Proxy herd with Python's asyncio

Uday Shankar Alla

Abstract

There are multiple scenario's that can prove to be a bottleneck when it comes to using a classic LAMP architecture. These scenario's include but are not limited to situations when clients move around a lot, when protocols other than HTTP are being used for access and when a large number of updates take place. In these situations, using a server herd might alleviate some of the bottlenecks by enabling servers to transmit data that is constantly changing directly to one another without the need for any intermediary infrastructure such as databases. The goal of this report is to investigate pythons asyncio and subsequently use it to implement an application server herd.

1 Introduction

The goal of the report was to examine asyncio's pros and cons. In order to do this I built a prototype of an application server herd that implements asyncio to communicate client's location data between 5 different servers called Goloman, Welsh, Wilkes, Hands and Holiday.

2 Asyncio Research

In order to implement the application server herd, I first had to do research on Python's asyncio and understand how it work's. The information for this research was primarily obtained from python's documentation on asyncio. (https://docs.python.org/3/library/asyncio.html)

2.1 Coroutines

One of the core aspects of asyncio, which primarily attributes to it's asynchronicity, is the correspondence between coroutines and event loops. Through the use of multiple points of entry, coroutines have the ability to generalize subroutines. What this essentially means is that coroutines can pause/resume execution of I/O operations or any other functions through these entry points. The code snippet, which is an example of a generator based coroutine, illustrates the power of coroutines:

```
def coro():
    hello = yield "Hello"
    yield hello
c = coro()
print(next(c))
print(c.send("World"))
```

The output of this snippet is the following:

```
Hello
World
```

From the code snippet above, the power of the keyword yield can be seen. The generator first pauses at *Hello* and then resumes at another entry point where it subsequently prints out *World*. It is important to note here that the line *c = coro()* stores the coroutine object in the variable *c*. Until c is scheduled/iterated upon, the call does nothing. The newer version of python has shifted the primary syntax from *@asyncio.coroutine and yield from* to *async and await.* They both essentially do the same thing, barring some lower level compilation code. "Yield from expressions and await expressions both accept coroutines while differing on whether they accept plain generators or awaitable objects, respectively."

Many of the functions in asyncio's `base_events.py` are used in the application server herd. `create_connection` for instance is one such function used, which uses async and await, which gives it the abilty to be paused and resumed. As seen from the documentation, this makes perfect sense as `create_connection` returns a coroutine.

2.2 Event Loops

In the code snippet provided above we used the next function to iterate over the generator. Another crucial aspect of coroutines is the ability to schedule them. Specifically, asyncio's API allows for the automatic schedule of coroutines through the use of event loops. Once an event loop is created, it continually waits for events to occur and subsequently dispatches messages and events.

Another way to illustrate the use of event loops is to look at JavaScripts event loop that is contantly being run on the browsers. The event loop is constantly listening for events such as clicks and hovers to occurr, upon which they receive these events and deal with it by invoking some appropriate callback functions. It can be thought of as an infinite while loop(While true). In order to make an event loop in python, we do the following:

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

Then, to schedule events we do the following:

```
loop.create_task(coroutine)
```

To start and stop loops:

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

This research on event loops served as a base for creating my server prototypes.

2.3 Asynchronocity

The challenging part of event loops is that the order of scheduled coroutines cannot be pedicted due to asynchronicity. Essentially these coroutines can be interlieved. Determining which one's will finish first is the hard part.

There are many advantages to such types of asynchronicity. For instance, in out proxy server herd application, when one of the servers makes an HTTP Get request to google it doesn't have to be blocked/wait to accept other connections. Essentially the server can continue execution after creating tasks.

3 Prototype

In this part of the report, I will describe briefly my prototype design of the application server herd and illustrate how I applied asyncio research to my application.
Essentially the entire application is implemented in *server.py* while some of the constants like port numbers and Google API key/end points are hardcoded in the file *config.py*.
The command to start a server:

```
python3.6 server.py <server_name>
```

3.1 Protocols & Transports

To implement the server herd, I used async's callback API, transports and protocols. Protocols in this case essentially represent the network protocols while the transports represent the communication channele of the various servers. Transports are used by protocol classes to write/read.

To represent the various types of network channels, I used 3 protocol subclasses:

1. `ProtocolServerToServer`
2. `ProtocolServerToClient`
3. `ProtocolHTTP`

The subclass *ProtocolServerToClient* is the main protocol associated with the server. The coroutine for the server is created by the `create_server` function in asyncio's AbstractEventLoop. The loop runs until a KeyboardInterrupt is identified. Therefore until there is a keyboard interrupt the server keeps listening to its assigned port.

Since *ProtocolServerToClient* is the main protocol for the server it contains the functions to handle IAMAT, WHATSAT and AT requests. It also contains three dictionaries that store each clients last time, location and AT stamp. When a server recieves an IAMAT request with valid parameters, *ProtocolServerToClient* first checks to see if the timestamp of the request is greater than the last seen timestamp of the same client and if it is, it updates the last seen time and propogates the data to the neighbouring servers through *ProtocolServerToServer* protocol. The *ProtocolServerToServer* protocol is minimal in the sense that it only propogates the data and doen't have to parse it or do any subsequent action.

When a server recieves a WHATSAT request with valid parameters, *ProtocolServerToClient* protocol makes a subsequent request to google. To do this, *ProtocolServerToClient* protocol first makes a HTTP raw request via TCP and builds the connection using the *ProtocolHTTP* protocol. The *ProtocolHTTP* protocol builds a connection to the google places API. This protocol differs from the two previously mentioned protocols as it utilizes two different transports to write. One transport to make the Google Places API request and another transport to write Google Places API's response back to the *ProtocolServerToClient*'s original client.

On an AT request, *ProtocolServerToClient* protocol checks to see of the timestamp of the request is greater than the last seen timestamp, and if it is greater updates it and propogates the data to neighbouring servers.

3.2 Hurdles with implementing prototype

Some of the hurdles I faced with implementing the application server herds were:

1. Implementing the flooding algorithm took some time to figure out. My flooding algorithm essentially keeps appending to the AT request with the names of servers that have already recieved the data. And so, when a server is propogating data to neighbouring servers it simply checks the end of the AT request and sends it to the neighbouring servers not present in the request.

2. Another hurdle was with building the raw HTTP GET request and parsing the json object recieved from the google places api.

4 Asyncio's Suitability for Server Herd

After doing a great deal of research on asyncio and subsequently incorporating it in my application server herd, it can be seen that asyncio works perfect for my prototype.

4.1 Asynchronicity in scheduling

An essential requirement to building the proxy server herd requires running a server that is continually looking for updates from other clients. In my 5 server prototype, each server on recieveing data/requests from clients needs to relay this information to the neighbouring servers so that every one of them have the latest information. For this type of application, asyncio's event loops would be perfect to use as (1) the connections are asynchronous which means that the servers are never blocked and (2) servers are constantly listening for updates and new connections.

If the application was synchronous then each server would technically be blocked while waiting for a connection to return. In our case let's say a server recieves a WHATSAT request, because of it's asynchronicity it can send out a HTTP request and while waiting for a response can still accept new requests from different clients.

4.2 Abstraction of Data

A lot of the underlying aspects and operations of the proxy server herd are abstracted away by asyncio's library which makes it a suitable option for us to use in the prototype. For instance, transports provide abstractions for network connection while protocols provide an abstraction for network protocols. Asyncio therefore makes it very easy to define our protocols.

4.3 Python as the language choice

Due to the flexibility of python as a language, it was easier to implement our prototype in python rather than languages like C++ or Java.

Pythons lists and string manipulations, such as split, strip, join etc. allowed for easy parsing of input and output. Therefore intuitively parsing JSON data is much simpler with using python as the main language.

In our networking application, python's anonymous functions prove to be extremely useful. If we look at our application server herd, we use anonymous function to pass generic protocol classes which are then instantiated at some other time. Therefore python's Lambda function serve to be very useful in such a networking application.

These are the two main reasons why we picked python over other languages.

5 Scaling issues

Multithreading, type-checking and memory management are possible areas that require a greater depth of understanding when examining pythons asyncio's suitability for our application server herd. In order to do this we compare our implementation of the server herd with a Java implementation of the same.

5.1 Multithreading

Python's asyncio uses event loops which run on a single thread. A Java implementation of the same would however more likely be multithreaded. This means that the performance of a Java based application will depend on the system on which it is being run; Based on the system's processors a Java implementation can either be slower or faster when comparing it to Pythons asyncio's performance.

Since Python's asyncio implementation performs the same on all machines it is better for "horizontal scaling." This just means that when more clients connect to server, there is only a slight impact on performace.However Java would win if the systems processors are powerful.

5.2 Memory Management

For an application server herd such as ours, pythons memory management is more efficient than Javas. For data allocation, both languages use the heap. What makes pythons memory management better than Java in this case is its Garbage collector algorithm. Python destroys objects that are no longer in use by reference count whereas Java destroys objects at some indeterminate time.

For our application, every time an object is no longer in use it gets deleted as opposed to Javas implementation which would have to wait for the Garbage collector in order to free memory. Thus pythons implementation is more memory efficient than a Java implementation of the proxy server herd.

5.3 Type-Checking

Python uses duck-typing - dynamic type checking. This means that only at run time are the types checked. Therefore this can give rise to run-time errors as opposed to a static type checking language. While duck typing makes code easy to write and run, what's hard is code readability. To deduce the type of arguments passed to a function, one must look at the body of the function in python as opposed to Java where the function definition contains all of the information.

6 Comparison with Node.js

Node.js and Pythons asyncio's framework are both single threaded and asynchronous used predominantly for networking applications. Node.js uses Promises for callbacks and barring some syntactic differences, they work just like pythons asyncio callbacks. However, Node.js use closure to encapsulate data to higher degree than what is seen in python. Both Node.js and Python are good choices when it comes to building networking application. Node.js might be better for web based networking application and pythons asyncio framework might be better for server based application like our server herd.

7 Conclusion

To conclude, python's asyncio framework is most suitable for our proxy server herd application. The fact that it is written on python and the level of abstraction it provides make it easy and suitable to implement in our application. Twisted and Node.js may be good alternatives, but my preference still lies with pythons asyncio.

8 References

1. https://docs.python.org/3/library/asyncio.html

2.https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/