Python Project: Proxy herd with asyncio

Scott Lee
University of California, Los Angeles
CS 131

Abstract

This paper discusses the asyncio asynchronous networking library as a candidate for replacing LAMP platform. As the LAMP architecture will be a bottleneck when updates to articles happen very often, access is required through various protocols, including HTTP, and clients tend to be more mobile, asyncio might be the good replacement for this architecture. By looking at the strength and weakness of asyncio library, justification will be provided. In addition, asyncio will be compared to another candidate, Node.js.

1. Introduction

Wikimedia Architecture uses LAMP platform based on GNU/Linux, Apache, MySQL, and PHP with multiple and redundant web servers behind a load-balancing virtual router for readability and performance. It took 30,000 HTTP requests per second during peak-time. In addition, 110 million revisions, which are basically the updates to articles by clients, were made, according to the report from Wikimedia Foundation Inc. in 2007. Most of requests by clients and media storage are based on HTTP. However, imagine that there are far more often updates to articles, accesses that are required through various protocols, not just HTTP, and more mobile clients. Then, this service will be a bottleneck in the LAMP architecture.

A potential candidate would be application server herd where the multiple application servers communicate directly to each other, instead of communicating via the core database and caches. In this mean, the inter-server communication can significantly increase the performance on rapidly-evolving data, such as GPS-bases locations and ephemeral video data. In order to implement the application server herd, this paper will look into asyncio asynchronous networking library in Python as well as a small prototype of the new architecture using asyncio.

2. Asyncio

Asyncio module 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.

2.1. Basic Components

The following code is an example use of asyncio library provided by Python asyncio library manual.

```
import asyncio

async def hello_world():

        print("Hello World!")

loop = asyncio.get_event_loop()

coro = hello_world()

loop.run_until_complete(coro)

loop.close()
```

The variable *coro* represents a coroutine object, specifically *async def* type of coroutine. It is obtained by calling a coroutine function and represents a computation or an I/O operation that will complete eventually. "Calling" the coroutine function doesn't start its code running; it doesn't do anything until it's scheduled its execution. The ways to let it start running are either calling *await coroutine* or *yield from coroutine* from another coroutine or schedule its execution using *ensure_future()* function or *AbstractEventLoop.create_task()* method. In the above example, an async def type coroutine *hello_world()*'s object is returned to *coro*, and the coroutine is triggered using *run_until_complete()* function. It blocks call which returns when the *hello_world()* coroutine is done.

The more details on asyncio library are provided along with the prototype in the following sections.

3. Prototype

While the application server herd can be used for any types of rapidly evolving data, this prototype only simulates inter-communication of GPS-based location. There are five servers: 'Goloman', 'Hands', 'Holiday', 'Welsh', and 'Wilkes', and the communication is bidirectional. The communication pattern is the following:

Goloman → Hands, Holiday, Wilkes

Hands → Wilkes

Holiday → Welsh, Wilkes

There are three commands for the message for communication: IAMAT, WHATAT, and AT.

The messages propagate through the servers with flooding algorithm. The flooding algorithm is a simple computer network routing algorithm where every incoming packet is sent through every outgoing link except the one it arrived on. For example, when IAMAT message is sent to Goloman from a client, then the location of the IAMAT message will be propagated to the connected server of Goloman (Hands, Holiday, Wilkes) first. Then, Hands, Holiday, and Wilkes start propagate the data to the neighboring servers except Goloman, as it's the server the message arrived on. In fact, the flooding algorithm actually "flood" the date from the top to the bottom. You can think this mechanism as Breath First Search (BFS).

In order to run a server:

python3 server.py <ServerID>

3.1. Design Overview

The overall communication is based on asyncio transports and protocols with callback-based API. Transports are classes provided by asyncio to abstract various kinds of communication channels. Every time a connection is made, *BaseProtocol.connection_made(transport)* is called. The argument, *transport*, is the transport representing the connection. Using this transport, the 'peername' is obtained, with *transport.get_extra_info('peername')*. The peername is the remote address to which the socket is connected. Beside the peername, socket itself or sockname can be obtained as well. Similarly, *BaseProtocol.connection_lost(exc)* is called when the connection is lost or closed. The argument is either exception object or *None* (EOF is received). Since *connection_made()* and *connection_lost()* are called exactly once per successful connection, it is very reliable for protocol implementation and logging. In addition, *Protocol.data_received(data)* is called when some data is received. At the time data is received on a server, it detects whether the first field of the message is IAMAT, WHATSAT, or AT, send the response, and finally drop the connection.

The following diagram shows the logic flow:

Start → *connection_made()* --(flood)→ *data_received()* → End

In this prototype, there are three Protocol subclasses: ProxyServerProtocol, ProxyClientProtocol, and HTTPProtocol.

ProxyServerProtocol handles most of jobs of location sharing. In *main()*, a coroutine, *server*, is created using *AbstractEventLoop.create_server()* with localhost IP address (127.0.0.1) and appropriate server port. Then, *server*, is run by a loop created by *asyncio.get_event_loop()*. The *server* loop is *"run_forever()"* until *AbstractEventLopp.stop()* is called. (When KeyboardInterrupt is detected, the loop stops). Therefore, the server constantly receives and responses any message or request on the assigned port.

As described above, ProxyServerProtocol receives message from client, sends back appropriated responses based on the command, update client information, and logs the actions taken.

ProxyClientProtocol serves the jobs on the client side with *connection_made()* and *connection_lost()*. Once a successful connection is made, it writes the message into transport and logs the action. Similarly, for the connection lost, it closes the transport and logs the action.

Finally, HTTPProtocol serves as a connector to the Google Places API. It contains *connection_made()* and *data_received()* likewise. When a successful connection is made with Nearby Search request in the form of manual HTTP GET, it writes the request into the transport. Then, when the Google Places API sends back the data based on the request, *data_received()* is triggered. At this time, it parses the json data from Google Places API using *json* library. The HTTP GET follows HTTP/1.1 format with the syntax of the following:

GET <url_target> HTTP/1.1

Host: <url_host>

e.g.

GET '/maps/api/place/nearbysearch/json?...' HTTP/1.1

Host: 'maps.googleapis.com\n\n'

Initially, I tried to make the request using aiohttp library. However, I failed to embed async *aiohttp.ClientSession()*, as *data_received()* cannot take coroutine as input data. In addition, to use aiohttp library, *data_received()* must be *async* function, which leads the whole structure broken. Therefore, I decided to manually make HTTP request with SSL credential using ssl library. My trial on aiohttp library is commented in *server.py*.

3.2. IAMAT Command

IAMAT command takes three arguments: client ID, latitude and longitude in ISO 6709 notation, and time sent in POSIX time.

i.e. *IAMAT <ClientID> <Location> <Time_sent>*

e.g. *IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997*

When this request is sent to a server, then the server will response with a message whose first field is AT. The arguments of AT message are the ID of the server that got IAMAT message from the client, the difference between the receive-time and time sent, and the remaining fields are simply copy of IAMAT message. The time difference is possibly negative due to clock skew.

i.e. *AT <ServerID> <Time_diff> <ClientID> <Location> <Time_sent>*

e.g. *AT Goloman +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997*

'Goloman.log' when the above IAMAT message is sent to Goloman (port = 11645):

```
Serving on ('127.0.0.1', 11645)
New connection from ('127.0.0.1', 63801)
Message received: 'IAMAT kiwi.cs.ucla.edu …'
Location updated for kiwi.cs.ucla.edu
Response sent: 'AT Goloman +8237559.384464264 …'
connection from ('127.0.0.1', 63801)
Connected to server Hands
Propagated location to Hands
Connected to server Holiday
Propagated location to Holiday
Connected to server Wilkes
Propagated location to Wilkes
Dropped connection to server Hands
Dropped connection to server Holiday
Dropped connection to server Wilkes
```

A special scenario is that once the first IAMAT message was sent, and then another IAMAT was sent. In this case, only the latest location will be updated.

3.2. WHATSAT Command

WHATSAT command contains three arguments: client ID, radius in kilometer, and the number of information requested.

i.e. WHATSAT <ClientID> <Radius> <Bound>

e.g. *WHATSAT kiwi.cs.ucla.edu 10 5*

It will return the result in json format from Google Places API Nearby Search based on the location of the client around the given radius up to the number of information. For example, if *WHATSAT kiwi.cs.ucla.edu 10 5* is sent, it will return five POIs within 10 km around the location of *kiwi.cs.ucla.edu*. Since its response requires the location of the client, AT message must be sent beforehand in order to update the location, or it will treat WAHTSAT message as an invalid message.

With the manually created HTTP request, SSL credential is also required to receive the result from Google Places API. Using ssl library, *create_default_context()* returns a new SSLContext object with default settings, and *ssl.CERT_NONE* sets how to verify other peers' certificates to the context. *CERT_NONE* means no certificates will be required.

3.3. AT Command

The "message" with AT command is not actually a message that can be sent by a client. It is purely for propagating IAMAT message throughout the server herd using flooding algorithm. The fields of AT message is exactly the same as IAMAT message, except the extra client ID(s) at the end to denote where the IAMAT message comes from.

3.4. Invalid Command

Invalid command manages literately the commands that are not one of IAMAT, WHATSAT, or AT command. The response of an invalid command is the following:

? <Invalid_command> <Rest>

e.g. *YOLO clientX 3.14 1592*

→ *? YOLO clientX 3.14 1592*

Besides the actual invalid commands in name itself, requesting WHATAT command before updating location with AT command will be treated as invalid command. In addition, if the arguments of AT command are not in the appropriate forms, it is also treated as an invalid command. For example, the *Location* argument must be in ISO 6709 format, and *Time_sent* must be in POSIX/Unix time format. Likewise, in WHATSAT command, the *Radius* argument must be non-negative and less then 50 km, and *Bound* argument must be non-negative and less than 20.

4. Recommendations

This section will discuss the comparison between Python and Java in terms of their suitability for server herd application, type checking, memory management, and multithreading. In addition, it will also compare Python's asyncio library to Node.js.

4.1. Suitability

The asyncio module contains a pluggable event loop with various system-specific implementations, transport and protocol abstractions, concrete support for TCP, UDP, SSL, subprocess pipes, and delayed calls, synchronization primitives, etc. Those built-in features are perfectly suitable to this kind of application. As the prototype has shown, they facilitate the implementation of the proxy server herd. With the generic features and functions provided in the manual, the implementation was smooth and easy.

Furthermore, implementing the application server herd in Python is much easier then implementing it in C++ or Java. As a small portion of comparisons, Python's built-in functions, such as *join(), split(), %,* etc, significantly increases efficiency in time and difficulty. If the prototype was written in C++, *split()* function would take some lines and be not flexible enough when parsing JSON data.

In terms of abundance of extra libraries from the community, Python has many libraries outside of the world, such as aiohttp library we used. As implementing the actual application of server herd, they will facilitate even more than they did in the prototype.

4.2. Python Type Checking

Python's variables are dynamically typed. In fact, you declare variables without giving them a specific data type. The types of variable are automatically assigned based on what data was passed to. While the dynamic types facilitate the implementation in terms of time, as we don't need to think of the type of a variable when declaring it, the dynamic typing decreases the code-readability. In statically-typed languages, the data types are explicitly specified when a variable is declared; therefore, when a person reads a function in the middle of the whole source code, the person is more likely to understand better with the data type written. However, in dynamically-typed languages like Python, in order to understand a function in the middle, the person has to trace back to the top function. This might sound not-a-big-deal. Even for me as a writer for this prototype, when I come back to continue developing after 2-3 days, I spent a good amount of time to get sense of where I was and what was the problem. The absence of data types also exists for functions.

4.3. Python Memory Management

Unlike the significant disadvantages from dynamic type checking compared to statically typed languages, Python's memory management is more efficient than Java's. The main difference is the different garbage collector strategies. Python uses a garbage collector based on reference counting. It frees objects as soon as they have no references. On the other hand, Java uses a Generational garbage collector. As the name indicates, it doesn't delete the memory allocation in the heap right away, but it keeps the underlying memory and reuses it for future object allocation. Considering this difference in term of a huge server herd application, Python is far more efficient than Java.

4.4. Python Multithreading

The server herd application using Python's asyncio library is based on a single thread, while a Java version must be multithreaded. There are obvious pros and cons on whether being single or multi-threaded. Python version, which is single threaded, will output the same performance in any hardware/system. Since asyncio only uses a single thread, the number of cores that are capable of using n-threaded application doesn't really matter. However, multithreaded application on the multiple cores on the hardware would give better output, proportionally to the number of cores. Assuming the availability of a powerful hardware, Java wins over Python in multithreading.

4.5. Asyncio vs. Node.js

Asyncio library and Node.js frameworks' implementations are very similar. They both are a single threaded asynchronous event driven. However, Node.js is more flexible in terms of event loops. Asyncio library's method to start an even loop is always a blocking call. For example, *create_server()* method in asyncio library is a blocking call, as it has to be done beforehand to execute the following codes. On the other hand, Node.js simply starts the event loop after executing an input script and exits the event loop when there are no more callbacks to perform. This difference is quite interesting; however, in perspective of application server herd, this difference doesn't make a huge difference. The staring and closing event loop usually take place once or few. The main factor that determines the application's performance is how the callback are managed and the asynchronous jobs are chained.

5. Conclusion

Providing explicit use of the Python's asyncio library and the general comparisons between Python and Java in various perspectives and asyncio library and Node.js, it is clear that the recommendation of Python with asyncio library is the best choice for Wikimedia-style server herd application.

References

[1] Eggert, Paul. Project. *Proxy herd with asyncio*, UCLA Computer Science Department, 26 May 2018,

http://web.cs.ucla.edu/classes/spring18/cs131/hw/pr.html

[2] Bergsam, Mark. *Wikimedia architecture*, Wikimedia Foundation Inc., 2007,

https://wikitech.wikimedia.org/wiki/File:Bergsma_-_Wikimedia_architecture_-_2007.pdf

[3] *18.5 asyncio – Asynchronous I/O, even loop, coroutines and tasks*, Python Software Foundation, 6 Jun. 2018,

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

[4] *Flooding (computer networking)*, 4 Apr. 2018, Wikipedia,

https://en.wikipedia.org/wiki/Flooding_%28computer_networking%29

[5] *18.5.4. Transports and protocols (callback based API)*, Python Software Foundation, 6 Jun. 2018,

https://docs.python.org/3/library/asyncio-protocol.html#asyncio-transport

[6] Fielding, R., et al. *5 Request*, W3, 2014,

https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html

[7] *18.2. ssl – TSL/SSL wrapper for socket objects,* Python Software Foundation, 6 Jun. 2018,

https://docs.python.org/3.6/library/ssl.html

[8] *About Node.js®*, Linux Foundation, n.d.,

https://nodejs.org/en/about/