Implementation of Server Herd Architecture with asyncio

Zicong Mo, *University of California, Los Angeles*

Abstract

When designing a web server, there are many possible architectures to consider, such as the LAMP platform of Wikimedia, each with their own performance implications. In the case where updates to a central server occur frequently, servers are accessed via various protocols, and clients are more mobile, the central application server of the Wikimedia architecture may serve as too much of a bottleneck. We propose a server herd architecture that handles this type of behavior smoothly, and consider the benefits and drawbacks of using Python's asyncio module as the framework to implement it.

Introduction

A server herd architecture consists of a group of servers that all communicate important information to each other, with each of them being able to serve client requests without needing to refer to a central database. Such a design avoids the bottleneck of an application server that needs to process all requests, leading to speedups in situations where updates occur frequently while maintaining performance when clients connect to multiple different servers. In this project, we implement a server herd that communicates the locations of clients with each other, allowing the client to request from any server a list of nearby attractions.

We consider the advantages of Python to other languages such as Java in the context of designing a server herd with asyncio, focusing on the differences in typing, memory management, and multithreading between these languages. Finally, we consider the benefits and drawbacks of using Python's asyncio framework compared to JavaScript's Node.js framework.

1. Design

1.1. Structure of Server Herd

There are five servers in the server herd, named Goloman, Hands, Holiday, Welsh, and Wilkes. Each server communicates only with a subset of these servers, but all client locations are propagated to all of the servers using a flood-fill algorithm. The graph of server-to-server communication can be expressed in table form:

Server	Communicates with
Goloman	Hands, Holiday, Wilkes
Hands	Goloman, Wilkes
Holiday	Goloman, Welsh, Wilkes
Welsh	Holiday
Wilkes	Goloman, Hands, Holiday

Each server accepts TCP connections from client asynchronously, and messages to the server are processed and handled asynchronously using co-routines. Each co-routine is added to an event loop, which processes each co-routine in its queue. In this way, the server can accept many connections or messages at the same time without slowing down significantly. To communicate with the server, the client can send an IAMAT or WHATSAT message, while CHANGELOC messages are reserved for server to server communication.

1.2. IAMAT Messages

IAMAT messages are sent from the client to any of the five servers and contain information about the reported location of the client and the time this information was sent. The messages have the following form:

IAMAT <client > <loc> <time_sent>

The <*client*> field is the name of the client sending the message, <*loc*> is the reported latitude and longitude of the client in ISO 6709 notation, and <*time_sent*> is the reported time that the client sent the message expressed in POSIX time.

Upon receiving a valid IAMAT message, the server will store the message into its client dictionary, along with the time the server received the message and the name of the server that received the message. The server will acknowledge that it has received the message by sending the client an AT message of the form:

AT <server> <time_diff> <client> <loc> <time_sent>

The <*server*> field is the name of the server that the client communicated with, <*time_diff*> is the difference between when the client sent the message and when the server received the message, <*client*> is the name of the client, <*loc*> is the reported latitude and longitude of the client in ISO 6709 notation, and <*time_sent*> is the time the client sent the message expressed in POSIX time.

In addition to sending the client an AT message, the server also communicates the newly received information to the rest of the servers through a flood-fill algorithm. The server sends a CHANGELOC message to each of the servers it communicates with, allowing each server to store the most recent client location. The process is described in more detail in section 1.4.

1.3. WHATSAT Messages

WHATSAT messages are sent from the client to any of the five servers and contain information about which client to search around and how much information to return. The messages have the following form:

WHATSAT <client> <radius> <num_entries>

The <*client*> field is the name the client to search for points of interests around, <*radius*> is the radius in kilometers around the client to search around, and <*num_entries*> is the number of points of interests to send back to the client. To be considered a valid WHATSAT message, <*client*> must be the name of a client that has previously sent its location to one of the servers, <*radius*> must be a value that can be cast to float, with a value between 0 and 50 inclusive, and <*num_entries*> must be a value that can be cast to integer, with a value between 1 and 20 inclusive.

Upon receiving a valid WHATSAT message, the server makes a request to the Google Places API to find points of interests around the client's last known location with the given radius. The request is made using the most recently stored latitude and longitude of the client and the provided radius (multiplied by 1000 to scale it to meters). The server uses aiohttp to make an asynchronous call to the Google Places API to get the information in JSON format. The server returns to the client an AT message along with the first <*num_entries*> entries of the results.

1.4. CHANGELOC Messages

CHANGELOC messages are not sent from the client to the server, but instead from server to server to propagate changes in client locations to all of the servers. The messages have the following form:

CHANGELOC <client> <new_loc> <time_sent> <time_received> <server_received>

The <*client*> field is the name of the client to update, <*new_loc*> is the new location of the client, <*time_sent*> is the time the client sent this location, <*time_received*> is the time the original server received the IAMAT message, and <*server_received*> is the name of the server that received the IAMAT.

Upon receiving a CHANGELOC message, the server checks the time stamp and the name of the client included in the message. If the server has no knowledge of this client, i.e. this message is the first time the server has a location for the client, then the server stores the message into its client dictionary, and sends the same message to all servers it communicates with. If the server already has a location for this client, then the time stamp of the message is compared with the time stamp of the stored location. If this location update occurred after the current stored location, the location is updated, and the server sends the same message to all servers it communicates with. Otherwise, the server has either already received the message (e.g. A sends to B, which sends back to A since communication is bidirectional), or the information is outdated. In either case, the server does not store the new location or send the message to all connected servers. This way of propagation ensures that no infinite loops occur when updating information but the information still reaches all servers efficiently, as each server receives each message at most twice.

1.5. Invalid Messages

Any message not of one of the three previous forms is considered an invalid message. A message can be declared invalid if it does not start with one of the predefined headers, if its data does not contain values that can be cast to the appropriate type, or in the case of WHATSAT, if a request is made for a client whose location has not yet been stored. Upon receiving an invalid message, the server simply returns the invalid message back to the client with a '?' prepended. For example, if the client sends the invalid message

BOGUS 1 2 3 4,

the server responds with the message

? BOGUS 1 2 3 4

and does not execute any code on behalf of the client.

2. Implementation using asyncio

2.1. Advantages of asyncio

The asyncio framework makes it easy to run servers that asynchronously handle requests. Each server contains an event loop, and can add a co-routine to the event loop when a new message or connection comes in. Because the server is asynchronous, the message is not processed until the co-routine comes to the front of the event loop. This gives the ability for the server to process many requests at the same time, as all it has to do is create a co-routine for the request and add it to the event loop. Adding a new connection can be done at the same time as processing an incoming message. Therefore, implementing the server herd infrastructure using the asyncio framework works extremely well, with basically everything we need from the server herd already implemented in the framework.

The asyncio framework also makes writing code for the server simple. Each server in the herd runs the same code, and therefore adding new servers is as simple as executing another program, rather than having to write new code for the server from scratch. A new server can be initialized simply by running the command *python3 server.py <server_name>*, This ability makes writing the server herd in asyncio easily scalable, as more servers can easily be added.

One functionality missing from the asyncio framework is the ability to make an HTTP request, as asyncio only supports the TCP and SSL protocols. However, this is not a problem, since Python has modules compatible with asyncio that can do exactly this. Since we need the ability to make an HTTP GET request to retrieve the information from the Google Places API, we use the aiohttp module to do so. The aiohttp module also works asynchronously, allowing us continue the asynchronous design of the server and execute co-routines only when they reach the front of the event loop. The module allows us to make a GET request to the specified URL, and retrieve the data in JSON form. Using the json library, we can process the JSON object, retrieve only the number of entries specified, and return it in a nicely formatted string to the client.

2.2. Disadvantages of asyncio

One disadvantage of asyncio and of asynchronous models in general is the fact that tasks are not necessarily processed in the order that they arrive. Not only does this make the programmer's job more difficult, as it makes the program harder to debug, it also introduces bugs and errors that would not exist in a synchronous model. For example, in a synchronous model, if a valid WHATSAT was sent after a valid IAMAT, the client could guarantee that the server would have a location stored for it, since the IAMAT message would be processed before the WHATSAT message due to the synchronous model. However, with asyncio, this guarantee cannot be made, as it is possible that the WHATSAT gets processed first, so the server would have no knowledge of the client's location and would return an error to the client even though the client sent the messages in the correct order.

The problem gets even worse when we have a server herd, as the potential for processing messages "out of order" increases significantly the more servers you have. For instance, consider a client sending an IAMAT message to the server Welsh, then sending a WHATSAT message to the server Goloman. Because the location information sent to Welsh needs to propagate to the rest of the servers, which takes two messages in this case since Welsh and Goloman are not directly communicating, the client could easily receive an error message from Goloman even if they delay the WHATSAT message for a period of time, as the time needed for the information to pass from Welsh to Goloman is highly dependent on the number of tasks the intermediate servers are processing. Note that although this would be a problem even for synchronous frameworks, the asynchronous model would encounter more of these problems. For this reason, the reliability of an asynchronous framework like asyncio would be lower than that of a synchronous framework, which would avoid bugs like the ones described. However, asyncio trades reliability for performance, as synchronous models are more easily bottlenecked by a large request, while asynchronous models can use asynchronous functions to yield execution more often and get more work done.

A problem specific to the asyncio framework is its lack of support for multithreaded servers. The server framework that asyncio runs is single threaded, which can have performance implications as it can only process tasks one at a time. A multithreaded model could create a new thread to handle each incoming task, then run it in parallel to all of the other tasks, greatly increasing performance. For small scale applications, such as the server herd implemented in this project, asyncio may be advantageous, since there is not likely to be much traffic on the servers and asyncio is so simple to use. However, as the number of requests grows large, the performance costs of asyncio may outweigh this ease of use, as the framework is not as scalable as a multithreaded server model due to the fact that it can only process tasks one at a time. This performance cost can be reduced by introducing server herds, as instead of creating new threads, we can easily introduce new servers that would reduce the amount of incoming traffic to each server. Again, if all of the traffic was being sent to one server in the herd, the server would bottleneck, and we would run into the same problem as before.

Finally, because each server in the server herd model runs the same code, using asyncio for a large server herd could lead to issues with maintaining the code for each server. In order to update the code running on the servers to fix a bug, each server must be shut down and restarted for it to run the new code. Even with just five servers in the herd, having to shut down each server and restart was annoying especially since client information would have to be resent as servers are not expected to retain information when they shut down. The problem would only grow with the number of servers in the herd.

Although this issue is not an incredibly major reason not to use asyncio, as the server restarts can be staggered and we can force the servers to log and reload client information on when shutting down or restarting, it is certainly annoying.

3. Comparing Python and Java

A large part of choosing the most suitable framework for the server herd is choosing the language itself. In particular, the language's implementation of type checking, memory management, and multithreading could cause problems as the size of the applications grows large. To illustrate how well Python handles these issues, we compare Python's implementation of these areas to Java's implementation.

3.1. Type Checking

Python and Java differ in the way that they handle types. While types in Python are dynamic, meaning that they are checked at runtime, types in Java are static, meaning that they must be declared and checked at compile-time[1]. One practical benefit of using Python over Java is the easy learning curve of creating a server. When setting up a prototype for a simple echo server, Python's typing ensures that an initial server can be set up without having to understand all of the types of the objects returned by function calls. For example, the programmer does not need to know that the call *await asyncio.open_connection* returns a StreamReader and StreamWriter instance to use them, since Python's duck typing allows the programmer to call *reader.read()* or *writer.write()* without necessarily having to know the exact types of *reader* or *writer*.

In contrast, even just to set up a server in Java, the types of function arguments and variable must be declared before use, requiring much more research and reading documentation just to start the project. However, once the module and framework are well understood, Python's typing may serve as a disadvantage, as the types of function arguments and variables also serve as documentation to help later developers understand the program better. Using the previous example, although developers understand how to use the returned reader and writer object, they may not know just from looking at the code the specific types of the returned objects. In summary, although Python helps developers unfamiliar with the framework start development, the language's dynamic typing may hinder future developers trying to understand the details of the project.

3.2. Memory Management

Memory management in Python involves a private heap containing all Python objects and data structures[2]. Python and Java have different types of garbage collectors, with Python primarily using the reference count method, and Java primarily using the mark and sweep method[3]. Each object in Python has a field called the reference count, which is increased or decreased when a variable is assigned or when the object is copied or deleted. When the number of references to the object is 0, there are no more variables that can reach the object, and it is therefore safe for deletion. Although reference counting has the advantage that objects can be immediately destroyed when they are no longer needed, saving memory, it fails to deal with circular references and also has a performance overhead[3], as each assignment must change the reference count in addition to performing the actual assignment. Since our application creates lots of variables that are only used once or twice before they are no longer needed, Python's ability to immediately return the memory works well for the server herd framework. In addition, the circular references issue for the reference count method is not a big deal for this framework, as they are uncommon due to how fast the application initializes then deletes variables. In any case, Python has an additional generational garbage collector that would free the memory periodically.

3.3. Multithreading

A big disadvantage of Python compared to Java is Python's lack of multithreading support. Python uses a global interpreter lock to synchronize threads such that only one thread can execute at a time, meaning it executes exactly one thread at a time, even on multi-core processors[4]. In contrast, Java is able to utilize multi-core processors in its threading applications to get more work done in the same amount of real time. As discussed earlier, not having a multithreaded server means that the potential throughput of the server is decreased, as less requests can be satisfied in the same amount of time. Consequently, a server running Python is less scalable than a server running Java, as the Java server has multithreading capability and would be able to process more requests and have higher throughput.

4. Comparing asyncio and Node.js

Both asyncio and Node.js are frameworks for writing server-side code, written in Python and JavaScript respectively. To illustrate some key differences between the two languages, we consider the performance, concurrency, and accessibility of both frameworks.

4.1. Performance

Arguably the most important aspect of its program after its correctness is its performance. The entire motivation behind the server herd is performance, as the central application server of a Wikimedia architecture server could potentially be bottlenecked. In this metric, Node.js is significantly faster than Python[5]. Node.js is based on Chrome's V8 engine, which is extremely fast and powerful. In addition, Python's performance slows significantly when working with memory intensive programs, although this is not a big concern with server herds as they are not particularly memory intensive.

4.2. Concurrency

Both asyncio and Node.js run the same single-threaded event-driven asynchronous architecture[5]. Many of the concepts are similar as well. For example, the Node.js analog of the Future is a Promise, which is a task that can be evaluated asynchronously sometime in the future. One advantage that asyncio has over Node.js is the concept of a co-routine, whose execution can be halted and reentered with new information. This gives asyncio the ability to be slightly more dynamic and flexible.

4.3. Accessibility

JavaScript possesses many of the same characteristics that make Python so easy to use. For instance, Python and JavaScript are both dynamically typed, so the duck typing that makes it so easy to set up a server in Python also makes it easy to start development on a project in Node.js. One advantage that Node.js has over Python in this area is due to the language itself. Since JavaScript is so popular in front-end development, writing the back-end in JavaScript as well would lead to fewer mistakes and errors than writing the back-end in Python[5].

Conclusion

From our investigation into the details of asyncio and Python as well as our implementation of the proxy server herd, we see that the asyncio framework is extremely suitable both theoretically and practically for the task of designing a server herd. The asynchronous nature of asyncio and the event loop provides the exact functionality that we want in the server herd, which is the ability to process and handle several requests simultaneously. Although we did not investigate Node.js and JavaScript in as much detail as we did asyncio and Python, a preliminary look shows that Node.js may be more appropriate for our server herd, with its increased performance and accessibility. However, before we can conclude that Node.js is the most suitable framework to use for the server herd, additional tests similar to the proxy server herd designed in this project should be conducted on the Node.js framework.

References

[1] Radcliffe, Tom. *Python vs Java: Duck Typing, Parsing on Whitespace and Other Cool Differences.* ActiveState Blog. Jan 26, 2016. Available: https://www.activestate.com/blog/2016/01/python-vs-java-duck-typing-parsing-whitespace-and-other-cool-differences

[2] *Memory Management.* Python Software Foundation. Available: https://docs.python.org/3/c-api/memory.html

[3] Golubin, Artem. *Garbage collection in Python: things you need to know*. Apr 05, 2018. Available: https://rushter.com/blog/python-garbage-collector/

[4] Beazley, David. *Inside the Python GIL*. Chicago: Chicago Python User Group. Jun 11, 2009 Available: www.dabeaz.com/python/GIL.pdf

[5] *Python vs Node.js: Which is better for your project.* DA-14. Aug 03, 2017. Available: https://da-14.com/blog/python-vs-nodejs-which-better-your-project