are assigned an extra field when stored in memory which denotes the number of times a reference to it has been created. When a reference is erased, the reference counter for the object that this reference points to is also decremented; vice versa, if a reference is created, the reference counter is incremented. This is a very fast way of keeping track of references and allows for immediate deallocation and allocation of objects. Periodic sweeps of memory free the objects with zero reference count. The problem with Python is its inability to deal with circular references. Two lists with elements within the list that point to each other cannot be deallocated by the reference counter method and leaves dangling pointers which can cause memory leaks that are uncontrolled by the developer [4].

Java, on the other hand, implements an efficient mark-and-sweep implementation where periodic sweeps of memory deallocates unused objects. Java's memory implementation requires periodic sweeps that go through all of memory which takes a large amount of memory and processor time. However, Java implements certain techniques that reduce the amount of work the garbage collector does. For example, Java has a generational system that upgrades certain generational blocks of memory if it survives multiples sweeps of memory so that they are examined less often by the garbage collector [4].

Overall, Python's memory management scheme fits the server-herd architecture better because server herds create and delete variables quickly in response to brief requests from clients. Circular data structures are also uncommon in the server herd architecture.

4.3 Multithreading

Python, as mentioned in the asyncio discussion, has a distinct disadvantage compared to Java when it comes to multithreading. Python is subject to a constraint called a Global Interpreter Lock or GIL and this lock is a global mutex that prevents Python from performing concurrent access to objects via multiple threads [2]. This effectively restricts Python from using any multithreading to execute jobs and also results in a significant performance hit as the amount of traffic a server model receives increases. Java, on the other hand, is able to use its efficient concurrency implementation to split tasks into multiple threads.

Java, can not only assign a single thread to garbage collection to not affect the overall performance of the program, but can also assign different tasks to different threads. Python, in comparison, becomes restricted in potential throughput and is less scalable overall; at a certain point, the asynchronous model which is essentially quasi-multithreading becomes bottlenecked whilst, Java can just create more threads and assign more processing power.

5 Asyncio vs Node.js

Both Node.js and asyncio are well-known and widely-used libraries in the field of asynchronous server development. Node.js like asyncio, features an asynchronous, non-blocking I/O model that instantiates an event-driven event loop [5].

Node.js and asyncio have very similar asynchronous characteristics. Both use a construct to yield to the main event thread: Node.js uses callbacks while asyncio uses coroutines. Events are interleaved in both languages and inserted to the event loop through chains of callbacks. Node.js also implements a Promise construct while Python uses the Future construct to ensure tasks are evaluated in the future [5].

However, Node.js and asyncio do differ on certain performance metrics. Node.js runs on an efficient Chrome V8 Engine that is extremely powerful. However, Node.js does not do well with CPU-intensive tasks that require a large amount of processor cycles [3]. Python, on the other hand, lags behind in memory management. Python slows significantly with scaling memory demands. Both are dynamically typed for ease-of-development but Python is a more portable language [5].

Both asyncio and Node.js are widely supported but they differ in their strengths of support. Asyncio is much more established as an older language than Node.js so it has much better documentation in many areas; however, Node.js is newer so it has faster update cycles and better maintenance of new web development standards [5].

Finally, Node.js has an ease-of-use advantage over Python since most applications have both front and back-end development. Since Javascript can be written in both the front and back end. Node.js provides easier compatibility and writing both the front and backend in Javascript reduces the potential for mistakes and errors. Python, on the other hand, does not have much front-end support so developers who want to develop full-stack would have to choose two different languages to develop the front and backend separately, raising certain compatibility issues.

Overall, asyncio seems the be more fit for designing a server-herd because most of the advantages of Node.js are irrelevant to our application. Since we are spawning multiple server instances, we won't have much memory pressure but we might have CPU pressure from a high number of requests to one particular server. We also don't have a front end when implementing the server-herd so we don't need to use two languages to implement the project. Therefore, asyncio and Python is the correct choice for development.

6 Conclusion

From the research we performed in this project, we have come to a conclusion that although Python and asyncio fall behind Java and Node.js in certain areas, these specific areas are less relevant to implementing a server-herd architecture. On the other hand, the advantages that Python and asyncio present fit the server-herd model well compared to these other implementations. For a Wikimedia-style application, implementing the server-herd architecture using Python and asyncio is the optimal decision.

7 References

[1] “Asyncio - Asynchronous I/O¶.” Asynchronous I/O and Concurrency Python Documentation, docs.python.org/3/library/asyncio.html.

[2] “Global Interpreter Lock.” TcpCommunication - Python Wiki, wiki.python.org/moin/GlobalInterpreterLock.

[3] Kayode, Badewa. “Running CPU Intensive Task in Nodejs.” Medium.com, Medium, 9 Aug. 2017, medium.com/@badewakayode/running-cpu-intensive-task-in-nodejs-db4f995db310.

[4] “Memory Management Reference.” 2. Allocation Techniques - Memory Management Reference 4.0 Documentation, www.memorymanagement.org/mmref/lang.html.

[5] Netguru. “Node.js vs Python Comparison: Which Solution to Choose for Your Next Project?” Netguru Blog on Machine Learning, Netguru, 27 Sept. 2018, www.netguru.com/blog/node.js-vs-python-comparison-which-solution-to-choose-for-your-next-project.

[6] Radcliffe, Tom RadcliffeTom. “Python vs. Java: Duck Typing, Parsing on Whitespace and Other Cool Differences.” ActiveState, 3 Feb. 2019, www.activestate.com/blog/python-vs-java-duck-typing-parsing-whitespace-and-other-cool-differences/.

[7] “Flooding (Computer Networking).” Wikipedia, Wikimedia Foundation, 6 Mar. 2019, en.wikipedia.org/wiki/Flooding_%28computer_networking%29.