CS131 Python Project Report

Shreya Raman, *UCLA*

Abstract

The purpose of this project was to examine whether or not *asyncio* is an effective way to implement an application server herd. To test this, we created a parallelizable proxy for the Google Places API. We analyze the benefits and fallbacks of *asyncio* in addition to comparing its suitability with Java and Node.js. We come to the conclusion that due to the nature of this application, being small-scale and single-threaded, *asyncio* is a good match.

1. Introduction

There are five servers in this server herd: Goloman, Hands, Holiday, Welsh, and Wilkes. Each server is assigned to ports 12040, 12041, 12042, 12043, and 12044, respectively. They can bi-directionally communicate with each other as follows

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

Clients can communicate with these servers using an *IAMAT* command. Information sent from the client to the server is propagated to the other servers with a flooding algorithm and an *AT* message. Clients can also communicate with these servers using a *WHATSAT* command, which requests information about nearby places to other clients.

2. Commands

The client can communicate with the server via *IAMAT* and *WHATSAT*. The server responds to the client via *AT*.

2.1 IAMAT

The *IAMAT* commands are sent from the client to the server and come in the form

IAMAT <client_name> <location> <time_sent>

in which *client_name* is the name of the client device that sends the message, *location* is the latitude and longitude coordinates of the client device in ISO 6709 notation, and *time_sent* is the time at which the client sent the message.

The server responds to this with an AT message as follows

AT <server_name> <time_diff> <client_name> <location> <time_sent >

Most of these parameters are the same from the *IAMAT* command, with the exception of *server_name* and *time_diff*. *server_name* is the name of the server which the client is connecting

to, and *time_diff* represents the time difference between the client sending the message and the server receiving the message.

When AT messages are sent between servers, they are as follows

AT <client_name> <server_name> <latitude> <longitude> <time_diff> <command_time>

2.2 WHATSAT

The *WHATSAT* commands are sent from the client to the server to get information about nearby places, and they come in the form

WHATSAT <client_name> <radius> <info_amount>

Again, *client_name* is the name of the client device that sends the message. *radius* is the radius around the client's last known location. *Info_amount* is the number of results that will be returned. To get these results, we use the Google Places API with an asynchronous HTTP request. This is done with *asyncio* and *aiohttp*. The format of this query is

https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=%s,%s&radius=%d&key=%s

The server responds to this with an AT message as follows

AT <server_name> <time_diff> <client_name> <location> <time_sent> <json_dump>

This is the same message that is being sent in response to a *IAMAT* command, with an addition of *json_dump*, which is a formatted JSON message.

3. Asyncio

Asyncio is an asynchronous Python library that is used to write concurrent code, and it supports server-server and server-client TCP connections. The main components of *asyncio* are coroutines and event loops. And event loop runs only one task at a time, and pauses a task when another coroutines needs to run. *Asyncio* has both pros and cons, but the benefits outweigh the drawbacks and it is a suitable choice for this project.

3.1 Pros

The *asyncio* library is fairly easy to use to implement the server herd because its documentation was readable and it had multiple APIs. Since it is asynchronous, the performance is fast; multiple requests can be processed because of *asyncio*'s use of coroutines and event loops. Synchronous code, on the other hand, would take a long time to process a high volume of queries and TCP connection requests.

Furthermore, *asyncio* works great with *aiohttp*, another Python module. These two modules working together allows for sending HTTP requests with *WHATSAT* commands.

3.2 Cons

While the asynchronous manner creates advantages, there are also drawbacks. With asynchronous code, multiple tasks

are run on a single core, meaning that a bottleneck could occur if there is too high a volume. Since multithreading is not supported, processes cannot be executed in parallel and this would be a problem in a situation with significantly more requests and servers.

Additionally, I ran into some difficulties while debugging. I had to use several print statements which created confusion because of the lack of order in processing.

4. Comparison to Java

Java is a popular programming language that has features quite different from Python. To address the question of which programming language is better suitable for this server herd, we compare the two languages in their type checking, memory management, and multithreading abilities.

4.1 Type Checking

The difference between Python and Java is that Python does dynamic type checking while Java does static type checking. In dynamic typing, the variable type is determined only at runtime. In static typing, on the other hand, variables types are checked during compile time. Java's static type checking makes code more descriptive and easier to debug. Additionally, there would be fewer mismatch errors during runtime because the errors would be caught during compile time. However, it also means that when writing code, you have to know types for all the variables, functions, and so on. With Python, you could set up the server without knowing the types, so its method of type checking makes Python more flexible than Java, especially for this project.

4.2 Memory Management

Python and Java both use garbage collection to allocate memory, but there are some differences. Python uses a link/reference count method, in which each object has a count that depends on the number of times that object is being referenced by another object. This object can only be de-allocated when its count is at zero. Java, on the other hand, uses the mark and sweep method. It sweeps through the objects and marks objects that are unreferenced, and then sweeps again to delete these marked objects. Because of this two-fold garbage collecting process, Java is slower than Python. Since variables are created and deleted often in this server herd project, Python surpass Java in terms of memory management.

4.3 Multithreading

Java uses multithreading and has support for concurrency and parallelism, which would be helpful for the server herd given the amount of requests being sent to the servers. Python, unfortunately, does not support multithreading. Instead, it uses a Global Interpreter Lock which prevents multiple threads from executing on an object simultaneously, therefore blocking parallelism. Due to this, Java would be better suited.

5. Comparison to Node.js

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine, and like Python, it writes server-side asynchronous code and is dynamically typed. It is widely used in web development and has many other similarities to Python. The Promise class of Node.js is analogous to asyncio's Future class, and async/await is implemented the same way.

One difference is that Node.js is inherently asynchronous so everything is automatically handled in an asynchronous manner. Python is only asynchronous when using the *asyncio* library. For this reason, using Node.js would make it easier on the developer to program this server herd. Nevertheless, this isn't a huge advantage since *asyncio* allows Python to be asynchronous. Additionally, Python is more robust and extendable with its multiple libraries. Therefore, Python better supports applications that use multiple libraries and better supports the server herd.

6. Conclusion

Based on the conducted research, Python's *asyncio* has proved to be the most suitable for implementing a small-scale application server herd like this project. The asynchronous framework efficiently handles a large number of requests, and along with its fast performance, automatic garbage collector, and dynamic typing makes *asyncio* a practical choice.