Application Server Herd Design using Python's asyncio

Sanketh Hegde, *COM SCI 131*

Abstract

A classic LAMP architecture might prove to be a bottleneck if (1) updates happen far more often, (2) accesses happen on protocols other than HTTP, and (3) clients are more mobile. An application server herd might solve this issue by allowing servers to communicate rapidly-evolving data directly to each other without having to speak to a database. This paper discusses a possible implementation of an application server herd using Python's asyncio library.

1. Introduction

In order to understand the strengths and weaknesses of using asyncio to build an application server herd, I built a simplified prototype that uses asyncio to propagate client location data among five servers named Alford, Ball, Hamilton, Holiday, and Welsh. This paper describes the design of this prototype and evaluates how well asyncio performs in practice.

2. Research on asyncio

In order to understand how asyncio would work in the context of an application server herd, I first did a fair amount of research on how to use asyncio and how it was implemented. Information was obtained from documentation, source code, CS 131 lecture, and CS 131 lab.

2.1. Coroutines

The part of asyncio that makes it truly asynchronous is the relationship between the event loop and coroutines. Coroutines generalize subroutines through the use of multiple entry points. They can use these entry points to pause and resume execution of some computation or I/O operation. Here's an example of a simple generator based coroutine:

```
def coro():
  msg = yield "Hello"
  yield msg
cr = coro()
print(next(cr))
print(cr.send("Bye"))
```

The output being:

```
Hello
Bye
```

This very simple example shows how the 'yield' keyword can be used to create a generator that pauses execution (after printing "Hello") and resumes at another point (to print out "Bye"). It's important to note that calling a coroutine does not start executing its code. As seen above, the line 'cr = coro()' does not execute the function coro, but simply stores the coroutine object in the variable cr. The call does not do anything until it is iterated over or scheduled.

Looking over the source code for asyncio, 'base_events.py' contains many of the functions used in the server herd prototype. An example is the function 'create_connection'. Its implementation contains 'yield' and 'yield from', which means these functions are actually generators that can be paused and resumed. This makes sense, because 'create_connection' is supposed to return a coroutine, as specified by the documentation.

2.2. Event loops

The example above uses Python's next function to iterate over the generator, but coroutines can also be scheduled, which is where asyncio's API comes in. Asyncio allows for the creation of an event loop that automatically schedules coroutines. Once started, an event loop waits for and dispatches events and messages in a program.

An example that I found easy to grasp is the JavaScript event loop that is running in every browser. When events like clicks and hovers happen, they are given to the event loop. The event loop deals with a certain event by calling some callback function designated to handle the event appropriately, and then it continues to loop in order to deal with any future events or messages. It can almost be thought of as a 'while True' loop that is constantly collecting and dispatching any occurring events or messages.

To create an event loop in Python:

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

Events can be scheduled using the following functions:

```
loop.create_task(some_coro)
asyncio.ensure_future(some_coro, loop)
```

The above functions schedule the execution of some coroutine within the event loop.

To actually start and close the event loop, respectively:

```
loop.run_forever()
loop.close()
```

The above five functions, in addition to some other variants, formed the foundation for my implementation of an application server herd prototype using asyncio.

2.3. Asynchronous

One of the confusing aspects of event loops is the fact that they are asynchronous. This means that the execution order of any scheduled coroutines is not known ahead of time.

One misconception that I had is that this means that the event loop is multithreaded; this is not true. An event loop created by asyncio runs, by default, on one thread. The fact that the event loop is asynchronous, then, means that coroutines are actually interleaved and it is impossible to know ahead of time which scheduled coroutine will finish first. The advantage of an asynchronous event loop comes with external calls to libraries and servers.

For example, if one of the five servers included in the prototype makes an HTTP request to the Google Places API, that server need not wait for an HTTP response before receiving other connections. In this way, a server can create a task and continue on with its normal execution without waiting for that task to complete. The use of callbacks through protocols, as described in the next section, ensures that responses are received when they do eventually happen.

3. Prototype design

This section describes the approach that I took to creating a simple application server herd using my research on asyncio above. Through descriptions of the prototype implementation, this section provides some of the background for Section 4, which addresses the question of whether asyncio is a suitable framework for an application server herd.

The majority of my implementation resides in *server.py*. The source file *config.py* contains some constants, like port numbers and API endpoints, that are used in *server.py*.

To start a specific server:

```
python3 server.py Alford
```

3.1. Transports and protocols

I decided to utilize asyncio transports and protocols, a callback based API, to implement the server herd. Transports are used to represent various server communication channels, while Protocols are classes that represent generic network protocols. Protocol classes use transports to read and write data.

I created three main Protocol subclasses that represent different types of network connections:

1) ProxyServerClientProtocol
2) ProxyClientProtocol
3) PlacesHTTPClientProtocol

ProxyServerClientProtocol serves as the primary protocol used to represent the server itself. The server coroutine is created using asyncio's AbstractEventLoop.create_server function. The loop is set to run forever until a KeyboardInterrupt is detected; thus, the server continuously listens to whatever port it has been assigned.

Being the protocol for the primary server entity, ProxyServerClientProtocol contains functions that receive messages, deal with them, and send appropriate responses. These messages include IAMAT, WHATSAT, and AT commands. This class also contains a static variable that maps client ID's to their location AT stamps.

On a valid IAMAT request, the ProxyServerClientProtocol instance updates the specified client's stamp if the given client timestamp is greater than the stored client timestamp. If a client's stamp was updated, it then propagates data to its server neighbors by creating a connection based on the ProxyClientProtocol, described below. Finally, the protocol instance responds to the original client with its AT stamp.

On a valid WHATSAT request, the protocol instance builds a raw HTTP request on top of TCP and creates a new connection based on the PlacesHTTPClientProtocol, described below. In this case, the ProxyServerClientProtocol instance itself does not respond to the original client; instead, the transport connecting to the client is passed on to the PlacesHTTPClientProtocol so that it can send the original client the JSON data when it arrives.

On a valid AT request, the protocol instance updates the given client's AT stamp if the timestamp is valid. It then proceeds to propagate the stamp to its server neighbors through the ProxyClientProtocol. To avoid infinite loops on propagation, the instance's server name is appended onto the propagated AT message, so that neighboring servers know which servers not to propagate to.

ProxyClientProtocol is a very simple network protocol that propagates a message to another server. It does not need to receive or parse any incoming data, so it's implementation is very minimal.

PlacesHTTPClientProtocol is a protocol that connects to the Google Places API and sends place information back to the original client. This protocol is unique because it uses different two transports to write data. It uses the transport obtained from connecting to Google's API to make the HTTP request on top of TCP. But it uses another transport, which is passed in to the protocol's constructor, to write Google's response back to ProxyServerClientProtocol's original client. In this

way, it serves as a type of callback from the connection to Google's API to write information to the right client.

Below are logs that show Alford receiving an IAMAT request (some details are omitted for brevity) and propagating data to Hamilton:

```
New connection from (...)
Received input data: 'IAMAT ...'
Updated location data for kiwi.cs.ucla
Sent output response: 'AT Alford
+1512367138.545446396 kiwi.cs.ucla
+34.068930-118.445127 13444.412014450'
Dropped connection from (...)
Connected to server Hamilton
Propagated location data to server
Hamilton
Connected to server Welsh
Propagated location data to server
Welsh
Dropped connection to server Hamilton
Dropped connection to server Welsh
```

3.2. Implementation problems

There were some obstacles that I encountered while implementing this application server herd, but most of them arose from my relative inexperience with using asyncio, and not from any actual problems with the asyncio library.

One of the problems I had involved propagating location data to neighboring servers. The implementation itself was not hard after going through the documentation of asyncio, but at first, propagating location data to servers made subsequent server-client connections hang for no apparent reason. After much deliberation, I realized that I was closing the transport in the ProxyClientProtocol too early, thus shutting off subsequent connections. Problems like these, which stemmed from my naïve understanding of transports and protocols, forced me to understand the workings of asyncio and its components at a deeper level.

Making the HTTP request to Google's Places API on top of TCP was by far my biggest obstacle. Formatting the raw HTTP request was easy enough given many helpful online resources, but creating the connection proved a much larger obstacle. My request seemed to be working using 'openssl' on the shell, but the same request hung during actual server execution. I later realized that this problem rooted from the fact that I was attempting to receive data from an HTTPS connection without the proper SSL credentials. Using Python's ssl library, I created an appropriate SSL context and passed it into asyncio's 'create_connection' function to get requests to work properly.

Again, the above problems stem from my relative inexperience with requests, asyncio, transports, and protocols. Asyncio is a more than suitable framework for creating an application server herd, which leads to the next section.

4. Suitability for application server herd

All of the above research and prototyping leads me to the following conclusion: Python's asyncio is a very suitable framework for implementing an application server herd. This section elaborates on why this is the case.

4.1. Asynchronous scheduling

Section 2.2 and 2.3 described an asyncio event loop and its asynchronous nature. After implementing my own prototype application server herd, it's clear that asyncio's event loop is a perfect fit for the type of architecture that an application server herd requires.

An application server herd requires a server that is constantly collecting updates from clients and other servers. In the case of the five-server prototype, servers propagate client location data to neighboring servers to ensure that all servers have updated values. Asyncio's event loop suits this type of server application perfectly because (1) it is constantly waiting for connections from clients and servers, and (2) connections are asynchronous, so one connection does not impede a server from receiving other connections.

If this architecture had been synchronous rather than asynchronous, a server would need to wait for a connection to return before evaluating another connection, which is unacceptable.

A case in which this asynchronous nature is necessary for the prototype is when the server receives a 'WHATSAT' request and has to make an HTTP request to Google's Places API. Because the event loop is asynchronous, the server which receives the 'WHATSAT' request does not have to wait on the JSON response of Google; it can continue accepting requests from other clients and servers. Once Google does respond, the PlacesHTTPClientProtocol will make sure the client that sent the original 'WHATSAT' request gets the JSON data. This asynchronous nature is also necessary when servers propagate information to each other; just because a server is receiving updated information does not mean it should be unable to process any current requests. This is one of the reasons asyncio is a suitable framework for an application server herd.

4.2. Data abstractions

Writing a coroutine server connection from scratch that performs just as a coroutine should while still handling all the data that passes through a connection would be

difficult and error-prone. Fortunately, asyncio's library abstracts a lot of the underlying operations so that writing an application server herd becomes much easier.

Transports and protocols, for example, provide abstractions for network connections and network protocols, respectively. It was very easy to represent the various types of connections my prototype server would make by simply defining different protocols. The ProxyServerClientProtocol, ProxyClientProtocol, and PlacesHTTPClientProtocol classes all read and write data differently. ProxyClientProtocol, for example, does not receive data at all, while PlacesHTTPClientProtocol receives data and writes it to a different place from where it was originally conceived. Asyncio makes it easy to define these protocols for different kinds of connections. Transports are also very useful abstractions of actual connections; asyncio allows them to be passed around, read from, written to, and closed at will.

Connections and servers themselves are also abstracted away through the 'create_connection' and 'create_server' functions. With only a protocol, host, and port, asyncio will create a connection at the given host and port, using the protocol as the base. Even SSL connections are abstracted away into a simple 'ssl=True' flag (though in the case of my prototype, I had to create my own SSL context and pass this instead).

Together, these abstractions make the implementation of something as complex as an application server herd easy to conceive.

4.3. Python language choice

The fact that the prototype was written in Python actually made it relatively easy to implement, when compared to a similar implementation in another language like Java or C++. This is due to the flexibility that Python is known for.

This flexibility is especially useful in parts of the implementation that parse large amounts of incoming data. Python strings and lists, for example, have many functions like 'join' and 'split' that make the conversion process extremely easy. This translates into very intuitive parsing for large amounts of JSON data, parsing that would be more difficult to implement in languages like Java and C++.

Anonymous functions in Python also prove extremely useful for networking applications. In the case of this prototype, they are used to pass around generic Protocol classes that can be instantiated at a later point. These lambdas allow intuitive abstractions of concepts like Protocols that would otherwise be harder to represent.

Python also has an immense collection of libraries and a very involved community to go along with it. Of course, Java and C++ are equally as impressive in these regards, but the combination of Python's libraries with its flexibility makes it an ideal choice for easily implementing an application server herd.

5. Possible scaling concerns

There are certain areas of interest that require a deeper dive when evaluating the suitability of asyncio in the context of larger applications; namely Python's type-checking, memory management, and multithreading. This section addresses these possible concerns, comparing asyncio's Python implementation to a hypothetical Java implementation.

5.1. Type-checking

Python uses dynamic type checking (duck typing), which means types are only checked at runtime. While dynamic type checking may result in runtime errors that are not possible in a statically typed language, it does not necessarily make code less reliable, especially when it comes to something like an application server herd. A good example is the popular backend web framework Ruby on Rails. Ruby is, in many ways, even more flexible of a language than Python, yet Ruby on Rails remains one of the most popular and reliable frameworks that companies use to this day.

In my opinion, the advantages of duck typing that reveal themselves during the development process overshadow any minor reliability concerns. Development in any dynamically typed language is faster because developers don't have to worry about dealing with different types of objects; they can just write code the way that they want. No time is being wasted ensuring that types match, so applications can be built out extremely quickly. The fact that applications can be built quickly means that they can be tested quickly and more often, which means that dynamically typed backend code can often be equally, if not more, reliable than statically typed code when put into practice. Writing an application server herd in Java, a statically typed language, would take much longer to develop and test because the compiler must be satisfied that all types match across the entire program.

Duck typing is still a double-edged sword, because with development speed comes a lack of readability. Statically typed languages like Java are very readable because functions (or methods) know their argument and return types exactly. A method can be looked at in Java and its usage can be deduced very easily most of the time. The same cannot be said of Python. Python arguments can be of any type. A person reading Python code must go through the function body to deduce what sort of types the function is looking for and what the function actually returns. As the codebase grows for an application server herd built in Python, it may become

less and less readable. The only way to ensure readability is continual and proficient documentation.

5.2. Memory management

Python's memory management actually makes it more efficient than Java for an application server herd implementation. Both of these languages use the heap for all data allocation, but their garbage collector (GC) algorithms differ. Java's GC destroys objects which are no longer in use at some indeterminate time, while Python's GC uses the reference count method to immediately destroy objects that are no longer in use.

Within the application server herd, this means that all created objects are quickly deleted when they are no longer referenced. For a similar implementation in Java, unused objects would have to wait for the garbage collector in order to be freed. Thus, memory management for an application server herd would actually be more efficient in Python. This memory efficiency would prove extremely useful for larger asyncio application server herds that use more memory.

5.3. Multithreading and performance

Asyncio uses a single threaded, asynchronous event loop. A similar implementation in Java would probably be multithreaded, which means its performance would be system dependent. This means that, depending on how many processors a system has, the performance of a Java implementation may be faster or slower than a Python asyncio implementation.

The Python implementation, on the other hand, will perform the same across all systems, and thus is actually better for horizontal scaling; more and more clients will be able to connect to an asyncio server with only a slight performance impact. The Java implementation would win in terms of horizontal scaling, however, because the use of more powerful server machines means the multithreaded approach will ultimately be more powerful.

6. Node.js comparison

Python's asyncio framework and Node.js are both asynchronous, single-threaded frameworks used for networking applications. For the most part, besides syntactic differences, they operate in similar ways.

Asyncio uses callback-based protocols to provide asynchronous operations. Node.js does something very similar with the use of promises. Promises also have callbacks and operate in much the same way that asyncio callbacks work, other than syntactic differences and the fact that the term 'callback' isn't as widespread in Python as it is in JavaScript.

However, one mechanism Node.js uses that is not found in asyncio is the closure. Closures in JavaScript encapsulate information to a higher degree than what is possible in Python. They allow for a rough implementation of 'public' and 'private' variables and functions, which can be useful in certain contexts.

On the other hand, Python, being an object-oriented language, has always had classes and objects. JavaScript only recently introduced these ideas with ES6, which means it's not as modular as Python.

Both Node.js and asyncio are relatively recent developments, and they have both proven to be extremely popular. The implementation of an application server herd could certainly be done with Node.js, however it's usage and implementation more properly aligns with web based applications. Asyncio, with all the modularity that comes with Python, is probably a better choice for server applications like a server herd.

7. Conclusion

Python's asyncio framework is incredibly powerful. It's data abstractions and the fact that it is written on Python also make it extremely easy to use. As seen by the above research and analysis, asyncio is more than capable as a framework for an application server herd. There are definitely available alternatives that may be worth looking into like Node.js and Twisted, but with all its strengths, I still highly recommend asyncio.

8. References

[1] 18.5. asyncio - Asynchronous I/O, event loop, coroutines and tasks. (n.d.). Retrieved December 04, 2017, from https://docs.python.org/3/library/asyncio.html

[2] Cannon, B. (2016, December 17). How the heck does async/await work in Python 3.5? Retrieved December 04, 2017, from https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/

[3] Diaz, Y. (2016, February 20). AsyncIO for the Working Python Developer – Hacker Noon. Retrieved December 04, 2017, from https://hackernoon.com/asyncio-for-the-working-python-developer-5c468e6e2e8e

[4] Saba, S. (2014, October 10). Understanding Asynchronous IO With Python 3.4's Asyncio And Node.js. Retrieved December 04, 2017, from http://sahandsaba.com/understanding-asyncio-node-js-python-3-4.html