CS 131 Project. Proxy herd with `asyncio`

Fang Yujie
University of California, Los Angeles

Abstract

This project uses asynchronous style of programming. The benefits of using single thread are listed as follows: first, there is no overhead of kernel context switch and hence no latency; second, there is no need for lock, mutexes or semaphores to prevent race condition because race condition is not a problem for single threaded program. Even though the content for coroutine and asyncio is new to me and hence difficult to implement, I would recommend it to boss because of the above benefits listed.

1. Introduction

This project is built on the framework of asyncio to implement an application server herd. Specifically, there are five servers, named ‘Goloman’, 'Hands', 'Holiday', 'Welsh' and 'Wilkes', which given restrictions about which servers are communicable. The client is allowed to send its location to the server using the following format:
<IAMAT> <client_ID> <location> <time>
for example:
`IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997`
and the server will reply with a message using the following:
<AT> <server_ID> <time_difference> <location> <time>
for example:
`AT Goloman +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997`

After the client reports his location, he can make a request using the following format:
<WHATSAT> <client_ID> <radius> <number_of_places>
for example:
`WHATSAT kiwi.cs.ucla.edu 10 5`
The server will response the same message as it does for ‘IAMAT’ followed by a new line and a JSON-format message.

It is important to note that when one server shuts down, other servers are allowed to operate as normal. Therefore, the implementation of the flooding algorithm is important. (This will be discussed in section 3.2.)

2. What is `asyncio`?

Python’s `asyncio` provides infrastructure for writing single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives. In this program, communication from client may arrive at any given time, asyncio helps to schedule the tasks without drawing into the complexity and confusion of multi-threaded concurrency while still reaping the benefit of sequencial scheduling.

pros:

1) Messages arrive and depart via the network at unpredictable times - asyncio lets you deal with such interactions simultaneously.
2) Callbacks are strictly serialized: one callback must finish before the next one will be called. This is an important guarantee: when two or more callbacks use or modify shared state, each callback is guaranteed that while it is running, the shared state isn't changed by another callback.
3) The program does not wait for a reply from network callsbefore continuing. Callbacks is running when the result of a network call is known. Meanwhile, the program continues to poll for and respond to other network related I/O events. Callbacks execute during the iteration of the event loop immediately after the expected network I/O event is detected.

cons:

1) Difficult to understand and write!!! Yes, I struggled with this project and spent a lot of time reading the manual.
2) It is slower than multi-threaded program. There is latency to deal with context switch.

3. Program Design

3.1. Overall framework

I choose to used object-oriented program. Before using object-oriented program, I tried to use non-object-oriented approach by found it difficult to trace different servers.

Basically, I set up a class called `Server` which is used to handle all five server objects.

Inside the `Server` class, I implement a nested class named `Protocol` which is used to handle all detailed implementation of input and output of different servers.

Inside the `Protocal` class, I introduce several parsing functions, namely:

```
parse_message
iamatInfo
whatsatInfo
```

The first function, `parse_message`, parses the message given by client, check the format of the message, determine if it is an 'IAMAT' message or a 'WHATSAT' message, and call respective parsing functions.

If it is an 'IAMAT' message, the parsing function, `iamatInfo`, is called. This function checks the message format and update the information into the server object. The information is stored in a dictionary, with the key being a string of the client_ID and the value a list [latitude, longitude, time]. To respond the client, it writes a line of a specific format.

If it is a 'WHATSAT' message, the parsing function, `whatsatInfo`, is called. This function checks the message format and request information from Google API. It is important to note that `asyncio.ensure_future` is needed to schedule the execution of a coroutine object.

3.2. Flooding Algorithm

To ensure that when certain servers shuts down, other servers can still operate as normal and acknowledge the most recent updated information, decided to call the flooding algorithm as soon as one server receives a message. That is, when one server receives an 'IAMAT' message, it will parse the message for its own server, and send the same message to other servers immediately. However, this will cause the flooding algorithm to run into infinite loop as there is no flag to signal servers if the message is already updated onto that server. Therefore, I added 5 digits after the message to signal if specific server has already been updated.

When a new message is sent from a client, it is immediately coded with five '0's after the message. When a server receives it, it replaces the corresponding '0' with '1'. Every time a server receives an 'IAMAT' message, it checks if the corresponding position in the list is '1'. If yes, it means that the server has already received this message and will do nothing. Otherwise, the server will update, replace '0' with '1' and send the new message to server that it allowed to communicate. Once all servers are updated, the flooding stops.

3.3. Google Places API

To query Google Places API, I imports the `aiohttp` library. Latitude and longitude is stored in `self.server.info` dictionary and the key is just the client_ID, which are easily accessible. The information for Google Places API is stored into a json object and is dumped and written using TCP.

3.4. Error handling

All wrong formatted messages are responded by '? ' followed by the original message. This line is implemented by `self.tcp.write(('? ' + message).encode())`.
It is recorded into the log with specific error messages.

One important error handling part is when one server fails and other servers cannot send message to it. If it is handled with `except`, then the flooding algorithm will pause when it catches the `OSError`. Therefore, every time one server opens a connection, it catches `OSError`, throws it, and continue the flooding algorithm.

4. Limitations

1) The servers do not handle historical location updates after a server shuts down and is brought up again. That is, even when this server gets an 'IAMAT' message and is updated, this information is not stored after it goes down and brought up. The server will not handle a 'WHATSAT' request with historical locations as expected. The new server will only start receiving new location updates.

2) When two 'IAMAT' messages with the same client_ID are sent to two different servers at exactly the same time, the servers do not agree on the value. For example, some servers may contain message A while other servers may receive message B and refuse to update itself with message A, depending on the priority in the dictionary `server_communicable`. Therefore, it is possible that two same 'WHATSAT' messages send to two different servers will yield different responses.

5. Discussion

What are the performance implications of using `asyncio`?

There is context switch issue about `asyncio`. A context switch in `asyncio` represents the event loop yielding the flow of control from one coroutine to the next. Comparing to multi-threaded application, single-thread approach should be slower intuitively because of server process is put to sleep when there is no communication and there is latency for context switch. However, for this project, the advantage of a single-thread program is that it is free of race condition as asynchronous code is running serially. `Asyncio` can yield the control of processor to other processes that are running. Therefore, to total throughput of `asyncio` is higher than multi-threaded application.

how about Java-based approach to this problem?

Python's dynamic type checking gives programmers more freedom compared to Java's static type check. For example, python allows me to define a Server class that can be used by most functions that expect a dictionary, a list or another type, very easily. On the other hand, Java is statically typed, type checking is done at compile time.

In terms of memory management, python memory management uses a heap. All the objects declared and the data structures involved are contained in the private heap. The heap management is done via the use of python memory manager and is performed by the interpreter itself and that the user has no control over it. Also, python uses reference counting to free objects as soon as they have no reference, meaning that most of time everything is cleaned up by reference counting. But reference counter requires extra bookkeeping and locking overhead - cyclic reference where two objects refer to each other so their reference count will never be zero. For java, memory is allocated only to objects. There is no explicit allocation of memory, there is only the creation of new objects. In addition, java run time has a garbage collector which is responsible for freeing the space which no longer is having any reference to an object.

How about Node.js?

Node.js asynchronous model allows the code to handle many concurrent connections with high throughput on a single thread. Using Node.js, the applications are extremely scalable due to asynchronous and event-driven processing.

`Asyncio` has done for python what node did for javascript. Asynchronous programming in python enables you to do new and powerful things, but like all technologies there are tradeoffs. It can significantly improve throughput on the same hardware. There is no need for semaphores. However this won't improve latency since awaited functions still have to wait for IO. While our awaited functions wait for IO, control is automatically returned to the event loop so other code can run.

For `asyncio` code, you need to run your code using an event loop. For this project, this usually means adding a bunch of print statements to whatever isn't working and re-running my code. This is a very unpleasant coding experience.

6. Conclusion

Writing this project using `asyncio` in python proves to be a difficult experience. Yet, I would still recommend using the `asyncio` module in Python. First, it is far more lightweight than any other multi-threaded implementations with the use coroutines and event loop. Python's dynamic type checking and memory management also contributes to the easiness of implementation as well as high performance.

However, there are so many functions in `asyncio` and it is difficult to determine which one to use. Since I am waiting the result of actions and perform some actions on the message, should I use `asyncio.gather`? Or `asyncio.wait` on future completion? Or should I have the future to `run_until_complete` or `run_forever`? Did I forget to write `async` in front of a function that has await? Did I forget the write `ensure_future` when I call other functions? The python `asyncio` is so error-prone that I spend more time on debugging than ever.