# Server Proxy Herd using Python asyncio

Millie Savalia, UCLA COM SCI 131 Winter 2018

## Abstract

A classic LAMP architecture might become limiting if updates happen fairly often, accesses happen on protocols other than HTTP, and clients are more mobile. A potential solution to this inefficiency is an application server herd which would allow servers to communicate rapidly-evolving data directly to each other, acting as intermediary proxies, without having to speak to a database every time. This paper provides an example of an implementation of a proxy herd using Python's asyncio library.

## 1. Introduction

While at first the task of designing an application server herd seemed daunting, the strengths and weaknesses of using asyncio within Python became more clear as I got further into prototyping the model. The model propagates client location data among five servers named Goloman, Hands, Holiday, Welsh, and Wilkes. This implementation contrasts with the current Wikimedia architecture which is based on LAMP and optimized for a significantly higher frequency of reads and writes, making it inefficient for the proposed application. The server herd model will help to evaluate how well the asyncio framework performs in such an application.

## 2. asyncio

Before I began on the design of the server herd, I first researched how to use the asyncio framework and how it was implemented. I examined the asyncio source code and documentation and also obtained information from CS 131 lectures.

### 2.1 Coroutines

A feature of asyncio that makes it asynchronous is the relationship between the coroutines and the event loop. Coroutines utilize multiple entry points to generalize subroutines. They then use these entry points to pause and resume execution of some computation or IO operation. Simply calling a coroutines does not execute its code. A call does not execute the function until it is iterated over or scheduled.

In the source code for asyncio, 'base_events.py' contains many of the functions used in the server herd prototype. One such example is the function 'create_connection', whose implementation contains 'yield' and 'yield from' implying these functions are generators that can be paused and resumed. This is in agreement with the documentation which specifies the function returns a coroutine.

### 2.2 Event loops

Coroutines can also be scheduled as well as iterated over in a generator. This is where asyncio's API comes in, allowing for the creation of an event loop that automatically schedules coroutines. An event loop will wait for and dispatch events and messages in a program once started.

An easy to understand example is the JavaScript event loop running in every browser. Whenever a click or hover happens, it is given to the event loop. The event loop handles the event by calling some callback function designated to handle the event properly, then continuing in the loop to deal with future events or messages.

In my implementation of the server herd prototype, I created event loops, and then used functions such as 'create_task', 'ensure_future', 'run_forever', and 'close'.

### 2.3 Asynchronous

One of the aspects I found to be confusing about event loops is that they are asynchronous, meaning the execution order of any scheduled coroutines is not known ahead of time. This does not mean the event loop is multithreaded. Coroutines are interleaved, therefore making it impossible to know which coroutine will finish first. The advantage of an asynchronous event loop then comes with external calls to libraries and servers.

If one of the five servers in the prototype makes an HTTP request to the Google Places API, that server doesn't have to wait for an HTTP response before receiving other connections. A server can therefore create a task and continue on with its normal execution without waiting for that task to complete. Callbacks ensure that the responses are received when they do happen.

# 3. Prototype design

The majority of my implementation for the server herd lies in server.py. The source file config.py contains some of the constants I used such as port numbers, API endpoints, and neighboring servers.

### 3.1 Transports and protocols

I utilized 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 and use transports to read and write data. I created three main protocol subclasses that represent different types of network connections which consisted of ProxyServerClientProtocol, ProxyClientProtocol, and PlacesHTTPClientProtocol.

ProxyServerClientProtocol is the primary protocol used to represent the server itself. The server coroutine uses asyncio's AbstractEventLoop.create_server function. The loop is set to run forever until a KeyboardInterrupt is detected, which means the server continuously listens to whichever port is has been assigned. Since this is the primary server entity protocol, it contains functions that receive messages, deal with them, and send responses. These messages include the IAMAT, WHATSAT, and AT commands. The class also contains a static variable that maps client ID's to their locations AT stamps.

On a valid IAMAT request, the ProxyServerClientPro- tocol instance updates the specified client's stamp if the given client timestamp is greater than the stored client timestamp. 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 PlacesHTTPClientProto- col, 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.

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 im- plementation 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. It also uses another transport, which is passed in to the protocol's constructor, to write Google's response back to ProxyServerClientProtocol' s original client.

### 3.2 Implementation issues

Most of the obstacles I encountered while implementing the server herd were due to my inexperience and unfamiliarity with asyncio, not problems with the library itself.

One of the issues I had involved propagating location data to neighboring servers. The implementation itself was not too difficult after going through the documentation of asyncio. At first, propagating location data to servers was causing subsequent server-client connections hang for no clear reason. After thinking through it, I noticed I was closing the transport in the ProxyClientProtocol too early and shutting off subsequent connections. Such problems which came from my lack of experience with transports and protocols, pushed me to understand the workings of asyncio and its components better.

# 4. Suitability for server herd

All of my research and prototyping leads me to the conclusion that Python's asyncio is a very suitable framework for implementing an application server herd.

### 4.1 Asynchronous scheduling

After implementing my own prototype application server herd, it's clear that asyncio's event loop is a great fit for the architecture that an application server herd requires.
A 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 greatly suits this type of server application because it constantly waits for connections from clients and servers, and the connections are asynchronous, so one connection does not impede a server from receiving other connections.
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. This asynchronous property is also necessary when servers propagate information to each other. When a server is receiving updated information, it should also be able to process any current requests.

**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 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 because asyncio allows them to be passed around, read from, written to, and closed at will.

# 5. Possible 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.

**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 re- mains one of the most popular and reliable frameworks that companies use to this day.
the advantages of duck typing that re- veal themselves during the development process over-shadow 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.

**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. 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 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

Python's asyncio framework and Node.js are both asynchronous, single-threaded frameworks used for networking applications. Aside from syntactic differences, they operate in similar ways.
Asyncio uses callback-based protocols to provide asynchronous operations. Node.js works very similarly 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.
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, allowing for a rough implementation of 'public' and 'private' variables and functions, which can be useful in certain contexts.
Python, an object-oriented language, has always had classes and objects. JavaScript only recently introduced these ideas with ES6, meaning it's not as modular as Python.
Both Node.js and asyncio are relatively recent developments and both have proven to be extremely popular. The implementation of an application server herd could certainly be done with Node.js, but 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 such as a server herd.

## 7. Conclusion

Looking overall at my research it is clear Python's asyncio framework is very powerful. It's data abstractions and the fact that it is written on Python also make it easy to use. As seen by my 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, but with all its strengths, I would recommend asyncio.

## 8. References


18.5. asyncio - Asynchronous I/O, event loop, coroutines and tasks. (n.d.). Accesseded March 14, 2017, from https://docs.python.org/3/library/asyncio.html.

Diaz, Y. (2016, February 20). AsyncIO for the Working Python Developer – Hacker Noon. Accessed March 14, 2018, from https://hackernoon.com/asyncio-for-the-working-python-developer-5c468e6e2e8e.

Github. python/cpython. Accessed March 14, 2018, from https://github.com/python/cpython/tree/3.6/Lib/asyncio/.