CS 131 Project
Proxy Herd with Python's `asyncio`

Arnold Pfahnl
University of California, Los Angeles

Abstract

In this project, the viability of utilizing an "application server herd" for replacing parts of Wikimedia's platform is explored. Specifically, we focus on Python's `asyncio` asynchronous networking library and test a simple and parallelizable proxy herding application with the Google Places API. This implementation is then compared to a Java-based approach as well as Node.js.

1 Introduction

The Wikimedia server platform is the basis for massive content-serving sites like Wikipedia, and is based on Debian GNU/Linux, the Apache web server, the Memcached distributed memory object cache, the MariaDB relational database, the Elasticsearch search engine, the Swift distributed object store, and a combination of PHP and Javascript. Redundancy is provided by the Linux Virtual Server load-balancer based on a cluster of servers with two levels of caching proxy servers for reliability and performance.

A new Wikimedia-style service for news has the following properties:

1. Updates to articles that happen far more often than Wikipedia.
2. Access required via protocols not limited to HTTP or HTTPS
3. Clients that tend to be more mobile.

The issue is that the PHP and Javascript application server appears to be a bottleneck, since it is difficult to add newer servers, and the application server has a response time that is too slow.

The "application server herd" is a prime candidate as an alternate architecture for the platform. Multiple application servers communicate directly with each other as well as via the core database and caches. Interserver communications should handle rapidly-evolving data from short strings of GPS location to large ephemeral video, while the database server is used for stable, less-often accessed data or data that is transactional in nature.

Python's `asyncio` asynchronous networking library has the potential to be a good match for this problem. The library's single-threaded concurrent programming, event-driven approach allows an update to be processed and forwarded rapidly to other servers in the herd. This project experiments with a five-server herd with an uneven communication spread to test the library against the new requirements.

2 Python Versus Java

Python's type checking, memory management, and multithreading are handled much differently than Java and the differences and issues are discussed below.

2.1 Type Checking

Python is dynamically typed, meaning that variable types are checked as the application is being run by the Python interpreter. This allows for an incredible amount of flexibility as variables can change types on the fly, and it makes prototyping, gluing together applications, and establishing a functional program much quicker. This also has the added benefit of increasing readability since the codebase tends to be much cleaner and concise. Python also provides type hinting that allows developers to indicate what types are expected for certain variables, so variable types can be checked before running the program to a certain extent. However, as the name implies, type hinting is more of an annotation rather than an actual check, so it doesn't serve as a replacement for static type-checking – you can program without following the type hints.

Java is statically typed, meaning that type checking is performed at compile time. Java will detect issues with type mismatches before the program is run, and this allows for more robust and predictable code. This is especially useful for applications that run for long periods of time, or need a

low failure rate, since this helps avoid many runtime errors. The major drawback is that the codebase has to be designed with particular types in mind, and changes to accommodate different types can be rather difficult.

2.2 Memory Management

Garbage collection (GC) is the basis for memory management in both Python and Java.

Many Java implementations have GCs that utilize a concept called generation-based copy collection [1]. In this approach, objects are initially allocated a chunk of memory, and when that memory becomes full, all live objects are copied into another block of memory, and the entire initial block is freed. When the second block becomes full, live objects are copied into a third block, and the same process happens over and over again. This exploits temporal persistence of objects in memory, the idea that if something is in memory for a long time, it will probably stay in memory for a long time.

The main issue with this is in the `finalize()` method for an object that's called by the GC when it's about to free an object's storage. This method is typically empty, but sometimes it points to other objects that also need to be freed. However, the generation-based copying collector doesn't know where that garbage is, so a traditional mark-and-sweep is typically used in tandem for objects with non-empty `finalize()` methods.

Mark-and-sweep is a classic algorithm that marks all live objects, usually with some form of depth-first traversal, and sweeps all unmarked objects into the free list. This is a simple but often slow process, hence the use of generation-based copy collection most of the time for Java.

Python's GC is simpler in that it uses reference counting to keep track of objects that are in use. A new pointer assignment means an increment or decrement of the object's reference counter, and this makes Python's GC rather slow. When the reference count drops to zero, the object isn't used by anything and it can be freed immediately. This works in most cases, but sometimes a cyclic list may be created where an object has a chain of references that leads back to itself, but nothing references any object in the chain. Thus, Python implementations like CPython utilize both reference counting, used in most cases, and mark-and-sweep for corner cases.

Overall, Python's memory management is slower than Java's, but both work well at managing memory.

2.3 Multithreading

Multithreading in Python, specifically the CPython implementation, is limited by the Global Interpreter Lock (GIL) that only allows one thread to execute any CPython bytecode at any time [5]. Thus, no parallel threading can occur, but this makes garbage collection much simpler to implement, since this prevents race conditions for reference counters. Unlike CPython, Jython and IronPython, which are implemented in Java and .NET respectively, don't have a GIL, and can "fully exploit multiprocessor systems" [5]. Multithreading in Python allows for concurrency, but not parallelism.

In Java, multithreading is a built-in capability of the language. However, this also comes with the difficulty of placating the Java Memory Model, ensuring that race conditions and the accidental negative effects of load/store reordering optimizations are mitigated with complicated features of the language like the `synchronized` and `volatile` keywords.

Implementing a multithreaded program comes with many caveats and is oftentimes not the correct choice when it comes to building reliable software. In Python, there are libraries that allow for intensive parallel computing outside the GIL that are robust and don't require any complex multithreaded programming knowledge from the developer's side. NumPy is a classic example that can perform vectorized mathematical operations with parallelization using Basic Linear Algebra Subroutines (BLAS) that can be compiled to take advantage of multicore machines with threading [6]. In essence many libraries provide multithreading capabilities without the programmer having to actually write out any multithreaded code themselves.

3 `asyncio` Compared to Features of Node.js

The Python `asyncio` library and the JavaScript runtime, Node.js, share very common features when it comes to asynchronous networking [3]. Both utilize an event loop construct that waits for and dispatches events in a program. The following keywords are shared by both:

async marks a function as a coroutine in Python. Coroutines can be suspended and resumed, and other coroutines can run when the currently executing coroutine is suspended. In Node.js, **async** marks a function that calls an async function that returns a promise. A promise is an object that represents the completion or failure of an asynchronous operation. In this sense, the marked Node.js function is similar to a Python coroutine.

await suspends the current Python coroutine until the awaited asynchronous function is finished. In Node.js, **await** similarly suspends the current routine until the promise from the async function is resolved or rejected.

Both Python and Node.js also have similarities in their difficulties with parallelized code. Python's CPython implementation does allow threading but only one thread at a time, and JavaScript doesn't support threads at all. In a sense, both were designed with single-threaded programming in mind, but both allow for concurrency through asynchronous programming. The single-thread, single-process cooperative multitasking model is a feature that both Python's `asyncio` library and Node.js share.

`asyncio` Necessity of Newer Features

The `asyncio.run()` function introduced in Python 3.7 creates, executes, and closes an event loop, and is something that should only be called once in a program [4]. It's the preferred point of entry for `asyncio` programs in Python 3.9. However, it is not a necessary function, and is more of an abstraction that packages lower level functions like `asyncio.new_event_loop()` and `loop.run_until_complete()` found in both current and older Python versions. Thus, getting by with older Python versions requires only a small bit of refactoring, a relatively easy task.

Recent version of Python, including Python 3.9 also allow for `asyncio` to be run from the command line with `python -m asyncio`. This creates a Python REPL with the following prompts (some details have been replaced with "..." for brevity):

```
asyncio REPL 3.9.5 (default, ...)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", ...
>>> import asyncio
>>>
```

All this does is create an environment to quickly prototype new asychronous functions and programs and isn't necessary at all for development with `asyncio`. Note that one of the features it mentions is that asynchronous functions can be called with the `await` keyword without having to explicitly start an event loop, since it was started with the REPL. Thus it's easy to to get by with older Python versions that don't have this feature.

4 Implementing an `asyncio` Server Herd

To test the viability of the Python `asyncio` library, I've been delegated the task of implementing a server herd that synchronizes data among a group of servers. Clients can share their location, and this location data is synchronized across all the servers. A querying client can request information about the locations of other clients-of-interest by querying any server. The server then responds with information about places near the client-of-interest using the the Google Places API with location information previously shared by the client-of-interest.

4.1 Server Herd Architecture

The prototype server herd contains five servers that accept TCP connections and communicate bidirectionally with the `asyncio` library in the following manner:

1. Riley talks with Jaquez and Juzang.
2. Bernard talks with everyone else but Riley.
3. Juzang talks with Campbell.

Servers communicate by propagating *AT* messages that will be described later. A simple flooding protocol ensures that once any server receives information about a client, all servers will receive that information. In my implementation, I've also made sure that all servers check the timestamps of messages being propagated against the message for a client that is currently stored. If the message was sent before or at the same time as the currently stored image, we halt the message's propagation. Otherwise messages might be propagated forever.

The server herd is also robust in the sense that a server will continue to operate if any of its neighbors go down. Ideally, the server herd graph should be connected at all times so that all servers can update with new information, but this can't be guaranteed.

The servers also log all actions with the Python `logging` library, with messages for informational and debugging purposes as well as error reporting. Each sever outputs its log to a file with the format `<server name>.log` where `<server name>` is a placeholder for the actual server's name.

4.2 Client Messages

Emulated mobile clients with IP addresses and DNS names can interact with servers through TCP connections through two message types: *IAMAT* and *WHATSAT*.

IAMAT messages are used to share location information with the server herd. The format is as follows:

```
IAMAT <client ID> <location> <time>
```

The following is an example *IAMAT* message:

```
IAMAT kiwi.cs.ucla.edu +34.068930-118.445127
      1621464827.959498503
```

The *client ID* is a unique identifier for the messaging client, the *location* field is longitude and latitude expressed in ISO 6709 notation, and the *time* field is the client's idea of when the message was sent, expressed in POSIX time.

WHATSAT messages are used to query the Google Places API based on the location of some client, not necessarily the querying client. The format is as follows:

```
WHATSAT <client ID> <radius> <upper bound>
```

The following is an example of a *WHATSAT* message:

```
WHATSAT kiwi.cs.ucla.edu 10 5
```

The *client ID* is the unique identifier of the client whose location is being used for the query. The information of for this client must already be stored by the servers for the query to be successful. The *radius* is the distance, in kilometers, to search for places from the client-of-interest, and the *upper bound* is the maximum number of results to be returned from the query. The maximum radius is 50 km and the maximum upper bound is 20.

4.3 Server Messages

A server responds to a client's *IAMAT* message with an *AT* message. The *AT* message has the following format:

```
AT <server name> <time diff> <client ID>
            <location> <time>
```

Note that there are no newlines in the message. The following is an example of an *AT* message:

```
 AT Riley +0.263873386 kiwi.cs.ucla.edu
+34.068930-118.445127 1621464827.959498503
```

The *server name* is the name of the server, *time diff* is the difference between when the server perceived it had received the client's *IAMAT* message and the *time* field of the *IAMAT* message, and the *location* and *time* fields are copies from the *IAMAT* message. The time diff field can be negative due to clock skew.

When responding to a *WHATSAT* message, the server responds with an *AT* message in the same format as described before, and it provides a JSON-formatted message with the place information provided by the Google Places API. The format of the message is the *AT* message followed by a newline, then the JSON message stripped of trailing newlines, then two newlines.

For invalid commands, the server responds with a question mark "?", a space " ", and a copy of the invalid command.

4.4 Querying the Google Places API

Since the Google Places API requires an HTTP request, and `asyncio` only supports TCP and SSL, my implementation uses the `aiohttp` library to create and send an HTTP GET request.

Performance and Recommendation

Overall, the `asyncio` library provides a great way to run and exploit server herds. With the simplicity of Python, it is fairly easy to write `asyncio`-based programs. Additionally, the library is fairly mature with many "batteries included" and this makes building an application fairly quick. Turning existing non-asynchronous programs into asynchronous programs is also a breeze since all it takes is a little bit of syntactic sugar to make the transition. Starting a server or client, handling requests and connections, reading and writing, and much more are made easy with the well-documented and intuitive API. The usage of coroutines to handle requests asynchronously allows for the server to be high-performance when there is lots of blocking I/O operations, and this is perfect for a use case where servers are expected to deal with heavy I/O loads.

Even with these benefits, `asyncio` is inherently limited by its single-threaded model, which can make scaling up a challenge. One notable drawback is the breakdown in performance when `asyncio` is used with CPU bound code [2]. If CPU intensive code is executing with `asyncio`, the CPU bound code will naturally degrade performance since both occupy the same thread and `asyncio` will have a tough time serving any request. There is also the disadvantage for certain use cases that could benefit from multithreaded parallelism that just can't be done in Python in general due to the GIL.

Conclusion

Through research and exploration of the features of Python's `asyncio` library and comparing it to Java and Node.js, `asyncio`'s viability for creating application server herd architectures was established. The prototype server herd demonstrates that the library has great potential as the basis for the new Wikimedia-style news service. In the future, further research may need to be conducted on larger scale simulations of the library to determine how well it can scale up in practice.

References

[1] Cornell. Copying garbage collection. https://www.cs.cornell.edu/courses/cs312/2003fa/lectures/sec24.htm. Accessed: 2021-05-30.

[2] Nathan Van Gheem. Scaling python web applications: Asyncio vs threads. https://www.nathanvangheem.com/posts/2019/06/11/scaling-python-web-applications.html. Accessed: 2021-05-31.

[3] Andrei Notna. Intro to async concurrency in python vs. node.js. https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315 Accessed: 2021-06-01.

[4] Python. asyncio – asynchronous i/o. https://docs.python.org/3/library/asyncio.html. Accessed: 2021-06-01.

[5] Python. Globalinterpreterlock. https://wiki.python.org/moin/GlobalInterpreterLock. Accessed: 2021-05-30.

[6] SciPy. Parallel programming with numpy and scipy. https://scipy.github.io/old-wiki/pages/ParallelProgramming. Accessed: 2021-05-30.