CS131 Project: Implementing Proxy Herd with Python's `asyncio`

Abstract

Python's `asyncio` library offers some useful tools for asynchronous processing. Without native support for multi-threading programming, `asyncio` provides an effective alternative to enhance performance with its unique advantages and disadvantages. In this report, we closely study and analyze the process of implementing a proxy herd prototype with Python's `asyncio` library. From that, we attempt to compare `asyncio`'s solution to that of Java-based approaches and `Node.js`, leading to a discussion of the pros and cons for each implementation.

1 Introduction

For this project, we attempt to implement a prototype of a server herd that satisfy the following usage requirements:

- server receives frequent updates from client
- access will be required via various protocols
- clients tend to be more mobile

Specifically, we are interested in building an "application server herd" where multiple applications servers directly communicate to each other, sharing information across all servers in realtime. This allows client information to be accessed from any of the servers. In this project, we choose to implement a server herd that receives clients' GPS location coordinates and allows clients to request nearby searches using the Google Places API based on their uploaded locations.

This report first presents an overview of the implementation for the above prototype. Specifically, we discuss how the servers respond to client messages, how inter-server communication is conducted, and how the servers request and process nearby search with Google Places API.

Afterwards, this report closely analyze and evaluate the advantages and disadvantages of implementing this prototype with Python and specifically the `asyncio` library, compared to approaches utilizing Java and Node.js.

2 Prototype Implementation

For this prototype, we intend to design a proxy herd with five servers, namely Goloman, Hands, Holiday, Welsh, and Wilkes. Every server communicates to some of the other servers bidirectionally and overall all servers are inter-connected by one or two intermediate connections. Each server accepts TCP connections from clients with information about their ID, coordinates, and timestamp. Upon receiving the message, a server should respond to the client with a confirmation message, store the information in its local database, and propagate this information directly or indirectly to all other servers. Later, a client can query for information in the format of nearby search on any users that previously uploaded its information to one of the servers. The server should respond to such query by requesting a Nearby Search through Google Places API with the specified user's information and the client's command parameters (search radius and max item count) as input. The result from Google Places should be processed and transferred to the client.

2.1 Handling `IAMAT` command

A client can upload its current location by sending a message to a server with `IAMAT` command with the following form:

```
IAMAT <user-id> <coordinates> <POSIX-time>
```

Following the command identifier is the client's user id, the latitude and longitude of client's current location using ISO 6709 notation, and the client's timestamp when the message is sent from the client expressed in POSIX time.

Upon receiving this command, the server should first respond to the client with a message of the following format

```
AT <server-name> <time-diff> <client-message-copy>
```

Following the message identifier "AT" is the server name that receives the message, the difference in time between server receiving the message and client sending the message, and a copy of client's message excluding the leading command identifier. This informs the client that the message is received at the specified server and verifies the received message to the client.

After sending the above message to the client, the server should store this information inside a local data structure and proceed to propagate this message to all other severs using a flooding algorithm.

To implement this first requirements, we first utilizes `asyncio.start_server` to start up a specified server for receiving client message using its specified port. By passing in a function defined as

```
def async handle_connection(reader, writer)
```

and following with

```
await server.serve_forever()
```

we essentially creates a coroutine `handle_connection` for handling all incoming and outgoing messages concurrently while the server loops forever. Finally, we can use `asyncio.run(main())` to initiate the process.

To handle the command, we first parse the client message into meaningful tokens, calculate the time difference, and rearrange the tokens for a response message. Then, a dictionary is created for this specific client with meaningful keys for its id, coordinates, timestamp, timediff, and host server. We store this dictionary at a local client list for future query reference. Note that we've chosen to use low-level dictionaries to avoid complications of defining and maintaining classes.

Finally, we propagate this information to all neighboring servers by constructing and sending a customized `INTERNAL` message with the following format:

```
INTERNAL <id> <coor> <time> <diff> <host>
```

essentially copying the values stored in the client dictionary to other servers via this message.

To communicate with neighboring servers, we again utilizes the `asyncio` library with:

```
reader, writer = await asyncio.open_connection(
    '127.0.0.1', servers[server])
```

This allows the server to communicate with other servers like a client, opening a new connection to its neighboring servers every time it receives new information about a client, and sending the customized message to transmit the information. Consequently, the server needs a dedicated function to handle the received internal messages.

2.2 Handling `INTERNAL` message

To allow inter-server communication, the servers are implemented to transmit information via sending a customized `INTERNAL` message and propagate such message through a flooding algorithm. Upon receiving a message with the command identifier `INTERNAL`, the server will again parse the message and extract the corresponding information into tokens. Same as in handling /textttIAMAT commands, the server store the received information in a client dictionary with meaningful keys corresponding to the stored values. The dictionary is then stored in the local client list of the server that received the message. This essentially copies the client information from another server into this server's local database.

After storing the received information, the server should proceed in spreading this information subsequently to all its neighboring servers to allow the information flood to all currently running servers. We discover that upon receiving a propagated information, if the user is already registered in the local database and the message has a timestamp earlier than the local record, this means the current database already has a piece of client information that is more recent than the information being propagated from other servers. Thus, in this condition the received /textttINTERNAL message can be safely discarded since it offers no useful information to the current server. If the received information is outdated for the current server, it would also be outdated for all other neighboring servers, since upon receiving a most recent client update, the client information is propagated immediately to all neighboring servers.

For this reason, we only store and propagate client updates that is most recent in the perspective of the current server. By comparing the timestamp of the received client information to the client record in the local database, every server is in charge of deciding which update should be discarded and which should be stored in its local data and propagated to neighboring servers. By establishing this constraint, not only do we prevent the situation when an older information overwrites a more recent client record due to connection delay, we also present a terminal condition for the flooding algorithm to terminate when every servers received client information that is no more recent than its local record. This avoids the flooding algorithm to loop infinitely.

2.3 Handling `WHATSAT` query

Clients can send queries to the server in the form of a `WHATSAT` command, asking about another client's nearby places. Such queries has the following format:

```
WHATSAT <user-id> <radius> <result-cap>
```

Following the command identifier, the client supports a user id for the server to fetch its most recent uploaded location, a radius parameter that determines the radius of nearby search for the queried location, and a maximum number of places results desired by the client.

Upon receiving this query, the sever should respond with an `AT` message that is exactly the same as the uploader received when it most recently updated its information to one of the servers. Following that is a JSON-format message exactly in the same format received from a Google Places Nearby Search request, with the results list trimmed according to the item cap provided by the client.

To implement this functionality, we again parse the query message into meaning tokens. With the user id provided as a key, we can search the local client list to see if there exists a record of this user (whether the specified user has sent a `IAMAT` command prior to this query). If not, we simply treat the query as an invalid command since the spec doesn't specify the behavior for this situation. If a record is found in the local database, we retrieve the client dictionary and subsequently all the fields in the dictionary. With the values from the client dictionary gathered, although we do not have the original `AT` respond from the original server that received the latest update, we could use the information from the client dictionary to construct the exact same message and send to the querying client.

With the coordinates of the specified user retrieved from database, we could then extract a pair of latitude and longitude of Google Places requests format from the original string.

Combining with the radius parameter, we use the `aiohttp` library to send such request to Google Places with:

```
async with aiohttp.ClientSession() as session:
    params = [('location', location),
              ('radius', tokens[2]+'000'),
              ('key', API_KEY)]
    async with session.get(
        PLACES_URL, params=params) as resp:
        jsonstring = await resp.text()
```

where `aiohttp.ClientSession()` creates an HTTP client session and `session.get()` makes an HTTP request with the provide URL and receives the JSON result from Google Places.

With the received result string in JSON format from Google Places Nearby Search with the user's provided parameters, the server could finally process the string with a pair of `json.loads` and `json.dumps` from the `json` library for decoding the result string into a json object, retrieving and slicing the result list to meet the required item cap, and finally encoding the modified json object into a message string. Appended to the previous `AT` message, the entire message is responded to the querying client.

With the above three functionality implemented, the prototype is thus complete. In the next section, we review the process of developing such a server herd using Python's `asyncio` library and evaluate its effectiveness.

2.4 Prototype review

With the completion of prototype implementation, we now review the process of building a naive server herd using functionalities provided by the `asyncio` library.

First, our server herd implementation heavily relies on the concept of `asyncio` event loop and coroutines. Without going to the details of low-level APIs for fine grain control on the event loop, we stick to the high-level APIs to handle the requests in the server herd, specifically with the usage of `async` and `await`.

By declaring a function with the `async` keyword, we essentially creates a coroutine object that can be pushed onto the task queue. When the task is invoked by the `await` keyword, the task is pushed onto the queue to run concurrently with all other coroutines, and the current coroutine pauses until the task returns. This allows the server herd to handle multiple client requests and inter-server propagations concurrently, thus the server herd is able to receive subsequent requests while the previous requests are still processing. Combined with the easy-to-use high-level APIs for TCP server and client connection, we succeeded to build an effective application server herd in an efficient and convenient manner.

2.4.1 Advantages of `asyncio`

With asynchronous processing provided by this library, we could conveniently implement concurrent programs as an effective alternative for multi-threading. While the currently running task is doing slow I/O intensive work (TCP and HTTP connections in this project), the program can put the task to sleep, move on to the next task and resume on the previous task after I/O is finished. This allows the CPU to be fully utilized at all time, cutting the CPU time wasted on waiting I/O to finish, thus the performance of I/O heavy programs can be greatly improved by the usage of `asyncio`.

On the other hand, the asynchronous nature of `asyncio` framework allows a python script to perform multitasking behavior. While the previous tasks are in I/O processes, the CPU is able to handle new tasks as they are provided. While the new tasks might not be processed immediately, they will be accepted and be pushed onto the queue for later process.

2.4.2 Disadvantages of `asyncio`

Again, due to the asynchronous nature of `asyncio`, new tasks are not processed immediately but rather pushed onto a task queue as future events. This in effect reduces the responsiveness of the program, since the time of processing for a specific task is determined by the event-loop and thus might not guarantee the desired latency.

On the other hand, since the tasks are handled asynchronously, the completion of each tasks depends on the event loop and are related to their individual I/O time. In effect, this can cause tasks to be not finished in their arriving order. For example, in this project, consider the situation in which a new client sends an `IAMAT` to a sever to upload its location and immediately after this another client sends a `WHATSAT` querying the previous client's nearby places to the same server. If the task for handling `IAMAT` is still in the process of I/O and thus the task is paused, the subsequent query, although arrives later than the update, would fail to find a record for the requested client since the task for storing that information is paused. Such a-synchronization induced errors can be difficult to understand and handle at times, rendering the execution of the program unstable and relies on individual I/O times.

Finally, with asynchronous processing as the substitute for multithreading, we reduce the power and efficiency granted by parallelization to that of concurrency. With all processing relying on a single CPU and all tasks processed asynchronously, the efficiency of the program is greatly reduced since it cannot utilize multiple CPU cores, whereas parallelization enables the program to fully utilize all cores and handle multiple tasks at the same time. This Introduces a significant limit on the performance of a program using `asyncio` as a multithreading alternative.

3 Comparing to Java

In this section, we compare our implementation of the application server herd to an alternative Java-based approach. Specifically, we analyze the differences between the two languages on the aspects of type checking, memory management, and multithreading.

3.1 Type checking

In the most obvious manner, the difference here is that Python uses dynamic type checking while Java uses static type checking. For Python scripts, the interpreter only determines the type of variables at runtime. Variable types are automatically assigned by the Interpreter and can be changed during the course of execution. This allows a simpler and easier workflow for Python programming, as the programmers can be free from declaring variable types and focus more on the implementation logic. As a scripting language, this advantage in ease of use is highly valuable, increasing both the writability and readability of python programs. This is hugely favored in wiring short scripts and prototypes where functionality is valued over robustness. However, the lack of compile time type checking can lead to complicated and hard to detect bugs when the project grows larger in scale.

On the other hand, variable types in Java is checked at compile time. The programmer must explicitly annotate the type for every variable and method, with any type violations result in a compile time error. This ensures that no type error would go through compile stage, enabling the programmer to detect type errors early in production for easier debugging. This in effect allows larger projects to be easier implemented and better understood by programmers as the types for all variables are statically define. This reduces the chance of logical errors introduced by runtime type error, allows easier debugging, and increases the robustness of programs. On the downside, it complicates the coding process by constantly requiring the programmer to specify and match types for all operations on the variables. This could hamper the efficiency of production if the need for compile time type safety is not required.

3.2 Memory management

For Python's memory management, everything is implemented as if they are structures. More accurately, Python allocates every variable, whether of intrinsic type of constructed type, onto the heap. Every variable is associated with a reference count that is used for its garbage collection. The reference count for every variable is maintained by the interpreter at runtime, and when the reference count for a variable reaches 0, the object will be deallocated from the heap. Without utilizing the concept of stacks, all variables are allocated on shared resources, and the python memory manager is in charge of managing the scope for each reference. To manage relying all memory on shared resources, Python utilizes the Global Interpreter Lock that locks the entire interpreter during memory management. This implementation introduces numerous debatable pros and cons to the language. In short, it enables high single-threaded performance at the cost of multithreading flexibility. [3]

On the other hand, memory management in the Java virtual machine is specified by the Java Memory Model. Unlike Python, the JMM divides memory between threads stacks and the heap. Each thread running in the Java virtual machine has its individual thread stack that is private to each thread. Within each stack contains the call stack: a record of method calls and return addresses. Local variables created by the thread are allocated onto its own thread stack, invisible to other threads. Local variables of primitive types are fully stored within the stack, while all objects created are allocated onto the heap regardless of which thread the creator is. References (similar to pointers) to objects are stored in the thread stacks for the threads to access the objects in the heap. It is worth noting that object members and static member methods are stored on the heap along with the object. The same object can thus be accessed by any thread that has a reference. Java also utilizes a garbage collector for easing the consideration of low-level memory management from the programmer's mind. The Java Memory Model specifies a more complicated management mechanism to promote the power and flexibility of Java's memory management at a possible cost of performance. [2]

3.3 Multi-threading

The fundamental difference between Python and Java in terms of multithreading is that multi-threading in Java achieves parallelization whereas in python it only achieves concurrency. As previously mentioned, due to pythons implementation of the Global Interpreter Lock, only a single thread can be executed at any given time to prevent memory race. On an implementation level, the GIL constrains the power of multi-threading in Python to concurrency, as to achieve parallel programming in Python one should utilize Multi-processing instead. By using the `threading` library, programmers could built multi-threaded programs that relies on a thread-based concurrency, as opposed to our concurrency approach with `asyncio`.

On the other hand, in Java each thread has a separated private stack only visible to the thread itself. This allows multiple threads to be naturally executed in parallel, and parallelization provides huge potentials for Java multi-threaded programs to improve their performance by fully utilizing multiple CPU cores. However, this requires programmers to manually deal with race conditions and explicitly handle synchronization implementations. Indeed, with great power comes great responsibility.

4 Comparing to Node.js

Node.js is runtime environment based on Chrome's V8 JavaScript engine. By choosing Node.js for backend, programmers obviously gains all benefit from the JavaScript language. Simple and efficient production, vast amount accessible tools, and easy code sharing and understanding between other JavaScript programmers makes Node.js a very popular choice for server implementations. With an event-based model, Node.js provides Non-blocking I/O and asynchronous request handling to achieve low-latency real-time applications, a great choice for implementing servers with constant data update.

Similar to Python's `asycio`, the weakness of Node.js lies at its poor scalability, as Node.js is also single-thread based. Without the ability utilize multiple CPU cores, which is commonly present at current server-class hardwares, this limits Node.js to be only suited for CPU-nonintensive, heavy I/O lightweight applications. However, in 2018 multithreading is introduced in Node.js as an experimental feature. With the help of workers thread module, Node.js can allow every CPU core to execute an individual thread, leveraging some performance setback due to JavaScript's single-threaded nature. [1]

5 Conclusion

By experimenting with building an application server herd prototype with Python's `asyncio` library, we gained valuable experience with handling server functionalities with an asynchronous approach. With this knowledge, we proceeded to analyze and evaluate the advantages and disadvantages of implementing the server herd with `asyncio`, as well as comparing our approach to that of Java and Node.js. With this project, we have developed a better understanding on the demands and limits for building application servers. We built an introductory insight on the difference between Python, Java, and Node.js's respective strength and weakness. With these information in mind, we have broadened our vision to be better prepared for future projects where we can efficiently and wisely choose how we should work on server backends with better knowledge.

References

[1] AltexSoft. The good and the bad of node.js web app development, May 2019.

[2] Jakob Jenkov. Java memory model, December 2014.

[3] Real Python. Memory management in python, Dec 2018.