Proxy Herd with asyncio

Abstract

In the LAMP architecture of Wikimedia, many web servers running behind the abstraction of a virtual load-balancing router can improve both performance and reliability [1]. The performance comes from distributing the traffic of requests to many servers and the reliability comes from the redundancy of storing copies of the same data across multiple servers. However, a new service designed for news will have frequent updates to a central server, various access protocols in addition to HTTP, and more mobile clients. A bottleneck could arise if the new use cases were used on the current Wikimedia Architecture. Thus, a server herd architecture is proposed to reduce the bottleneck of these new factors. A server herd architecture provides eventual consistency with a flooding algorithm across multiple servers and asynchronously processes and responds to both client requests and server flood requests. The solution will be built with python and asyncio. The advantages and disadvantages of python and Java, as well as asyncio and Node.js will be discussed below in order to analyze which language or framework provides the best solution.

1 Introduction

The concept of a server herd architecture consists of multiple servers that relay information to each other while simultaneously responding to client requests without the need of a centralized database. The motivation behind this architecture is to split up the workload of a single server into multiple servers while ensuring eventual consistency in order to eliminate the bottleneck of a single server processing all requests. In this project, I implemented a server herd architecture by exploring asyncio, a python library for writing concurrent code with the async and await syntax tokens. This report will discuss whether or not asyncio is a suitable framework for this architecture. It also compares the language choice of python with that of Java, comparing how the type checking, memory management, and multithreading properties of both languages affect scalability and reliability.

2 Design Decisions

The project specifies that five different servers named Goloman, Hands, Holiday, Welsh, and Wilkes run on five different assigned ports such that when a single client sends information to a server, the information gets propagated to each server's reachable neighbor servers. The idea of reachability is defined by which servers are allowed to talk to each other in a bidirectional manner. See *Table 1* below for the table that defines the reachability of each server and *Table 2* for the port numbers associated with each server.

Server ID	Communication Links
Goloman	Hands, Holiday, Wilkes
Hands	Goloman, Wilkes
Holiday	Goloman, Welsh, Wilkes
Welsh	Holiday
Wilkes	Goloman, Hands, Holiday

Table 1: The table of each server's allowed bidirectional communication links.

Server ID	Port
Goloman	11670
Hands	11671
Holiday	11672
Welsh	11673
Wilkes	11674

Table 2: The port numbers of each corresponding server.

I will briefly discuss the design and implementation decisions in order to implement the project functionalities.

2.1 Server Herd

The server herd design consists of two files, config.py and server.py. Config.py contains the API key for making HTTP requests to the Google Places API, the assigned port numbers associated with each server name, and the communication links between the servers. Server.py consists of the logic that handles client messages and queries as well as flood operations from other servers. The server asynchronously accepts TCP connection requests from clients and receives messages from the client with co-routines. Client messages that begin with IAMAT simply update a backend map that associates client IDs with a data record and WHATSAT looks up the coordinates of the most recent saved location of the client in the backend map and returns the results of the Google Places API request for locations near the coordinates. Finally, whenever a server receives an IAMAT or FLOOD message and successfully updates its backend, it propagates the received information to any reachable neighboring server.

2.2 asyncio

As referenced previously, part of the solution in reducing the bottleneck of the server herd architecture is the ability to perform tasks asynchronously. The asyncio python framework provides an interface to

create python application servers that asynchronously handle client requests. Below are some advantages and disadvantages of implementing the server herd architecture with the asyncio framework.

2.2.1 asyncio Advantages

The framework offers a simple interface to develop the server. Because the asyncio library calls for each server are contained within the server.py file and each server runs the same task, adding a new server is as simple as running another instance of the server program.

Within the server.py file, the asyncio framework easily supports and complements the design pattern of a server. Every server runs a loop and adds a co-routine whenever a new message or client request comes in. This means that a server can simultaneously handle many messages and requests by simply invoking a co-routine each time a new message is received. Each message is queued and does not need to be processed until it is at the front of the event loop. The basic requirements of the server herd are already efficiently implemented in the asyncio framework, especially the idea of adding a co-routine after each message and running each co-routine at the same time [2]. Even the functionalities of starting a server at a specific IP address and port and running a certain task loop with any function are already implemented and abstracted into an interface for the developer.

Asyncio is highly performant. Instead of needing to poll in order to continuously call reader.readline() in order to receive a message, asyncio offers the await keyword so that the co-routine simply continues execution once something is able to be read from a client. The overhead in needing to constantly poll is reduced by having await wait for a notification to determine when to resume normal execution.

2.2.3 asyncio Disadvantages

Unfortunately, although asyncio has many advantages, especially in the context of creating correct server behavior, there are many disadvantages and limitations of the framework.

A major disadvantage of asyncio is the absence of supporting multithreaded servers. Although there is an event loop that juggles multiple co-routines, there is only ever one thread running at a time [3]. This means that there could be potentially faster performance if multithreading was supported, as tasks run in parallel could save time. Parallel threads could be utilized by creating a thread each time a co-routine task is initiated instead of just adding the task to the event loop. However, the amount of time saved in a multithreaded framework generally scales with the size and complexity of the server. For this project, the functionalities are simple and the number of requests a server receives is not expected to be high. However, as the load the server experiences scales, the more of a bottleneck the server code becomes. This is because many requests are being queued but only one task can be done at a time. A parallel, multithreaded framework could perform tasks in parallel, reducing the potential queue size of the tasks.

Another limitation is the lack of guaranteeing a strict order of execution. In fact, asyncio offers no promise of the order in which tasks complete, despite knowing the order in which the tasks began. This implies that the programmer must design and develop their program without relying on the order of processed tasks, which reduces flexibility. This issue also causes the program to be difficult to debug, especially when dealing with client requests that arrive at almost exactly the same time. For example, assume that an IAMAT message was previously received and that there is latitude/longitude data saved for client A. Next, assume the server receives an IAMAT and a WHATSAT request back-to-back. In a synchronous model, one would expect that the WHATSAT returns the data associated with the newest IAMAT message specifying client A's new location. However, if the WHATSAT query is processed before the newer IAMAT message, data associated with the older IAMAT message will be returned. This will produce unexpected results, especially because no errors or runtime exceptions actually occur. Thus, the incorrect response may even be unnoticed, producing incorrect results without a warning. The consequences of this issue increase as we scale the number of servers in the server herd as well as increase the number of clients or client requests coming into the servers. Out of order execution means that multiple trials of the same exact client queries could produce different results. This lack of consistency decreases the effectiveness of the server herd.

2.3 IAMAT Design

The IAMAT message has a format of:

IAMAT <client_id> <lat_long> <time_sent>

where the <client_id> identifies the client sending the message, <lat_long> is the latitude and longitude of the location the client wants to send, and <time_sent> is the client's idea of when it sent the message in POSIX time. The server, after successfully receiving the entire message, cleans the input text by removing all white space characters from around and in between the parameters and builds a list from the

input. Then, each parameter is checked to make sure they have expected values. If all checks pass, then we know we have expected input values. I simply store the message's information in a simple map object representing the server's backend data store. The key to the record is the <client_id>, and the record itself includes all the received information including the <server_id> that receives the message and the time difference between when the client sent the message to when the server receives it. If there is already information in the backend for this <client_id>, I save the most recently sent out IAMAT message's data into the backend. The server responds with the appropriate response to the client by echoing back the information along with the server and time difference values. If a successful record was added or changed in the backend, then a FLOOD statement is sent out to each reachable server to broadcast that the client's information has changed.

2.4 WHATSAT Design

The WHATSAT message has a format of:

WHATSAT <client_id> <radius> <max_entries>

where <client_id> identifies the client we care about for querying a location, <radius> represents the distance in kilometers away from the client's location, and <max_entries> represents the maximum number of places that we want to receive from the query. After the server cleans and parses the input, it replies to the client with the same AT response that the IAMAT message gets but also attaches the JSON results of the HTTP call to the Google Places API call made with the aiohttp library. If the client record cannot be found in the server's backend map, then we simply respond to the client with the default error message as specified:

? <original_input_message>

When the client sends a WHATSAT query, data is fetched but no data is set, so no flooding occurs. This means that no data will be propagated from server to server after a WHATSAT query, so no FLOOD messages will be sent to neighboring, reachable servers.

2.5 FLOOD Design

The FLOOD message is internal to the server herd and will not be sent be a client. The purpose of this message format is to implement a flooding algorithm. Whenever any record in the server's database is successfully updated (stored a more recent record or stored a record for the first time for this client), a FLOOD message will be sent to all reachable servers as specified by *Table 2*. The FLOOD message has a format of:

FLOOD <server_id> <time_diff> <client_id> <lat_long> <time_sent>

where the <server_id> is the identifier of the server that the client talked to for that specific record, <time_diff> is the difference between the client's idea of when the message was sent and the server's idea of when the message was received, <client_id> is the identifier of the client that caused the creation of this record, <lat_long> is the location of the client, and <time_sent> is the client's idea of when the message was sent. There are two cases where a FLOOD message would be sent from a server to all of its reachable servers. The first case is any time an IAMAT message successfully creates a new record or replaces an old record in the backend database. The second case is any time a FLOOD message successfully creates a new record or replaces an old record in the current server's backend database. Flooding for any server stops once it's propagated this new information once to each of its reachable servers. This implementation guarantees that any valid update to one server's backend eventually propagates this change to all the other reachable servers. Eventual consistency is thus achieved with this implementation. This successfully prevents the bottleneck of just having a single server by allowing a herd to collectively receive client requests but sacrifices the luxury of having immediate consistency that a single server would be able to achieve.

2.6 Error Checking

The error checking design verifies the format of any input received by the server. First, the program sanitizes the input by trimming any unnecessary white space characters before, in the middle, and after a message. Then, I split the input into a list and check each parameter. For the IAMAT messages, I check that <client_id> is not all white space characters, <lat_long> is comprised of exactly two positive or negative concatenated floats, and that <time_sent> can be converted to a valid float variable. For the WHATSAT queries, I perform the same check for <client_id> and ensure that <radius> can be converted to a float and that <max_entires> can be converted to an int with a try except structure. For the FLOOD queries, I check that <server_id> exists in the recognized server name map, <time_difference> has a valid sign prefix along with a float suffix, and the other fields I check exactly the same way I check any inputs received from IAMAT. If any parameters cannot be correctly parsed, I return ? <original_input_message> as specified.

Lastly, I also check to ensure that the HTTP request made with the aiohttp library is successful and parses

a correct JSON file. If either the HTTP request fails or if the response is not a valid JSON object, I also return ? <original_input_message> as specified.

3 Node.js vs. asyncio

Node.js and asyncio are both frameworks for developing server-side programs. Node.js is compatible with JavaScript while asyncio is compatible with python. Although the frameworks are both single-threaded, there are significant differences in terms of co-routines, efficiency and ease of use.

3.1 Single-threaded

Node.js and asyncio are frameworks that both implement server-side capabilities with a single thread [4]. The asynchronous nature of reacting to real-time events is similar across both frameworks. Thus, both frameworks do suffer from worse performance in the case of a busy server with a high amount of client-server traffic due to the lack of parallelism.

3.2 Co-routines

Node.js does not support the concept of co-routines, while asyncio does support this property. A co-routine allows for certain executions of a task to be stopped and resumed at will, especially when new data arrives. This data can be directly fed into the currently executing task, allowing asyncio to provide a more flexible framework in terms of interacting with dynamic data.

3.3 Efficiency

In terms of efficiency, the server herd architecture sacrifices immediate consistency for performance. Thus, it is important to evaluate which framework provides faster performance. The Chrome V8 engine written in C++ is used to power Node.js [5] and offers a much more performant solution. Python also has slower performance when it comes to memory-intensive programs. When a server herd begins to save more and more data (from clients and propagations), the program may slow down as time goes on. Overall, Node.js seems to be more performant.

3.4 Ease of Use

In general, both JavaScript and python are easy to develop with. Both languages do not require compilation to run, but instead rely on dynamic type checking. The duck typing property [6] of the languages make it easy to set up a server relatively quickly. In terms of the number of lines of code required to get a basic server working, both Node.js and asyncio require fewer lines than a C++ implementation of a server. Node.js does have an advantage over python because of current trends. Because these frontend frameworks like Angular.js and React.js develop the user interface in JavaScript, it is generally easier and less error-prone to also develop the backend with JavaScript and Node.js. However, this is a minor advantage and well-written servers in python can correctly interact with a frontend written in JavaScript.

4 Java vs. Python

Both of these languages are extremely popular when it comes to writing server-side programs. In this section, I will examine the similarities and differences between the languages. Specifically, I will compare their properties of type checking, memory management, and multithreading, and how each property affects the performance and reliability of the program.

4.1 Type Checking

The types of variables are relatively similar across both languages. However, the way that these types are checked happen at very different times. Java is statically typed, meaning that all types must be declared and checked during the compilation stage. On the other hand, python is dynamically typed, meaning the program is allowed to run and types are checked only right before the statement is executed [7]. This is known as duck typing. The advantage Java has is that when the program runs, the compiler guarantees that no type errors will happen during runtime. This means that the program will not crash due to incompatible types and any potential cause of a crash will be caught in the compilation phase. The disadvantage is the overhead for the developer. The developer must compile the program every time before they want to test it. For large programs, this could take a long time. Furthermore, the developer must declare and know each variable's type and each return type of all the functions and library calls being made before assigning the results to a variable. The advantage of python is that developing is quick and easy. The exact return type of a function need not be known during run time and assigned into a declared variable. Thus, a simple program can be more quickly developed. The disadvantage of python is the potential risk for a runtime crash. A small type error that rarely gets run in a conditional branch could be overlooked by simple tests and could result in a crash later. Duck typing could cause developers to fail to understand exactly how the functions and library calls they are making work. Failing to research the exact return types of the functions being called could result in not understanding how to use a function properly.

4.2 Memory Management

Both languages have some sort of garbage collector whose job is to free up memory no longer being used. The Java garbage collector generally implements the mark and sweep method where all objects being referenced get marked, and then a final sweep frees unmarked objects [8]. On the other hand, the python garbage collector uses the reference count method, where a count is associated with each reference to an object [9]. Whenever a reference is created, the object is initialized with a count of 1. When a new reference points to an existing object, the count increments. This can happen when a variable is assigned or when some object is copied. When a reference to the object is freed, the count decrements. The memory is finally freed whenever the reference count of an object reaches 0. A major flaw with python's garbage collector is the failure to correctly handle circular references. Furthermore, the reference count method is relatively slow. In my python implementation, objects are created and destroyed relatively quickly. Thus, memory is freed the moment the memory is no longer needed. In comparison, a Java garbage collector periodically marks and sweeps. If the time in between each mark and sweep is long, unneeded objects will not be freed for a while. Python also has a generational garbage collector that periodically frees objects in memory.

4.3 Multithreading

Java has a variety of multithreading options. Java can take advantage of multiple processors with the synchronize key word [10] or even explicit locks depending on the use case. A multithreaded server can potentially improve the performance of the server herd as tasks can be done in parallel. Care must be taken to ensure no two threads execute in critical sections that involve reads and writes to the database. However, in the worst case, the performance will match that of a single-threaded application, as long as care is taken to avoid deadlock. Thus, Java's multithreading option offers a more scalable way to develop programs compared to python, especially a server herd that could experience high traffic. python does not have the option to create multithreaded programs. A built-in global interpreter lock ensures that only a single thread can run at once, even when python is running on a machine that has multiple cores.

5 Conclusion

After implementing a simple server herd architecture with python and asyncio, comparing asyncio with Node.js, and comparing python with Java, I learned that asyncio is well-suited for developing a server herd architecture. Asynchronous behavior allows the servers to interact with each other and the clients simultaneously. Whenever a server is not doing handling a client's request, it can work on propagating flood messages to improve the perceived efficiency of the system as a whole. Thus, it seems clear that for the purposes of a simple server herd architecture, python may be more suitable due to its ease of development and fast performance for simple, single-threaded tasks. However, as the servers scale up in complexity and size, a Java implementation may ensure reliability and performance due to its support for multithreading. After briefly examining Node.js, it seems like a good candidate for a server herd architecture. JavaScript in the backend and frontend could reduce the errors and dependencies required. Furthermore, it is built on the Chrome V8 engine, which is extremely performant. In order to more conclusively determine which framework is better, a Node.js version of the server herd architecture could be developed and tested against the python implementation. However, this is not generally realistic, which is why it is important to understand the theory behind the strengths and weaknesses of different languages and frameworks.

6 References

[1] *Project. Proxy herd with asyncio*, https://web.cs.ucla.edu/classes/winter19/cs131/hw/pr.html.

[2] *Coroutines and Tasks*, https://docs.python.org/3/library/asyncio-task.html.

[3] *Developing with asyncio*, https://docs.python.org/3/library/asyncio-dev.html.

[4] *Understanding Event Loop in NodeJS*, https://codeburst.io/how-node-js-single-thread-mechanism-work-understanding-event-loop-in-nodejs-230f7440b0ea.

[5] *Understanding How the Chrome V8 Engine Translates JavaScript into Machine Code*, https://medium.freecodecamp.org/understanding-the-core-of-nodejs-the-powerful-chrome-v8-engine-79e7eb8af964.

[6] *Duck Typing*, https://devopedia.org/duck-typing.

[7] *Java vs. Python: Coding Battle Royale*, https://stackify.com/java-vs-python/.

[8] *Garbage Collection in Java*, https://www.geeksforgeeks.org/garbage-collection-java/.

[9] Garbage Collection in Python, https://www.geeksforgeeks.org/garbage-collection-python/.

[10] *Synchronized Methods*, https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html.