CS131 Project Report: Proxy Herd with asyncio

Zhengtong Liu
UID: 505-375-562

Abstract

In this project, we implement an application server herd in Python using the asyncio asynchronous networking library. Additionally, we researched to evaluate whether asyncio is a suitable framework for this kind of application. We also compare the overall approach of asyncio to that of Node.js.

1. Introduction

Wikipedia and related websites are based on the Wikimedia server platform, which is based on Debian GNU/Linux, the Apache web server, the Memcached distributed memory object cache, the MariaDB relational database, the Elasticsearch search engine, the Swift distributed object store, and core application code written in PHP+Javascript. While this works well for Wikipedia, the Wikimedia application might not be suitable for a new Wikimedia-style services for news, where updates to articles will happen more often, access will be required via various protocols, and clients will tend to be more mobile.

Therefore, we look into an "application server herd" architecture, where the multiple applications servers can communicate directly to each other as well as via the core database and caches. The inter-server communications are designed for rapidly-evolving data whereas the database server will sill be used for more stable data that is less often accessed. We will also investigate the asyncio asynchronous networking library in Python, which allows an update to be processed and forwarded rapidly to other servers in the herd due to its event-driven nature. In this project, we will do some research on the asyncio library as a potential framework for the application server herd by (1) building a prototype proxy herd, where five servers communicate to each other, send and receive TCP messages from clients, and make HTTP requests for location information from the Google Places API and (2) evaluating the efficiency and suitability of this library in this kind of application by comparison with a Java-based approach.

2. Implementation of Server Herd Prototype

We implemented a simple server herd prototype for the Google Places API. The prototype consists of five servers, with server IDs, 'Riley', 'Jaquez', 'Juzang', 'Campbell' and 'Bernard'. The five servers communicate to each other bidirectionally as follows: 1) Riley talks with Jaquez and Juzang, 2) Bernard talks with everyone else but Riley and 3) Juzang talks to Campbell. Each server accepts TCP connections from clients, and clients would send IMATAT or WAHTSAT requests to the servers. Valid messages would be responded by the servers specified below, and invalid messages should be sent back to the client with a question mark ('?') at the front. Moreover, each server logs input and output to a file named '<server_id>.log'. The logs contain notices of new and dropped connections from other servers.

2.1 IAMAT Requests

Clients send their location in coordinates via the IAMAT request to the servers. A valid IAMAT message uses the following format:+

IAMAT <client_id> <coordinates> <timestamp>

Note that a client id may be any string of non-white-characters; the coordinates consist of the latitude and longitude in decimal degrees using ISO 6790 notation; and the timestamp is expressed in POSIX time, consisting of seconds and nanoseconds since 1970-0101 00:00:00 UTC. Fields are separated by one or more white space characters and do not contain white spaces.

The server respond to clients with a message using the format:

AT <server_id> <time_difference> <client_id> <coordinates> <timestamp>

The server id is the name of the server that got the message from the client, time difference is the difference between the current time for the server and the time stamp received from the clients, and the remaining would be the copy of the message received from the client.

2.2 WHATSAT Requests

Clients query for information about places near other clients' locations using the WHATSAT request of the following format:

WHATSAT <client_id> <radius> <max_result>

The arguments to a WHATSAT message are the name of another client, a radius (in kilometers) from the client, and an upper bound on the amount of information to receive from Places data within that radius of the client. The radius must be at most 50 km, and the information bound must be at most 20 items as specified in the spec.

The sever responds with a AT message in the same format as before followed by a JSON-format message which is in the same format that Google Places gives for a Nearby Search request.

2.3 Server-to-Server Communication

Severs communicates to each other to share the location information of the clients. Specifically, they send AT messages (the same format as above) to other neighboring servers via TCP connection. A flooding algorithm is adopted to ensure the proper propagation of messages: a dictionary is kept as a class attribute in the Server class to store the (<message>, <servers>) pairs, where 'message' is the AT messages, and 'servers' is an array keeping track of the servers that have received the messages. Once a new message is shared to a server, this server's id is appended to the 'servers' array and the message is further propagated to the neighboring servers. If a server receives an old message, it stops propagation.

Note that servers continue to operate if their neighboring servers go down.

3. The Suitability of asyncio for Proxy Herd

3.1 Pros and Cons of asyncio in Sever Herd

One of the advantages of asyncio in this application is that it supports asynchronous operations. Note that in our application, tasks may start and complete independent of each other. For example, while a server is waiting for a response from the Google Places API, it may receive new requests from clients at the same time. If we are only allowed to do the synchronous operations, a server has to wait for the response from the Google Places API before doing anything else. In contrast, by marking the response as an asynchronous operation, we make sure that waiting for the response does not block other tasks, while the response is taken care of by the event loop in the background. This saves the CPU time and makes our program more efficient, and the servers can response to the clients once the message is updated.

Moreover, asyncio allows finer control of the task switching by developers. With interfaces like 'start_server', 'open_connection' and the concepts of coroutines and awaits, asyncio handles the power of controlling the task switching to the programmers. In this project, for example, we mark the I/O bound tasks as async while keeping other tasks as normal to achieve maximum exploit of the server herds. This direct control of task switching allows programmers to maximize the concurrency in the program easily in Python, which a single-threaded language.

Another advantage of asyncio is that it scales far better than threading. Each task takes far fewer resources and less time to create than a thread, so creating and running more of them works well. This enables our application to be scalable.

However, one downside of using the asyncio library is that we are limited to compatible libraries with it. For example, to get the TCP messages from the Google Places API, we have to use libraries that can send HTTP requests asynchronously like aiohttp instead of synchronous libraries like urllib or requests.

Overall, the asyncio library fits into the server herd application very well. Firstly, it improves the performance and makes the program more scalable by employing the event loop to run independent tasks concurrently with a single-threaded context. Moreover, the asyncio library is rather simple and intuitive comparing to the traditionally multi-threading, thus making this library easy for the developers to explore and to use on the new applications like this server herd application.

3.2 Dependence on asyncio features of Python 3.9 or later

Note that the asyncio features of Python 3.9 or later, like asyncio.run and python -m asnycio, are fairly important to our program. Take asyncio,run() as an instance. Although we can get by with older versions of Python by creating an event loop manually using asyncio.new_event_loop(), and then executing the loop with asyncio.run_until_complete() and other builtin functions, asyncio.run() would be much simpler and easier to use as it takes care of managing the asyncio event loop. The other new feature we mentioned above, python -m asyncio, launches an asyncio REPL loop. This allows us to test some asyncio functions, or execute asyncio commands directly. While this feature might not be so important as the previous one, it also makes the asnycio library in Python more convenient and portable to use.

3.3 Problems Encountered

One problem I encountered was the about the await expression in the asyncio library. I used a function 'parse_message' to parse the message received from a client. However, I forgot to make it an asynchronous function at first. Since the IAMAT request can be responded immediately, there is no need to make the 'parse_message' function a coroutine at that moment. However, when I added the code to process the WHATSAT message, the function failed: while I use the await expression to wait for response of the HTTP GET request from the Google Places API, I did not use the await expression to get the return value from the asynchronous

function that handles the http get request. To solve this problem, I modified the ‘parse_message' function by adding ‘await’ keyword before the return value from ‘handle_whats_at’ (the function handles WHATSAT request). I also added ‘await’ in front of the return value from ‘handle_i_am_at’ (the function handles IAMAT request) and marked ‘handle_i_am_at’ and ‘parse_message’ function as ‘async’ for consistency.

Another problem I run into was about the communication between servers. Initially, I thought that the server ‘Juzang’ only talks with ‘Campbell’, as the spec says "Juzang talks with Campbell.” However, as I tested my implementation of the server.py using the test script from the TA hint repository, I consistently got 9/10 for the score of the correctness of advanced flooding test cases. Then I drew the communications between servers in a graph, and realized that ‘Juzang’ should indeed connect to ‘Riley’, ‘Campbell’ and ‘Bernard’. The problem was solved after I made this modification.

4. Comparison between Python and Java

4.1 Type Checking

Python is a strongly but dynamically typed language. This means that Python interpreter does type checking only at runtime, and the type of a variable is allowed to change over its lifetime. This makes declaration of variables in Python easy and concise. This also makes Python code more readable and flexible, as the type annotations are not necessary and the variables and objects can be passed between function without having to know their types.

In contrast, Java is a statically typed language. To be specific, the type of a variable or object is checked during the compile time. Once a variable is declared to be certain type, the compiler will ensure that it is only evert assigned values of that type. The statically type checking enables the compiler to eliminate errors before runtime, thus ensuring the reliability and correctness of the program.

In this project, the way Python does type checking would be an advantage. As we aimed to explore the “application server herd” but not actually implemented a server herd in production, the size of the program and corresponding tests should be relatively small. Although the flexibility of the type checking in Python is achieved at the cost of performance, this problem is not that significant for this small application. To compensate the performance issue in the real large applications of the server herd prototype, we might need to migrate to Java.

4.2 Memory Management

Note that memory is managed by the Garbage Collectors both in Python and Java. Garbage collection process makes sure that the memory is freed up when it is not in use to make it available for other objects. However, two languages implement the Garbage Collector in different ways. Specifically, Python utilized the idea of reference counting. The memory manager keeps track of the number of references to each object in the program. When an object's reference count drops to zero, the Garbage Collector automatically frees the memory from that particular object. Note that this algorithm is simple, and relatively easy to implement correctly. However, if a group of objects contain a pointer cycle, or form a circular reference, their reference counts can never reach zero and therefore never be reclaimed.

In contrast, Java implements the garbage collection via the mark-and-sweep algorithm. To be specific, we access and use one bit per object. In the MARK phase, we clear all he bits of all objects, and do a deep-first traversal to mark all the objects in a tree addressed by a common root. In the SWEEP phase, all the unmarked objects are freed and the memory is put back to the free list.

Note that in this project, the reference counting in Python would be a better idea. Notice that circular referencing is unlikely to happen in our implementation. However, as we learned in lecture, allocation of memory is normally fast but occasionally slow using the mark-and-sweep algorithm. This makes it unsuitable for real applications, like the application server herd, as the performance is unpredictable.

4.3 Multithreading

In Python, the Global Interpreter Lock(GIL) is used to synchronize the execution of threads so that only one thread can execute at a time. This increases the speed of a single-threaded programs, as threads do not need to acquire or release locks on data structures separately. Also, the Global Interpreter Lock is easy to implement and allows the use of interfaces of the some C libraries that not thread safe.

In contrast, Java is a multithreaded programming language, where two or more parts of a program is allowed to execute currently for maximum utilization of CPU.

Notice that in this project, multithreading is not that necessary. The single threaded version works fine with the help of the asyncio asynchronous library, and requests to the server herd are replied pretty fast in tests. Therefore, we do not need the multithreading in Java which requires the complicated synchronization methods, as it might cost overhead for this simple application. If we hope to scale up the application and serve the requests for a large number of clients, we might need to consider to write a multithreaded program in Java.

5. Comparison between asyncio and Node.js

Python and Javascript are both dynamic languages. Python’s GIL and Javascript’s intentionally single-threaded design

makes threads-base parallelism almost impossible in those two languages. Asyncio for Python and Node.js for Javascript, are the asynchronous libraries to enable certain level of concurrency in those two languages. Both frameworks allow independent tasks to run asynchronously. Specially, when some code is waiting for I/O (disks, network etc.), other code is not blocked and can be run concurrently. Both frameworks have event loops, and both have the keywords including async and await. In fact, more equivalent concepts exist in two frameworks: awaitables in the asyncio library v.s. promises in Node.js, coroutines in the asyncio library v.s. callbacks in Node.js, etc.

Two frameworks differ in terms of performance and applications. Note that Node.js is based on Chrome's V8 engine, which is very fast and powerful. Moreover, Python programs often require large amounts of memory, which also makes Node.js perform better than Python with asyncio. However, Node.js is more suitable for developing scalable web applications, as the MERN stack enables developers to write frontend and backend code both in Javascript. For small applications like our server herd application, it would be enough to use Python with asyncio, which is relatively simple and easy to understand.

6. Conclusion

In summary, Python with asyncio proves to be a suitable framework for the application server herd architecture. Features like dynamic typing and Garbage Collector in Python allow the exploration of the new idea to be productive, as programmers do not need to spend much time in understanding the syntax and managing the memory space. Although multithreading is unavailable in Python, the asyncio library enables the tasks to run concurrently to achieve I/O parallelism. This improves the efficiency of the server herd, making it possible for the servers to handle a large number of requests and to update messages to the clients frequently. If we want to build the actual server herd for large applications, we may need to store the clients' messages in a database rather than keeping them inside the Python program. Moreover, further experiments are needed to decide whether we should migrate to Java, Javascript with Node.js or other programming languages to implement the large applications in practice. At current scale, Python with the asyncio library performs very well and is suitable for the new Wikimedia-style service.

7. References

https://docs.python.org/3/library/asyncio.html

https://docs.python.org/3/library/asyncio-task.html

https://docs.python.org/3/library/asyncio-eventloop.html

https://web.cs.ucla.edu/classes/spring21/cs131/hw/pr.html

https://www.javatpoint.com/memory-management-in-java

https://www.geeksforgeeks.org/memory-management-in-python/

https://realpython.com/python-type-checking/

https://wiki.python.org/moin/GlobalInterpreterLock