Proxy Server Herd Implementation Analysis

Ameya Thakur, 004933653

Abstract

In this paper, I compare the implementation of a server herd application in Java, Python, and Node.js. Due to the asynchronous nature of server operations, the efficiency of Node.js, and its simplicity to use due to dynamic typing, Node.js is likely the best runtime for this application.

1 Introduction

There are many web server architectures that can be chosen for a given project. Each has varying complexity, performance, reliability, and scalability. One possible architecture is a server herd. In this architecture, a client can send queries to any server in the herd, and it will receive a response based on the latest version of the data. In order to achieve this, the servers in the herd communicate with each other so that each one has the latest version of the database at all times (discounting some synchronization latency). We will test a server-herd architecture in this project by creating a proxy server herd for the Google Places API. Clients will communicate their location to any server, and it will be updated on all of the servers in the herd. The servers will keep track of the locations of clients, and if the client requests information about places near the location of a client, the server will query the Places API and respond with the result. A proxy like this is useful in order to prevent clients from abusing the API by accessing it directly. Furthermore, having multiple servers allows for load distribution and scaling up by adding more servers to the herd.

Servers need to be able to operate asynchronously in order to accept connections from clients while already handling older connections. The goal of implementing this project is to analyze the effectiveness of Python's asyncio framework for asynchronous operations [3]. We will compare the Python implementation with possible implementations in Java and Node.js in order to determine the optimal system to use for this application.

2 Server Design and Implementation

2.1 Server Herd Architecture

For this project, we were instructed to implement a herd with 5 servers: Goloman, Hands, Holiday, Welsh, and Wilkes. Each server runs the same code, but a command line argument is passed when running the server.py file to specify which server is being created.

Each of the servers has a specific subset of servers that it floods to. Flooding is the process in which a server sends the location data it just received to other servers in order to synchronized their data. The routes that each server floods are stored in a dictionary so that each server can look them up at runtime based on the command line argument for the server name. The routes are as follows:

```
Goloman -> Hands, Holiday, Wilkes
Hands -> Goloman, Wilkes
Holiday -> Goloman, Welsh, Wilkes
Welsh -> Holiday
Wilkes -> Goloman, Hands, Holiday
```

Each server communicates to the others in order to synchronize their location data after a query from a client modifies one of their stored locations dictionary.

2.2 Interface

Each server takes two commands from clients and one from other servers to handle flooding. The first query that the server takes from clients is `IAMAT`, which includes an ID, location, and timestamp. This query simply notifies the server of the client's location, and the server sends a simple `AT` response back to indicate it has received the message and the time between the message being sent by the client and received by the server. The server stores the response in the locations list in order to be able to use the information to respond to the second query.

The second query that a client can make to the server is `WHATSAT`, which includes an ID, a radius in kilometers, and a cap on the number of places to return. This query is requesting information about the places that are near the location of the client specified by the ID. This information is gathered by querying the Google Places API with the location of the client according to the locations dictionary and the radius in meters.

In order for data to be synchronized between the servers, each server needs to flood messages to the others it is connected to. This flooding occurs by a server establishing a connection to each of the servers in the dictionary of routes corresponding to its name. The server then sends the `AT` response that it sent as a response to the client to the other servers. The other servers detect the `AT` response and interpret it as a flooding update. The receiving servers check if they already have newer data by comparing the timestamps of the current query and the location with the same ID in the locations dict, if it exists. If it is a location that the server has not already received, it stores it in the locations dictionary and floods it to the servers it is connected to as well. By flooding in this way, all servers should be updated in less than 3 steps.

As instructed by the TAs, the server assumes that each connection will send only one command. This allows commands to contain any white space, including newlines, because the command will terminate with an EOF. However, this makes the flooding implementation unrealistic. Rather than a server keeping open connections to the routes it needs to flood, connections are repeatedly opened and closed when updates need to be made. This comes with significant overhead, and in a situation with many simultaneous queries, flooding would likely be a bottleneck. However, the one message per connection implementation of flooding is simple and good enough for a research application such as this.

2.3 API Queries

When a `WHATSAT` command is received by the server and the arguments are validated, an asynchronous function is used to query the Google Places API. It is important for this function to be asynchronous because the latency for internet queries is very high compared to local operations. If API queries were blocking operations, the server would experience extremely slowdowns while it waited for an API response. Asynchronous operation allows other code to execute while the server is awaiting a response.

To query the Places API asynchronously, I used the aiohttp library, which provides asynchronous HTTP support. I also used the standard library's regular expression module to modify the location coordinates so that they matched the API's specifications. The aiohttp library sends a request with the specified parameters and awaits the result [5]. The JSON result is then decoded into a python object so that the number of results can be modified according to the arguments in the `WHATSAT` command. The python object is then converted back to a pretty-printed JSON string by the json module. The server responds to the `WHATSAT` command with the saved `AT` location response with the JSON response after it, terminated with two newlines.

3 Asyncio Analysis

The python asyncio framework creates an event loop in which coroutines can be executed concurrently. This means that while one routine is waiting for a response from an API call, the server can still accept new connections from clients. This framework also has very useful high level wrappers. I used the streams API to create the server and keep it running very easily. The server reads from the client until it reaches EOF, at which point the input is processed and a response is sent. Using the stream API abstracts the event loop completely, and allows for very rapid prototyping. Implementing the server herd's flooding and API calls was straightforward. However, Python is a dynamically typed language, meaning there is potentially less stability if exceptions are not handled properly. Dynamic typing may also lead to some security issues for servers if a client provides input that is interpreted in an unexpected way.

A potential problem with asynchronous operations is that operations may complete out of order. For example, if one server is flooding while another server is receiving a request from a client, it is possible for a newer location from a client to be overwritten in favor of an older flooded message. This is undesired behavior, but it is acceptable in this situation because it is reliable enough. The locations may not be precise and the responses may be slightly out of order sometimes, but the performance gained by asynchronous operations makes it a significantly better choice for servers.

Python, with the asyncio framework, is a very good choice of language to prototype servers and test different architectures easily. The downside to Python is that, as an interpreted language, it will likely not be as fast as an implementation in a compiled language like C. These tradeoffs make Python an excellent choice for the early stages of development, but it may not be wise to choose Python to write a server in a production environment.

4 Comparison with Java

One potentially strong language to write a server in is Java. Java also has asynchronous I/O support through the NIO API [6]. This API allows the server to create asynchronous socket channels, which implement non-blocking communication. This API is not significantly more complex than the Python asyncio framework. However, there are several other differences between Java and Python that can affect the server.

4.1 Type Checking

Java is a statically typed language, while Python is dynamically typed. This means that writing code with Java will likely be more time consuming, but it may be more stable in the long run. Rapid prototyping is easier in Python, but static type checking makes Java a better option for production grade servers which need reliability.

4.2 Parallelism

Java has strong support for multi-threading built into the language. Python somewhat supports parallelism, but the Global Interpreter Lock prevents multi-threading from being efficient. The GIL effectively allows only one thread's code to be interpreted at a time, which means multi-threading is not very useful in Python [2].

Server code needs to handle as many simultaneous requests as possible with given hardware. Multi-threading allows a single server process to make use of all of the many cores in modern high-performance server hardware. It is possible for the Python server to use a server with many cores as well. Given the server herd model, one physical server can run one python server process per thread. However, this comes with additional disadvantages such as the overhead of additional interpreters and the additional ports required. Overall, the strong parallelism support in Java makes it a better choice for servers due to its ability to scale well and provide high performance.

4.3 Memory Management

Python and Java have different memory management models. Python uses a link count based garbage collector. In this model, every object has a link count of the number of pointers to the object. The advantage to this is that garbage collection can be done on the fly as soon as the link count goes down to zero, so large pauses do not have to occur during operation. However, a significant problem with this is that any changes to pointers will require multiple changes to link counts. However, Python also uses a Java-style mark and sweep garbage collection model when space gets low in order to handle cycles, which cannot be addressed by a link count model.

Java uses a mark and sweep memory model. In this model, the garbage collector periodically marks objects that are still in use, and then sweeps objects that are not in use by marking them as free. This process can sometimes require a pause while garbage collection is performed, but with advances such as mostly concurrent mark and sweep and parallel mark and sweep, it is still performant. Furthermore, Java also uses generational garbage collection, which significantly speeds up garbage collection on objects that are rapidly created and destroyed [7].

A server will rapidly create and destroy data structures as queries are received, processed, and returned. Consequently, the memory management model must be fast before all else. Server hardware generally has large quantities of memory, so slightly wasteful use of memory is preferred to slower operation. As a result, the Java memory management model will likely be a better choice for servers under high load.

4.4 Execution

Python and Java are both interpreted languages, but Java's use of a Just In Time compiler means that Java code can often be almost as fast as native code. Python, on the other hand, has no JIT compiler and must always be interpreted from the original source. It is sometimes possible to compile Python to machine code ahead of time, but this is still not as performant as Java code. Overall, Java is a faster language which is key for servers.

From nearly all perspectives, Java is a better choice than Python for production servers. Python may be better in the initial stages of a project for rapid prototyping, but it will have major detriments if used in an environment where performance is the highest priority.

5 Comparison with Node.js

Node.js is a Javascript runtime based on the Google v8 Javascript engine. Javascript is an asynchronous language built around the event loop concurrency model [4]. This means that writing server code with asynchronous support is very simple in Javascript using promises or other libraries.

Javascript and Python are both dynamically typed languages, which makes them easier to write. However, the execution of Javascript in Node.js and Python differ. The v8 Javascript engine does not simply interpret Javascript. Rather, Javascript is compiled to machine code and optimized, leading to execution that can be faster than Python [1]. Node.js and Python are both single threaded and focus on concurrency for speed increases rather than parallelism.

Overall, Node.js and Javascript are a good balance between the ease of development of Python and the high performance of Java. Because the language is built around an asynchronous event-loop model, it is even better at running servers with many simultaneous requests. The disadvantage of Node.js is that it is still single-threaded, so CPU intensive applications may be slower.

6 Conclusion

Asynchronous I/O is necessary for a server to be able to serve many requests simultaneously. Most programming languages provide some API or library for asynchronous operations, but there are specific features that provide a significant benefit to servers and server herds in particular.

Python is a highly writable language; dynamic typing and simple syntax make it ideal for rapid prototyping.

Python's main disadvantage is speed. Servers need to be able to serve as many requests as possible as fast as possible, and Python is simply not the ideal language for this use case.

Java is less writable than python due to static typing and a more complex syntax. However, Java has a significant speed advantage over Python due to JIT compilation and AOT compilation to bytecode. Furthermore, the Java memory management model would likely be better for high performance servers due to its low cost per operation. Java also has multi-threading support, which enables it to have even higher performance on CPU bound operations. Static typing also potentially provides a reliability and security improvement.

Node.js is the ideal balance between Python and Java. It is dynamically typed, making it faster to develop in, but it is also very fast. It is single threaded, but the architecture of the server herd means that one server can be created per thread on a physical server to distribute the load. The main advantage with Node.js is that asynchronicity is fully built into the language, making it ideal for a server which needs to field lots of connections.

A server herd is a viable architecture which can effectively distribute load in systems with lots of connections and updates. Asynchronous operation is extremely important for the server herd architecture, which is why Node.js should be chosen for this application rather than Python. Python is a good option for rapid development in the early stages of a project in order to test ideas, but it should not be used as a long term solution in a production environment.

References

[1] Digging into the turbofan jit. 2015. https://v8.dev/blog/turbofan-jit.

[2] Python global interpreter lock. 2017. https://wiki.python.org/moin/GlobalInterpreterLock.

[3] Asynchronous i/o. 2018. https://docs.python.org/3/library/asyncio.html.

[4] Javascript event loop. 2019. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop.

[5] Welcome to aiohttp. 2019. https://aiohttp.readthedocs.io/en/stable/.

[6] Manoj Debnath. Understanding asynchronous socket channels in java. 2017. https://www.developer.com/java/data/understanding-asynchronous-socket-channels-in-java.html.

[7] Pankaj Kumar. Java (jvm) memory model memory management in java. 2017. https://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-javamemory-management-in-java-8211-java-garbage-collection-types.