# CS131 Project Report

Bryan Wong
Discussion 1A

## Abstract

In this paper, we will explore the Python asyncio library for asynchronous programming, in the context of a proxy server herd application. We will evaluate its effectiveness in creating an effective and scalable solution, and compare it to various alternative implementations, including Java and Node.js. We will also take a deep dive into various elements of async.io, including its performance and effectiveness as a framework for a proxy server herd.

## 1. Introduction

Wikipedia is an online encyclopedia website that hosts over 6.2 million articles and serves over 1.7 billion unique monthly visitors from across the globe. Built using GNU/Linux, Apache, MariaDB, and PHP+JavaScript, the Wikimedia server platform uses multiple, redundant web servers behind a load-balancing virtual router and caching proxy servers for reliability and performance.

While this solution works well for Wikipedia, we will explore an alternative Wikimedia-style news service with the following differences: it will receive far more frequent updates to articles, access will be required with more protocols (rather than just HTTP), and clients will tend to be more mobile. With the existing Wikimedia server platform, the PHP+JavaScript application server will become a bottleneck and adding newer servers will have poor scalability due to the large number of cell phone clients broadcasting their GPS locations.

A potential solution that optimizes this news service is an application server herd, where multiple application servers communicate directly to each other as well as via the core database and caches. Interserver communications allow for rapidly-evolving data (such as GPS locations) to stay updated across the herd. The database server will be used for less-accessed stable data or data that requires transactional semantics. Under this architecture, all servers will be able to communicate with each other even if some servers are not directly connected. By propagating messages (such as GPS locations), all servers will be updated after a few interserver transmissions without needing to consult a central database.

In order to implement this solution, we will explore Python's asyncio asynchronous networking library. Since asyncio is event driven, this will allow for an update to be processed and rapidly forwarded to other servers in the herd. In this paper, we will explore the pros and cons of using asyncio and examine a prototype application that proxies the Google Places API.

## 2. Comparing Python and Java

### 2.1 Type Checking

One of the most apparent differences between Python and Java is type checking: while Python is dynamically typed, Java is statically typed. Variables in Java must have a defined type at compile time, while variables in Python have flexible typing and are only checked at runtime.

While this dynamic typing makes Python code easier to write and more concise, it also comes with a number of downsides. Ambiguity in typing can make code less readable and more difficult to maintain long-term, especially if variables are not descriptive. This may be an issue if our application ends up having a relatively large codebase where multiple engineers are needed to maintain it. Additionally, dynamic typing may result in error-prone code that results in type errors caught only at run-time. On the other hand, static typing in Java makes type errors that would normally cause the program to crash easy to catch at compile time.

**2.2 Memory Management**
Both Python and Java have automatic memory management systems: the Python memory manager and Java Virtual Machine (JVM), respectively. These memory management systems automatically allocate and deallocate chunks of memory as needed by programs. The main difference between these systems is how garbage collection is handled. In a garbage collection system, the programmer does not need to manually free allocated memory— instead, a garbage collector automatically reclaims memory from objects that are no longer being used by the program.

The Python implementation of garbage collection uses reference counting. Every time an object is referenced or dereferenced, a counter is adjusted accordingly. When the counter is decremented to zero, the object's memory is freed. However, this can be a relatively inefficient system, since the interpreter must check the counter each time an object is referenced/dereferenced. This system is also unreliable, as it fails to address circular dependencies where objects that are actually garbage are unable to be freed. Due to this system, a Python implementation of our application may suffer from slower response times.

In contrast, the Java implementation of garbage collection uses a concurrent mark sweep (CMS) approach. The algorithm starts at the roots of object trees and traverses all objects, marking them along the way. Any objects that are unable to be reached are labelled as "dead" and moved to a chunk of memory that is later swept all at once. This results in a better performance, albeit at the cost of utilizing more memory. If our application finds itself frequently short on memory, it may be worth it to use the Python reference counting approach to garbage collection instead.

**2.3 Multithreading**
Another significant difference between Python and Java is the presence (or lack) of multithreading. In Java, the Java Virtual Machine (JVM) supports multithreading on multiple CPU cores. This allows tasks to be performed using true parallelization, and significantly optimizes CPU-intensive tasks like arithmetic calculations. Java includes several keywords, such as 'synchronized' and 'volatile', that are designed for concurrent programming and prevent race conditions.

On the other hand, Python does not allow for concurrent execution on multiple CPU cores. This is due to the Python global interpreter lock (GIL), which ensures that only one thread is allowed to run at a time. This is done to ensure that Python reference counting is consistent by preventing race conditions. In addition, Python is implemented using C libraries that are not thread safe. The consequence of this is that Python multithreading is not effective at optimizing CPU-parallelizable tasks, and is ideally utilized for IO tasks. Since our application is likely to be bounded by different IO tasks (receiving messages, propagating messages, and performing API calls), the lack of true parallelization is not a huge issue.

**2.4 Conclusion**
Though Python has a number of downsides when compared to Java, many of these issues are not large issues in the context of our application. Since our server herd is not performing any easily CPU-parallelizable tasks, the Python GIL model of multithreading is suitable for our IO-bounded system. We can utilize the Python model to perform useful tasks while polling for messages or waiting on API calls. Since the current implementation is a prototype with a small codebase, Python's dynamic typing is a good fit. However, if we are looking to expand and scale our application to a larger codebase with multiple engineers, it may be a good idea to use Java's static type checking instead. Also, Python's reference counting garbage collection model results in a slower but more memory-efficient system than Java's CMS garbage collection.

## 3. asyncio and Node.js

### 3.1 An Overview of asyncio

asyncio is a Python library that takes a nonparallel single-threaded approach for concurrent programming. It was introduced in Python 3.4, and allows tasks to voluntarily take breaks and allow other tasks to run. This makes it well suited to the needs of our application, where other tasks can be performed while waiting on an IO task to complete.

### 3.2 Implementation

The asyncio library is implemented using an event loop where tasks voluntarily take breaks and cycle through a queue of waiting tasks. Functions can be designated as a coroutine using the ‘async’ keyword. This allows it to suspend its execution and give control to another coroutine. Additionally, coroutines can use the ‘await’ keyword to suspend its execution until the awaited function completes. This allows asyncio to be a perfect fit for IO-bound network code, like the proxy server herd we are trying to implement.

asyncio reader/writer streams also allow for the creation of TCP servers that can send and receive data. By using asyncio coroutines, these TCP servers can manage multiple requests in an efficient manner. Later, we will discuss a prototype application that utilizes asyncio TCP servers in this manner.

### 3.3 Comparing asyncio and Node.js

Since both Python and Javascript heavily discourage thread-based parallelism (with Python’s GIL and Javascript’s single-threaded design), it is useful to compare each language’s concurrency solutions. Both Python’s asyncio and Javascript’s Node.js provide asynchronous options.

While the Python asyncio library enables asynchronous tasks in an otherwise synchronous programming language, Node.js is a runtime environment for Javascript that is inherently asynchronous. This means that many Python libraries have blocking, which makes them incompatible with asyncio. Node.js, on the other hand, is designed with asynchronous execution in mind first and foremost.

Both asyncio and Node.js utilize event loops to parallelize tasks, allowing for optimization of IO-bound tasks. Both also share the ‘async’ and ‘await’ keywords, which are functionally approximately the same. However, one difference is that Node.js uses callbacks, which specify functions to run when a task is completed. This prevents blocking and allows other code to run in the meantime. While asyncio uses awaitables (including coroutines), Node.js uses Promises, an equivalent object that represents the future completion of an asynchronous object and its return value. Finally, the asyncio library has several higher level functions, like asyncio.sleep and server commands, while these must be implemented by the programmer in Node.js.

### 3.4 asyncio for Server Herds

When evaluating asyncio for use in writing a server herd application, we can see that there are several tools that make writing such programs easy and convenient. The asyncio.start_server() function allows a TCP server to be easily started by specifying a handler function, an IP address, and a port. By calling ‘await server.serve_forever()’, the server can receive and send messages continuously. In our prototype application, we use these functions as well as reader/writer streams to parse and response to client commands. By using the ‘await’ keyword to asynchronously read and parse messages, we are able to handle multiple concurrent commands. For our application the ‘await’ keyword is also used to asynchronously propagate “AT” messages to neighboring servers and to make calls to the Google Places API.

In addition, there may be some scalability issues with our server herd implementation. As we increase the number of servers (perhaps to accommodate more visitors), data freshness will suffer as the number of interserver transmissions increases. This may result in responses to WHATSAT queries referencing outdated information. However, this scalability issue isn’t inherently due to asyncio, we may see the same thing happen with an alternative framework.

### 3.5 Performance Implications

For the purposes of our prototype, asyncio is a great fit. Since our application is bound by lengthy IO tasks, voluntary yielding of tasks allows us to create a responsive and robust server. However, if we wanted to perform more CPU-intensive tasks, such as complex arithmetic computations, the single-threaded nature of asyncio would be a large bottleneck. If we were to use a framework that instead allows true multi-core parallel processing, CPU-intensive tasks could be significantly optimized. While asyncio is great for handling IO tasks, it also prevents the use of true thread-based parallelism, which could hurt the performance of some applications.

### 3.6 Reliance on Python 3.9

In Python 3.9, various changes were made to asyncio, though none of them are too significant. asyncio.run() was updated to use the shutdown_default_executor() coroutine, that schedules a shutdown for the default executor. A new useful tool, the coroutine asyncio.to_thread(), was added. It allows IO-bound functions to run in a separate thread without blocking the event loop. However, the 'run_in_executor()' function can be used alternatively in older versions.

Otherwise, there were a few other miscellaneous changes. The 'reuse_address' parameter of asyncio.loop.create_datagram_endpoint() was removed due to security concerns, so programs using this functionality in older versions should consider removing it. Ultimately, Python 3.9 provides some new features and optimizations, but developers should be able to manage using older versions.

## 4. Conclusion

After evaluating Python's asyncio library against several alternatives, we have come to the conclusion that asyncio is recommended for the proxy server herd application, conditional on no CPU-intensive tasks being added. asyncio allows for various IO-bound tasks to be asynchronously performed by having tasks voluntarily yield control while waiting for completion. Compared to Java, Python has some inefficiencies in its garbage collection model and less strict type checking, though these disadvantages are offset by the convenience of writing code to set up asyncio TCP servers. If more CPU-intensive arithmetic tasks are needed in the future, asyncio should not be used. Since asyncio is a single-threaded application, the efficiency gain from using a true multithreaded parallel approach is lost. Additionally, it may be a good idea to use Java for stricter type checking and a more efficient memory model when scaling the application to a larger codebase.

## 5. References

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

Brad Solomon. Async IO in Python: A Complete Walkthrough
https://realpython.com/async-io-python/

Brett Cannon. How the heck does async/await work in Python 3.5?
https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/

Tom Radcliffe. 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/