CS 131 Project: Proxy Herd with asyncio Report

Abstract

This report examines Python's asyncio asynchronous networking library as a way to implement an application server herd architecture. Ultimately, it concludes that asyncio is a suitable framework for this kind of application, primarily because its asynchronous approach allows it to efficiently process many requests and connections at once.

Introduction

A proxy herd, or application server herd architecture, consists of multiple application servers communicating directly to each other as well as via the core database and caches. This means that for rapidly evolving data such as GPS-based locations, this architecture can avoid the performance bottleneck associated with accessing a central database. Instead, the servers exchange the relevant information so they can each serve client requests. This project, in which clients can provide readily-updatable GPS-based locations then request nearby attractions, therefore lends itself to an application server herd design.

The asyncio library provides a set of both high-level and low-level APIs to write concurrent code making use of Python coroutines and event loops. This paper considers the pros and cons of the asyncio library, as well as whether it is well suited to this kind of application. Further, this paper discusses the pros and cons of Python and Java regarding type checking, memory management and multithreading, and also briefly compares asyncio to Node.js.

1. Implementation of Application Server Herd

This project consists of five servers within the server herd, with server IDs Hill, Jaquez, Smith, Campbell, and Singleton. Each server is designated a distinct port number. Each serve can communicate bidirectionally with only a subset of all the servers, and thus location updates are propagated across the server graph by using a simple flooding algorithm. The servers continue to operate even if a neighboring server goes down. Each server asynchronously accepts TCP connections from clients. Messages between a server and client, as well as between two servers, are also sent and received asynchronously. Each asynchronous event is a coroutine, which is added to the event loop. This event loop sequentially goes through and processes each coroutine in its queue. Thus, the server herd can handle many connections at the same time without getting bogged down waiting for messages.

Server	Communicates With
Hill	Jaquez, Smith
Jaquez	Hill, Singleton
Smith	Campbell, Singleton, Hill
Campbell	Smith, Singleton
Singleton	Jaquez, Smith, Campbell

1.1. Server-Client Communications

Clients can send two types of reqeusts to the server: IAMAT and WHATSAT requests.

IAMAT requests are used for the client to provide data about itself to the server herd. The format of these requests is:

IAMAT <client_id> <location> <time_sent>

The client_id is used to identify a given client, location is the latitude/longitude pair of the client in decimal degrees using ISO 6709 notation, and time_sent is the client's idea of when it sent the message expressed in POSIX time.

When a server receives an IAMAT message, it stores relevant data about the client as well as when the message as received and the name of the server that received it. It then propagates this information to the rest of the servers by sending FLOOD requests. It responds to the client with a response of the following form:

AT <server_name> <time_diff> <client_id> <location> <time_sent>

where server_name is the name of the server that originally received the IAMAT request, time_diff is the difference between the server's idea of when it got the message and the client's time stamp, and the rest of the parameters are the same as above.

Clients can also send a WHATSAT request, which is used to query information of nearby attractions. The format of these requests is:

WHATSAT <client_id> <radius> <num_places>

where radius specifies a radius in kilometers from the client, num_places specifies an upper bound on the number of nearby attractions, and client_id is the same as before.

When a server receives a WHATSAT message, if the server has data on that given client based on its client_id, it will

make an asynchronous request to Google Places API to find nearby attractions using aiohttp. It then responds to the client with a response of the following form:

AT <server_name> <time_diff> <client_id> <location> <time_sent>

<JSON_format_msg>

where JSON_format_msg are the formatted results from the Google Places API Nearby Search request, with a maximum number of results as specified by num_places. The rest of the response is identical to the response to an IAMAT request.

Any client-server requests that do not follow this format are not processed, and return a message of the following form:

? <invalid_command>

1.2. Server-Server Communications

Servers communicate by using FLOOD commands to propogate updated client data. They are of the form:

FLOOD <server_name> <client_id> <location> <client_time_sent> <server_time_received> <original_server> <time_diff>

where server_name is the name of the current server sending the request, original_server is the original server which received information about this given client, and the rest of the parameters are obvious.

When a server first receives a FLOOD message, it updates or adds information about that client then sends the FLOOD message to all servers is communicates with. This flooding stops when a given server recognizes that the information it is receiving has already been processed, and it does not continue to send FLOOD messages. In this way, each server in the server herd receives the updated information about a given client.

2. Encountered Problems

The first problem I ran into is servers would continue to recursively send each other FLOOD requests with the same information. To resolve this, each server checks if it has already received that information, and if it has it will not continue to send FLOOD requests.

Another problem I ran into was that the given test script gave results that varied depending on the ports assigned to given client IDs on the UCLA linux server. As the tests worked fine on my personal machine, I assumed that another student mistakenly used my designated ports.

3. Discussion of asyncio

The asyncio library provides a set of both high-level and low-level APIs to write concurrent code making use of Python coroutines and event loops. It is a cooperative multitasking library, in which tasks decide when to give up control. The event loop is a single threaded approach for concurrency, in which coroutines are stored in a queue and processes sequentially. When suspending a task, such as when waiting for I/O, the event loop can run the next task in the queue to improve performance.

Python 3.8 introduced asyncio.run(), which can be used to execute a coroutine and return the result while automatically managing the event loop. This project uses the old method of manually managing the event loop, which I used because I found more documentation for it before I learned about asyncio.run(). While the old approach works for the purposes of this project, going forward it would be beneficial to use asyncio.run(), as the implementation is significantly more complex than just a shorthand of the old method. According to the Python docs, asyncio.run() should be the preferred way of running asyncio programs.

3.1. Advantages of asyncio

Asyncio is very suitable for this kind of application, as it makes it easy to asynchronously handle events such as connections and requests. Each server maintains its own event loop. Thus, rather than processing a message as soon as it is received, it can simply add it as a coroutine to the event loop and process it later. This allows each server to asynchronously process a lot of requests at the same time, making it ideal in an application where clients can send a lot of requests (i.e. location updates). It can also accommodate a lot of connections at the same time, which is ideal for this kind of scalable application.

Since each server can have its own event loop, the asyncio library allows this server herd design to be easily scalable, as each server uses the same code. In the instance of this project, more servers can easily be added by adding them to the data structure of known and connected servers, then running it.

While the asyncio library only supports TCP and SSL protocols, it is supported by other libraries such as aiohttp, an asynchronous HTTP client/server. Thus, it is easy to implement the Google Places API as was done in this project.

3.2. Disadvantages of asyncio

The biggest disadvantage of asyncio, and more generally of asynchronous models as a whole, is that the order of requests and connections received is non-deterministic. This causes multiple problems that synchronous models do not have to worry about.

In the context of this project, consider the case where a client sends an IAMAT request to server Hill, then a WHATSAT request to server Jaquez. In a synchronous model, since the IAMAT request came before and was therefore processed before the WHATSAT request, it can

guarantee the correct behavior. However, in this asynchronous model, the Hill server must propagate information from the IAMAT request to other servers, and thus this model cannot guarantee that Jaquez has received the most recent information on the client, and thus the return value of the WHATSAT request is non-deterministic. This problem gets worse with a greater number of servers, as there will be more FLOOD requests and thus more variance in the ordering of how to process all the requests.

It is worth noting that although the above means that asynchronous models are less reliable than synchronous models, asynchronous models have significantly better performance. This is because synchronous models can suffer from the performance bottleneck of processing each request as it is received, while asynchronous models can suspend tasks (for instance, when they are waiting for I/O) and thus get more work done.

Another disadvantage of asyncio is the lack of support for multithreading. Since asyncio uses an event loop, which is a single threaded approach for concurrency, a multithreaded model could divide tasks in the event queue to multiple threads and process them in parallel. Thus, the multithreaded model would have much better performance than asyncio's single-threaded approach.

4. Python vs Java

This section discusses the pros and cons of programming languages Python and Java in regards to type checking, memory management and multithreading.

4.1. Type Checking

Python uses dynamic type checking, where type errors are only reported at runtime, while Java uses static type checking, where variable types must be declared and checked at compile-time.

Dynamic type checking avoids the complexities that come with variable types, which can make writing code a lot easier for beginners. This is especially the case in dealing with new libraries or APIs. Thus, Python's duck typing is very useful for this project, as it saves the programmer the time of delving too deeply into the documentation to understand library-specific types.

The advantage of static type checking is that a lot more checking can be done by the compiler, and therefore a lot more trivial errors can be caught without running the program.

4.2. Memory Management

Memory management in Python involves a private heap containing all Python objects and data structures, where the private heap is managed internally by the Python memory manager. To contrast, Java uses an automatic memory management system called a garbage collector. Both are done without the programmer having to implement memory management logic in the application. However, the two languages use different types of garbage collectors. Python uses the reference count method, and Java uses the mark and sweep method.

In the reference count method, each object has a reference count which is modified whenever that variable is accessed. When the number of references to that object is 0 (i.e. no objects can reach this variable) then it is deleted. This approach involves overhead during references and fails with circular references. However, for the purposes of this server-herd project it is ideal, as there are many variables that are allocated frequently and only used a few times. Thus, this approach recovers memory quickly.

The mark and sweep method locates and detects unreachable objects, then frees that space from memory. Since this method has to traverse all reachable objects to mark and sweep, it involves significantly more overhead than the reference count method.

4.3. Multithreading

A big disadvantage of Python is that it doesn't support multithreading. Python's Global Interpreter Lock (GIL), which avoids race conditions in Python's memory management, avoids deadlocks and allows it to use C libraries that are not thread-safe, mean only one thread can execute at a time. Thus, even with multiple cores, Python is limited as it cannot achieve real parallelism.

On the other hand, Java supports multithreading and can thus make use of the benefits of parallelism. In the context of this project, multithreaded support would allow servers to process multiple requests at the same time, greatly increasing performance.

5. Asyncio vs Node.js

This section discusses the pros and cons of the asyncio and Node.js frameworks in regards to performance and concurrency. They are written in Python and JavaScript respectively, and are both asynchronous frameworks for writing server-side code.

It is worth noting that Node.js has the added benefit that many web developers are proficient in JavaScript for front-end design. Thus, Node.js can be a more natural choice as opposed to learning Python.

5.1. Performance

Node.js is based on Chrome's super-fast V8 engine, and is thus significantly faster than Python's asyncio. Python's performance also suffers with memory-intensive programs, but as server-herds are not particularly memory intensive this is not a concern for this project.

5.2. Concurrency

Both Node.js and asyncio use the same single-threaded asynchronous architecture, and are thus similar in regards to concurrency. Where asyncio uses an event loop, Node.js uses somewhat analogous Promise objects. The biggest difference is that Node.js does not have coroutines, which can be halted and modified, meaning asyncio can be more dynamic and flexible.

Conclusion

This report concludes that asyncio is a suitable framework for this kind of application. Asyncio's asynchronous approach is perfect for implementing a server herd, as it makes it easy to create multiple servers which can each process a lot of requests and connections at a time. Python's dynamic type checking also makes it easier for beginners to incorporate APIs and libraries.

Node.js also seems like a suitable framework for this kind of application, but it was not studied in depth.

References

https://docs.python.org/3/library/asyncio.html

https://docs.python.org/3/whatsnew/3.8.html

https://www.freecodecamp.org/news/nodejs-vs-python-choosing-the-best-technology-to-develop-back-end-of-your-web-app/

https://docs.aiohttp.org/en/stable/

https://www.activestate.com/blog/python-vs-java-duck-typing-parsing-whitespace-and-other-cool-differences/

https://www.javatpoint.com/memory-management-in-java

https://docs.python.org/3/c-api/memory.html

https://nodejs.org/en/about/