Suitability of Asyncio as a Framework for Application Server Herds
CS 131 Project Report

Joseph Picchi
UID: 605-124-511

Abstract

This project provides a prototype of a simple application server herd that is implemented in Python using the asyncio library. The application server herd features five different servers that run on different TCP ports and communicate simple messages directly to each other and to clients. The ease of development with Python and simplicity of the application server herd source code testifies to the suitability of asyncio as a framework for this type of application. Furthermore, research into the workings of Python and the performance implications of Asyncio reveal that the single-threaded, event-based, asynchronous execution model of asyncio is ideal for I/O intensive server requests. This execution model also allows the server herd to overcome limitations imposed by high-latency python operations (e.g. garbage collection) and lack of parallelism by scaling up the server count with ease. Overall, although the interpreted Python bytecode introduces performance concerns, these concerns can be addressed by the herd structure and asynchronous event handling of asyncio, thus making asyncio a suitable framework for a server herd application.

1 Introduction

Wikimedia-style server platforms work effectively for many websites, but their advantages and drawbacks are dependent upon platform implementation [Egg]. For example, the wikimedia server platform that supports Wikipedia is implemented with a network that incorporates Debian GNU/Linux, an Apache web server, Maria DB relational databases, and application code written in PHP+Javascript, amongst many other components.

Although this implementation is effective for Wikipedia, it presents considerable concerns for a wikimedia-style news service that seeks to support frequent updates to articles, accessibility with various protocols (not just HTTP/HTTPS), and compatibility with mobile clients [Egg]. Using a Wikipedia-style approach to implement such a service may introduce variety of issues: bottlenecks from PHP+Javascript application code, difficulty modifying software to add more servers with ease, and slow response times due to the Wikimedia application server acting as a bottleneck on core clusters.

As an alternative to the Wikipedia-style approach, an application server herd prototype was implemented. The application server herd allows clients to connect to any one of the servers in the herd to receive desired services. Each server communicates nominal information directly to the other servers through a basic "flooding protocol" in order to quickly spread information across the network and avoid the bottleneck imposed by storing and pulling information from a central database.

This project implemented a prototype of the application server herd that features up to 5 servers that can receive, propagate, and service messages from clients. The application is written in Python using the asyncio library, which provides an asynchronous event loop to which events can be added and eventually serviced in a concurrent fashion. The 5 servers communicate with each other through TCP connections, flooding client interaction updates across the network as they are received in order to support data consistency across all servers in the network.

2 Implementation Details

The server is implemented as a single file `server.py`. To start up one of the five servers, the script is run with the command `python3 server.py <server-name>`.

In the `server.py` file, the main function processes command line arguments, creates an object instance of the Server class, and calls `asyncio.run()` to asynchronously run an async member function of the Server class.

The Server class then starts up a server on our desired port using `asyncio.start_server()`, and it calls the `asyncio.serve_forever()` method to enable the server to handle requests.

Every time a client connects to the port on which our server is listening, an async function in the Server class handles

the request, using asynchronous methods to read data from the TCP connection, query the Google Places API, and send messages back to the server via TCP. Each asynchronous request is added to the event asyncio event loop, and requests are sequentially removed from the event loop and processed by the server.

2.1 Flooding Protocol

Once an AT message is sent back to the client, that AT message is stored in the server that created it. That server then opens TCP connections to each of the servers to which it can transmit messages, and it sends the AT message that it just created to those servers.

Every time a server receives an AT message, it checks whether the AT message it has received has a newer client timestamp than the AT message currently in its database for that client. If not, the server does nothing. Otherwise, it adds the AT message to its database, overwriting any previous AT message that contains the same client ID, and it propagates the AT message to each of the servers to which it can transmit messages via TCP. This process continues for every server that subsequently receives the message until every server that can be reached either already contains the AT message or already contains a more recent AT message from that client, at which point the message ceases to propagate.

At this point, all servers that can be reached have stored the AT message, so they can respond to WHATSAT queries with the most recent information about the client whose ID appears in the WHATSAT.

3 Python vs. Java

3.1 Type Checking

The Python language is dynamically typed [sna]. This means that variables in Python need not be bound to variables of only one declared typed. Instead, Python variables can be bound to objects of different types throughout the execution of a program. For example, one variable could be bound to an integer, then be reassigned to a string later in execution.

Dynamic typing makes the python language highly flexible because variables can be reused for different types of objects. It also makes the syntax more clean and readable because programs are not muddled by type names and type checks. Dynamic typing also makes the development process more convenient because the programmers do not have to labor over type considerations, and they can focus more on computational logic.

Dynamic typing is dangerous because type checking occurs at runtime [sna]. Therefore, program bugs that are caused by type errors will not be caught at compile time, and they may cause programs to crash during execution. For our purposes, this would cause servers to crash if they have not been exhaustively tested beforehand.

In contrast, Java uses static typing. The type of each variable is explicitly declared when the variable is created, and Java variables can only be assigned to objects that share the same type as the variable [sna]. Type checking occurs at compile time, which means that errors from programming logic that manifest themselves in type errors are quickly caught before the program ever runs. This is a clear advantage for large programs, since the probability of errors increases as program complexity increases.

Since our prototype program is relatively simple, dynamic typing provides convenient development with low error probability. However, for a large-scale server system with considerable software complexity, static checking would be more advantageous because it would ensure that program bugs that manifest as type errors are quickly caught during compilation, thus preventing server crashes without the need for extensive testing.

3.2 Memory Management

Python uses a garbage collector and memory allocator to manage its program memory [Egg21]. When objects are allocated, the memory allocator finds memory in the free list and stores the object in that memory. When objects are freed, their memory is returned to the free list.

The Python garbage collector is premised upon reference counts [Van] . Each object has an extra word that counts the number of pointers that address the object. Every pointer assignment to an object has the additional computational cost of an extra increment and decrement to that object's word. Once the reference count of an object reaches 0 (i.e. it is no longer accessible to any part of the program), the object is freed using its internal deallocation function.

The only issue with this is that the reference count tracking method is ineffective against circular references (e.g. A is an object that contains a reference to B, and B is an object that contains a reference to A, but neither is referenced by any other object) [Egg21]. In this case, neither A nor B will be freed because their reference counts are nonzero. Python deals with this by occasionally running mark and sweep to clear out objects with circular references. This is not an issue for our program because circular references can be easily avoided. They would only occur by accident, and the cost of removing them is minimal.

In contrast, Java uses a generation-based copying collector for most objects, and it uses mark and sweep for objects that lack a `finalize()` method [Egg21]. Java allocates memory from the beginning of a chunk of free space. When that free space runs out, it copies allocated objects to a different region in memory, separating those copied objects into "generations", based on how old they are compared to other objects in the program. This copying procedure avoids fragmentation, and

typically it is only performed in the nursery (i.e. the generation with the newest objects and the free space) because this is the region that most likely requires garbage collection.

The Python memory management method is advantageous for our program because the cost of tracking references is relatively low, and it results in relatively homogeneous memory access and edit times. The lack of necessity for circular references in our program avoids the need for costly mark and sweep procedures. The Java memory management method is slightly more costly when the copy collector is executed to free up memory, leading to memory access and update times that are slightly more homogeneous. This could be an issue as the number of clients to our servers scales up, since this would lead to greater memory usage. Thus, the Python memory management method advantageous for our application.

3.3 Multithreading

The Python Global Interpreter Lock (GIL) locks the entire Python interpreter while a thread is executing [Mon21]. This means that no 2 threads can ever execute in parallel: only one can execute at a single time. The GIL avoids race conditions in memory allocation and garbage collection, since two threads running in parallel may simultaneously access a shared resource and non deterministically update its reference counts, thus stimulating undesired actions by the garbage collector to free or fail to free the object in question.

Due to the GIL, Python does not allow threads to run in parallel [Mon21]. They instead must be run concurrently (e.g. using asyncio). This results in fast single-threaded code and support for C libraries that are not thread-safe, but it limits parallelism-driven throughput speedup.

In contrast, the Java garbage collector can be run in multithreaded programs in such a way that race conditions are avoided [Mon21]. Therefore, Java has extensive support for multithreaded applications, though it requires careful use of synchronization primitives to produce deterministic outcomes. Thankfully, Java synchronization abstractions are highly object-oriented, making it easy to use them in many applications.

The lack of parallel thread execution in Python is not a huge concern for our application. Since the wikimedia server is being implemented as a server herd application, each server can run on as a separate program and operate independently of the others, thus operating in parallel without the need for parallelism via threads. Furthermore, as long as the computational requirements to service each client request is low, concurrent execution is satisfactory because requests can be distributed across the server herd relatively easily in order to serve multiple client requests through different servers simultaneously.

Figure 1: Coroutines are added to the event loop and executed asynchronously [Mon21].

4 Asyncio Library

Asyncio is a single-threaded library that supports concurrent execution [Mon21]. It operates through cooperative multitasking, in which tasks voluntarily yield the CPU to allow other tasks to process.

Developers mark coroutines explicitly by defining them with the async keyword [Pyta]. These coroutines can be temporarily suspended and resumed, where their suspension permits other coroutines to run. Coroutines are run by adding them to an event loop, from which they can be extracted and executed, possibly adding other coroutines to the event loop in the process. Such a process enables coroutines with high I/O requirements to wait (via the "await" keyword) while other concurrent processes are executed in the meantime.

4.1 Ease of Development

The asyncio library methods are highly encapsulated and object-oriented, making it extremely easy to set up a server and handle requests from the event loop. Similarly, the `async` keyword clearly distinguishes coroutines from non-coroutines, which improves code readability and enforces error checking when adding events to the event loop [Pyta]. The `await` keyword also improves code readability and clarity because it specifies exactly when a function can yield, and it distinguishes coroutine calls from other method calls in yielding functions.

Overall, asyncio's object-oriented nature hides the details of event loop management, making it extremely easy to set up a server herd.

4.2 Difficulties Encountered

Although the object-oriented nature of asyncio makes it convenient to use, it cultivates a variety of different ways to implement the same functionality. This can decrease code readability and increase complexity because styles across developers are allowed to vary. For example, to run a collection of coroutines concurrently, one can either create the coroutine instances and call `asyncio.gather()` with each of the coroutines as an argument, or one can call each of the coroutines from a single separate coroutine and use `asyncio.run()` on

that top-level coroutine to set up an event loop [Pyta]. These are just two ways amongst many to produce the desired result. This issue is further accentuated by the fact that developers can achieve the same result by using either the high-level API or the low-level API of asyncio.

The fact that many operations exist to achieve the same result has a few implications. One is that the encapsulated asyncio functions hide the details of their internal workings to the extent that the results of the computations are equivalent, but the performance is dependent upon the application. This would require the development team to research which asyncio method is most optimal for the server herd application by analyzing the asyncio source code. Another implication is that this features is positive for individual programmers because it supports flexibility in the server implementation, though this may make team collaboration on the same application difficult.

4.3 Performance

The obvious performance concern related to asyncio is lack of support for parallel computation [Mon21]. This limits the throughput of a single server because it cannot spawn threads to service multiple coroutines at once. Instead, it must service coroutines one at a time, yielding whenever that coroutine must wait for some process (e.g. I/O or another coroutine) to complete.

However, given that the load can be balanced across multiple servers, this is not a huge problem for a server herd implementation. Since each server operates independently from other servers, only relaying occasional through message transitions and database retrievals, servers in the server herd can essentially operate in parallel. Thus, asyncio services individual coroutines quickly using fast single-threaded code, and it exploits high-latency operations as opportunities to execute other coroutines. At the same time, parallelism is still simulated in our server network because each server has its own event loop that is managed and serviced in parallel to all of the other servers.

Therefore, although the throughput of each individual server is limited by lack of parallelism, the throughput of the network is still high as long as requests are properly load-balanced across the herd.

4.4 Reliance on Python 3.9 Asyncio Features

It is not extremely important to rely on asyncio features of Python 3.9 or later. `Asyncio.run()` was updated to use the new coroutine `asyncio.shutdown_default_executor()` in Python 3.9 [Pytb]. This update is unimportant as long as a `ThreadPoolExecutor` is not used in the server herd. The prototype implemented in this project demonstrates that a `ThreadPoolExecutor` is not necessary to achieve the desired functionality, but even so, it could still be incorporated into the application if necessary by using alternatives to `asyncio.run()`. Alternatives in older Python versions are slightly less concise, but they are nonetheless easy to use. For example, we can manually create and execute event loops with `asyncio.new_event_loop()` and `asyncio.run_until_complete()`, adding coroutines to the event loop as desired.

Similarly, other Python 3.9 asyncio features can be easily avoided in favor of alternatives. `asyncio.PidfdChildWatcher` is unimportant for our server herd because there is no immediate need to poll unix file descriptors, `asyncio.to_thread()` can be can be avoided by using `asyncio.run_in_executor()`, since the former essentially a high-level version of the latter, typeErrors can be manually detected without the update Python 3.9 update to `ssl.SSLSocket` [Pytb]. Lastly, `python -m asyncio` is relatively unimportant for the server herd application because the server implementation is stored in files that do not require a REPL.

4.5 Comparison to Node.js

Both Asyncio and Node.js are event-driven server architectures that are capable of asynchronous event handling [Noda]. Javascript is single-threaded in design due to its support for only one call stack, which means that parallelism across threads is limited in both Node.js and Asyncio. Just like the low-level API of Asyncio, Node.js offers asynchronous I/O primitives that prevent blocking and promote asynchronous execution of other tasks during I/O latency. Node.js has similar object-oriented functions to create and manage event loops. For example, it has its own `create_server()` method that creates a new HTTP server, similar to the Asyncio method `Asyncio.create_server`.

Asyncio and Node.js are both implemented with event loops, making almost all operations non-blocking so as to repeatedly complete asynchronous tasks in the event loop while the program waits for high-latency operations to return [Noda]. Node.js also supports the "async" and "await" keywords in contexts that are very similar, if not equivalent, to the contexts in which they are used for Asyncio [Nodb]. Async functions in Node.js differ in the sense that they always return a promise, which is a construct that does not appear to exist in Asyncio. Promises are placeholders for values that will eventually become available, but which allow the calling function to continue executing until they result in a resolved or rejected state.

A difference between Node.js and Asyncio is the prevalence of callbacks in Node.js, which are parameters passed to functions that are called at the completion of the function to which they are passed [Nodc]. These enable programmers to call other methods or operations that should be placed on the event queue once the current function finishes executing. In

contrast, asyncio does not rely heavily on callbacks, if at all. Asyncio instead repeatedly adds events to the event loop and asynchronously executes them, waiting for prerequisite coroutines to finish before the "await" calls in other coroutines return.

Overall, the approaches of Asyncio and Node.js are relatively similar in the sense that both use event loops to coordinate single-threaded asynchronous execution, but they differ in the sense that Node.js has slightly different methods and constructs that differ from the Asyncio control flow.

4.6 Pros and Cons of Asyncio

A major pro of using Asyncio is the ease of development. The object-oriented nature of the library and Python language in general makes it easy to set up a server with an event loop and create TCP connections with automatic input and output stream generation. This simplicity leads to more readable, bug-free code.

Another advantage is memory management. Comparative to other languages (e.g. Java), Python's garbage collector is relatively efficient and produces homogenous memory access and update times due to its reference count deallocation procedure. This behind-the-scenes nature of the garbage collector contributes to the simplicity of the server implementation, and it reduces the probability of memory leaks that might occur in other languages.

A final advantage is performance. Although Python is single-threaded, the application server herd structure enables the creation of multiple servers with independent processors and event loops that can operate in parallel. This mitigates the need for thread-based parallelism, providing the benefits of asynchronous execution while balancing the computational costs of client requests across servers.

The single-thread nature of Python could also be considered a con. Without thread-based parallelism, individual servers cannot perform operations in parallel. However, this is relatively unimportant because our server performs I/O intensive, high-latency operations, which means that much of the execution time of threads would be spent waiting or spinning anyways. Furthermore, the more servers we add to the herd, the less we require thread-based parallelism because individual servers can run in parallel to one another.

Another con is the fact that Asyncio operates through cooperative multitasking [Mon21]. If an operation or coroutine is computationally intense and never yields, then the server will remain hung on that operation and will starve the other requests. Thus, special attention must be paid to avoid server preoccupation on non-yielding tasks.

A final con is that the execution order of asynchronous tasks may be nondeterministic. Because the server tasks are I/O intensive and coroutines are scheduled according to an event loop, coroutines might not be executed in a predefined or predictable order. The order depends on the variable latency of awaited operations. This might result in coroutine starvation if some requests do not get to execute in a timely manner. This problem should be mitigated as the number of servers scales up and the load of requests is balanced across the server network.

5 Recommendations and Conclusions

Asyncio is a suitable and convenient framework to implement application server herds. The Python language supports simple, readable programs, and the asynchronous nature of the asyncio event loop is ideal for high-latency operations that yield frequently. Although Python's dynamic type checking is convenient for development, it requires extensive application testing before launching the servers to ensure that unanticipated runtime errors do not occur. Lack of parallelism can easily be solved by scaling up the number of servers in the server herd, which is a relatively easy task from a software perspective when asyncio is used. The main drawback of Python is that the language itself has slower runtime performance compared to other languages (e.g. C) due to its semi-interpreted nature and behind-the-scenes computational complexity (e.g. from garbage collection). These drawbacks can be mitigated by scaling up the number of servers and balancing the network request load across all servers in the herd. Overall, Asyncio makes it easy to implement server herds, and its major drawbacks can be remedied by scaling up server count and optimizing load balance.

6 Acknowledgments

Thank you to Professor Eggert and the amazing TA team for putting together this project and helping us complete it!

References

[Egg21] Paul Eggert. *CS 131 Week 9 Wednesday Lecture*. May 2021.

[Mon21] Amit Mondal. *CS 131 Week 9 Discussion*. May 2021.

[Egg] Paul Eggert. *Project. Proxy herd with asyncio*. URL: https://web.cs.ucla.edu/classes/spring21/cs131/hw/pr.html.

[Noda] Node.js. *Introduction to Node.js*. URL: https://nodejs.dev/learn.

[Nodb] Node.js. *Modern Asynchronous JavaScript with Async and Await*. URL: https://nodejs.dev/learn/modern-asynchronous-javascript-with-async-and-await.

[Nodc] Node.js. *What are Callbacks?* URL: https://nodejs.org/en/knowledge/getting-started/control-flow/what-are-callbacks/.

[Pyta] Python. *Asyncio - Asynchronous I/O*. URL: https://docs.python.org/3/library/asyncio.html.

[Pytb] Python. *What's New In Python 3.9*. URL: https://docs.python.org/3/whatsnew/3.9.html#asyncio.

[sna] snapLogic. *Python vs Java*. URL: https://www.snaplogic.com/glossary/python-vs-java.

[Van] Alexander VanTol. *Memory Management in Python*. URL: https://realpython.com/python-memory-management/.