CS131 Project. Proxy herd with asyncio

Calvin Chen
University of California, Los Angeles

Abstract

Wikimedia Architecture uses a LAMP platform, which uses redundant web servers for reliability. This leads to the lack of efficiency in (1) updates to articles will happen far more often, (2) access will be required via various protocols, not just HTTP, and (3) clients will tend to be more mobile service. To dealing with rapidly-evolving data, this paper gives a prototype of application server herd that used Python *asyncio* library as a replacement for LAMP platform and shows the potential of using asyncio to implement an application server herd.

1. Introduction

This project is to implement an application server herd with Python *asyncio* library. This application server herd should be able to update data more often and can be accessed with various protocols. Also, when updating data, it should be capable of propagating updates to all the other servers. This project shows whether *asyncio* a great fit to implement an application server herd or not in Python with comparison with other languages such as Java and Node.js.

2. Asyncio

In this section, I will briefly describe the research on asyncio as a potential framework for the application server herd.

2.1. Event Loop

The event loop is the central execution device provided asynio. Event loop allows users to create server and client transports for TCP, UDP, SSL and so on. In the document's protocol example, TCP echo server and client use the *AbstractEventLoop.create_server()* and *AbstractEventLoop.create_connection()* method respectively to receive and send data and close the connection.

2.2. Coroutines

Coroutines in asyncio can be generated by *async def* statement or by using generator. When calling a coroutine, the coroutine object doesn't do anything till scheduling its execution. Coroutines use *yield* to pause and reenter the routine. In the document's example in TCP connection, they create server and client as a coroutines and scheduled by event loop.

2.3. Asynchronous I/O

In asyncio, the event loops are actually asynchronous. Thus, the execution order of the coroutines in event loop can not be determined in advance. However, the execution of event loop is sequential, the data in each execution is asynchronous. In asyncio, we can access shared memory but still have to wait for I/O. The event loop can still run other code while awaited function waiting for I/O. In the case with los of I/O, the saving time can be substantial.

3. Implementation

This section describes the prototype of the application server herd. The application of server herd is based on python version 3.6.5. and is basically contained in *server.py* file. Users can use *python3 server.py [server_id]* to start a server called server_id. In *config.py*, I stored some constant that can be modified such as google api key and server_ids and ports. In *client.py*, I simply build a TCP client that users can called *python3 client.py [server_id] [message]* to send data to server_id. The following shows the application of the server herd.

3.1. Transports and Protocols in *asyncio*

I looked into transports and protocols, which are callback based API in *asyncio* to send and receive data. I created a server protocol called *EchoServerClientProtocol* that create the server that can be access by clients and other servers. The created server can run and wait for connection forever until a *KeyboardInterrupt* signal sending to server. The followings are going to show how the *EchoServerClientProtocol* deals with following incoming message.

3.2. IAMAT

IAMAT message is a message sent from a client begins with IAMAT and followed by client_id, location, and timestamp. Thus, when the message begins with IAMAT, I first check all the arguments given being valid or not. It should have three components: client_id, location, and timestamp. Also, the location should be the ISO 6709 notation and the timestamp should be sxpressed in POSIX time. If one of the arguments is invalid, the server will send a message back to the client with "?" and the input message. If all the arguments are valid, I then check the timestamp of the message and decide to update the client location. Since Python is dynamic type language, its object is runtime type bind-

ing, to check the arguments are valid, I also handle the exception if the arguments cannot be casted to desire type.

To save the client location, the server has a dictionary that map a client_id to its location. After updating client location, the server creates a message beginning with AT and followed by server_id, difference of timestamp, client_id, location, and timestamp. This AT message is then propagated to all the neighboring servers. When propagating message to other servers, I concatenate all the neighboring servers after the AT message, so that other servers won't send this message to the other servers that have been propagated by current server. This is how I designed my flooding algorithm, the detail of flooding will be discussed in later part. In this case, I can make sure that the propagation will not go into an infinity loop.

3.3. WHATSAT

WHATSAT message is a message sent from a client begins with WHATSAT and followed by client_id, radius from the client, and the bound on the amount of information received from Places API. Like IAMAT message, I also first check the arguments are valid or not. The radius and the bound should be at most 50 and 20 respectively. If one of the arguments is invalid, the server will send a message back to the client with "?" and the invalid command.

After checking the arguments, I query the Google Place API with the given client_id, client location, and the radius. Note that client location can be accessed by client_id. If the client_id is not existed, the server sends back the message with "?". To access Place API, I use asynchronous *aiohttp.ClientSession()* to access google map information with the arguments and API Key. In Python, this method run the code fragment in a single thread. Compared to other language that calls can be made in multi-thread such as Java, this method seems to be not efficient. However, this method actually does not block the server from receiving other incoming messages. Since Python uses the global interpreter lock to protects access to Python objects, preventing multiple threads from executing Python bytecodes at once, this *asyncio* method is actually a great fit in implementing application server herd to perform non-blocking parallelism.

3.4. AT

AT message is a message sent from other server begins with AT and followed by server_id, difference of timestamp, client_id, location, timestamp, and a list of flooded server which I appended in IAMAT. Like IAMAT message, I also first check the arguments are valid or not in case some clients send invalid AT message to the server. If one of the arguments is invalid, the server will send a message back to the client with "?" and the invalid command.

After checking the arguments, the server checks if the client stamp should be updated or not. If needed, the server updates and propagates the message to neighboring servers. Note that I've sent the flooded servers in the incoming message, so there is no need to propagate to the flooded ones. When propagating, the server does the same thing as IAMAT step, which is to concatenate the neighbors of this server.

3.5. Flooding Algorithm

To propagate message to neighboring servers, I define *EchoClientProtocol* to connect neighboring servers as a client so that the server can send message to other servers. In order to prevent infinity loop in sending message, I append all the neighboring server_ids to the message, and when the receiving server going to propagate AT message to its neighbors, it skips the server_ids that are appended after the AT message, and added its neighboring server_ids as before.

4. Comparison

In this section, I will compare Python implementation of server hear with Java and Node.js.

4.1. Compared with Java

To begin with, let's first look into Python's type checking. Python is a dynamically typed language, which determines the object type in runtime. When building a prototype for this project, this feature can lead to quick implementation since there is not much assumptions on the message. However, if we want to scale this application, static typing language like Java would be powerful to help developers express their assumptions on their implementation.

Then let's look at the memory management system. Python's memory management system makes it easier to develop because Python is a heap based memory model unlike Java, which has to allocate objects by developer. In this application server herd, when the objects are no longer referenced, the system deletes it on its own. This memory management system makes the implementation by *asycio* more efficient and also suitable for scaling the application.

Now let's look at the parallelism in this two languages. As mentioned in the previous part, Python uses the global interpreter lock to protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. *Asyncio* uses only single thread, but it uses asynchronous event loop to perform non-

blocking parallelism. In this case, if the system itself has many processors, Java may perform better than Python *asyncio*, which means it benefit on vertical scaling by increasing the number of CPU cores. However, *asyncio* itself performs well across systems, this benefit on horizontal scaling, which is a great fit for the application server herd.

4.2. Compared with Node.js

Node.js operates on a single thread like Python *asyncio*, using non-blocking I/O calls, which supports several concurrent connections without incurring the cost of thread context switching. Both Python *asyncio* and Node.js are both asynchronous and single-threaded. According to the creator of Node.js, this language was built to handle a lot of concurrent connections. However, compared to *asyncio*, Node.js is primarily used to build Web application. Python itself seems to have a larger number of packages, which may be a better fit for application server herd.

5. Conclusion

From above analysis, *asyncio* seems to be a great match for implementing the application server herd in various aspects. To conclude, Python itself is a great language to build a non-blocking asynchronous server, and for horizontal scaling the application, the asynchronous method is a better match than multi-threading method.

References

[1] 18.5. asyncio — Asynchronous I/O, event loop, coroutines and tasks. See https://docs.python.org/3/library/asyncio.html

[2] Python GlobalInterpreterLock. See https://wiki.python.org/moin/GlobalInterpreterLock

[3] Flooding (computer networking). See https://en.wikipedia.org/wiki/Flooding_(computer_networking)

[4] Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages. By Erik Meijer and Peter Drayton from Microsoft Corporation. See https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/rdl04meijer.pdf

[5] Node.js. See https://en.wikipedia.org/wiki/Node.js