CS131 Project: Proxy herd with asyncio

name
University of California, Los Angeles

Abstract

Python recently introduced a package called asyncio, which enables the code to execute asynchronous I/O. This is very useful because it could potentially speed up the application dramatically. In fact, when we compare asyncio package with other options available (e.g. Java and Node.js), Python with asyncio package seems to be a pretty good choice for developing an application server, specifically a proxy herd.

Introduction

An application server herd is an architecture, where multiple application servers communicate directly to each other as well as via the core database and caches. This is useful when we want to fetch constantly changing and needed data. For example, in this project, we will be developing a server herd, where users can ask the server for places near their GPS locations. Users will continuously send their location to a server. Users can request a server for information about places near their location, and the server must respond to it using Google Places API. It would be too slow if we were to store the user location on the database since the data is rapidly changing (i.e. the server must request the location to the database frequently). Thus, we can let the servers talk to each other and propagate the user location to every other server. We will be using Python's asyncio for this project. Later we will discuss the pros and cons of this framework compared to a Java-based approach. We will also briefly talk about how it would be different if we use Node.js.

1. Specification

Let's briefly talk about how we are going to set up the project. Essentially, we want the users to be able to send their GPS locations to the application server herd. The server will make an Google Places API request on behalf of the client and send its response back to the users.

1.1. Servers

We will have five servers in the herd, named Hill, Jaquez, Smith, Campbell, and Singleton. They communicate to each other as follows:

- Hill talks to Jaquez and Smith
- Jaquez talks to Hill and Singleton
- Smith talks to Hill, Campbell, and Singleton
- Campbell talks to Smith and Singleton
- Singleton talks to Jaquez, Smith, and Campbell

Notice if server A talks to server B, then server B talks to server A (i.e. the communication is bidirectional).

1.2. Commands

There are three commands the servers must be able to handle. First, it must be able to handle the IAMAT command. This command has format as follows:

IAMAT <clientid> <latitude><longitude> <time>

The clientid uniquely identifies an user. The user location is represented as latitude and longitude. The time represents the time the message was sent from client to server in POSIX format. The servers must store the user location and respond to the user as follows:

AT <serverid> <timediff> <clientid> <latitude><longitude> <time>

The serverid uniquely identifies the server that received the request from the client. The timediff represents the difference between the time the server received the request and the time the client sent the request. Second, the servers must be able to handle WHATAT command. This command has format as follows:

WHATSAT <clientid> <radius> <upperbound>

The radius specifies how far from the user we want to search for places. The upperbound specifies the maximum number of places the client wants to get as the response. The server will then use Google Places API and respond as follows:

AT <serverid> <timediff> <clientid> <latitude><longitude> <time>

Basically it's the same response as the one for IAMAT command except we also send the response from the Google Places API to the client. Finally, the server must be able to handle AT command. AT command is not sent from the client but from another server. When a server receives an IAMAT command to add/update an user location, that information must be propagated to neighboring servers (i.e. servers where communication is established) by AT command. This command has the format as follows:

AT <serverid> <timediff> <clientid> <latitude><longitude> <time>

Note it is exactly the same as the response for IAMAT command. In fact, even the data is the same, but timediff and time are not supposed to be used. We will only need the clientid and the location to propagate the information we want. For any other invalid commands including wrong arguments and invalid clientid, the server should respond as follows:

? <command>

where command is the data sent from the client (i.e. message the server couldn't process).

2. asyncio

Asynchronous IO is the idea that when a process is waiting for IO to be completed, we should execute different processes in the meantime to keep the CPU busy. This allows us seemingly to be running concurrent code with a single thread and a single CPU core. asyncio is a Python package that does this for us. A very important concept we need to know about asyncio is a coroutine. A coroutine is a function that can stop its execution at any point and come back to it later. When a coroutine suspends its execution, it gives the control to another coroutine. Event loop will take care of the monitoring of the coroutines for us. It checks if a coroutine is idle and should be switched to another coroutine. Consider a simple example below:

```
async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

await asyncio.gather(count(), count(), count())
```

There are three coroutines running in this program. When it encounters the keyword "await", the coroutine gives up the CPU and lets another coroutine execute its code. Thus, the output of this program looks as follows:

```
One
One
One
Two
Two
Two
```

Let's explore where this behavior of async IO can be helpful/not helpful in the project.

2.1. Advantage

Async IO is very powerful and useful for this project for reasons as follows:

- When the server writes to the buffer for response or waiting for the request from the clients, it's wasting the CPU time. However, if we use async IO, the server can switch to different coroutines, and effectively continuously receive requests from clients without the need to complete the execution from previous request.
- When the server tries to connect to neighboring servers, it takes some time for the connection to be established and every connection takes different time. Async IO allows us to attempt to establish the connections all at once (i.e. each attempt is a coroutine). Thus, we can flood the user location data all at once. No need to flood them one by one.
- When the server makes a Google Places API request, we don't want to keep waiting for the response and do nothing. With aiohttp Python package, which is designed to work well with asyncio package, we let the API request be a coroutine. Thus, when the server waits for the response of the API call, it can switch the task to another coroutine.

2.2 Disadvantage

Async IO can also introduce some problems regarding the order of the execution. We don't know in what order the coroutines are executed, which makes it extremely difficult for debugging and may cause unexpected behavior. In the project, the user must send an IAMAT message to the server to send user location data first. After that, the user can send a WHATSAT message to the server to request places near the user location, which the server received previously. However, since every user request is binded to a coroutine, WHATSAT message could be processed first before IAMAT message is processed. The table below depicts the scenario where it could be problematic.

Client send	Server process
IAMAT message WHATSAT message	WHATSAT message IAMAT message

From the client's perspective, the server should respond with some places based on the user location. However, from the server's perspective, the WHATSAT message contains unknown clientid (because it is processed before IAMAT

message), and thus cannot process the request and send an error message to the client.

2.3. Usage of asyncio As Server Herd

To wrap up, asyncio is suitable for server herd application because it avoids the CPU from being idle, and thus maximize the performance of the server in terms of handling requests from the client. There is a little downside of it in terms of order of execution, but the user can simply make the request again (e.g. send WHATSAT message again), so the problem is not severe.

3. Python v.s. Java

Now, let's compare the difference between Python implementation of this project v.s. Java implementation of this project. We will focus on language-related issues such as type checking, memory management, and multithreading.

3.1. Type Checking

Python is dynamically typed language, meaning type of variables are determined during runtime. On the other hand, Java is statically typed language, meaning type of variables are determined during compile time. Statically typed languages have better performance because the types are determined during compile time, the compiler can optimize the code. Also, it makes the programmer's life easier for debugging because types of every variable is known. Although dynamically typed languages lose all the benefits just mentioned, it has its own advantages. Dynamically typed languages are less verbose (since there is no need to declare the types), which makes the code easier to read. Another benefit of Python is duck typing, which states that the class of an object is less important than the attribute/method it defines. For example, when we call x.quack(), we don't care if the class of x is actually Duck. As long as it has quack funcion, we are allowed to do so. This would have taken a lot of extra code to implement in statically typed languages. However, this can be easily done in dynamically typed languages like Python. At the end of the day, choosing statically typed language or dynamically typed language comes to the preference of the programmers. It's about whether they want the code to be more flexible or more under the control in terms of types.

3.2. Memory Management

Both Python and Java have automatic memory management systems, namely garbage collection. Unlike in C++, programmers don't need to explicitly free the objects in the heap. Garbage collection will find unused objects and free the memory for us. The difference between Python's garbage collection and Java's garbage collection lies in its implementation. Java's garbage collection uses an algorithm known as mark-and-sweep algorithm. The algorithm traverses all object references and marks every object found as alive. All of the heap memory that is not marked as alive are freed. The disadvantage is that it consumes additional memory to run this algorithm. Python's garbage collection uses an algorithm called automatic reference counting. Every object will contain an information of how many references it has. When the reference count becomes 0, we free the object.

Generally Java's mark-and-sweep algorithm is better than automatic reference counting in terms of because automatic reference counting does not handle retain cycle. It's the programmer's responsibility not to have a retain cycle when writing the Python code. Thus, for reliability reasons, Java's memory management seems like a better option.

3.3. Multithreading

Python does not work well with multithreading. Generally speaking, multi-threaded programs in Python are slower than single threaded programs. This is because Python uses Global Interpreter Lock (GIL) to implement automatic reference counting. When a count of the object is updated, it is susceptible to a race condition. We don't want two threads trying to update the count at the same time. We prevent this race condition by introducing GIL, which essentially locks the entire interpreter while the count is updated (because adding locks for every object is costly and could cause deadlocks). Therefore, Python is not suitable for multithreading programs because most of the time the thread will be competing for GIL. Java, on the other hand, works well with multi-threaded applications and makes use of the hardware to its full potential.

4. asyncio v.s. Node.js

Just like Python's asyncio package, Node.js allows us to execute asynchronous code except it is designed to be event-driven and enables asynchronous I/O. Also Node.js is interpreted very fast due to V8 engine, which is heavily invested by Google. Thus, Node.js application runs faster than Python application with asyncio. In terms of the type of the project we are doing, it boils down to the personal preference of Python v.s. JavaScript.

Conclusion

For the type of the project we are doing, Python with asyncio package is a solid choice. It lets the server continuously process the requests from the clients as well as communicating with other servers. Compared with different

languages such as Java, it does have some disadvantages, but also have some advantages over them such as readability and quickness to develop an application. With an introduction of asyncio package, although Python is still slower than Node.js, it can execute asynchronous code just like Node.js. Considering the flexibility and popularity of Python, the performance issue is negligible/acceptable.

References

[1] Solomon, Brad. Async IO in Python: A Complete Walkthrough. https://realpython.com/async-io-python/

[2] Gros-Dubois, Jonathan. Statically Typed vs Dynamically Typed Languages. https://hackernoon.com/statically-typed-vs-dynamically-typed-languages-e4778e1ca55

[3] Vantol, Alexander. Memory Management in Python. https://realpython.com/python-memory-management/#garbage-collection

[4] Sridharan, Mohanesh. Garbage Collection vs Automatic Reference Counting. https://medium.com/computed-comparisons/garbage-collection-vs-automatic-reference-counting-a420bd4c7c81

[5] Ajitsaria, Abhinav. What Is the Python Global Interpreter Lock (GIL)? https://realpython.com/python-gil/#why-was-the-gil-chosen-as-the-solution

[6] Romanyuk, Oleg. NodeJS vs Python: How to Choose The Best Technology to Develop Your Web App's Back End. https://www.freecodecamp.org/news/nodejs-vs-python-choosing-the-best-technology-to-develop-back-end-of-your-web-app/