# CS 131 Project Report - Proxy Herd with `asyncio`

Benson Har
*University of California, Los Angeles*

## Abstract

The Wikipedia server platform is good for Wikipedia's purpose as an online encyclopedia. However, applying this approach to our application, a Wikipedia-styled service for news, will not work as well due to inherent bottlenecks and code inflexibility in the approach. We tackled these issues by trying a different server architecture called "application server herd", and we researched whether the library `asyncio`, an asynchronous networking library, would be suitable for implementing this. To test out these ideas out, we implemented a prototype Proxy herd for the Google Places API using Python and `asyncio`. Then we compared this approach to a Java-based approach, as well as Node.js and `asyncio`.

## 1 Introduction

Wikipedia and related websites are based on the Wikimedia server platform, which is built using on Debian GNU/Linux, Apache, MariaDB, and PHP+JavaScript. While this works well for Wikipedia, we are interested in building a new Wikimedia-style service designed for news. This means that updates to articles will happen far more often, access will be required via various protocols, and clients will tend to be more mobile. As a result, the PHP+JavaScript application server looks like it will be a bottleneck, and it seems as if it will be too much of a pain to add newer servers (e.g., for access via cell phones).

A potential solution is a different architecture called an "application server herd", where the multiple application servers communicate directly to each other as well as via the core database and caches. The interserver communications will allow for rapidly-evolving data while the database server will still be maintained for more-stable data that is less-often accessed.

We explore this idea by implementing a prototype proxy herd for the Google Places API in Python and the asynchronous networking library `asyncio`. Python's `asyncio` is an asynchronous networking library that is single-threaded approach for concurrency. With this we build a proxy herd consisting of five servers, allowing us find out how suitable Python and the `asyncio` library are for building the new Wikimedia-style service.

## 2 asyncio

`asyncio` is a library to write concurrent Python code, mainly for I/O-bound and high-level structured network code, with async/await syntax.

`asyncio` centers around declaring and running coroutines, which are tasks that can be executed, suspended, and then resumed when a response is returned. They can be implemented with the `async def` statement and `await`. These coroutines are queued into an "event loop" which then executes them concurrently.

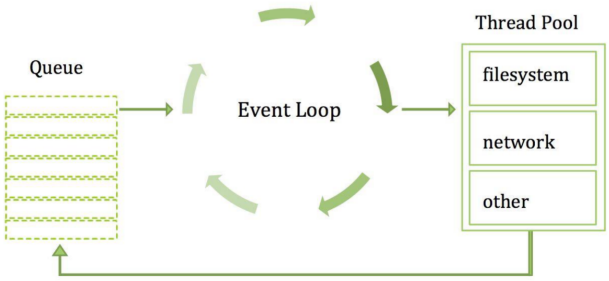


Figure 1: Diagram of the Event Loop from the TA Slides

In addition, `asyncio` provides an easy to use interface for writing servers. To illustrate in a simple outline:

1. To set up the server, pass the TCP port and IP address of the host to `asyncio.Server.start_server`.

2. Then, to run the server, pass the server invoking `asyncio.Server.server_forever` into the function `asyncio.Server.run`.

3. Finally, to close the server, use `asyncio.Server.close`.

## 3 Importance of asyncio Python 3.9 or later features

It is not important to rely on `asyncio` features of Python 3.9 or later, but it would be more convenient and would result in more concise code. This is especially evident in the function `asyncio.run` and the change in Python 3.9 where it uses the new coroutine shutdown_default_executor. The new coroutine makes it so that the executor will automatically be scheduled to shutdown and wait for the threads to join in the ThreadPoolExecutor. In versions earlier than Python 3.9, we would have to do those manually. In essence, we would have to stop the loop with `asyncio.loop.stop` and close the loop with `asyncio.loop.close`.

As for `asyncio.run` itself, it was introduced in version 3.7. According to the `asyncio` documentation, it automatically executes the passed coroutine, manages the event loop, and closes the thread pool when done. It is possible to replicate its behavior by setting up the event loop with `asyncio.new_event_loop`, run our coroutines with `asyncio.loop.run_until_complete`, and closing the event loop with the procedure mentioned before. Also, there are other ways to create tasks through the use of `asyncio.create_task` and other built-in functions that can result in `asyncio.run`'s functionality.

In addition, Python 3.9 introduced `asyncio.to_thread` that runs a function asynchronously on a separate thread. This can also be done manually by copying the context with `contextvars.copy_context` and running the function in the event loop.

As for `python3 -m asyncio`, it is a feature introduced in Python 3.8 that launches a natively async REPL, so commands can directly executed one at a time. However, since our application relies on executing python files, it is not useful. As a result, while the new features in Python 3.9 and up are convenient, they are not important as there are other ways to obtain the same behavior.

## 4 asyncio vs Node.js

As it turns out, `asyncio` and Node.js are very similar. They are both are libraries for high-level, dynamic programming languages (Python for `asyncio` and JavaScript for Node.js) and are concurrent, non-blocking libraries that excel in I/O-bound tasks. To circumvent the single-threaded nature of both Python and JavaScript, `asyncio` and Node.js employ an event loop to queue up tasks, which `asyncio` calls "coroutines" and Node.js calls "callbacks". Either way, both frameworks use features called `async`'s and `await`'s to deal with asynchronous code, though JavaScript has other options, like the feature "Promises", to deal with it, as well. In general, `asyncio` and Node.js are very similar.

## 5 Considering Python with Respect to Java

In this section, we consider Python's approach to type checking, memory management, and concurrency compared to Java's approach.

### 5.1 Type Checking

Python is dynamically typed. This means that the type of a variable is determined with run-time values. A major benefit to this is that functions can pass around objects without having to know or declare their types. As a result, this makes Python easier to write with and makes the code somewhat more concise. That makes the code more readable. Typically, this also means that development can be faster, especially with the elimination of compilation errors. Therefore, it makes writing the "application server herd" simpler, as it allows us to focus more on the logic of using the functions in the `asyncio` library to construct our application.

On the other hand, Java is statically typed, meaning the type of a variable must be known at compile time. Usually the compiler does this checking. This is done conservatively to make sure that the typing of arguments and parameters are consistent before a function is allowed to use them. The main advantage is that type errors and other trivial bugs are caught early, eliminating those type of errors during run time. This makes Java a bit safer for larger applications, as it ensures that these errors do not occur during run time.

### 5.2 Multi-threading

Python multi-threading is limited by the GIL, which is a lock that allows only one thread to hold the control of the Python interpreter. This means that for CPU-bound code (code that is CPU intensive), using more threads versus a single thread will not make a difference in performance as it would in Java. It essentially performs like a single-threaded application. It is not realistic to disable this feature since it would break legacy code and C extensions, so this is something that would limit the performance of our application.

Despite this issue, there are some solutions and benefits to this. For one, the GIL provides a performance increase to single-threaded applications due to only one lock needing to be managed. Moreover, the application we need for the Proxy herd is more I/O-bound than CPU-bound, meaning the program spends more time waiting for input to come from databases, other servers in the network, and clients. Specifically, this is seen in the IAMAT, AT, and WHATSAT commands that are sent to the server network in our prototype. Thus we can still achieve better performance by writing concurrent code, which we did through the use of `asyncio`. As

`asyncio` states in the introduction to its documentation, it is well suited for IO-bound network code. `asyncio` employs the use of event loops to get the effects of concurrency. This means that `asyncio` has the thread essentially queue up tasks in a loop and deal with the response when the task returns (ie. the server or database responds). This switching between tasks and dealing with them when they respond create an effect of parallelism due to the fact that multiple requests or responses would be active simultaneously in the event loop.

In contrast, multi-threading is allowed for Java. This allows Java to perform way better on CPU-bound tasks and still allow for the functionality `asyncio` provides Python. However, this introduces the risk of race conditions and deadlocks. This requires complicated synchronization methods to mitigate, even with Java's Memory Model.

With respect to our server herd, Python's single-threaded approach will be suitable, as each server receives and responds to messages (I/O bound tasks) rather than do heavy computations (CPU bound tasks).

### 5.3 Memory Management

Both Python and Java use garbage collectors (GC). In comparison to Java's GC, Python's GC is simple. It's method involves reference counting, where as soon as the reference count for an object reaches 0, the object's memory gets freed. This is a simple and cheap way to free memory. However, Python's GC will have issues with cyclic data structures, as they will never have a reference count of 0. Also, there is added overhead to keeping track of the objects since the reference counts have to incremented and decremented, but that matters less since interpreted languages like Python are slow already. Our application most likely will not have cyclic references, so Python is still suitable in this case.

On the other hand, Java's GC is more complicated. In essence, it runs the mark-and-sweep algorithm regularly to clean up the heap. This algorithm has the GC traverse the roots of the object trees and mark all objects that can be reached. Then it goes through the heap freeing objects that are not marked. Java's GC is a bit more efficient than that, since it uses a generation-based copying collector, where the GC focuses more on freeing newer objects and using a strategy where it copies over currently referenced objects into a new piece of memory and freeing the old chunk of memory. It is more complicated to implement and run but it deals with Python's GC issue with cyclic references.

## 6 Prototype Proxy Herd Details

Our prototype is an "application server herd" for the Google Places API using the `asyncio` library. Connections between clients and servers are done via TCP. Clients can connect to one of five application servers with IDs 'Juzang', 'Bernard', 'Jaquez', 'Johnson', 'Clark' and send IAMAT or WHATSAT messages. All other messages are considered invalid and will elicit the following response from the server:

```
? <INVALID MESSAGE>
```

The servers communicate with each other bidirectionally on the locations of clients with AT messages. In the following figure, we can see the pattern in which they communicate:

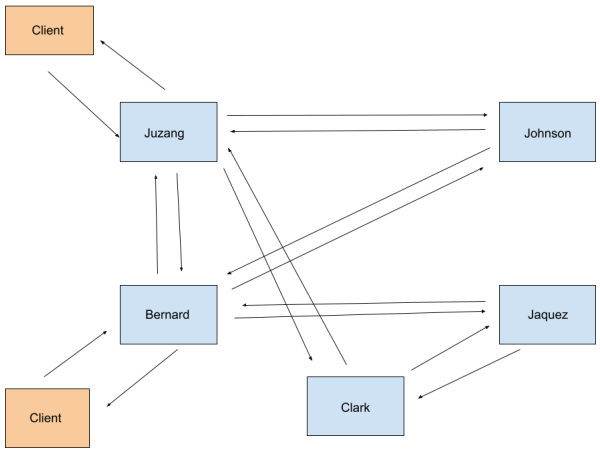


Figure 2: Each box is a server (blue) or client (orange). The arrows indicate the direction in which the they communicate.

Individually, the servers may query the Google Places API for places near clients. As for server-to-server communication, they propagate AT messages of clients to each other with a flooding algorithm.

### 6.1 IAMAT Message

IAMAT messages are sent from the client to a server. The basic format is:

```
IAMAT <CLIENT_NAME> <COORDS> <TIMESTAMP>
```

where COORDS is the latitude and longitude using ISO 6709 notation and TIMESTAMP is when the client sent the message in POSIX time.

The IAMAT message tells the server where the client is. This is kept tracked of by the server and propagated as AT commands to the other servers using the flood algorithm.

### 6.2 AT Message

AT messages are sent from the server to client and from server to server. The basic format is:

```
AT <SERVER> <DIFF> <CLIENT> <COORDS> <TIMESTAMP>
```

where DIFF is the difference between the server's idea of when it got the message from the client and SERVER is the server name.

The AT message is the response that the server sends to the client when it receives an IAMAT message.

It also is the message that the server propagates to its neighbors to let them know of the client's location. This is achieved by having the server connect to its neighbors with `asyncio.open_connection` and then using that to send the AT message.

### 6.3 WHATSAT Message

WHATSAT messages are sent from the client to a server. The basic format is:

```
WHATSAT <OTHER_CLIENT> <RADIUS> <MAX_INFO>
```

where RADIUS is the radius of the client in kilometers, OTHER_CLIENT is the name of the client queried, and MAX_INFO is the upper bound on the items that the Places API returns.

The server queries the Google Places API with the location and radius using `aiohttp`. If successful, it may prune the results to fit the upper bound and return the AT message of the client in question with a JSON-format message containing the response.

## 7 Benefits of Using asyncio

As stated before, `asyncio` made it relatively easy to write the servers, making the process to implement the server herds relatively straight forward. That and the fact that there are built-in functions for communication via TCP leads to the conclusion that it would not be difficult to add more servers, meaning that server herds much larger than five are feasible. Furthermore, if the application needs to query an API, it is relatively simple to use additional libraries like `aiohttp` to send HTTP requests to the API.

## 8 Negatives of Using asyncio

While `asyncio` has some major advantages, it also has some cons. Notably, it will not be useful if the bottleneck in performance is due to heavy computations done by each server. As stated before, this is due to the GIL and the resulting single-threaded nature of Python.

Another potential issue is how `asyncio`'s event loops use cooperative scheduling which is when tasks voluntarily give up control of the CPU. This may lead to issues where a task never gives up control resulting in the rest of the tasks in the event loop to never be executed. Even in the less extreme case where the task eventually gives up control of the CPU, performance can be severely throttled if multiple of this kind of task is queued.

## 9 Issues Encountered

Asynchronous code is always difficult to write and debug. The way I approached this was break down and test as much of my code as I could before integrating them together. So, if the IAMAT message to the server did not return a correctly formatted AT message, I could focus on one function/feature itself. This was a similar approach that I took with making sure that queries to the Places API was correct, as well. Moreover, using the sample hint code from the TAs helped me get started and quickly gain a better understanding of how `asyncio` worked, eliminating some of the time I would need to get used to the technologies.

## 10 Conclusion

In this project, we researched how suitable Python's `asyncio` library would be for building an "application server herd" kind of program. To explore this, we implemented a proxy herd for the Google Places API with `asyncio`. Notably, we found that the ease in which servers can be set up and integrated into the herd and the built-in support for client-to-server and server-to-server communication are major advantages. In addition, asynchronous strategies (like event loops) that `asyncio` employs to achieve concurrency are perfect for our Wikimedia type application, as the tasks that need to be done are mainly I/O bound. Though Java may have multi-threading capabilities and a more consistent GC, we find that those features are less relevant, especially if we avoid having the servers perform expensive computations and use cyclic data structures. Therefore, we find that Python and its `asyncio` library are suitable for the task.

## References


Node.js Documentation
https://nodejs.org/en/docs/

asyncio Reference Page
https://docs.python.org/3/library/asyncio.html

Abhinav Ajitsaria. What Is the Python Global Interpreter Lock (GIL)?
https://realpython.com/python-gil/

JavaScript Event Loop
https://www.javascripttutorial.net/javascript-event-loop/