# CS 131 Project: Proxy Herd with asyncio

Bradley Mont
UID: 804-993-030
*University of California, Los Angeles*

## Abstract

In this project, we implement an application server herd for a Wikimedia-style service with a few extra constraints. We choose Python's `asyncio` library to implement the application server herd due to its event-driven nature. Additionally, we evaluate the fit of our choice to use `asyncio` for this application, and we focus on the tradeoffs between Python and Java for implementing server applications. Finally, we examine the fundamental differences between the approaches taken by `asyncio` and `Node.js`.

## 1. Introduction

Traditionally, the Wikimedia Architecture uses a LAMP (GNU/Linux, Apache, MySQL, and PHP) platform featuring one virtual router in addition to several redundant web servers. For this project, we are being asked to implement an architecture for a site similar to Wikipedia besides the following changes: more frequent updates to articles, access by several other protocols in addition to HTTP, and mobile clients. The Wikimedia Architecture platform would not be ideal for our site because of the bottleneck of always needing to contact the application server, especially with several updates from constantly moving clients.

Therefore, we instead investigate using an application server herd which utilizes multiple servers that can communicate amongst one another and flood messages to all adjacent servers; this way, the servers can quickly all receive updates from each other without the need to contact a central database. In order to determine if an application server herd is an appropriate implementation for our architecture, we use an application server herd to make a parallelizable proxy for the Google Places API. This application features five bidirectionally connected servers that can receive TCP messages from clients, send TCP messages to clients and other servers, and make HTTP requests for location data from the Google Places API.

We choose Python's `asyncio` asynchronous library to implement the application server herd due to its ability to quickly flood updates to all servers in the herd. Focusing on maintainability and reliability, we then conclude whether or not `asyncio` is appropriate to use for this task. Since we are also concerned about scalability and larger applications, we compare how type checking, memory management, and multi-threading are implemented in Python and Java, and we determine which implementation is better-suited for large applications. Lastly, we compare `asyncio` with `Node.js`, a Javascript runtime with functionality comparable to `asyncio`.

## 2. Proxy herd Implementation

Our implementation of an application server herd consists of five servers with the following IDs: Goloman, Hands, Holiday, Welsh, and Wilkes. Each server can bidirectionally communicate with a predetermined subset of the other servers. The servers can also send and receive TCP connections from clients.

Each client represents a mobile device with location coordinates, an ID, and an idea of what time it is. To inform a given server of its location, a client can send an `IAMAT` message to one of the five servers, and that server will then flood the message to all other servers so that all active servers have the client's data. The server will also respond with an `AT` message containing that client's data as well as the server name and approximate response time.

To query for which places are located nearby another client's location, a client can send a `WHATSAT` message to one of the five servers, and the server will respond with an `AT` message containing the results of a Nearby Search request to the Google Places API. The server queries the Google Places API via an HTTP GET request using Python's `aiohttp` library.

Any invalid messages sent to the servers will receive a response of just the original message with a question mark prepended to it. Finally, the servers log all of their input and output to a file to keep track of all connections.

### 2.1. IAMAT Command and AT Response

The `IAMAT` command contains the following fields: the client's ID, the client's latitude and longitude in decimal degrees using ISO 6709 notation, and a POSIX time expression (seconds and nanoseconds since 1970-01-01 00:00:00 UTC) representing when the client believes it sent the message.

In response to receiving an `IAMAT` command, the server responds with an `AT` message containing the following fields: the server's ID that received the message, the difference between when the server believes it received the message and when the client reported it sent the message, and then a copy of the client's fields in its `IAMAT` message.

In my `server.py` implementation, I maintain a global dictionary `clients` that maps a client ID to the following details that represent the most recent time that it updated its location with an `IAMAT` message: coordinates,

time client sent, time server received, and the name of the server that received the message. This way, I can easily access and update a client's coordinates when new IAMAT messages and requests for clients' locations come in.

### 2.2. UPDATE_CLIENT Command and Flooding Updates

Additionally, the client's information must be flooded to all other servers so that a client can query information from any functioning server and receive a consistent response. We accomplish this by having the server send an UPDATE_CLIENT message to all of the servers that it is in direct communication with. The UPDATE_CLIENT message is solely used for inter-server communication, and it contains the following fields: the client's ID, its updated coordinates, the time the client sent its message, the time that the server received the client's message, and the name of the server that received the client's message.

Upon receiving an UPDATE_CLIENT message, a server checks for the following criteria: either the server has no known information of that client in its clients dictionary, or the server simply hasn't yet been informed of the client's new information (accomplished by checking timestamps). If either of these are true, then the server updates its clients dictionary with the information given in the UPDATE_CLIENT message, and then it calls the flood function to update its servers that it is in direct communication with as well.

Specifically, the flood function connects to all of the servers in direct communication with a given server, and it sends UPDATE_CLIENT messages to them as well. This process recursively continues until all servers have been updated with the client's newest information.

### 2.3. WHATSAT Command

Additionally, a client can send a WHATSAT command to a server with the following fields: the ID of another client, a radius (in km) from that client's location, and an upper bound on the amount of nearby places that the client wants to receive back from the server.

In response to receiving a WHATSAT command, the server responds with an AT message (in the same format as before) containing the data about the client referred to in the WHATSAT command. However, this AT message also contains the results in JSON format for a Nearby Search request to the Google Places API with the client's location and the radius as parameters to the request.

In my server.py implementation, I first retrieve all of the necessary information about the requested client by searching that client in my global clients dictionary. This gives all of the information to form the first part of the AT message. Then, using Python's aiohttp library, I send an HTTP GET request to the Google Places API, and then I use Python's json library to return the results as a formatted JSON. The AT message plus the JSON are then returned back to the client.

## 3. The Suitability of asyncio for a Proxy herd

### 3.1. Overview of Asynchronous Programming and asyncio

First off, when running code purely sequentially, that means that a task must wait idly while previous tasks are executing. If there is one large task at the beginning, that could starve all of the other tasks and bottleneck performance. With asynchronous code, the program can allow other tasks to execute while waiting for a time-intensive operation such as I/O to occur, which can greatly increase the efficiency of programs.

For our implementation of a proxy server herd, we used the asyncio library, which is described as "a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queries, etc" [1]. Opposed to preemptive scheduling with a built-in timer to force a task to give up the CPU after a certain time interval, asyncio relies on cooperative multitasking, in which the tasks volunteer to give up the CPU so other tasks may run as well. It is also important to note that asyncio allows for concurrent programming, which could be considered as almost an illusion of multithreading and parallel programming [2].

Two of the most important keywords in the asyncio library are async and await. When we prepend the async keyword to the definition of a function, that function becomes a coroutine. A coroutine can voluntarily halt its execution and allow another coroutine to execute instead, which allows us to use cooperative multitasking with asyncio. This is also accomplished using the await keyword, which stops the current coroutine from executing until the awaited function is finished executing. Additionally, there is an event loop that picks the next task while a task is awaiting [2].

asyncio provides high-level and low-level APIs to implement asynchronous programming. The high-level APIs allow programmers to "run Python coroutines concurrently and have full control over their execution," "perform network IO and IPC," and "control subprocesses" [1]. The low-level API allows programmers to "create and manage event loops, which provide asynchronous APIs for networking, running subprocesses, handling OS signals, etc." [1]. Not to mention, asyncio also allows programs to establish reliable TCP connections between clients and servers.

### 3.2. Advantages of asyncio in Proxy server herd

For a proxy herd, I believe that asynchronous programming is crucial because having all of the servers sequentially process one message at a time would be extremely slow. Due to asyncio's focus on coroutines and asynchronous programming, as well as its thorough documentation on creating asynchronous TCP clients and servers, I found

asyncio to be very suitable to the proxy herd implementation.

Using the low-level API, I use asyncio.get_event_loop() to create an event loop for each server. The event loop chooses asynchronous tasks and then executes them; in our case, the event loop handles new connections to the server and processing messages. Using the high-level API, I use the async/await syntax to define all of my functions as coroutines that can voluntarily stop executing. This combination of an event loop for each server and coroutines for each task allows for much more efficient, asynchronous code.

The asynchronous event-driven nature is so important for this application because servers can work on processing multiple tasks at the same time. Since all tasks are coroutines, the server simply adds each task to the event loop, and then the event loop can asynchronously execute all of the tasks inside the loop. If any coroutine such as a large I/O operation is extremely slow, it will not bottleneck the performance of the rest of the program because we can execute the next coroutine in the event loop while waiting. Since we are expecting frequent updates to clients in our Wikipedia-based service, the ability to take in more messages as we are currently processing messages in the event loop allows for much more efficient, concurrent code.

Additionally, our asyncio implementation of a proxy server herd solves the main bottleneck of Wikimedia Architecture's traditional LAMP architecture: the central application server. If we're receiving a large influx of messages and the servers have to talk to the central application server every time they receive a message, that will severely bottleneck performance. With our implementation, the servers recursively propagate the message to all of the other servers by using the flood coroutine to pass on UPDATE_CLIENT messages to all of the servers. This way, each server will have consistent and updated information about all of the client data, and there is no bottleneck of each server having to sequentially talk to one central database.

Therefore, due to its simple syntax, TCP compatibility, and API for creating an asynchronous event loop of coroutines which eliminates bottlenecks, asyncio is extremely compatible to use for our application server herd.

### 3.3. Disadvantages of asyncio in Proxy server herd

For our specific application server herd, we need to send an HTTP request to the Google Places API (in the event that a client sends a WHATSAT message). However, asyncio only supports TCP and SSL protocols, so we must include an additional library, aiohttp, to send an HTTP GET request to the Google Places API to get location data.

Additionally, it is important to remember the difference between concurrency and parallelism. Parallel code takes advantage of multiple cores or multiple CPUs and actually executes multiple threads or processes at the same time. However, concurrency is simply an illusion of parallelism; multiple things are not actually running at the same time. Due to problems with race conditions when updating reference counts to memory, Python only supports concurrent code, not parallel code [2]. Therefore, asyncio's concurrent implementation is less performant than another implementation that can take advantage of parallelism and multithreading.

Finally, due to asyncio's asynchronous nature and event loop, there is the possibility that coroutines could be completed in non-chronological order. This could lead to problems of servers accessing stale data. For example, say a client sends an IAMAT message updating its location, then another client sends a WHATSAT message querying about that client's location. If those messages are processed out of order, then the WHATSAT message will return stale location data about that client.

### 3.4. Overall Recommendation for asyncio in Proxy herd

After analyzing the pros and cons of the asyncio library for our proxy server herd, it is apparent that the benefits heavily outweigh the drawbacks. To reiterate, asyncio's use of an event loop and coroutines allows for servers to process multiple tasks at a time and prevents one slow task from starving all of the other tasks. Additionally, the ability to asynchronously flood messages to all servers eliminates the bottleneck of a central database.

Even though asyncio does not support HTTP requests, the aiohttp library is extremely compatible with asyncio, so that is an extremely small inconvenience. Plus, asyncio's concurrent nature, although maybe not as fast as a multithreaded implementation, greatly increases the efficiency of the code. Finally, even though our code might be a little less reliable than synchronous code, the performance benefits greatly outweigh the slight decrease in reliability. Overall, asyncio is an extremely suitable option for our proxy server herd.

### 3.5. Problems implementing Proxy herd

My biggest problem during my implementation of the proxy server herd was flooding a message to all servers. It took me awhile to understand how the flooding process works and how it eliminates the need for one central application server. Finally, I came up with the idea of creating a new type of message, UPDATE_CLIENT, that is only used for communication between servers. Even with that, my code kept infinite looping until I remembered to put in constraints for when to continue flooding the message and when to stop.

## 4. Comparison between Python and Java

### 4.1. Type checking

Python and Java handle type checking in extremely different ways. First off, Python is a strongly, but dynamically typed language, meaning that all type consistency checks happen at runtime, not compile time. This allows Python's code to be much simpler because we do not need to specify type annotations when creating variables and functions. Plus, Python code is more flexible as well because the same code could potentially work when ran with different types of data. However, this approach is also less safe because Python won't catch some type errors at compile time that a statically-typed language would normally catch. This also makes Python harder to debug than a statically-typed language since a statically-typed language will show us the error before even running the code. Finally, Python uses what is known as duck typing, where we care about the functionality, not the type of the variable [3]. This loses a bit of reliability but adds simplicity and readability to the code.

On the other hand, Java is a statically typed language, meaning that type checking occurs at compile time, before the program starts running [3]. Additionally, type annotations are required to indicate the type of each variable. This makes code more verbose as we can see exactly what type each variable is, and it also adds reliability to our code since we know what types variables are. Plus, Java is easier to debug since type errors can be caught before having to worry about run-time errors. However, Java code can be considered less simple and less flexible than Python due to the type annotations required.

For our use, Python is optimal because its dynamic type checking makes our code simpler to write, and it doesn't make us assign types ahead of time like Java would. This is a fairly small application, so the greater difficulty in debugging Python should not be much of a factor.

### 4.2. Memory Management

First off, Python implements automatic memory management through reference counting, which means that it keeps track of the amount of times that an object in memory is referenced [2]. This allows for quick and simple garbage collection: when an object's reference count drops to zero, its memory can be immediately claimed by the garbage collector. However, this approach fails when links form cycles because the objects' reference counts will never become zero since they refer to each other. As a result, this would create a memory leak, so this requires Python to periodically run the mark and sweep algorithm as well. Overall, Python's memory management could be considered slow because global reference counts must constantly be updated and checked for garbage collection, but its approach is memory-efficient.

On the other hand, Java uses the mark-sweep algorithm for garbage collection which first marks all objects that are pointed to by something and then reclaims ("sweeps") all unmarked objects. While this approach is faster than Python's because we don't have to constantly access and update reference counts, it is more memory-intensive. It also works for cyclic memory references, unlike Python's reference count approach. Finally, Java periodically garbage collects, while Python constantly checks reference counts.

For the purpose of our project, Python is superior because we do not create any cyclic memory references, and Python's approach will save memory. Plus, since Java's memory management is more sporadic and unpredictable, it could lead to unexpected delays in server execution, which could definitely cause data inconsistency.

### 4.3. Multithreading

Since Python's garbage collection relies on reference counts, Python cannot have any race conditions when updating reference counts. Race conditions could cause inaccurate reference counts, which could either result in memory leaks or memory being freed that is still in use. Therefore, only one thread can execute at a time in Python, so parallelism is not possible [2]. Instead, Python uses a Global Interpreter Lock that only one thread can hold at a time. Even though this does not allow for parallel code or taking advantage of multiple cores, this does have its own advantages: this simple schema allows for extremely fast single-threaded code and there isn't overhead from creating several locks. This is also optimal since Python takes advantages of several C libraries that are not thread safe [2].

Conversely, Java does have support for multithreading and parallel code with its `Thread` class. Plus, the Java Memory Model defines the behavior of multithreaded Java applications to satisfy the "As-If Rule," which states "you can implement a language any way you like, so long as it's done as if the model were executed." For short-term locks, Java has a `synchronized` keyword to put around critical sections and prohibit races inside of them. For longer-term locks, Java has the `wait`, `notify`, and `notifyAll` keywords. Finally, Java contains an extensive high-level synchronization library containing classes such as `Semaphore`, `Exchanger`, `CountDownLatch`, and `CyclicBarrier` [4].

For the needs of our project, Python's form of concurrency is optimal because since our project is relatively small, the cost of possibly creating multiple locks in Java would not be amortized. Python's single-threaded, asynchronous approach would have less memory and performance overhead and still give us a performance boost through running the code asynchronously.

## 5. Comparison between asyncio and Node.js

`Node.js` is described as an "asynchronous event driven Javascript runtime … designed to build scalable network applications" [5]. `Node.js` and `asyncio` are similar in the fact that they are both single threaded, so they both run code concurrently, but not in parallel. Additionally, they are both event-driven and contain an event loop that can asynchronously execute tasks. `asyncio` utilizes coroutines in its event loop that can voluntarily halt their execution and allow other coroutines to execute. `Node.js` uses callbacks in its event loop which serve similar functionality [5].

In terms of performance, `Node.js` takes the edge because it is "based on Chrome's V8 which is a very fast and powerful engine" [6]. Additionally, Javascript applications are known to perform better than Python applications for applications that require large amounts of memory [6].

For developing scalable web applications, both approaches have their pros and cons. With `Node.js`, the frontend and backend code could potentially both be written in Javascript, which would allow for the frontend and backend to be very well integrated with each other. However, `Node.js`'s non-blocking single-threaded architecture does not handle I/O intensive tasks very well [6].

Overall, due its greater ability to handle I/O intensive tasks (which is a necessity in our server herd) and its greater readability as well, `asyncio` is preferable over `Node.js` for our application server herd.

## 6. Conclusion

After using it to implement the application server herd and researching alternative approaches as well, it is evident that `asyncio` is the optimal framework for a proxy server herd and for our Wikipedia-based website. Its asynchronous, event-driven nature allows it to efficiently handle a large influx of client requests in a reliable manner. Although other approaches such as Java or `Node.js` do have some benefits, we can see based on our observations that `asyncio` is the optimal choice for our application.

## References


[1] *Python asyncio library Documentation*: https://docs.python.org/3/library/asyncio.html

[2] *TA Kimmo Karkkainen's Slides on Python and asyncio*: https://piazza.com/ucla/spring2019/cs131/resources

[3] *Python vs Java: Duck Typing, Parsing On Whitespace And Other Cool Differences*: https://www.activestate.com/blog/python-vs-java-duck-typing-parsing-whitespace-and-other-cool-differences/

[4] *Java 7 java.util.concurrent Documentation*: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-ssummary.html

[5] *Node.js About Page*: https://nodejs.org/en/about/

[6] *Python vs Node.js: Which is Better for Your Project*: https://da-14.com/blog/python-vs-nodejs-which-better-your-project