Project: Investigation into Aysncio as a Server Herd Framework

Computer Science 131, Spring 2019
Professor Paul Eggert
Taylor Lee, *604790788*

Abstract

This report investigates the practicality, efficiency, and feasibility of using the `asyncio` Python library as a framework for a large-scale server system such as the LAMP Wikimedia Architecture. After comparing Python's type checking, memory management, multithreading, and ease of use with Java and with Node.js, I recommend `asyncio` as a viable and efficient server herd framework.

1. Introduction

1.1 asyncio

`asyncio` is a python library primarily used to write concurrent and asynchronous code. `asyncio` is often used to provide the foundation and framework for Python networks, servers, database connections and more. Using the `async/await` syntax, developers can call `asyncio` coroutines that can be run concurrently. This concurrent programming can be called using the `aysncio.create_task()` function, which runs coroutines concurrently as `asyncio` tasks. The capacity for concurrent tasks and coroutines is the core of the `asyncio` library.

1.2 Project Purpose

This server prototype was developed to investigate the merits and practicality of using the `asyncio` library to replace a larger-scale server system (such as the LAMP Wikimedia architecture). This new server herd protype is distinct in several ways: it can accommodate for rapidly-evolving data from mobile devices; it can access a large database server that contains stable information; the individual application servers do not have to have complete connection with each individual server, but holistically the server herd will be able to distribute all new information in an efficient and concurrent manner.

2. Server Herd Prototype

My server herd prototype contains five distinct individual servers, respectively named with the following IDs: Goloman, Hands, Wilkes, Holiday, and Welsh. Each of these IDs refers to a specific port number on the seasnet servers. Because I was assigned distinct port numbers for the seasnet servers, I hard-coded each ID name with an individual port number. Goloman, Hands, Holiday, Wilkes and Welsh correspond to port 12138, 12139, 12140, 12141, and 12142 respectively. To simulate a server herd that doesn't have complete inter-server communication, the servers are limited to the following restrictions:

1. Goloman talks with Hands, Holiday and Wilkes.
2. Hands talks with Wilkes.
3. Holiday talks with Welsh and Wilkes.

These server communication restrictions are bi-directional (ex. If Goloman can talk with Hands, Hands can also talk with Goloman).

2.1 Server Design

Each individual server can be constructed using a command line input such as the following:

```
python3 server.py <server_name>
```

This command line python3 call contains two arguments, 'server.py', which is the name of the server python file, and `<server_name>`, which can be any of the five servers that are listed above. If any command line call lists a server that is not one of the five, the log file will throw an error and exit the `main` function.

To construct an `asyncio` server, I used the `asyncio.start_server()` function. This function creates an `asyncio` server with the port that was given in the command line argument. It actively reads and writes to any client that connects to it.

2.1.1 Inter-Server Connections

My servers communicated through the `asyncio.create_task()` function call. I used this concurrent `asyncio` function in combination with a simple flooding algorithm to ensure concurrent transfer of information between servers. This inter-server communication is the most essential concurrent code because on a larger-scale server herd with more servers and more inter-server calls, it is extremely important that an individual server can contact all its neighboring servers concurrently and not sequentially.

The flooding algorithm I implemented was simple. When a server received a `UPDATELOC` message from another server, it checked to see if the new location, ID, and timestamp

contained in this message was already in its dictionary of locations. If so, it did not update the redundant message and did not send the UPDATELOC message to its neighbors. On the other hand, if the UPDATELOC contents were different from the one found in the locations dictionary, the server updated the location and sent the same UPDATELOC message to its neighbors. This flooding algorithm ensures that the inter-server communications do not enter an infinite loop.

2.2 Client Requests & Server Responses

Clients can connect to servers using the following command line call:

```
nc localhost <portnumber>
```

The portnumber is the port that corresponds with the server ids. In addition to using nc, a client can use telnet or other methods of contacting port, but for the purpose of this project, I tested clients using nc. The servers accepted the following client calls:

2.2.1 *IAMAT*

A client can send the following location call to a server:

```
IAMAT <client_id> <client_location> <client_timestamp>
```

The <client_location> must be given using ISO 6709 notation and the <client_timestamp> must be expressed in POSIX time. When receiving an IAMAT client call, the server adds the client's information to its locations dictionary, sends an UPDATELOC message to its neighboring servers, and responds to the client with the following message:

```
AT <server_name> <time_difference> <client_id> <client_location> <client_timestamp>
```

2.2.2 *WHATSAT*

Additionally, clients can send the following message to a server:

```
WHATSAT <client_id> <radius> <upper_bound>
```

The <radius> argument defines the mile range in which the server should search, and the <upper_bound> argument limits the amount of results the server returns.

After receiving a WHATSAT call, the server queries the Google Places API using a http get call. Since asyncio doesn't accommodate for http calls, my servers called the API by using the aiohttp library. Because Places API requires a key, I hard coded my API key into my code. The server responds to the client with the same AT response as above, followed by the results of the API call in json form (cutting off the results to be within the <upper_bound> limit) .

2.3 Special Conditions

In the case that the WHATSAT or IAMAT calls are in invalid form, the servers respond with:

```
'? <invalid_client_call>'
```

Since the servers are the only ones using the UPDATELOC call, I did not need to check if the UPDATELOC calls were valid.

2.4 Documentation & Logging

All server input, outputs, errors, and inter-server communications are logged in a log file. Each server logs their files in "<server_name>_log.txt", which saves in the local directory that server.py is in. This documentation and logging proved to be an essential way to test if the servers were truly running concurrently. The log entry is formatted as follows (with time being formatted as %d-%b-%y %H:%M:%S):

```
%(asctime)s - %(levelname)s: %(message)
```

3. Evaluation of asyncio

3.1 Usability

Overall, asyncio is an easy library to use. I had no prior experience with asyncio before this project, but after a quick reading of the documentation and source code, I was able to execute a simple server and client. Because asyncio has high-level APIs, much of the code I used was simple. For example, to start the server and have it accept incoming connections, I just needed to use the simple function call: start_server(). The await and async function calls allow for typical function definitions to be called asynchronously, which is a substantial upside to the asyncio library.

In addition to the high-level APIs, asyncio provides low-level APIs that allow developers to retain more control of the loops and coroutines that the library implements. While such low-level APIs were not necessary for this project, they will probably prove to be useful for more extensive frameworks.

3.2 Performance

asyncio allows for functions calls to be performed asynchronously. For this project I took advantage of this concurrency by concurrently calling between servers. When testing the performance of this concurrency, I was impressed by the efficiency that the servers could communicate with each other. To test this concurrency, I added the following line to the function that was being called concurrently:

```
await asyncio.sleep(1)
```

This line calls for the function to sleep for one second before executed the function. Without concurrency, the function

would be called sequentially, and if I called the function four times, a total of 4 seconds would pass during the four function calls. However, because this function was called concurrently using the `asyncio.create_task()` function, all calls to this function were called in one second.

You can see in the following log lines from "Goloman_log.txt" that all three communication calls to Goloman's neighboring servers occur concurrently:

```
06-Jun-19 18:30:55 - INFO: Port Wilkes is down. Cannot send
kiwi.cs.ucla.edu at +34.068930-118.445127 to Wilkes.

06-Jun-19 18:30:55 - INFO: Sending kiwi.cs.ucla.edu loca-
tion at +34.068930-118.445127 to Hands.

06-Jun-19 18:30:55 - INFO: Sending kiwi.cs.ucla.edu loca-
tion at +34.068930-118.445127 to Holiday.
```

While this test was limited to a server herd of only five servers, such concurrency performance is extremely valuable. `asyncio` allows for concurrent server communication, which is a huge bottleneck in other server configurations.

3.3 Disadvantages of asyncio

One of the key disadvantages of `asyncio's` asynchronous calling is the fact that the tasks are run asynchronously and the calls are not run in order. While this disadvantage is not obvious in server herds that are small, scaling this server may prove to be difficult because of such out of order communications between servers. There may be some race conditions between the servers depending on the amount of the servers and the extent to which users are updating their location. However, this disadvantage is not extremely severe, and with a server herd of that scale, a developer can create tests and functions that account for the asynchronization.

3.4 Comparison with Java

Java and Python are two of the most popular coding languages currently used. However, they differ on some basic levels, and when creating a network of servers, it is important to determine which language would be best suited for the networks needs.

3.4.1 Memory Management

Python's memory management differs from Java's in that each variable refers to an object, and when that object has a reference count of 0, the garbage collector collects it. On the other hand, Java allocates memory for variables depending on the type of variable and instantiates that memory with the value assigned to that variable. This fundamental difference in memory management actually favors Python's implementation in my server herd prototype.

For my prototype, since I only use variables once or twice in each function, Python's memory management is advantageous because there is not much extensive overhead in collecting objects with a reference count of 0 and because memory is continuously being freed through the garbage collector. The servers will be allocated more memory and will have greater bandwidth for more client calls.

3.4.2 Multithreading

Unlike Java, Python does not support multi-threading, even on machines that have multiple cores. This comes at a huge disadvantage for hosting servers, as the capacity to multithread increases the throughput that servers can intake. In this comparison Python falls short of Java because Java can support multi-threading, which allows for the servers to have greater bandwidth for the client calls it receives.

However, the `asyncio` library proves to be a nice workaround for Python, as it allows concurrent programming, much in the same way multi-threading in Java allows for the same amount of work to be done in less time. Both are viable ways of increasing efficiency in an asynchronous manner.

3..3 Type Checking

Python and Java also differ in the way that they check types. Java supports compile-time checking whereas Python supports dynamic checking. In developing this server prototype, I found Python's lack of static checking detrimental at times, especially since in order to make sure that all elements and functions of a server were correct, I needed to specifically check each individual part of the code dynamically. Many times, I would receive a type error only during a test whereas Java would have warned me of the type error at compile time. This made debugging a little cumbersome and frustrating.

However, because Python is so flexible with its type checking at compile time, programming in Python is extremely simple. I did not have to explicitly declare the specific type of certain variables as long as I knew that they were of the same type and would not cause a type error when tested. On the other hand, if I were coding in Java, I would have to know the specific type of every variable and object that I used in creating my server. While such specificity may lead to a more nuanced understanding of low-level Java APIs, Python's type flexibility proved to be very easy to use.

3.4 Comparison with Node.js

Node.js is another framework for server/client communications. Node.js is extremely popular and is written in JavaScript. Node.js is primarily based on the concept of event-driven programming, where the flow control of a program is reliant on a user's events (such as a mouse click or input). This allows Node.js to be a suitable server host, as such a framework is scalable and still efficient.

Like Python, Node.js does not support multithreading. This is another disadvantage for concurrent programing, as multi-threading in a language like Java allows for more efficiency and performance. However, like Python, Node.js makes up for this disadvantage by supporting concurrency via asynchronous calls. At times, this concurrency allows Node.js to be even more efficient and faster than other multi-threaded servers. Additionally, salient to Node.js is its non-blocking I/O, which allows it to maximize CPU and memory.

After reviewing Node.js and asyncio, I conclude that both are suitable frameworks for various server applications. Node.js may be preferred for projects that include front end development, since JavaScript is the only language used throughout the entire development, whereas Python may be preferred when usability and clarity is preferred, since Python is a highly readable and understandable language.

4. Conclusion

Through this server herd prototype as well as an investigation into Python and the `asyncio` library, I see that Python and its `asyncio` library is an extremely suitable library for a server architecture such as our server herd. Because `asyncio` allows for concurrency, server-server communications are extremely efficient. Additionally, Python is an extremely understandable and easy language to develop in. After comparing Python to Node.js, I cannot come to a definitive conclusion on which framework would be better suited for our server herd. However, after a precursory look into Node.js, I see that Node.js is an extremely useful server framework, and further investigation may reveal more advantages.

5. References

Eggert, Paul. *Project. Proxy Herd with Asyncio*, web.cs.ucla.edu/classes/spring19/cs131/hw/pr.html.

"Event-Driven Programming." *Wikipedia*, Wikimedia Foundation, 11 Apr. 2019, en.wikipedia.org/wiki/Event-driven_programming.

"Node.js vs Python Comparison: Which Solution to Choose for Your Next Project?" *Netguru Blog on Python*, www.netguru.com/blog/node.js-vs-python-comparison-which-solution-to-choose-for-your-next-project.

Real Python. "Async IO in Python: A Complete Walkthrough." *Real Python*, Real Python, 29 Apr. 2019, realpython.com/async-io-python/.

"Streams¶." *Streams - Python 3.7.3 Documentation*, docs.python.org/3/library/asyncio-stream.html#asyncio.start_server.