Project Report: Proxy herd with *asyncio*

Jinjing Zhou
University of California - Los Angeles

Abstract

In this project, we are asked to investigate the efficiency and suitability of implementing the recently released Python module, *asyncio*, in a new Wikimedia-style service designed for news so that updates to articles will happen more often, accesses will be made through various protocols and clients will tend to be more mobile. This report includes the introduction of an implementation of the proxy herd application using *asyncio* module, the evaluation of the pros and cons of using Python *asyncio*, and comparison between using Python and using other possible languages.

1 Introduction to *Asyncio*

Asyncio in Python provides infrastructure for writing single-threaded concurrent code (which avoids overhead to a large extent) using coroutines, running clients and servers as well as other resources. The core of asynchronous programming with *asyncio* in our case is that, communications from clients may arrive at depart at unpredictable times, *asyncio* the project to deal with such interactions simultaneously without using potentially confusing and complicated "threaded" concurrency while retaining the benefits of strictly sequential code.

1.1 Pro

First of all, *asyncio* has a concept of event loop, which is a queue of events and a loop that continuously pulls jobs off the queue and runs them[2]. For our project, there are multiple functions which need to be done interleavingly. Instead of coding context switches manually and have the kernel to do them which causes overhead – one of the bottlenecks in concurrent programming, the event loop allows the project to have application controlled context switch without involving the CPU.

Moreover, *asyncio* allows the project to have coroutines instead of actual threads, and running them on event loop for scheduling. In other words, the execution of functions that are defined as asyncio.coroutine can be paused and the event loop will decide when to continue. Our project can take advantage of this as it will need to monitor multiple activities, as a server must be listening to a client, a server may be talking to a client, a server may be talking to another server, and in addition, a server must be listening to all other servers, which can all be defined to be asyncio.coroutine functions. The event loop will decide which coroutine should be running next if the last coroutine is finished and there are more in the event queue.

In addition, using *asyncio* solves some of the problems that would take place in our project if we implement it with multi-threading.

If our project implemented multi-threading, in the case where multiple servers receive information about the same client, and a client sends a "WHATSAT" query to one of the server, there is no guarantee that the newest information will be replied to the client. However, since *asyncio* runs only a single coroutine at a time and switches only at points as defined by the code, there will not be any race condition.

1.2 Con

Asyncio also has some disadvantages however. To begin with, if two coroutines are supposed to wake each other, which may happen in our project when servers are trying to reconnect to each other or continuously listening to each other, if the code was not examined carefully enough. This will result in a deadlock situation.

Besides, though coroutines are all run on a single thread, which don't require extra sockets or memory, making it less possible to run out of resources, the "executor pool" of the module actually resembles a thread pool[1]. As a result, if the code puts too many executors into the pool, the resource starvation situation can still take place. However, this is not so likely in our current project, which only involves five server, but in other situations where more are involved, this may become a concern at some point.

2 Project Implementation

My project uses *asyncio* to make a distributed service, where the five implemented server will communicate to any clients as well as among themselves via TCP. A flooding algorithm is implemented so that servers update information from each other, and clients can give information to one server and ask for it from another.

Overall, it was relatively easy to write *asyncio*-based programs that run and exploit server herds. Though the concept of asynchronous programming seems hard at first, after understanding the process, I found that asyncio allows me to take advantage of the event loop, so that I do not need to keep track of the scheduled events or decide when to wake or hang a certain event. The two functions that I used mainly to do the scheduling are *asyncio.ensure_future* and *yield from*. The first one is used for all tasks that needs to be arranged to be completed in the future, but does not need to be completed immediately so that the program is still asynchronous, while the second one is used for tasks that needs to executed immediately. Together with the two functions, the event loop will ensure all tasks to be executed at some point while the process is never blocked.

2.1 Project design

In this section, I will go through how my code for the project works.

2.1.1 Connection

First of all, I declared some global dicts to indicate how the flooding algorithm works. One dict indicates for each server, what are the servers that it should communicate with. Another dict indicates for each server, the list of servers that it should "take the initiative" to connect with them and try to reconnect with them if they ever go down.

When a server is brought online, the coroutine *handle_connection* will be added to the event loop, which is function that keeps reading data from different sources, and parse the message received and call the respective handler for that certain type of message, and ensure that the writer is drained before the connection is closed.

Also, it will be asked to connect to all servers that it should be in charge of managing the connection as indicated in the global dict. After getting connected, it will send a message to the other server starting with "*IAmServer*", and store the respective pair of reader and writer into its member variable as well as using *asyncio.ensure_future* to call on *listen_to_server*, so that while not blocking the process, the function will be called as determined by the event loop. How *listen_to_server* works will be elaborated in the next paragraph. However, if it cannot connect to a certain server, it will repeated call itself using *asyncio.ensure_future*, so that the server will attempt to connect with the other until it is connected, and at the same time, it is not blocking any other job the server needs to do.

If the message received starts with "*IAmServer*", the handler for it will be called. The message indicates that this message is sent from a server that just got connected to this server, and it will store the given pair of the reader and writer into a member variable of the server class, so that the reading and writing between these two servers can happen later. This function will also call another function, *listen_to_server* that make sure this server starts to listen to this connection between the two servers, so that if any update message is sent, it will be able to get the info update them through another function, "*updateServer*", which is called using *asyncio.ensure_future* so that the process is asynchronous. In addition, it also monitors if the other server has gone down, which will be indicated as the reader to that server returns an empty string. In this case, the function will delete the set of reader and writer pair for this server. If this server is in charge of reconnecting, it should also add the job of reconnecting into the event loop by calling *asyncio.ensure_future* on the function *start_server_connection*.

2.1.2 Communication

Going back to the *handle_connection* coroutine, it will also need to handle messages from clients: "*IAMAT*" and "*WHATSAT*" message. The "*IAMAT*" handler, after verifying the validity of the command, will send an "*AT*" response with the format as required in the spec to the client, and then check if the information has a timestamp that is later than what it has for the client in its member variable, *client_info,* a dict that stores each client's name paired with info regarding that client. If it does or that client is not in the dict yet, it will need to update that information and call *asyncio.ensure_future* on *updateServer*, a function that send this new info to all the servers that this server is connected using an "*AT*" message.
As for "*WHATSAT*" messages, after checking it is a valid command, the server will immediately call (using

yield from) a function *getGoogle* with the parameters required to do the information retrieval from Google. Inside the function *getGoogle*, it uses *asyncio.open_connection* with the Google place to get a reader-writer pair, and after formatting the query, it uses the writer to write to it and uses the reader to get the reply. Then it formats the reply as required in the spec and send it back to the client that sent this "*WHATSAT*" message.

There is still another type of message that a server can get, which is the "*AT*" message that used for communication for updates between servers. This kind of message is handled in similar way as "*IAMAT*" messages as the server will check if it has received this update before. If it has, it not propagate this message so that the project does not ends up in infinite loop; if it has not, it will call *asyncio.ensure_future* on *updateServer.*

2.2 Problems that I ran into

One of the problems that I ran into was when ensuring that servers continuously "listen" to each other. While listening to clients is guaranteed using *asyncio.start_server* in *main()*, *listen_to_server* needs to be manually coded to repeat. Initially I thought to call it with *asyncio.ensure_future* at the end of each iteration. However, this did not work out and I deduce that it may be so due to the deadlock or resource starvation situation as described before. Therefore, I switched to a while True loop which only breaks when the server is disconnected.

I was also confused at how to handle the reply for a certain "*IAMAT*" message if a newer "*IAMAT*" message (a newer timestamp) has been received before (either at other servers or the server itself), the reply should be the newer information or the one received now. I then decided to use the one received now, since this makes more sense as the server must indicate to the client that it has received this instance of "*IAMAT*" message. Also, if two messages with identical timestamp from the same client are received at different servers, my code chooses to not consider the one received later as the update since this kind of situation is unlikely to happen in real-life situations.

Another minor issue I had was that my GET query to Google always return fewer results than expect. Later I found that it was because the query parameter for radius should be meters instead of kilometers, which is used in the "*WHATSAT*" messages.

2.3 Performance implications

I would consider using *asyncio* to make a distributed service to have good performance since *asyncio* is much more lightweight compared to other multi-threaded solutions. In addition, it will never have any race condition so it does not need to use any mutexes or semaphores, which further simplifies the solution and speeds up the process while retaining the benefits of strictly sequential code.

3 Comparison with Java

Python's dynamic type-checking nature simplifies the development process compared to Java. Programmers do not need to strictly type each object but directly use them as long as the type stays constant each time the code is run. Compared to Java's static typing, where programmers will have to define the type of each object strictly before using them, Python gives programmers more degree freedom as well as makes development easier. Though this is done at the expense that the code may be harder to read since types are not clear for each object, this will not be a problem if naming for each object is done carefully enough and a reader can get information about objects from that.

The memory management system of Python uses a reference-counting approach where the number of references for each object is recorded, and objects with zero reference will immediately be deleted from memory. This makes garbage collection efficient and much more convenient for programmers compared to Java, where programmers have to manually allocate (or even delete for C++) objects on the heap. It also makes the code more direct and easier to understand without codes that are needed for allocation of objects. However, there is the problem with cyclic reference where two objects are referring to each other so their reference count will never be zero. Still, this may not be a big deal since programmers can easily avoid doing that when writing the code.

Lastly but not least, as mentioned before, compared to any multi-threaded implementation of this server herd, the single-threaded implementation of Python makes the process more light-weight as it is event-driven and requires a lot less overhead. In addition, it also avoids, to a large extent, race conditions and deadlock situations which are common in multi-threaded implementations in Java.

4 Comparison with Node.js

Both Node.js and the *asyncio* module in Python are asynchronous and single threaded, which contributes to their excellent speed and performance. However, if we compare between these two, Node.js may beat our *asyncio* implementation due to that fact that Node.js is based on Chorme's V8, a fast and powerful engine that can easily integrate with web applications. In addition, in general, Python does not perform optimally in memory intensive applications[3].

However, Node.js is a bit more strongly-typed compared to Python which makes it easier to use. Furthermore, the exception-handling in Python is easier, making it faster to debug.

While our project only implements five servers, if we are to have more server, which is highly likely in real-life applications, we also need to consider if we can maintain the performance even with increasing number of requests that need to be handled. Both Python and node.js are likely to work in such situations since *asyncio* in Python allows asynchronous processing to be easily achieved and the fact that Node.js creates a single-thread asynchronous architecture with I/O operations completed outside the thread without blocking it makes it suitable for web applications.

5 Conclusion

I would recommend using the *asyncio* module in Python for the implementing the proxy herd as it is far more lightweight than any other multi-threaded implementations when we take advantage of the concept of coroutines and event loop. Python's dynamic type checking and memory management also contributes to the easiness of implementation as well as high performance. Though one can argue that Python is intended for synchronous scripting, it is obvious that with the *asyncio* module has now enabled Python to achieve more.

6 Citations

1 Humrich, Nick. "Asynchronous Python – Hacker Noon." Hacker Noon. September 23, 2016. Accessed December 04, 2017. https://hackernoon.com/asynchronous-python-45df84b82434.

2 Cannon, Brett. "How the heck does async/await work in Python 3.5?" Tall, Snarky Canadian. December 17, 2016. Accessed December 04, 2017. https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/.

3 "Python vs Node.js: Which is Better for Your Project." Python vs Node.js: Which is Better for Your Project | DA-14. Accessed December 04, 2017. https://da-14.com/blog/python-vs-nodejs-which-better-your-project.