Report on CS 131 Project: Proxy herd with asyncio

Hermmy Wang
704978214

University of California, Los Angeles

Abstract

The project is expected to provide a Wikimedia-style service that can accommodate frequent updates, access via various protocols and mobile clients. Therefore, we need a server application that can support these functionalities as efficiently as possible. An "application server herd" is an architecture where clients can send queries to one server and multiple application servers can communicate directly to each other without talking to the database. In this project, the server herd architecture is implemented in Python using the asyncio asynchronous networking library. The asyncio library provides APIs to support concurrency for high-performance network and webservers, database connection libraries, distributed task queues, etc. [1]

This paper introduces the design of the server herd application, analyzes the pros and cons of using the asyncio library in Python to implement the proxy herd, addresses several techinal issues regarding the special features of this library, and compares the application's performance with other frameworks. We have reached the conclusion in the end that the asyncio library is a suitable framework to implement the Wikimedia-style service with multiple servers that require rapid intercommunication.

1. Introduction

Wikipedia and Wikimedia projects are run from multiple servers. The Wikimedia system architecture uses a LAMP platform based on Linux Virtual Server (LVS), Varnish caching proxy servers in front of Apache HTTP server, PHP/JavaScript to support performance and reliability. This project aims to build a new Wikimedia-like server proxy that can support faster and more frequency article updates, access from various protocols other than HTTP, and more mobile clients.[2]

For faster transmission and better performance, we need asynchronization. In general, if we execute in Synchronous manner i.e one after another we unnecessarily stop the execution of those code which is not depended on the one you are executing. Asynchronous does exactly opposite, asynchronous code executes without having any dependency and no order. This improves the system efficiency and throughput. [8]

The server herd is implemented using Python 3.7.2. Asyncio is a library in Python to write concurrent code with the async/await syntax, where async is used in defining functions and await is used in calling the async functions. There are high-level APIs and low-level APIs available in this library: high-level APIs are primarily used to start servers and open connections, run coroutines currently, perform interprocesses communications, and synchronize; low-level APIs are used to create event loops and implement protocols. [1] In the following sections, I will introduce how the program utilizes both the high-level APIs and the low-level APIs of the asyncio library to implement the server herd.

2. Server design

2.1 Prototype

The prototype consists of five servers in the proxy herd: "Goloman", "Hands", "Holiday", "Welsh", and "Wilkes", where Goloman talks to Hands, Holiday, and Wilkes; Hands talks to Wilkes; and Holiday talks to Welsh and Wilkes. The communication between servers are bidirectional.

Server	***Connnections***
Goloman	Hands, Holiday, Wilkes
Hands	Goloman, Wilkes
Holiday	Goloman, Welsh, Wilkes
Welsh	Holiday
Wilkes	Goloman, Hands, Holiday

The program is built using the following command:

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

2.2 Commands and Messages

IAMAT

IAMAT <client ID> <+/-latitude+/-longitude> <time>

The IAMAT command is sent by the client to tell the server its location and time. The server will then add the client to the clients and propagate the message to its connections.

WHATSAT

WHATSAT <client ID> <information amount> <radius>

The WHATSAT command is sent by the client to query the server for the information near the given client's location within the specified radius. Since we use the Google Place API, the maximum amount of information cannot exceed 20 entries, and the maximum radius cannot exceed 50km.

Error message

? <unknown command>

When the server receives an invalid command from the client, it outputs "?" followed by the invalid command. Brief debugging information is also output to the standard error.

3. Advantages of using asyncio

Asynchronization allows the CPU to do other useful work while the it waits for other slow operations such as I/O and network requests.

We have primarily used streams primitives in the high-level APIs of the asyncio library because streams are associated with network connections. Sending and receiving messages with streams are more convenient without detailed and complicated low-level code. I have used the following stream functions in the program:

asyncio.open_connection(host_address, port_number, event_loop)

This function establishes a network connection and returns (reader, writer) objects. If the connection succeeds, the program continues to write encoded message to the designated port.

asyncio.start_server(handle_query_function, host_address, port_number, event_loop)

This function starts a socket server. Whenever a client is initiated, the function handle_query_function that takes a pair of StreamReader instance and StreamWriter instance will be called.

We can see from the above stream function usage that the asyncio library provides very handy interface to establish safe and controllable network connections.

The event loop will continue running until the instance of the Future object has completed.

The key feature of the asyncio library is the event loop, which runs asynchronous tasks and callbacks, perform network I/Os, and run subprocesses. An event loop is created to start the server.

We want the server to be on until the user presses the keyboard interrupt (^C). Therefore, the event loop of running the server should run forever and only stops when the program detects a keyboard interrupt. After the event loop is stopped, we can safely close the loop at the end of the program.

The asyncio library also provides Future objects which connects low-level callback-based code with high-level async/await code. The program primarily uses the future function

asyncio.ensure_future(flood_function(param))

The function takes the return value of my function that implements the flooding algorithm as its argument.

These low-level APIs are the essential underlying features of the asyncio library. The event loop

4. Disadvantages of using asyncio

First, since the asyncio framework is asynchronous, the tasks are not processed in their arriving order. Out-of-order task processing can cause unexpected error and makes the program harder to debug. For example, if we have two clients communicate to the same server at the same time. One client introduces a new client ID via the IAMAT command, while the other client queries WHATSAT that newly created client ID. Since the tasks arrive at the same time and the program is asynchronous, it is possible that WHATSAT command is processed before IAMAT. This causes an error since the client ID is not stored in the designated server database yet.

Also, the asyncio library provides framework for writing single-threaded concurrent code using coroutines. The library only supports single-threaded programming but not multi-threaded programming or parallelism. Concurrency does not require multiple CPUs; instead, on a single CPU, concurrency allows an application to work on multiple tasks at the same time. A single CPU does not need to finish a task completely before starting on the next task. On the other hand, in parallelism, the tasks in an application are divided into smaller pieces which are executed on multiple cores at the exact same time. Parallelism can greatly increase an application's efficiency. Since the asyncio library only supports concurrency, the program cannot benefit from running on a multiple-core machine.

tho

5. Python vs. Java

5.1 Type-checking issue

Python is a dynamically typed language, while Java is a statically typed language. Python checks the variable type only when the code starts running. The variable's type can change dynamically throughout its lifetime, and no type needs to be specified in the variable declaration. On the other hand, Java checks the variable type during the compile time, and the type is not allowed to change in general. In the project, dynamic type checking of Python allows easy parsing of the received message. Since the message is received as a string, I can easily split the string into words and cast the desired word into a floating number. However, dynamical type checking introduces more type errors since the bugs are not notified during the compile time.

Since the asyncio library does not support parallelism, the risk for race conditions is significantly reduced. The tasks are processed sequentially, which prevent the memory to be changed by multiple threads at the same time. [2]

5.2 Memory management issue

Unlike C/C++ where programmers need to manually deallocate dynamically allocated objects, both Python and Java have garbage collectors that take care of this issue. The variables in Python are all dynamically allocated on the heap. Python utilizes link count to keep track of the free space: whenever a variable is created in the memory, the link count to that memory location is incremented. When the link count is decremented to zero, the memory space is considered freed.

On the other hand, only objects are on the heap in Java. The garbage collection algorithm implemented by Java is "mark-and-sweep", where the garbage collection process is divided into two phases. When an object is first initialized, its mark bit is set to 0 which indicates it is not yet reachable. During the mark phase, its mark bit is set to 1 which indicates this memory location is now reachable. A depth first search is required to perform this marking phase. During the sweep phase, all the unreachable objects are cleared. The objects whose mark bit is 0 are cleared so that we can reclaim the heap memory.

Overall, both garbage collection algorithms have disadvantages regarding consuming additional memory (RAM) to run those algorithms and this has a major performance impact. Further, the objects aren't garbage collected at the very instant. It takes its own time it has a performance impact as well.

5.3 Multithreading issue

In Python, the threading module is available to do multithreading programming. The async/await syntax was introduced in PEP492. The async def syntax marks a function as a coroutine. Internally, coroutines are based on Python generators, but aren't exactly the same thing. Coroutines return a coroutine object similar to how generators return a generator object. Once you have a coroutine, you obtain its results with the await expression. When a coroutine calls await, execution of the coroutine is suspended until the awaitable completes. This suspension allows other work to be completed while the coroutine is suspended "awaiting" some result. In general, this result will be some kind of I/O like a database request or in our case an HTTP request. [12]

On the other hand, Java provides Thread class to achieve multithreaded programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface. [11]

In comparison, Java uses multiple cores but provides no thread safety. Python has GIL (global interpreter lock) and will only use one CPU-level thread of execution; therefore, on the machine level, Python does not have multithreading but rather concurrency.

6. Python vs. Node.js

The approach used by Python and Node.js to do asynchronization is very similar in their nature. Both Python and Node.js are dynamic languages, which in general do not execute code in parallel. They achieve I/O optimization via asynchronization rather than parallelism. Node.js is asynchronous in nature as its design pattern ensures non-blocking code execution. Python achieves concurrency via async/await coroutines. Regarding performance, Node.js outperforms Python since Node.js is based on Chrome's V8 engine which is more powerful. Python's performance is slowed down by memory-intensive programs.

7. Conclusion

In conclusion, the project successfully provides a Wikimedia-style service that can accommodate frequent updates, access via various protocols and mobile clients. Therefore, we need a server application that can support these functionalities as efficiently as possible. An "application server herd" is an architecture where clients can send queries to one server and multiple application servers can communicate directly to each other without talking to the database. In this project, the server herd architecture is implemented in Python using the asyncio asynchronous networking library. The asyncio library provides APIs to support concurrency for high-performance network and webservers, database connection libraries, distributed task queues, etc. This paper introduces the design of the server herd application, analyzes the pros and cons of using the asyncio library in Python to implement the proxy herd, addresses several techinal issues regarding the special features of this library, and compares the application's performance with other frameworks. We have reached the conclusion in the end that the asyncio library is a suitable framework to implement the Wikimedia-style service with multiple servers that require rapid intercommunication.

8. Reference

[1] The Python Standard Library » Networking and Interprocess Communication » asyncio — Asynchronous I/O.
https://docs.python.org/3/library/asyncio.html
[2] Wikimedia servers
https://meta.wikimedia.org/wiki/Wikimedia_servers
https://docs.python.org/3/library/asyncio.html
[3] Hjelle, Geir Arne. Python Type Checking (Guide)https://realpython.com/python-type-checking/
[4] Wikimedia architecture
Mark Bergsma
https://upload.wikimedia.org/wikipedia/labs/8/81/Bergsma_-_Wikimedia_architecture_-_2007.pdf
[5] Mark-and-Sweep: Garbage Collection Algorithmhttps://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/
[6] Garbage Collection vs Automatic Reference Counting. Mohanesh Sridharan
https://medium.com/computed-comparisons/garbage-collection-vs-automatic-reference-counting-a420bd4c7c81
[7] PYTHON VS NODE.JS: WHICH IS BETTER FOR YOUR PROJECT. https://da-14.com/blog/python-vs-nodejs-which-better-your-project
[8] Asynchronous programming in Node.js.
https://codeforgeek.com/asynchronous-programming-in-node-js/
[9] Cameron SimpsonMulti-threading in Python vs Java https://mail.python.org/pipermail/python-list/2013-October/657460.html
[10] Multithreading in Python.
https://www.geeksforgeeks.org/multithreading-python-set-1/
[11] Python Multithreading and Multiprocessing Tutorial. https://www.toptal.com/python/beginners-guide-to-concurrency-and-parallelism-in-python
[12] Multithreading in Java.
https://www.javatpoint.com/multithreading-in-java
[13] Andrei Notna. Intro to Async Concurrency in Python vs. Node.js.
https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36