# CS 131 Project Report


## Abstract

In this report, I will be taking a look at using Python's asyncio library to implement an application server herd and compare its performance with that of a LAMP architecture. The LAMP architecture suffers from bottlenecks resulting from mobile clients, frequent updates, and accesses from non-HTTP protocols. This may be fixed by using an application server herd to propagate updates to data directly between servers rather than having to communicate with a database, and so this paper examines the effectiveness of using the asyncio library in implementing this idea.


## 1. Introduction

To examine the effectiveness of Python's asyncio library in implementing an application server herd, I built a prototype consisting of five servers which communicated bidirectionally to propagate client location data. I will be describing how I designed this prototype and evaluate its performance in this report.

## 2. Analysis of asyncio

In order to evaluate the effectiveness of asyncio, I needed to do some research on how to use the library and especially on how it was implemented. I did this by taking a look at the source code and documentation.

### 2.1. Coroutines

Coroutines are an integral part of asyncio as they allow for its asynchronous nature by providing entry points to pause and resume execution of some task. Examining the source code for asyncio, we can see that coroutines are implemented using Python's *yield* and *yield from* keywords to create generators. This makes sense, as the main use of generators in Python is to be able to pause execution and maintain its state so that it can resume later without losing track of its work.

### 2.2. Event Loops

The main way asyncio works is by creating an event loop that looks for and responds to events that occur throughout the lifetime of the program. It is similar to how Javascript works with the use of event listeners and callbacks to run upon detection of a particular event.

## 3. Prototype Implementation and Design

In this section, I will be describing the implementation and design of my application server herd prototype. I used the performance of the prototype to evaluate whether or not asyncio is a suitable library to implement a server herd, and if it is a good alternative to using a LAMP architecture.

The bulk of my prototype code is in the *server.py* file, and most of the constants required are in and imported from the *config.py* file.

To start any specific server, you use the following command:

*python3 server.py Goloman*

### 3.1. Transports and Protocols

My implementation of the application server herd is built upon asyncio's support for transports and protocols which utilizes a callback-based API rather than a stream-based API. I chose to use transports and protocols because they offered less abstraction and more control over the behavior of the server through the asyncio.Protocol class, and this made sense since the protocol should have control over the handling of messages based on predefined formats.

The high-level design of my prototype features the following classes:

1. ServerClient
2. ServerClientProtocol
3. ClientServerProtocol

The ServerClient class was used to handle the overall state for any particular server, and it contained the ServerClientProtocol class, which inherited from the asyncio.Protocol class. I chose this design since it made sense to me to have a high-level class to handle the overall server state while the protocol class should handle each individual connection and check the message formats.

I also used a ClientServerProtocol class which inherited from the asyncio.Protocol class for use by each server to propagate messages for updated client

locations to neighboring servers, as this was the main purpose of the prototype.

The ServerClient class created the ServerClientProtocol class using asyncio's AbstractEventLoop.create_server() function. The event loop was set to run until Ctrl+C is pressed, and until then it would continue to listen for events and process them accordingly.

Inside the ServerClientProtocol class were three callback functions along with other functions I defined. The three callback functions were the connection_made(), data_received(), and connection_lost() functions. These functions ran when certain events were detected and these events can be inferred from the name of each function.

Once a connection from a client was made, the ServerClientProtocol class obtains an instance of asyncio's Transport class so that it can send messages to the client. When data is received by the ServerClientProtocol class, it is put into a buffer, since the data_received() callback does not guarantee to hold an entire message when it is called. I then called my own handle_lines() function to process the received data, which looks for '\n' characters to denote the end of a particular message.

The handle_lines() function makes sure that the incoming message is valid by checking its fields, and if the message is determined to be invalid, the server responds to the client to indicate that the message it sent was invalid.

If the first part of the message is IAMAT, the ServerClientProtocol class passes the message onto the process_IAMAT_message() function, which takes the message and creates a client stamp based on the arguments of the message. The ServerClient class is then used to keep track of the client using a dictionary mapping the client ID to its given location, and if the IAMAT message contains a later timestamp compared to the stored timestamp, the client's stamp is updated and the server propagates its location to its neighboring servers according to its predetermined flood list. It then responds to the client with its own AT message.

If the first part of the message is WHATSAT, the ServerClientProtocol class retrieves the information for the particular client that the WHATSAT query is interested in and uses its information to build the necessary parameters for an HTTP request to the Google Places API. It then uses a coroutine to send the request asynchronously and adds a callback to the event loop to run when the request is complete and the proper response is constructed.

If the first part of the message is AT, the ServerClientProtocol knows that this message is coming from a neighboring server containing updates to a certain client's location. It uses this message to update its own stamp for the client and propagates the update to the neighboring servers as specified by its own flood list. Since these updates can lead to an infinite loop from each server attempting to send AT messages to each other and then further passing along the AT message, I modified the AT message to include the sending server's name at the end so that each server could keep track of which servers already received the message and refrain from propagating the AT message to them. This approach worked and prevented any infinite loops from occurring while still ensuring that all servers received the updated location.

As stated before, the ServerClient class used the ClientServerProtocol class to handle the propagation of messages, and this was done asynchronously by creating a coroutine to run on its own and create a connection to the specified neighboring server. When a particular server finished propagating the AT message, it would send a response confirming that it had updated the client's location, and the ClientServerProtocol class would receive this response and subsequently close its connection to the server.

The following log messages show how the server handles IAMAT messages and the propagation of the client's location to other servers with details omitted:

*(Welsh) => New incoming connection from (...)*
*(Welsh) => Message received: 'IAMAT ...'*
*(Welsh) => Successfully updated client stamp for ...*
*(Welsh) => Propagating updated client stamp for ...*
*(Welsh) => Received update list: ['Welsh']*
*(Welsh) => Connecting to server Holiday*
*(Holiday) => New incoming connection from (...)*
*(Holiday) => Message received: 'AT ...'*

*(Holiday) => Incoming AT message from (…) is server Welsh propagating updated client location for …*
*(Holiday) => Successfully updated client stamp for …*
*(Holiday) => Propagating updated client stamp for …*
*(Holiday) => Received update list: ['Welsh', 'Holiday']*
*(Holiday) => Connecting to server Goloman*
*(Goloman) => New incoming connection from (…)*
*(Goloman) => Message received: 'AT …'*
*(Goloman) => Incoming AT message from (…) is server Holiday propagating updated client location for …*
*(Goloman) => Successfully updated client stamp for …*
*(Goloman) => Propagating updated client stamp for …*
*(Goloman) => Received update list: ['Welsh', 'Holiday', 'Goloman']*
*(Goloman) => Connecting to server Hands*

and so on…
Eventually the received update list is *['Welsh', 'Holiday', 'Goloman', 'Hands', 'Wilkes']* and then the propagation stops.

### 3.2. Implementation Difficulties

The main obstacle I encountered when implementing my prototype came when I was trying to propagate client info between servers. For some reason, upon receipt of an initial IAMAT message, the algorithm ran smoothly and every server correctly updated their stamps for the client. However, upon receipt of a second IAMAT message, for some reason only one of the servers would correctly receive the AT message, and the other servers would get the connection but received no data.

Upon further digging, I realized that this issue came from closing the transport too soon, and so I changed my implementation to send a confirmation message once the server was done propagating the message, and then the ClientServerProtocol class would receive the confirmation message and close the transport.

Another obstacle I faced was figuring out how to make the HTTP request asynchronously using Python's aiohttp library, and I solved this by looking at the documentation and realizing that I had to use the create_task() and add_done_callback() functions from asyncio. I then created an asynchronous function using the *async* keyword which I passed to the create_task() function, and a function get_response() as the callback to run once the HTTP request returned and the response was constructed.

## 4. Application Server Herd with asyncio

Using the performance of my prototype and the overall experience of implementing it, along with asyncio's support and use of coroutines, callbacks, and event loops, I have concluded that asyncio is a suitable library for implementing an application server herd. Furthermore, it was very simple to read the documentation and figure out how to utilize asyncio's provided classes.

### 4.1. Asynchronous Scheduling of Events

Looking at asyncio's use of coroutines, callbacks, and event loops, it is clear that its asynchronous nature is perfect for implementing an application server herd, as each server should be able to propagate updated client info while still being able to process new incoming client messages.

The use of callbacks in the asyncio.Protocol class allows it to continuously listen for new client connections, which are handled by new instances of the ServerClientProtocol class, allowing old instances to continue processing their client's messages and newer instances to be created.

The support for coroutines is very important in allowing the server to create and send the HTTP request to the Google Places API asynchronously, and still be able to respond to new incoming messages. It also allows the server to propagate updated client info asynchronously, which also frees it up to handle new connections.

### 4.2. Abstraction

Using asyncio's library was extremely helpful in abstracting away all the details necessary to implement a robust client-server architecture, and it allowed me to focus on the details necessary to meet my prototype's specification.

Instead of choosing the higher level of abstraction offered by the stream-based API of asyncio, I chose to use the lower level Transport and Protocol classes, as they provided more robustness and control over the handling of client connections. They still provided enough abstraction, however, as the use of transports made reading and sending data very easy, and the protocol callbacks made it simple to reason about how the server should respond to certain events.

## 5. Drawbacks

Although asyncio is very suitable for implementing an application server herd, there are some concerns regarding Python's type-checking, memory management, and multithreading as compared to Java's.

### 5.1. Type-checking

Python is a dynamically-typed language, which means that types are not specified explicitly and are instead checked at runtime. The problem with this comes with readability, as it may be harder to infer the types of various functions or variables, and to understand Python code requires more effort to go through the code and infer the types of everything. With Java, this is not a concern since it is statically-typed and all types are explicitly coded so that it is very clear what types are required by a method and what types a method returns. To combat this, it is necessary to maintain good documentation of the code to make it extremely clear how to reason about the code and its types, and it becomes increasingly important to update this documentation as the code base grows.

Dynamically-typed languages have their advantages, though, namely in ease and speed of development. By not having to specify the types of everything, developers do not need to constantly worry about dealing with specific types and simply write code according to how they think it should be written. This contrasts with Java, where developers constantly have to worry about matching types and appeasing the compiler, which makes development time longer and adds to frustration when coding in Java.

### 5.2. Memory management

With regards to Python, memory management is much more efficient compared to Java, since Python's garbage collector maintains a reference count to immediately destroy objects when they are no longer being used. Java's garbage collector only destroys objects after some time that they are not being used, and so this leads to a less efficient implementation of memory management.

Relating this to the application server herd, it is clear that this feature is important since it is necessary to be able to create and destroy connections quickly when they are made and lost, and so this feature of Python makes it desirable when implementing an application server herd.

### 5.3. Multithreading

The asyncio library uses a single thread to run its event loop, whereas a Java implementation would definitely utilize multiple threads. This means that with Java, performance would depend heavily on the specs of the system the application is running on, whereas with Python, performance would be similar across different systems.

This has important implications for horizontal scaling, since multiple clients are able to connect to an asyncio application server herd without much performance impact. However, in the long run, Java will perform better, since you can simply add more powerful systems to run the servers and thus achieve better performance through the use of multithreading.

## 6. Comparison of asyncio with Node.js

Looking at asyncio and Node.js, it is clear that both utilize a single-threaded, asynchronous event loop approach to create network applications. The callbacks and coroutines of asyncio are similar to the promises of Node.js, and they operate in much the same way.

Since Python is an object-oriented language, its implementation of classes and objects is much more robust and natural. Javascript only introduced these ideas in ES6, and from my experience coding in Javascript, the use of classes is awkward and unnatural.

However, since Node.js uses Javascript, the use of closures is available. This allows Node.js to encapsulate information better than Python and can achieve functionality similar to using the *public* and *private* keywords.

Overall, it is definitely possible to implement an application server herd with Node.js, but it is geared more towards creating web applications, and so I think that using asyncio is a better approach to implementing an application server herd due to its abstractions and ease of use in development.

## 7. References

[1] 18.5 asyncio — Asynchronous I/O, event loop, coroutines and tasks. (n.d.). Retrieved June 3, 2018, from https://docs.python.org/3/library/asyncio.html