Application Server Herd Implementation using Python Aynscio

Yuhao Tang - UCLA
billt4ng@gmail.com

ABSTRACT

In an application server herd architecture, new updates are flooded directly between servers, bypassing the need to store all new updates in a central database and route all data requests to the same database server. This paper discusses the merits and drawbacks of using Python 3.6.3's asyncio framework to implement a simple application server herd, based on my personal experience of building a sample implementation as well as research into the mechanisms behind the Python language and asyncio framework. With these pros and cons in consideration I conclude that the asyncio framework would make an effective and efficient choice when implementing an application server herd architecture

Introduction

The underlying motivation behind the implementation of an application server herd comes from the desire to avoid bottlenecking a service architecture at the application servers. In our implementation of the application server herd architecture, we have 5 servers, each with a predetermined set of other servers that it can communicate with. Each server can handle three types of messages: 'IAMAT', 'AT', and 'WHATSAT'. Clients share information about their current location using 'IAMAT' messages, and this location information is flooded to each server along the predetermined paths. Each server is responsible for maintaining its own copy of the location data, and is responsible for propagating this location data when appropriate. Finally, clients can query for information about places near other client locations with the 'WHATSAT' command. Each server is responsible for using the location data stored on itself to query the Google Places API and retrieve the relevant JSON data regarding the corresponding location of the queried client. To start each server, we use the command `python3 server.py [server-name]`

This report will discuss the process of building such an application using Python's asyncio framework. We consider the advantages and disadvantages provided by native Python features such as its dynamic typing, reference-count based memory management system, as well as the implementation of asyncio's asynchronous IO capabilities, when compared to alternative frameworks that exist in the Java language as well as the NodeJs framework. Finally, we will provide the reasoning behind our conclusion that Python's asyncio framework is a good choice for implementing an application server herd service.

1. Implementation Design

1.1 High Level Design

The high level design for each server involves an asyncio event loop that runs a coroutine generated by asyncio's build in `start_server` command. This coroutine accepts incoming connections and calls a callback function on each one, called `handle_message`. This function is responsible for receiving the full incoming message in a read loop, then determines which type of message it has received and awaits the appropriate helper coroutine. If an invalid message is received, it calls the `handle_error` coroutine.

Each server maintains its own copy of location data for clients, which is maintained in a dictionary called `location_data` which maps each unique client name with a 3 element list consisting of the location coordinates, the time that the client sent the message, and the difference in time between when the IAMAT message was received and the AT response by the server was sent. Each server also remembers its own name, as well as the ports of the servers it is allowed to communicate with in a list called `neighbor_ports`.

1.2 Handling IAMAT Messages

The `handle_iamat` coroutine is responsible for handling IAMAT messages. It is given the received message split into list format, as well as the writer to the client. The coroutine then determines if the newly received location coordinate is either for a new client, or if it is the most recent for some existing client. If one of these conditions is met , the new data is stored in `location_data`. Regardless of whether the data is stored or not, an AT message is sent back to the client

Next, if the received data was stored, the coroutine attempts to propagate the new data to its neighbor servers. For each of its neighbor ports, we attempt to create a connection and send a flooding AT message. In our flooding algorithm implementation, each time a server sends out a flooding AT message, it appends its port number onto the end of the message. This way, when servers received flooding AT message in the future, they can determine which servers the message has already been propagated to, to avoid infinite flooding loops. If a connection fails in our implementation, we simply catch and log the connection and continue on, as the spec indicates that we want our server to continue operation even if other servers are offline.

1.3 Handling AT Messages

The `handle_at` coroutine is responsible for handling AT messages. It works similarly to the IAMAT handlers, except that it does not send back any messages to the server that sent the AT message.

First, we check if the newly received data is the most recent for the corresponding client, or if it is for a new client, and if so, we either add it or update the entry in `location_data`. Then, we follow the same procedure as the IAMAT handler when it comes to propagating the information. If the information was accepted into the cache, we attempt to open up connections to the neighbor ports and propagate the message. However, there is one additional condition, where we check the list of ports that the incoming AT message is terminated by. As we noted earlier, each server appends on its own port number to the end of an outgoing flooding AT message. Therefore if the port number of a server is already in the terminating list of an AT message, we do not want to propagate the AT message to them again, to avoid an infinite flooding loop.

We note that our algorithm does not completely avoid redundant AT messages. Due to timing inconsistencies as well as the existence of multiple paths between servers, we still send out occasional redundant AT messages. However, redundant messages will not be processed into the server cache and will not be further propagated, therefore eliminating any possibility of an infinite flooding loop occurring.

1.3 Handling WHATSAT Messages

The `handle_whatsat` coroutine is responsible for handling WHATSAT requests from clients. It takes in the received message split into list format, as well as the writer to the client. We first determine if the request is for a valid client in our cache, and if the arguments for radius and maximum number of items are valid. If not, we call and await the `handle_error` coroutine.

Otherwise, if a valid client is request, we call and await the `query_api` coroutine with the coordinates parsed into Google API format, as well as the search radius in meters, and the maximum number of results we want to receive. This is the main bottleneck of our coroutine, since the query and response to the Google Places API is by far the slowest part of our procedure. After the response is received, we generate the appropriate AT message to the client, and append on the filtered JSON result from the Google Places API.

1.4 Handling Google Places API Query

The `query_api` coroutine is responsible for querying and parsing the response from the Google Places API when we want Nearby Search information regarding a given coordinate.

The coroutine first opens up an ssl connection with the Google Places API. Then we send an HTTP get request using TCP with a registed API Key to the appropriate URI on the maps.googleapis.com host. Next we await the reply from the Google API Server. We note the incoming response has two parts, the HTTP header, and the JSON response data, which is the part we are interested in. To make sure we read until we have received all of the relevant information, we `readuntil` we have hit the '\r\n\r\n' sequence for the second time, and take the data read in during the second iteration of this read. This data

corresponds directly to the JSON data we are interested in.

Finally, the coroutine parses the received JSON data, and filters the results entry to have only up to the number of items that the client originally requested. This data is dumped into string format and returned.

1.5 Handling invalid messages

The `handle_error` coroutine is responsible for handling invalid messages received by the server. This function takes in the received message in split list form, as well as the writer to the client.

This coroutine simply appends a "? " onto the front of the invalid message, and sends that back to the client through the writer transport.

2. Evaluation of Python and Asyncio

2.1 Effectiveness of Asyncio Framework

The asyncio framework provided many useful abstractions when it came to ease of use as well as effectiveness when implementing our server herd. The asyncio framework also comes with solid documentation and community support, making it a good choice for introducing to developers who have not used any asynchronous library before.

The most important abstraction provided by asyncio is the implementation of the event loop and coroutine creation. The `async def` delaration makes it very easy to introduce nonblocking coroutines to our event loop. This combined with the `await` feature makes the process of managing non-blocking function calls very straightforward.

Asyncio also provides several useful abstractions related specifically to TCP IO. Our implementation makes use of the asynchronous `open_connection, start_server` coroutines as well as the asynchronous reads and writes provided by the StreamReader and StreamWriter classes. These classes make it significantly easier to manage asynchronous TCP IO, which was a major part of our server herd implementation.

One shortcoming I encountered with the asyncio framework was its lack of direct support for HTTP requests, which was relevant when we wanted to make asynchronous HTTP requests to the Google Places API. It was very important that these requests were made asynchronously, as the latency of communicating with the Google Places webserver as well as the download of the response JSON data were the two major time intensive operations in our application. Unfortunately asyncio does not provide any abstractions for HTTP requests, which required us to manually establish a SSL TCP connection and manually generate HTTP requests over TCP to query the Google Places API.

2.2 Python vs Java

When it comes to comparing an implementation of a server herd like ours using Python or Java, it is most relevant to consider the differences between typing, memory management, and thread management between the two languages.

When it come to typing, Python uses a dynamic duck-typing system as opposed to Java's statically typed system. In Python , variable names are bound to objects at execution time, and can be bound to different types of objects throughout execution [1]. This makes writing Python simpler than Java, but can make debugging more of a challenge. In our application, the issue of variable typing did not play a major role in the development process due to the relative simplicity of the program. However, on a much larger scale application it is easy to see how a developer might find it much easier to develop using Python's dynamic type system as opposed to Java's much stricter typing. However, there are also people who might prefer a statically typed system due to the more robust debugging and potential for cleaner refactoring in the future.

The major difference between Python and Java's memory management lies in their approaches towards garbage collection. Python utilizes a reference tracking approach when it comes to determining when an object's memory space should be freed [2]. This allows for objects to be freed as soon as they are no longer referenced by any variable, maximizing the use of memory space. Java on the other hand, utilizes a generational garbage collection approach that runs in the background, and sweeps through the memory freeing objects that are unreachable periodically [3]. When comparing the two approaches, reference counting will likely outperform java in terms of memory usage, since objects are freed immediately, whereas generational

garbage collection will likely win out in terms of performance, since it has less of an overhead when it comes to maintaining/constantly checking reference counts for objects. In the case of our application, I would argue that Python's garbage collection strategy is preferable, since on our servers would be much more likely to be limited by memory space rather than execution speed, since each server must maintain its own copy of the database cache on its own, and a processing requests does not require a significant amount of CPU resources.

Finally, we consider the approach to thread management in an application like ours. Python's asyncio framework utilizes a single-threaded, asynchronous approach to non-blocking IO. After a slow IO request is issued, for example, the request to the Google Places API, execution is yielded to the event loop, and another function is allowed to run. In Java, these processes would likely be implemented through multithreading. Where the request to the API is executed in a new thread. In our application, the single threaded asynchronous approach should be greatly preferred, since our tasks are limited by IO-bound rather than processor-bound, meaning that the benefits we would receive from multithreading would be insignificant compared to the cost of managing context switches and scheduling [4]. We can imagine a scenario where each incoming connection to the server spawns a new thread. Since servers likely have to manage huge amounts of users, the overhead from managing thousands or millions of threads would quickly become cumbersome. Furthermore, in our single threaded application we do not have to deal with concerns with maintaining thread safety or shared resources. Since only one thread is being executed at a single time, we don't have to worry about other threads causing race conditions when writing or reading to our server cache.

Overall, I would argue that Python is a preferable alternative to Java in our server herd application, due to its ease of development, memory-efficient garbage collection, and most importantly, support for single-threaded asynchronous IO management.

2.3 Python Asyncio vs NodeJs

Python's asyncio framework and NodeJS share many similarities, as they are both take a single-threaded asynchronous approach to non-blocking IO.

When it comes to performance, NodeJS runs on the powerful Google V8 JIT-compiler, which should generally outperform asyncio when it comes to managing large amounts of users.

When it comes to development efficiency, it likely comes down to personal developer preference. Many developers might favor NodeJS because it means that they can use the same language when building the backend and frontend for their application. Python and NodeJS both have extensive community and library support, although in the category of web development JavaScript appears to be the faster growing community. However, developing with NodeJS and JavaScript often involves utilizing and managing many third party tools that can be difficult to setup and learn, since the NodeJS package itself is fairly minimalistic in nature[5]. There is a strong argument that Python is a much easier to use out-of-the-box solution.

Overall, it is difficult to recommend a solid favorite between NodeJS or Python. In the end, it would likely come down to developer preference, and the exact type of application being built. In a completely backend application like our sample server herd, using Python is likely the easier choice, but in web-application involving UI NodeJS might be preferable.

3. Conclusion

Python's asyncio framework provides a great deal of effective and useful functionality when it comes to implementing IO-bound programs such as our application-server herd. Combined with the ease of development in the Python language, as well as the efficiency of asyncio's single-threaded asynchronous approach to non-blocking IO, I would concluded that it would make a good choice for implementing server herd applications. NodeJS provides a strong alternative, especially in cases where performance plays an important role.

4. References

[1] "Static vs. Dynamic Typing of Programming Languages." *Python Conquers The Universe*, 8 Apr. 2012, pythonconquerstheuniverse.wordpress.com/2009/10/03/static-vs-dynamic-typing-of-programming-languages/.

[2] "Memory Management¶." *Memory Management — Python 3.6.3 Documentation*, docs.python.org/3/c-api/memory.html.

[3] “Diagnostics Guide.” *Understanding Memory Management*, 27 Jan. 2010, docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html.

[4] Brij. “Concurrency vs Multi-Threading vs Asynchronous Programming : Explained.” *Code Wala*, 9 Sept. 2017, codewala.net/2015/07/29/concurrency-vs-multi-threading-vs-asynchronous-programming-explained/.

[5] Bolin, Michael. “JavaScript vs. Python in 2017 – Hacker Noon.” *Hacker Noon*, Hacker Noon, 20 Mar. 2017, hackernoon.com/javascript-vs-python-in-2017-d31efbb641b4.