# Asynchronous Server Herd Implementations

Sriram Balachandran
University of California, Los Angeles

## Abstract

There are various frameworks and architectures that exist for implementing web servers, but this project specifically requests the use of a server herd. This can be implemented in a variety of manners, but this paper deals with the implementation of a server herd utilizing the Python *asyncio* library.

This paper will discuss the usability of the *asyncio* library, the compatibility of the library with this particular task, the potential downfalls of using Python, and a comparison to an alternative option, *Node.js*.

## 1 Introduction

The task at hand is a Wikimedia-style service that will need to handle frequent article updates with mobile clients. As such, to deal with the mobility of clients, the specification has suggested the use of the Server Herd Architecture.

The Server Herd Architecture can be represented by a connected undirected graph, where each node represents a server. This allows one to concurrently handle requests from a variety of clients (both in number and geographic location), and then propagate this information to other servers per links defined in the adjacency list for each server.

The *asyncio* module in Python provides a means of single-threaded concurrent execution and access to sockets, along with Server and Client models. Since it can use non-HTTP Protocols and concurrently process updates, as well as provide convenient ready-built Client and Server Models, *asyncio* seems to be an ideal solution to our problem. The primary advantage of *asyncio* is that it is single-threaded and asynchronous, allowing it to handle multiple requests concurrently without complex multithreading operations. This will allow it to outperform a more traditional synchronous runtime environment for the given task.

In order to evaluate the suitability of *asyncio* for this particular Wikimedia application, this paper will consider 3 factors primarily: code readability, documentation/community support, and most importantly, performance. To do this, a prototype application server herd was built with 5 nodes that are capable of communicating with one another and external clients through a TCP connection.

## 2 Server Herd Prototype

### 2.1 High Level Overview

A total of 5 servers was set up in order to simulate the server herd (Hill, Jaquez, Smith, Campbell, and Singleton). In order to allow communication between servers, two pieces of hardcoded information were needed – the port numbers of each of the named servers, as well as the adjacency lists of each of the named servers.

```
PORT_NUMBERS = {
    "Hill": 11860,
    "Jaquez": 11861,
    "Smith": 11862,
    "Campbell": 11863,
    "Singleton": 11864,
}
```

[1] Python dictionary mapping each server name to its corresponding port.

```
LINKS = {
    "Hill": ["Jaquez", "Smith"],
    "Jaquez": ["Hill", "Singleton"],
    "Smith": ["Hill", "Singleton",
"Campbell"],
    "Campbell": ["Singleton", "Smith"],
    "Singleton": ["Jaquez", "Smith",
"Campbell"],
}
```

[2] Python dictionary mapping each server name to the named servers it communicates with.

By mapping each of the servers to a list of server names as opposed to a list of port numbers, we allow for more flexibility should the port numbers of the named servers need to be changed. Each server interfaces directly with other servers as well as clients. Clients will update the server with location info and request information that the server resolves using the Google Places API. Servers talk to each

other using a similar messaging protocol. When the server receives a message, the command type is extracted and then the message is passed along to the appropriate command handler function. The commands are as follows:

CLIENT
*IAMAT [client name] [+/-lat+/-long] [posix time]*

*WHATSAT [client name] [radius(km)] [results limit]*

SERVER
*UPDATE [data]*

### 2.2 Data Storage

Each server also has to hold relevant information for each client ID. This information is stored in a Python Dictionary keyed by client id. The information is stored in the following schema:

```
{
 "server": Data origin server,
 "client": Client ID,
 "coordinates": Location Coordinate,
 "timestamp": Client Sent Timestamp,
 "delta": (Server Receipt Timestamp -
Client Sent Timestamp),
 "message": AT Message for this data
entry,
}
```

[3] JSON object schema describing how the server stores data for each client.

### 2.3 Flooding

In order to propagate messages in between servers and their adjacent nodes, a flooding function was built that accepts a message to propagate and an origin server. We then pull the list of adjacent servers, the temporarily remove the origin server from the copy of the adjacency list. We then loop through this temporary list of servers. For each server in the list, a client connection is opened, the message is sent, and then the client connection is closed. In the case of connection failing, it is caught in a try-except block

### 2.4 IAMAT Command

When the server receives an *IAMAT* command from the client, the parameters are passed into the handler built for the *IAMAT* command. This handler will first update the server's dictionary entry for the received client id. The data origin server for this entry is specified as the name of the server processing the *IAMAT* command. The message property for this entry is built using the information received from the client. After this, this JSON string encoding of this dictionary entry is propagated through an asynchronous flooding algorithm to adjacent servers, with the receiver of the *IAMAT* command being specified as the origin server. Then the appropriate *AT* message is returned to the client.

### 2.5 WHATSAT Command

When the server receives an *WHATSAT* command from the client, the parameters are passed into the handler built for the *WHATSAT* command. The handler will check if there is an entry in the data dictionary for the requested client id. If there isn't a corresponding entry, then we return an error message to the client. If there is an entry, we return the *message* property under the entry.

### 2.6 UPDATE Command

In order to accommodate for server-to-server updates, support was integrated for an additional command that would only be sent by servers, the *UPDATE* command. When the server receives the *UPDATE* command, it will pass the received *data* parameter to the update command handler. The *data* parameter is a JSON string representation of the client dictionary entry. If there is no entry for that particular client id, then the dictionary entry for that client is populated with the received data. Otherwise, if the *timestamp* property of the received entry is greater than the *timestamp* entry for the existing entry for that client id, we then replace the entry (since it is "newer" data). The *server* property in the received entry represents the server that the *UPDATE* command was sent from. This server property is passed into the flood function to ensure that the *UPDATE* is not forwarded to the server that sent us the *UPDATE*.

### 2.7 Invalid Commands

If a command is received that is not supported by any of the handles, a basic error message is returned to the client.

## 3 Asyncio Analysis

### 3.1 Asyncio Benefits

Using *Asyncio* allows one to create single-threaded processes that can handle multiple requests in parallel asynchronously. In this architecture, where a server not only needs to receive requests from clients but also propagate received entries to other servers, this

easily usable parallelism was an excellent fit for this use case.

*Asyncio* affords task concurrency without having the programmer consider thread safety and race conditions. When a task is scheduled by the *Asyncio.run()* function, it operates similar to a yield-based operating system scheduler. Coroutines run through the *run* function will not allow another coroutine to use the main thread until it intentionally yields control.

Upon further inspection of the *Asycnio* implementation, we see that there is not only support for the tradition asynchronous programming paradigm, *await*, but there is also the option for *lock*. These two mechanisms achieve the same effect but in different ways. In the case of *await*, it forces the task to wait for the *Future* (or *Promise* as it is more commonly known) to be resolved or rejected. *Lock* places a lock on the value of the asynchronous expression until the *Future* is resolved. This way, even if the task's instruction pointer continues, it will not be able to execute any instructions using the locked value until the *Future* is resolved, thereby releasing the lock on the value. This offers the programmer further versatility and flexibility.

Another great advantage of the *asyncio* module is that it comes with built in TCP and SSL support, along with basic server and client models. This eliminates the need to interact with lower-level socket logic.

### 3.2 Drawback of Asyncio

*Asyncio* also comes with its fair share of difficulties. For programmers not familiar with asynchronous programming, it can be difficult and unintuitive to figure out the paradigm. *Future* logic, as well as handling *Future* resolution and rejection can be tricky to understand and handle for beginners.

In addition to difficulty getting accustomed with asynchronous programming, there is also the non-deterministic nature of the order of execution introduced by the simulated concurrency afforded by yielding coroutines.

If a programmer wishes to make a small part of their code asynchronous, this will unfortunately result in the introduction of a large number of *await*s in the rest of the synchronous code, which then requires that code to be marked by the *async* keyword somewhat unnecessarily, since any function using the *await* keyword must be marked as *async*.

### 3.3 Asyncio Summary

Overall, *asyncio* provides to us a relatively intuitive interface (for those familiar with asynchronous programming) that can be integrated into server logic to quickly and effectively improve performance. The performance boost afforded by the simulated concurrency is great for the particular Wikimedia-style application that expects frequent updates in high volume, as high network latency tasks won't affect the handling of other requests.

Unfortunately, since Python is rapidly iterative language, there are also a number of major versions of *asyncio* in use professionally that can make it difficult to switch if one is not aware of the API changes. These changes made to the API do afford much better code readability and compactness, but can be detrimental for programmers experienced with a previous version of the *asyncio* module.

## 4 Comparison of Asyncio with Node.js

The implementations of *asyncio* and Node.js rely on an Event Loop. An event loop will queue operations and execute them sequentially. Operations that take a disproportionate amount of time, like HTTP Requests, I/O, and database transactions, will be dispatched so that they don't block the execution of the remaining program. After the dispatched operation is resolved, subsequent code (known as callback functions in Node.js) will be placed in the Event Loop again and will be executed.

A key difference between the two is that *asyncio* is a module that needs to be installed and imported to make Python capable of handling asynchronous programming. Node.js, built upon Chrome's V8 engine, is inherently asynchronous, and will execute asynchronously unless specified otherwise. *Asyncio* forces the programmer to explicitly define the event loop and the asynchronous coroutine to be considered in the event loop.

There is great online support for both of the options, and package installation with either development platform is a simple one-line command with the respective package managers (pip for Python, npm for Node.js).

In terms of functionality, one can say that *asyncio* and Node.js bring the same options to the table. In terms of performance, it is difficult to say which would perform better without perhaps

reimplementing the prototyped server herd in Node.js as well. However, in terms of working with asynchronous calls, since Node.js is inherently asynchronous, I believe there is more versatility in Node.js's asynchronous programming. A popular Node.js module known as *Bluebird* affords the programmer a variety of *Promise* (or *Future* in Python) manipulating functions, that can greatly reduce the amount of code that needs to be written. For example, using the *Bluebird* library, it is possible to rewrite the flood function written for the *asyncio* prototype implementation in 2 lines.

## 5 Conclusion

The architecture suggested to be implemented in this Wikimedia application should most definitely be implemented in an asynchronous manner. Event-based concurrency is ideal for large volumes of web requests, and asynchronous frameworks give programmers access to this powerful feature with great usability and readability. *Asyncio*, with its built in server and client models, along with the fact that it's a Python module, a language already considered to be highly readable, makes a great choice for this particular task. However, it cannot conclusively be determined if *asyncio* is the best framework for this task compared to alternatives like Node.js without further testing.