CS131 'Proxy Herd with asyncio' Project Report

Brendan Rossmango
Discussion 1C

Abstract

In this report, we investigate the pros and cons of using Python's asyncio as a framework for an application server herd network, by comparing a Python-based approach to a Java-based approach, and by comparing asyncio to Node.js. In doing so, we will recommend using asyncio for I/O-intensive programs in Python, due to its relative ease and benefits in handling concurrent tasks.

1. Introduction

To create a server platform, different architectures and approaches are used. One traditional approach is to simply create multiple repetitive web servers as the server platform for the clients to use. Wikipedia, the free encyclopedia, and other related websites use this approach; referencing the project specification and the Wikipedia Infrastructure page, Wikipedia is based on Debian GNU/Linux, the Apache web server, the Memcached memory object cache, the MariaDB database, the Elasticsearch search engine, the Swift media object store, and core code written in PHP+JavaScript, which all run on multiple web servers behind load-balancing software on the Linux Virtual Server. This platform also has two levels of caching proxy servers, Varnish and Apache, that route user requests through Wikipedia's infrastructure levels and increase reliability and performance.

This approach works well for Wikipedia mainly because updates to the vast number of Wikipedia articles are few and far in between; since only a very small amount of Wikipedia articles are heavily visited, a very small number of the articles are frequently updated. This is also true for the various related Wiki pages for other fandoms and communities, as most of these pages contain very little information and are flagged for being stubs. In general, Wikipedia pages are not used for news updates, but instead of used as archives for past information. However, for a Wikimedia-style service designed for frequent news updates, this approach will falter.

Such a service designed for much more frequent news updates requires a different architecture and approach for the server platform, since updates to articles will happen much more frequently, access will be required via more protocols than just HTTP, and clients are likely to be more mobile. An alternative to the previous approach is needed because the PHP+JavaScript application platform will be a bottleneck, adding new servers for mobile clients will be tedious, and overall, the aforementioned server platform will just be too slow.

This alternative can come in the form of an application server herd, where multiple application servers communicate with each other, the core database, and caches. With this approach, the multiple servers communicate with each other to access rapidly evolving data, instead of being bottlenecked waiting for the central database for small data that needs to be updated quickly; the database server still holds data that is accessed less often. To update the news articles quickly, one application server can receive the update and directly communicate with other servers so they can also receive the update, instead of them having to wait to access the central database. Additionally, servers can propagate messages to their neighboring servers, which then send messages to their own neighboring servers, and so on; even if one server drops connection or the central database is unable to be accessed, the updates will be propagated to multiple different servers.

Python's asynchronous I/O asyncio library is a good candidate to replace the previous Wikipedia server platform with an application server herd. Due to asyncio's event-driven nature, an update can be processed propagated quickly to other servers in the herd, and the asyncio library is relatively easy to use. To examine the pros and cons of Python's asyncio as a potential framework to implement an application server herd, we wrote a parallelizable proxy for the Google Places API using asyncio.

2. Google Places Proxy server herd using asyncio

2.1 Overview

To test whether asyncio is suitable to implement an application server herd, we made a prototype that consists of five servers that bidirectionally communicate with each other. Additionally, we made a client that also uses asyncio to open a connection with the server.

Clients send their locations to the server using an IAMAT message, which includes their client name, latitude-longitude coordinates, and the time sent. The server responds to the client with an AT message, which includes the ID of the server that got the mes-

sage from the client, the elapsed time from the client's time stamp, and the IAMAT message's client name, location, and time sent.

Servers propagate both the IAMAT and AT messages to their neighboring servers, and if a server has not already received a message about the client and location from previous IAMAT/AT messages (either messages that they received from the client or other servers), the server updates its dictionaries about the client's location and timestamp of message.

Clients also can ask the servers about locations of other clients using WHATSAT messages, which include the name of the client, the location, and an upper bound on the number of places the Google Places API should return. The server responds with an AT message of this location and a JSON-format message in the format of Google Places Nearby Search request.

The importance of the application server herd is that servers can respond with Google Places information about locations that a client did not directly query them about, but instead, a neighboring server propagated information about. Additionally, servers continue to operate even if neighboring servers drop connection.

2.2 Initial Evaluation of asyncio

Going strictly from the implementation of the Google Places API proxy using Python's asyncio, asyncio works well in practice and is relatively easy to use. The discussion slides were very helpful in establishing a baseline on how to implement a simple server using asyncio's serve_forever() and run(), and using the await and async keywords to define functions that suspend execution and give control to other coroutines to go along with the asyncio functions was not a challenge. It is easy to read and understand how asyncio implements the server and client just from looking at the code. Additionally, propagating IAMAT and AT messages to other servers in the server herd was easy; all that is needed is to await a coroutine that opens connections to each neighboring server and writes to them.

3. Comparison of a Python-based approach to a Java-based approach

3.1 Python vs. Java overview

In the context of serving as a potential framework for an application server herd, Python and Java's main differences are in type checking, memory management, and multithreading.

3.2 Type checking

Python is dynamically typed, meaning variables are only checked at runtime and there is implied casting, while Java is statically typed, meaning variables are checked at compile time and can only be changed usually with a cast. In Python, variable names are only bound to an object, but in Java, variable names are bound to an object and the type. Both languages are strongly typed (meaning type exceptions will be raised when trying to coerce unrelated types, like trying to concatenate a string and an int; in this case, one must explicitly cast the int to a string).

Dynamic typing allows for more flexibility and makes code easier to write since the programmer does not have to worry about casting variables all the time, like they would if they were to use Java.

For small projects like the proxy Google Places API, it does not matter if one were to use Python over Java or vice versa regarding type checking. The problem in using dynamic languages may arise for maintaining very large projects, where it is harder to read and understand the code, especially since the variable names are only bound to the object, not a type. For the Google Places proxy, there were times when I confused myself on my variables for keeping the timestamps, specifically on if the type bound to the variable name was a float or a string, and this problem would only be magnified in larger projects where variable names do not have the type in their names. While code would be easier to write in Python since variables are flexible, if the code is not well-documented with comments, it would be harder to read, so it may be better to use the statically typed Java to maintain a large amount of code.

3.3 Memory management

The Python memory manager uses a heap for its objects and data structures, whereas in Java, there is no explicit allocation of memory, only the creation of new objects.

The major difference between Python and Java regarding memory management is the garbage collection methods employed. While both automatically reclaim memory that is no longer needed by the program (meaning the programmer does not need to manually free allocated memory like one would in C++), Python uses reference counting to reclaim memory from unneeded objects whereas Java uses an approach called concurrent mark sweep (CMS).

In Python, each time an object is referenced or dereferenced, the object's reference count must be incremented or decremented. Reference counting is very slow (in comparison to CMS) because each time an object is referenced, the interpreter or virtual machine must check if the reference count reaches zero.

The overhead of checking if the count is zero each time is large. Additionally, reference counting is unreliable because of circular references; even if an object is unneeded, it may have a circular reference to itself, which means its reference count is not zero, so it will not be garbage collected and it will waste memory. The pro of reference counting is it takes less memory than Java's CMS.

Java uses a concurrent mark sweep, which checks all reachable code; any unreachable object is marked as dead. The Java Virtual Machine then moves these dead objects to a new part of memory which is swept all at once. Additionally, the concurrent mark sweep can be executed by a separate thread. As a result, the concurrent mark sweep approach used by Java is much faster but also requires more memory.

In short, Python's garbage collection is slower but uses less memory, while Java's garbage collector is much faster but uses more memory.

3.4 Multithreading

One major difference in general between Python and Java is the handling of multithreading. A programmer would not choose Python over Java because of its support of multithreading.

Python uses a Global Interpreter Lock (GIL), which is a lock that can only be held by one thread at a time; this means that no other thread can execute while a thread holds the GIL. However, Python still supports multithreading – instead of running the threads simultaneously, Python frequently switches what thread has control of the GIL. Python uses a GIL mainly because of its garbage collection method – reference counting. Many problems due to race conditions could arise if multiple threads could increment or decrement the reference counts of objects at the same time. A naïve solution to this problem would be to just lock all reference counters of each object, but this would be very inefficient and can lead to deadlock. Python also is implemented using C libraries that are not thread-safe. As a consequence of the GIL, single-threaded code is very fast in Python, but multithreading does not improve the performance of CPU-intensive tasks.

Java, on the other hand, uses the Java Virtual Machine which has large support for multithreading. Java programmers can make use of the synchronized and volatile keywords, Thread objects, the Runnable interface, thread pools and thread pool executors, semaphores, etc. to achieve high performance using true parallelization on CPU-intensive tasks. For the multithreaded gzip compressor in homework 3, it is natural to use Java to imitate the performance of pigz, the multi-threaded gzip compressor written in C, due to its support of multithreading, whereas it is not natural to try to use Python for multithreading.

For the Google Places API proxy, it is okay to use Python instead of Java because this application is very I/O-intensive, not CPU-intensive, due to the large number of writes and reads needed to update and propagate the messages for each server.

3.5 Conclusion

To implement the Google Places proxy server herd, there is no reason to prefer Java over Python or vice versa. Regarding type checking, the project is small enough that Python's flexibility and ease outweigh the possible confusion when it comes to variable names. As for memory management, Python is slower since it must check if the reference count is zero each time an object is referenced, but it uses less memory because its approach does not mark all dead objects and place them in a new chunk of memory to sweep like Java's concurrent mark sweep approach does. Finally, for multithreading, the benefits of Java's possibility for true parallelization are not necessarily useful because true parallelization only leads to high performance for CPU-intensive tasks, not I/O-intensive tasks, and the proxy server herd requires a high number of I/O requests as each server receives messages to read and writes updates to each server it propagates the message to. So, it is okay to use Python here too, as Python's single-threaded code is very fast.

For a much larger project application server herd, it might be better to use Java over Python, because Java is much faster at garbage collection at the cost of more memory.

4. Comparison of asyncio to Node.js

4.1 Overview of asyncio

Python's asynchronous I/O library asyncio is a single-threaded approach to concurrent programming. It uses the keywords async and await to define coroutines and establish cooperative multitasking, where the coroutines suspend their own execution to give control to another coroutine. The async keyword defines a coroutine which is a function that can suspend its own execution to give control to the awaited function (which uses the keyword await) until the awaited function finishes. It is best to use asyncio in programs where many tasks are needed to be done while waiting for an I/O operation to finish executing. Python's asyncio uses an event loop that runs tasks that are waiting to be executed.

4.2 Comparison of asyncio to Node.js

Like asyncio, Node.js is designed to be event-driven and asynchronous. It is a single-threaded JavaScript

runtime designed to build scalable network applications. Since networking with multiple threads is inefficient and difficult to use due to the I/O-intensive operations, Node.js instead uses a single thread to do concurrent tasks just like asyncio. Since there are no locks in Node.js, there is no deadlock, also like asyncio. Additionally, like asyncio, Node.js uses the keywords async and await.

Node.js uses callbacks to implement concurrency; when a task finishes, another function in the callback can run; this is in a similar manner to how asyncio's coroutines suspend execution of async functions until the awaited function is completed.

Almost all functions in Node.js do not directly use I/O, so the program never blocks. This is a difference from asyncio, since asyncio has a function called sleep, which blocks, and other functions that a Node.js user would have to code themselves.

Another difference between asyncio and Node.js is that Node.js explicitly makes use of an object called Promise. Async functions in Node.js are just syntactic sugar for Promises, which are proxies for values that will eventually become available in the future. So Node.js uses await when calling functions that return Promises to wait for their return value. Once a promise has been called, the calling function continues executing while the promise is pending until it resolves or rejects. Promises are different than asyncio's Future objects; when a Future object is awaited, the coroutine will wait until the Future is resolved in some other place.

4.3 asyncio-based programs for server herds

To implement a framework for an application server herd, using asyncio is easy. Asyncio can be used to write TCP clients and servers using the start_server() function, which takes the handler function that returns a reader and writer, the hostname, and the port number as arguments. Then, asyncio has a function called serve_forever() which allows the program to endlessly receive and serve messages from the client, until the server is ended with a keyboard interrupt, for example. The handler function can parse messages it receives and send them to the other servers using an awaited function. The handler function is an async function, and after it finishes parsing the message, depending on if it is an IAMAT, AT, or WHATSAT message, it suspends its execution and gives control to an awaited function, which propagates AT messages to the other neighboring servers, if the message is an IAMAT or AT, or to the Google Places API awaited function, which uses the API to find the places in the format of a Google Places Nearby Search Result. Additionally, each I/O operation (each read and write) is an awaited function, so the handler function that receives (reads), parses, and sends (writes) messages can do these tasks concurrently.

4.4 asyncio performance evaluation

It is best to use asyncio for programs that are mainly I/O-intensive, since asyncio thrives in allowing the program to execute tasks concurrently instead of waiting for long I/O operations to finish; using asyncio to implement an application server herd is a good idea over not using any asynchronous functions at all. Because a simple server herd does not use CPU-intensive functions, then there are hardly any performance issues with using asyncio, other than the performance issues one would face when using Python, due to its slow garbage collector, but this is not the fault of asyncio. Using Python's asyncio over, say, a Java-based approach or Node.js, would only be a bad idea if the network application requires CPU-intensive operations, since Python multithreading does not improve the performance of CPU-intensive tasks.

4.5 Python 3.9 asyncio features

Heavy reliance on asyncio features from Python 3.9 or later is not needed. A change comes to asyncio.run() which now uses the new coroutine shutdown_default_executor(), which schedules a shutdown for the default executor that waits on the ThreadPoolExecutor to finish closing. Another change is that the to_thread() coroutine was added; this coroutine is used to run IO-bound functions in a separate thread to avoid blocking the event loop.

Other very small changes include a change to wait_for(), which now waits for cancellation in the case when timeout is less than or equal to 0, and removing the reuse_address parameter in a function due to security concerns.

None of these changes are significant; it is easy to get by with older versions of Python.

5. Conclusion

Due to the relative ease and plentiful benefits of using the Python asyncio library for network applications, specifically application server herds, it is recommended to use asyncio as a framework for network applications on Python. Using asyncio, a network application can run multiple tasks concurrently while awaiting I/O operations, so single-threaded code in Python can run with very high performance. In comparison to other approaches in other languages, Java has a more efficient garbage collection system and has much better support for multithreading, but for an I/O-intensive application like a network, Java is not a good idea to use since thread-based networking is difficult and ineffi-

cient. Asyncio, on the other hand, is perfect to use, since Python is flexible and asyncio makes it very easy to start a server herd, receive and send messages to neighboring servers, and start a client that opens a connection to the server.

6. References

[1] Python Standard Library. *asyncio – Asynchronous I/O* documentation. https://docs.python.org/3/library/asyncio.html

[2] 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/

[3] Brad Solomon, *Async IO in Python: A Complete Walkthrough*. https://realpython.com/async-io-python/

[4] Serdar Yelgulalp. *How to use asyncio in Python*. 2020. https://www.infoworld.com/article/3526429/how-to-use-asyncio-in-python.html

[5] Wikipedia. *Wikipedia infrastructure*.

https://wikitech.wikimedia.org/wiki/Wikimedia_infrastructure

[6] Some Dude Says. *Static vs. Dynamic Typing*. 2019. https://medium.com/swlh/static-typing-vs-dynamic-typing-83f0d8b82ef#:~:text=Static%20typing%20refers%20to%20when,true%20for%20compiled%20languages%20either.

[7] Node.js Dev. *Understanding JavaScript Promises*.

https://nodejs.dev/learn/understanding-javascript-promises