Application Server Herd Design with Python and Asyncio

Ken Gu, University of California Los Angeles

Abstract

With a classical LAMP architecture, the central bottleneck of the core application server becomes a problem when client updates are often, accesses require various protocols and clients are mobile. An "application server herd" in which proxy servers communicate directly to each other for updates and to the core database is promising for constantly evolving data. I implemented a prototype using Python's asyncio library. Through the process of working and understanding this library I compare this framework to other methods of implementation such as a Java-based approach and an approach using Node.js. I detail possible language specific concerns such as Python's type checking, memory management etc. as well as overall performance.

1 Introduction

While the LAMP model of web service stacks is suitable for dynamic web applications such as Wikipedia, to service the needs constantly updating clients, such as smart phone applications which repeatedly update their GPS locations, the approach falters. With many database accesses all at once, the one application server becomes a severe bottleneck and adding new servers is too costly from a software development perspective. The application server herd architecture, in which multiple application servers are set up to have direct communication with each other for rapidly-evolving data and to the core database server for more stable data appears to be a good candidate for this problem. Rather than communicating to the central database when receiving rapidly evolving data, an application server propagates this data to the other servers in the herd such that all servers have the most up to date data.

Additionally, Python's asyncio asynchronous networking library is a plausible method of implementing the application server herd architecture. However, it is yet to be known how asyncio works in practice. Questions about application maintainability, reliability, and develop difficulty, need to be answered. Thus, to determine whether asyncio was a suitable choice I researched about asyncio and built a simple prototype to understand asyncio's validity in fulfilling this server herd platform. Everything researched and built with asyncio was with Python 3.6.4.

2 Asyncio

Asyncio is a Python concurrency module. Its main selling point is that it can easily implement asynchronous IO using coroutines while using event loops to schedule these coroutines. An understanding of these characteristics is necessary to discern the pros and cons of asyncio.

2.1 Coroutines

Coroutines are an extension of generators which enable cooperative multitasking on one thread. They allow code consuming a generator to send a value back into the generator. In effect it allows a function or computation to communicate with its caller, passing information between computations during the middle of a program. Consider the following example modified from Brett Chalabian's discussion:

```
1  def echo():
2      while True:
3          echo_val = (yield)
4          print("Coroutine recieved: {}".format(echo_val))
5
6
7  coro = echo()
8  next(coro)
9  count = 0
10 while count <= 3:
11     coro.send(count)
12     count = count + 1
```

Here, the *yield* keyword is used in line 3 so that the variable *echo_val* in echo can receive the value of *count* outside. In line 7 *coro* receives a generator. Only in line 8 does *echo* run and it only runs until the yield is reached in which control is handed back to the caller. Thus, a generator object does not do anything until it is iterated over or scheduled. In the above example the coroutine is iterated over and the resulting output is:

```
Coroutine received: 0
Coroutine received: 1
Coroutine received: 2
Coroutine received: 3
```

This example illustrates how the consuming code can take the generator output and can perform other computations while also sending and receiving values from other

coroutines. As a result, many different parts of code may seem to all be running simultaneously as coroutines are being yielded back and forth.

In Python 3.6.4 to implement a coroutine, the word async must prefix a function. For example, *async def func()* defines a coroutine. Likewise, rather than using the yield from syntax, asyncio requires the *await* keyword. For instance, in *a = await func(x)*, *a* will receive whatever *func* yields.

2.2 Event Loops

Event loops provide an API to allow coroutines to be scheduled. It runs on the main thread and once started, coroutines can be passed to it for the event loop to execute. To understand event loops further, it is necessary to examine the source code.

Looking at the source code on GitHub for the *asyncio.get_event_loop()* method in asyncio/events.py on lines 596 to 603, asyncio checks whether it is on the main thread and no other loop exists before it creates and returns a new event loop[5]. If it is not on the main thread it raises a runtime error. This means that only one loop can run and on only one thread enforcing the fact that asyncio is single-threaded.

```
596         if (self._local._loop is None and
597             not self._local._set_called and
598             isinstance(threading.current_thread(), threading._MainThread)):
599             self.set_event_loop(self.new_event_loop())
600         if self._local._loop is None:
601             raise RuntimeError('There is no current event loop in thread %r.'
602                                % threading.current_thread().name)
603         return self._local._loop
```

Once a loop is received, the loop can now run coroutines. Some methods to note include the *AbstractEventLoop.run_until_complete(future)* method which when passed a coroutine runs the coroutine until it is finished and the *AbstractEventLoop.run_forver()* and *AbstractEventLoop.stop()* pair, the former of which runs until the latter is called. Likewise, to explicitly schedule coroutines, asyncio offers the *AbstractEventLoop.create_task(coro)* method. Thus, an event loop can be created and ran with the following:

```
loop = asyncio.get_event_loop()
loop.run_until_complete(coroutine)
```

2.3 Server Related Methods

In addition to understanding coroutines and event loops, it is important to see what support and APIs asyncio offers for various protocols and server functionality.

Asyncio conveniently offers an *asyncio.start_server(client_connected_cb, host=None, port=None, *, loop=None, limit=None, **kwds)* coroutine method. The key thing to note is that the *client_connected* argument is a callback function which takes a StreamReader and StreamWriter that can read and write to the client and returns a coroutine that can be run in the event loop. The use of callbacks ensures that client messages are received when they eventually happen. Therefore, all the functionality of the server can be implemented within this callback passed into *start_server*. Additionally, there is an *asyncio.open_connection(host=None, port=None, *, loop=None, limit=None, **kwds)* coroutine which takes a host server and returns a StreamReader and StreamWriter to communicate with the server. Finally, there are other libraries such as aiohttp which run on top of asyncio to enable easy asynchronous HTTP functionality.

3 Prototype Design

The herd server design follows closely the specifications mentioned in the CS131 project spec [3]. Understanding the implementation work necessary is important to answering the question of whether Python and asyncio is suitable for an application server herd.

The main file which contains the server implementation is *server.py* which contains a *Server* class that handles incoming messages, logs connection information, and stores relevant client data. In *config.py* server names and port numbers are stored in addition to Google's API key and request URL. Finally, in *parse.py* there is a class *ParseMessage* which parses the incoming messages to the server into its various component fields.

3.1 Asyncio for Server IO

I decided to use mainly the methods mentioned in Section 2 to implement the server as once those methods were understood the API was relatively straightforward to implement.

Upon creation of a server object and event loop, I pass the *server.handle_client* coroutine, the main coroutine to be ran, to *asyncio.start_server* and put the coroutine returned to the *loop.run_until_complete* method to run in the loop. Furthermore, this loop is then set to run forever until a keyboard interrupt is detected such that it continuously listens for incoming messages. Thus, with a few lines most of the

IO part is set up and all that is left is the implementation of the asynchronous functionality of the server. This is achieved through more coroutines and using await on these coroutines to pass control between computations. The next section details the server implementation and a common theme to note is how functions and methods await over many levels to achieve asynchrony.

3.2 Server Implementation

Servers communicate with each other through a flooding algorithm upon receiving IAMAT requests. In addition, servers can send HTTP requests to Google Maps API when receiving an WHATSAT request which the herd server then passes what Google returns to the client. Thus, the server must know how to handle IMAT and WHATSAT commands. For my implementation I also defined a FLOOD command that the server must handle, which tells the server to store the information as well as which neighboring servers it needs to propagate the message to.

Server.handle_client is the top level coroutine that once called by the event loop passes the StreamReader and StreamWriter to the *Server.handle_maintained_client* coroutine and schedules it in the event loop using *loop.create_task*. With this implementation *Server.handle_client* is enabled to handle new incoming clients while Server.*handle_maintained_client* deals with opened connections.

Server.handle_maintained_client then awaits on the StreamReader and passes the decoded message to *Server.handle_message* and awaits for the response message. Once a response message is received Server.handle_*client* either writes the response messages into the StreamWriter and awaits on itself for future messages on that connection or closes the connection if it knows the message received was a FLOOD message.

Server.handle_message then create a *ParseMessage* object from the incoming message and determines which class of command it just received. On a valid IAMAT it awaits on *Server.handle_iamat*, which determines whether to store and propagate the client's information based on the timestamp through calling *Server.most_recent_client* and gets the returned response message. If *Server.handle_iamat* needs to flood nearby servers it awaits on *Server.flood_neighbors*. On a valid WHATSAT request, *Server.handle_message* awaits on *Server.handle_whatsat* which if the client's information exists on the server, it creates a request URL based on the WHATSAT query and awaits on *request_gmaps*. This then uses the aiohttp module to send a HTTP request to the Google Maps API and gets the JSON response. Finally, on a valid FLOOD command, *Server.handle_message* calls *Server.most_recent_client* awaits on *Server.flood_neighbors*.

In short, the prototype implementation used asyncio's event loop and coroutine functionality extensively with the *await* and *async def* syntax to create coroutines and yield from coroutines. Although difficult to learn at first, once understand, asyncio provides simple interface for asynchronous functionality.

3.3 Implementation Problems

Most implementation problems faced during implementing the application herd server was due to a misunderstanding of how asyncio and coroutines worked. A big roadblock was figuring out how to maintain a connection with a client after a client has connected to the server. All the examples provided online such as in the official documentation had the servers closing the connection after writing to the client. However, with a bit of more research and testing on an echo server, I was able to figure out that I could create a new task that run a coroutine to specifically handle a maintained connection such that the server is still able to handle new connections. I also tried to maintain the connection between servers for their flooding messages but could not figure out a working implementation. Thus, due to time constraints I decided that the server sending a flood message to a neighboring server would oversee opening the connection and then closing the connection once the message is sent. One final major problem faced was figuring out when to stop reading client data. Asyncio offers several methods for reading, namely read, *readline*, and *readuntil*. I tried *readline* at first, but it seemingly read empty strings for no reason. Upon further inspection, I realized that *readline* does not consume the newline and thus on the next iteration of *readline* it would read the newline instead. Therefore, I implemented read with *readuntil* which reads up until a newline and consumes the newline.

4. Suitability for Application Herd Server

Through the process of implementing the prototype and understanding the details of the asyncio library, asyncio offers several advantages that make it suitable for application herd severs.

4.1 Asynchronous IO

The hallmark of asyncio is its API for implementing asynchronous IO. As Section 2 detailed, asyncio offers several options for implementing a server that handles its requests asynchronously. Likewise, in Section 3, asyncio is used extensively to suit the functionality of the application herd server.

As an application herd server receives constantly changing updates from multiple clients it needs a server model that

can handle multiple connections with other herd servers and clients. While Apache in the LAMP model can handle multiple connections, it handles multiple clients with multiple threads, with each new client connection becoming a separate process. However, as the number of connections increase such as with the example of many smartphones needing to update their GPS location, the multiple threads method becomes resources heavy and hard to manage.

On the other hand, asyncio's event loop and coroutine scheduling allows it to asynchronously handle incoming messages. As noted in Section 2.2, everything is running on one thread so difficulties involving multithreading such as synchronization does not apply. Therefore, while a server is handling a client, it can connect with another client and receive updates by passing control through scheduling of coroutines. For instance, in the prototype, while a server is waiting on the response of a HTTP request to Google's servers, it can handle new client connections, accept flood messages from other herd servers or process other clients' IAMAT and WHATSAT commands.

4.2 Abstraction

From a programmer's perspective, asyncio is also favorable in that it abstracts a lot of the lower level networking details away from the implementation. As a student who has never taken any networking class or has had exposure to network programming, I found it reasonably simple to understand how to implement a basic echo server and client from the official documentation. The exact details of the protocols are abstracted in which the programmer is only left to deal with a StreamReader and a StreamWriter. As mentioned in Section 2, only a few event loop methods, asyncio server coroutines, in addition to understanding the async def and await keyword are needed to implement a fully functioning server. Although it was not easy to understand the completely new abstract concept of coroutines and event loops at first, once understood, implementation is simple, and the methods provided are powerful enough to suit most needs.

4.3 Python

Python is a strong language for handling the logic involved with received messages. Compared to other popular languages such as Java and C++, Python's powerful library of built in types such as dictionaries, strings, and lists makes it extra easy and to implement the functionality of handling server messages. For instance, Python's *str.split* method and the *str.join* method allowed easy parsing of the individual fields of client messages. Python also offers a wide range of libraries such as json, csv, pandas etc. to handle data formats that could be passed around between servers. For instance, with the prototype, I used the json library to take in Google's response and cleanly indent it. In addition, Python has useful features such as list comprehension and lambda functions to provide quick functionality. For example, to remove a series of two or more newlines in the JSON response, I used a list comprehension that implemented the conversion in one line.

For a programmer, Python is a huge win in saving programmer time as it abstracts the lower details and allows the programmer to focus on the higher-level problems.

5 Addressing Concerns of Scaling with Python

Despite Python being a powerful language for implementing the functionality of an application herd server there are still several concerns about how Python's type checking, memory management and multithreading would affect the application as it scales. This section aims to address these concerns.

5.1 Type Checking

While Python's dynamic type checking enables fast prototyping of small programs by saving the programmer time from explicitly defining the types of variables, it is more susceptible to runtime errors that are related to mismatched types and operations on mismatched types that a statically typed language would not have. However, in my experience of programming the application herd server, this does not affect the reliability of the code if the code is well tested in the all the possible ways it can perform. Nevertheless, for development, the lack of static type checking may lead to unexpected runtime errors that are hard to debug. For instance, sometimes I received runtime errors that told me I was performing a string method on a list. Thankfully these error messages are relatively easy to parse but I can see how for programs built on a larger scale, with longer runtimes, it may be hard to debug or test all the possible run time errors. Similarly, without specifying types in the code, the code may be hard for readers to follow. In languages such as C++ and Java where types are explicitly written, a reader would likely have an easier time following the flow of programming.

Thus, for projects with very large code bases and upmost reliability standards, Python may not be the best language of choice. In the case of the application server herd, Python is still a great language as the project scales mainly with the number of clients and servers and not the code base.

5.2 Memory Management

Python manages heap memory through reference counting and assigns different variables who have the same values to the object in memory. As opposed to Java's generational and mark and sweep method for its garbage collection

which checks for unused objects at certain time intervals, Python's method is advantageous when there are not a lot of memory resources to use on the server. This is because unused objects are freed immediately when the reference count is zero and only one instance of an object is used in which multiple variables can refer to.

However, with Python constantly having to check an object's reference count, it also adds overhead which may slow down the performance of the program. There is also the possibility of reference cycles in which unused memory is not freed. Thankfully, this specific issue is addressed in the latest CPython implementations of Python where Java-style garbage collection is added when memory is low.

Thus, memory management is not a big issue and so long as the programmer is cognizant of the sizes of the data structures and objects the programmer is using, memory usage should also not be an issue.

5.3 Multithreading

As mentioned earlier a Python asyncio implementation would only be running on one thread. Therefore, issues associated with concurrency and synchronization are avoided. However, this means asyncio does not allow vertical scaling by increasing the number of CPU cores. Meanwhile, a Java implementation would most likely be multithreaded. Although the work that these threads can do is dependent on the system hardware but for large scale applications, an implementation that can effectively use high end multi-processor hardware would enable performance gains.

However, in terms of handling client requests, asyncio can scale well too. As Pawel Miech has tested in his blog post, asyncio is able to handle 1,000,000 client requests in 9 minutes on his host machine, averaging 111,111 requests per minute [4].

6 Asyncio vs Node.js

Asyncio is like Node.js in that both approaches use event-driven asynchronous IO solutions running on a single thread for networking applications. From the programmer's perspective the two are largely the same with asyncio using Python and Node.js using JavaScript with both languages being dynamically typed and offering support for object-oriented programming, event loops, and scheduling of code through callbacks or coroutines.

At a lower level, Node.js is a runtime environment built on Chrome's V8 JavaScript engine. Asyncio is a Python library which runs on a Python runtime environment. With the V8 engine JavaScript code is compiled directly to machine code. Meanwhile, Python and asyncio is mainly interpreted into bytecode. Likewise, the V8 engine uses a generational garbage collector as opposed to the reference count method mentioned for Python.

Both approaches offer similar functionality so the language choice for implementing an application server herd is mainly up to the programmer. However, with Python's strong library of helpful built-in methods and types, it may be the slightly better option.

7 Conclusion

Python is a language that offers many tools and abstractions to the programmer. However, with that freedom, there comes a greater responsibility to have code is well managed, documented, and tested as it scales. Although Java's static type checking and use of multithreading for network applications may improve the performance of the application, regarding the application server heard, the main bottle neck is still network speeds. Thus, while actual computation time for processing messages may be longer with a Python asyncio implementation, the end user would barely notice. Ultimately given that a programmer's time is becoming an increasingly expensive resource and computer hardware is getting cheaper every year, the Python asyncio combination is a good choice for implementing an application server herd.

8 References

[1] "18.5. Asyncio - Asynchronous I/O, Event Loop, Coroutines and Tasks¶." *18.5. Asyncio- Python 3.6.4 Documentation*, docs.Python.org/3/library/asyncio.html.

[2] Cannon, Brett. "How the Heck Does Async/Await Work in Python 3.5?" *Tall, Snarky Canadian*, Tall, Snarky Canadian, 17 Dec. 2016, snarky.ca/how-the-heck-does-async-await-work-in-Python-3-5/.

[3] Eggert, Paul. "Project. Proxy Herd with Asyncio." *Project. Proxy Herd with Asyncio*, web.cs.ucla.edu/classes/winter18/cs131/hw/pr.html.

[4] Miech, Pawel. "Making 1 Million Requests with Python-Aiohttp." *Python Programming, Web, Data Science*, Pawel Miech, 8 Nov. 2016, pawelmhm.github.io/asyncio/Python/aiohttp/2016/04/22/asyncio-aiohttp.html.

[5] Python. "Python/CPython." *GitHub*, 10 Mar. 2015, github.com/Python/cPython/tree/3.6/Lib/asyncio.

[6] Ronacher, Armin. "I Don't Understand Python's Asyncio." *I Don't Understand Python's Asyncio | Armin Ronacher's Thoughts and Writings*, 30 Oct. 2016, lucumr.pocoo.org/2016/10/30/i-dont-understand-asyncio/.