# COM SCI 131 Project: Proxy Herd with asyncio

Alex Chen, *UCLA*

## Abstract

Wikipedia and many other websites are based off the Wikimedia server platform. Constructed of many different system components, the one this project seeks to analyze is the redundant web server, single application server design and how it can be a bottleneck in a new Wikimedia-style service with different requirements. In this new service, updates occur more frequently, access by protocols further than HTTP need to be supported, and mobile clients exist. Thus, with such a design, the application server is a bottleneck because it is a single server attempting to serve frequent requests from clients.

This project attempts to solve this design issue by implementing an application server herd for this specialized Wikimedia-style service requirement. It uses Python's `asyncio` library to implement an application server herd, where the servers communicate with each other to send updates without the need to communicate with a central database and application server. Furthermore, we focus on the tradeoffs between Python and Java, and lastly, compare Python's `asyncio` to that of Node.js.

## 1. Proxy Herd Implementation

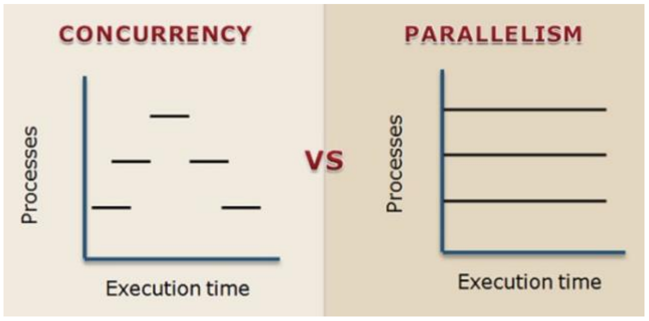
My implementation of the proxy herd utilizes five application servers. Each server has a bidirectional connection with a few other servers and can send and receive TCP connections from clients.

Upon receiving an update from a client about the client's location, the server will "flood" that update to the other servers in the herd. Then, whenever further clients may make a request, which requires that data, to a different server, a valid response can still be provided. Thus, this is how each server can communicate with each other in order to have the same state and information as other servers.

## 2. Asyncio

### 2.1 Usage

We use the `asyncio` library to aid our application herd implementation. With the single Python interpreter that exists, Python cannot have true parallel programming in which more than one task executes at the same time. However, the library allows concurrent programming to occur by allowing tasks can yield control of the interpreter to other tasks. Only one task will be executing at once, but by yielding, we can extract greater efficiency out of the interpreter.



The two important keywords where `asyncio` and Python come in play are `async` and `await`. The `async` keyword defines a function or certain expressions as a coroutine, while the `await` keyword is used by coroutines to suspend execution. Thus, together, `async` and `await` allow us to define coroutines that yield to each other. Whenever `await` is used, the current coroutine suspends execution, or yields, and allows another coroutine to utilize the interpreter / CPU.

The value in this is if the expression preceding `await` is some computationally heavy computation, such as a large database query or API request. Then, by yielding, while the expensive computation is executing, we can utilize the interpreter / CPU further.

### 2.2 Recommendation

My recommendation is that `asyncio` is a suitable framework for an application server herd. Once implemented, it was clear that many features of this design benefit from concurrency. First, concurrent server to client connections can be made, serving multiple clients at "once." Next, concurrency can be extracted from the Google Places API requests as well as the many server-to-server TCP connections occurring in flooding. While API requests are being made, or while servers are opening up connections with each other, the `await` action will allow other coroutines to be executed. This is important because API requests can be expensive, and TCP connections may also be expensive in the case of high network traffic. Thus, the interpreter / CPU will not just be idle. There is a large difference as opposed to having servers process sequentially.

In terms of the bottleneck in the original Wikimedia server design, we no longer have the application server bottleneck. With the `asyncio` implementation, servers can concurrently propagate messages to other servers upon receiving client requests, instead of relying on one central database / server.

### 2.3 Problems

In terms of problems I ran into, I would say there is a bit of a learning curve to writing asynchronous programs in Python, although it is not large. Learning how the event loop works and how to properly use the keywords is important to understanding how to add asynchronicity. I found myself confused by how `await` added asynchronous behavior because it would wait for whatever is being awaited. However, once I thought about it in terms of yielding, it made sense, and I was able to more easily determine where to add the keywords. It was also important to understand that coroutines may be executed out of order due to the library's asynchronous nature. For example, if an `IAMAT` request is made before a `WHATSAT` request, it is possible that these messages are processed out of order and stale results are returned. Lastly, because `asyncio` only supports TCP and SSL protocols, I had to use an additional asynchronous HTTP library, `aiohttp`, to make HTTP requests.

## 3. Python and Java Comparison

### 3.1 Type Checking

Type checking is an important point of comparison between Python and Java. In Python, duck-typing is used. This means that to determine what type an object is, certain tests are ran on that object. If the criteria for an object to be a duck is for it to quack and have white fur, for example, any object that quacks and is white will be treated as a duck. This is done at run-time In Java, static typing is used. This means that, at compile time, type is determined by looking at type annotations and performing graph traversals. The graph is one that marks relationships between different types and/or classes, and by traversing the graph, we can see if one type is a subtype of another. The benefit of static typing is that it is easier to debug and type errors can be caught before run-time, thus making many applications more reliable. Further, because these checks do not need to occur at run-time, the run-time performance will be better, although compilation will be slower.

The benefit of duck-typing is that code is simpler to write, both syntactically and semantically. If we want to declare a variable, we can omit a type annotation, and if we want to re-use a variable as an integer and an array, for example, we are able to do that without re-declaring a new variable. Duck-typing also has potential performance improvements for small programs, where only few checks need to be done. Static typing would potentially require searching through large DAGs to determine the relationships and typing.

For our use case, Python would work just fine, because the program should typically be small. Duck-typing should not have too much of a performance impact, while we retain the benefit of simplicity in coding, which positively impacts development speed and maintainability.

### 3.2 Memory Management

Memory management also has important differences. First, Python uses reference counts. This means that the number of times an object in memory is referenced is tracked, allowing for simple garbage collection. When an object's reference count reaches zero, its memory can be reclaimed. This does lead to memory leaks in a specific case, which is when object links create cycles. In these cases, reference counts will never reach 0. Overall, this memory management system is slow, because to update reference counts requires a load, increment / decrement, and store, which is three additional instructions and becomes expensive. For Java, mark-and-sweep is used to garbage collect. This first uses pointers on the stack to mark all objects that are pointed to, then reclaims all the unmarked objects. This approach is faster than updating reference counts, and does not leak memory as easily, but it is more memory intensive.

For our use case, Python's method is superior because there are not cyclic memory references created, so there will be no significant difference in memory leaks. This method will also save more memory and is more dependable. Java's memory management system can be more unpredictable and lead to unexpected delays in server execution and data inconsistency.

### 3.3 Multithreading

Lastly, multithreading works differently in Java and Python. First, multithreading does not truly exist in Python. In CPython, the most prevalent implementation of Python, there exists a global interpreter lock (GIL), which prevents multiple threads from executing Python bytecodes simultaneously [1]. This prevents race conditions and is necessary because CPython's memory management (reference count) is not thread-safe [1]. For example, multiple threads may access the same resource, causing a memory leak or memory that is still in use to be reclaimed [1]. For this reason, we can coroutines which provides concurrency, or even better, we can use the `multiprocessing` module which runs multiple processes, each with their own GIL [1].

On the other hand, Java has multithreading support inherently designed within it. There exists the `Thread` class, and to add locks to critical sections and prevent race conditions, there is the `synchronized` keyword. There also is the `wait`, `notify`, and `notifyAll` keywords for longer term locks. Lastly, Java also contains a synchronization library containing classes including `Semaphore`, `Exchanger`, `CountDownLatch`, and `CyclicBarrier` [3].

For this project, I would recommend Python where multithreading is concerned. The project has a few, defined places where concurrency is beneficial, and all these places can greatly benefit from cooperative multitasking, which `asyncio` provides. I do not think true multithreading is required because it is more useful in a use case where processing needs to be done by the CPU and the cores. In this use case, the expensive tasks can continue to run in the network or in I/O and do not require the CPU. However, if the latter approach which utilizes multiple cores is desired, we can always use `multiprocessing`. The benefit of utilizing Python, then, dominates, where those benefits are having less memory and performance overhead due to a single-threaded approach.

## 4. Comparison between asyncio and Node.js

`asyncio` and `Node.js` have many similarities. First, asyncio is based upon Python's single-threaded design, while `Node.js` also is designed to be single-threaded [5]. However, they still provide concurrency by utilizing an event loop which asynchronously executes tasks and allows for non-blocking behavior when expensive I/O is required. In `asyncio`, coroutines are used in the event loop, while in `Node.js`, callbacks are used which have a similar functionality [4].

`Node.js` has greater performance, since it is based upon Chrome's V8, which is fast and powerful [2].

Overall, concurrency in `Node.js` is very difficult to implement in complex applications, since anything beyond simple callback logic was not intended. Node encourages only using callbacks you understand well enough to do asynchronous programming [5]. In Python, it is a bit easier and simpler to implement since there are many supporting modules to utilize.

## 5. asyncio features of Python 3.9+?

The `asyncio.run()` feature was added in Python 3.7. It allows event loops to be created in a simple and safe way, thus also making it easier to write and debug asynchronous code.

An improved async `REPL` was added in Python 3.8, which makes it easier to use `asyncio` in the Python console, by running the command `python -m asyncio`.

These features are not crucial to our use case, or in most general cases. We can always workaround the `asyncio.run()` feature by manually creating an event loop with `asyncio.new_event_loop()`, and executing it with `asyncio.run_until_complete()`. For the `REPL` feature, it depends on if we are working with Python files or using the console. However, most use cases will ultimately use Python files.

Collectively, this means that it is not too important to rely on `asyncio` features of Python 3.9 or later.

## References

[1] *Is multithreading in python a myth?:* https://stackoverflow.com/questions/44793371/is-multithreading-in-python-a-myth

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

[3] *Package java.util.concurrent*: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

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

[5] *Intro to Async Concurrency in Python vs Node.js*: https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36