Proxy Herd with Python's `asyncio` library

Derek Vance
UCLA CS 131 Project Winter 2021

Abstract

Applications doing any significant amount of IO, especially network IO, face serious performance degradation without supporting concurrent execution through asynchronous IO operations. This project looks at the development and performance of a concurrent proxy herd application that implements asynchronicity using Python's asyncio library.

1. Introduction

The problem that we were initially asked to consider was one based on a Wikimedia-style service for news. In this service, article updates happen frequently, accesses must be supported for various protocols, not just HTTP and HTTPS, and adding servers to improve service to mobile clients should be straightforward. The system used by Wikipedia, the Wikimedia server platform, seems ill-suited for this task. The PHP and Javascript application server is a bottleneck for this design and won't scale well with the additional servers and increasing numbers of updates.

Instead, the solution explored here will use an application server herd. Many distributed servers will be made available to clients, and each server can establish connections with a subset of the servers in the herd. For rapidly evolving data like is required by this service, the servers in the herd will communicate new data to each other. The more stable data can still be accessed in the central database. This way, we decrease the traffic to the central database, improving response times and server redundancy. Python's ability to support this type of application server herd is further explored.

2. Techniques

Python is an interpreted, garbage-collected and dynamically-typed language. Many of Python's techniques for implementing parallelism are ill-suited for the task of developing a concurrency-enabled proxy herd. Python does support multiprocessing, although for this application the overhead of interprocess communication between the network operations and the main execution would be too costly from a performance perspective and would be inefficient in terms of computing resource utilization. It's support for multithreaded applications is inadequate for this task as well because of Python's global interpreter lock, a safeguard which prevents multiple threads from executing the same Python bytecodes at once. This prevents a multi-threaded program from utilizing more than one CPU core at once, negating one of the most attractive features of multithreaded programs, while also introducing the development hurdles that such programs bring. Instead, we will utilize Python's asyncio library and related async and await keywords to implement a single-threaded concurrent server program.

2.1. Python's Support for Concurrency

Rather than multiprocessing and multithreading, the approach we can best exploit for this problem is single-threaded concurrency. Such programs can execute parts of the program out of order without affecting the final result, all on a single thread and without the synchronization headaches of multithreaded programming. Python supports this program design very effectively through the asyncio library and the async/await keywords.

The async keyword is used to define coroutine objects. Coroutine objects are similar to normal functions but themselves cannot be called by themselves. Instead, coroutine objects must be *awaited* or scheduled by certain asyncio functions before being run by the event loop. The function asyncio.run(coroutine) sets up an event loop, runs the coroutine argument, and closes the event loop afterwards, thus acting as a main entry point for the concurrent program. The await keyword suspends the execution of the awaited object and lets the program know that it's ok to do some other task on the event loop before finishing the awaited object. This is perfect for long latency operations like network IO, which this proxy herd application uses frequently. The python interpreter can initialize the network IO and do some other task while waiting for its result and then return to the original task eventually once the network IO is complete.

The library specifies a number of ways to put new tasks onto the event loop, the most pertinent to this application being the asyncio.gather(*awaitables) function, which schedules each awaitable coroutine as a task. This is a key function to ensuring concurrency. We can't simply loop synchronously through the awaitables and use await on each. Doing this, the program would be waiting for the first

coroutine to finish, painfully slow network IO and all, before the second coroutine could even start its own network IO, and on and on, for example. Instead, by using asyncio.gather, each coroutine will be scheduled as tasks on the event loop. When the first task first awaits the result of its network IO, it will give up control to the event loop, which then runs the second task, which also can start its network IO before giving up control to the event loop to another task that can do useful work. This way, multiple network IO operations, which are the largest bottleneck in the program, can be occurring concurrently.

Lastly, asyncio supports a number of coroutines that support networking. The function asyncio.open_connection opens a TCP connection to an IP address and port, and the asyncio.start_server function starts a server that accepts client connections and handles their messages using an asynchronous coroutine.

3. Prototype

To test the techniques outlined above in a prototype a similar but simpler problem to the introduction was devised. Five servers named Riley, Jaquez, Juzang, Bernard, and Campbell make up the server herd, the central database is omitted, so the servers only have to respond to clients and send location data between themselves, and each server keeps a cache of clients it knows the location for. Additionally, the server herd has a topology such that Juzang, Bernard and Jaquez are connected to each other, Riley and Jaquez are connected, Riley is also connected to Juzang, and Jaquez is also connected to Bernard. The connections are bidirectional. Thus, it should take a maximum of three intraserver transmissions for a server to propagate a message to the rest of the servers in the herd.

3.1. Client Messages

Each server needs to handle two types of client commands. The first, IAMAT, lets a server know the client's name, location in coordinates, and the time it sent the command. The server responds with an AT message that includes the same information as the IAMAT command as well as the difference between when the server received the message and the time the client sent the message, as well as the server's name the client connected to. The server must then save the client's location data and propagate this information to its neighbors in the server herd.

The second command, WHATSAT, also includes a client name, a radius in kilometers, and the number of results the response is limited to. The server that receives this command consults its cache to determine the location of the client and the AT message received by that client after their most recent IAMAT command. It then makes an HTTPS get request to the Google Places API Nearby Search endpoint, using the command radius as a parameter, and responds to the client with the json returned by the API, limited by the number results specified in the command, appended to the AT message.

If a received client message doesn't conform to these command formats, the message, appended to a single '?' is sent back to the client. Additionally, if a server goes down, messages should still be able to propagate to the rest of the servers in the herd, so long as the connection topology allows it.

3.2. Implementation

To implement the above specification I wrote an asynchronous server program called server.py. It can be called on the command line like:

```
$ python3 server.py <server_name>
```

This enables the named server to handle client commands. Every network IO operation is awaited to allow for concurrent execution. The server's internal cache of client location information is implemented in a Python dictionary. When a server receives a IAMAT message, it needs to propagate the information to the rest of the servers without incurring a broadcast storm, so it needs an efficient flooding algorithm. To do so, the server can send a slightly modified AT message to each of its neighbors. When a server receives an AT message, it checks if it is newer information than is stored in its internal cache, and if so propagates the message to all of its neighbors except for the server it got the message from. It does so using the asyncio.gather function to schedule each of the propagation network IO's concurrently.

To handle WHATSAT messages, the asynchronous-enabled Python library aiohttp is used. The network IO to the Google Places API is awaited to allow other tasks to run in the meantime, which exploits the concurrency enabled by asyncio.

Each server's input and output is logged synchronously. The asyncio library itself doesn't have asynchronous support for file operations, but compared to the IO latency associated with the network, file system IO is far less.

4. Results

A metric for this implementation's performance is the field returned in the AT message, or the difference between when the client sent the message and when the server received the message. By itself, this value consists mostly of network latency, but if multiple messages are sent in rapid succession, then this value should not vary much because of the benefits of concurrent execution. With synchronous execution, we would expect the second AT message to have a difference roughly twice as large as the first, because it experiences the latency of both the first message, since the program would block during this time, and its own message.

However, these difference measurements stay at roughly 300 milliseconds during testing.

The server also is able to remain operational even after one server goes down. When Juzang was put offline, an IAMAT message from Riley was successfully propagated to Campbell, requiring three transmission hops.

5. Analysis

Developing a concurrent application in Python using asyncio as done here was relatively straightforward and pain free. The async and await keywords were very convenient and frequently used. One keyword allowed a function to cede control to the event loop, something that with a multithreaded application coded in Java, say, would have had to be manually done using some type of queuing mechanism. Also, the one line asyncio.gather function was very convenient for scheduling multiple tasks concurrently onto the event loop. My only gripe with the asyncio library is that it doesn't have functions for file system IO.

Python's dynamic type checking was both a blessing and a curse. The curse of it is that you'll only be notified of type errors at runtime, and only if that execution reached the error in question. I dealt with this by using frequent try catch exception handling blocks. In the end, it wasn't a huge problem because most executions of the program used most of the code base, so I could be aware of issues in the same amount of time it would take to compile a statically-checked program. The blessing was the ease of writing code. I could easily compose lists and tuples of different types, change the types of variables as needed, and make use of the flexibility of the dictionary type, things that in Java would've taken lots of extra boring code. Dynamic type checking has performance implications, however, and may incur longer response times than the statically-typed Java.

Python's reference counting method of garbage-collection was very convenient as well. It was well suited to this application since no circular references were used, which is a big bane to reference counting algorithms. This is part of the reason why Python's support for multithreaded applications is weak, however, since reference counting across threads will experience way too many race conditions to ensure data consistency. This problem is papered over by the global interpreter lock, which is a solution that limits the multithreaded programming capabilities that Java, which uses the mark and sweep garbage collection algorithm instead, does support.

5.1. Comparison to Node.js

Node.js is a javascript runtime that is well tailored to the type of problem we solved in Python. For starters, the runtime is itself a single-threaded event loop. It also natively supports asynchronous functions that can be awaited. It does so using Promise's, or, abstractly, values that are promised to be returned but haven't yet. It encourages a programming style that uses callbacks, or functions that should run after a specified operation has finished. So instead of awaiting a network IO read, then handling the input later on in the same function, Node.js is more explicit in how it articulates asynchronous code. When you look at a function call that includes a callback, it's clear that an asynchronous function is going to be called, and when it's done, this callback will handle the result. I noticed that I wrote my Python code in a similar type of way, which may be an indication that the way Node.js supports asynchronous code is more intuitive.

Node.js is quite similar to Python in that it is dynamically type-checked and interpreted, which results in slower CPU execution than a statically type-checked and compiled language, like Java. However, for this application, where the latencies have to do with network IO and not heavy CPU execution, both can mask this performance shortcoming by exploiting concurrent programming, rendering these shortcomings negligible while also benefiting from the ease of development.

5.2. New and Other Python Features

The new asyncio function asyncio.run was very useful, as it gave me an easy high level abstraction of the lower-level event loop initialize and close operations. The

```
$ python3 -m asyncio
```

command was useful as well for getting the hang of the async and await keywords and the useful asyncio functions. It's not necessary but does reduce the barriers to getting up and running and trying out asynchronous Python code in a REPL.

A feature that would be interesting to try would be the type annotations that newer versions of Python support, in conjunction with a static type-checker for Python like mypy. This would be a convenient blend of static and dynamic type checking, since type annotations could be optionally given to certain vulnerable sections of code and checked by mypy statically, before execution. This wouldn't increase performance, since the interpreter would still do dynamic type checking, but might enhance developer experience by allowing earlier and more reliable discoveries of type errors.

6. Conclusion

Python's asyncio library was well suited to create single-threaded concurrent programs. By using the asyncio event loop and the Python async and await keywords, asynchronous operations were easily offloaded to the event loop while their long latency network IO operations were handled. This allowed a single server thread to handle multiple client messages concurrently. Messages could be propagated to their neighbors in an asynchronous fashion using the asyncio.gather function, again allowing a single

server thread to send multiple messages concurrently. Furthermore, it was very easy to make asynchronous HTTPS get requests to the Google Places API using these same methods. The performance drawbacks of using a slower language like Python are irrelevant given the high network IO latencies that are the bigger bottleneck for an application like this and do not eclipse the benefits Python and its asyncio library give in terms of ease of development.

7. References

https://wiki.python.org/moin/GlobalInterpreterLock

https://realpython.com/python-gil/

https://docs.python.org/3/library/asyncio-task.html