Proxy Herd Server

Jennie Zheng - UCLA
November 30, 2017

Abstract

The async event-driven networking library works as an adequate but imperfect candidate for the Wikimedia platform code. The asyncio library is difficult to grasp, but once grasped, it provides a solid foundation for asynchronous programming in Python 3.4 and above. It allows a single kernel thread to control multiple user threads without explicit locks or mutexes. One downside is that Python is a dynamically typed, interpreted language, which means it's harder to develop large applications in and faces slower execution speeds. Python's Asyncio is a great language for prototypes, but not for large scale products.

1. Parallelism

Traditionally, parallelism can be accomplished with threads. Threads, however, suffer from slow-downs upon context switches due to kernel overheard and uncontrolled switching with may cause deadlocks. Another way of solving synchronization issues is to use user-space threads with cooperative yielding (In other words, the threads themselves decide when to yield to other threads). The Asyncio library is intended to make parallelism easier. It uses a single event loop with multiple coroutines which each yield when appropriate.

2. Prototype

a. Prototype Basics

In order to research Asyncio, I developed a prototype application with five servers communicating with each other, with clients, and with a Google Places API service. Note that since each server must connect, read, and write to multiple sockets, parallelism is a must. The servers may receive IAMAT messages from clients, and the server must respond by acknowledging to the client that it received the message, storing the location of the client, and flooding the message to the servers which it communicates to. Furthermore, servers may receive WHATSAT messages from clients, upon which the server should refer to its client locations to find out where the target client is located, call the Google Places API, and retrieve a list of attractions near the target client. Lastly, servers may receive SERVERMSGAT messages, which servers respond to by storing the message and flooding it if they haven't received the message already.

Note that a major problem with distributed systems is that they may not all agree about a particular piece of data. The prototype uses a simplistic solution. If a single client sends different data to multiple servers, the servers will keep the data with the latest client-time. If different pieces of data have the same client-time, the servers will each keep the data they obtained first.

b. Prototype Structure

The prototype is structure as a ProxyHerdServer class with multiple functions within it. The __init__ function requires a server number (0-4) as well as an event loop to initialize the class. Next, the start method, which must be called externally, will start a server with asyncio.start_server on the given event loop. From there, the main, asynchronous function, handle_message, will receive all messages, from clients and servers, and then handle them appropriately.

If a server needs to communicate to another server, it will try to simply write to the writer. If that fails, the server knows that the connection has been lost. Then, it will run asyncio.run_coroutine_threadsafe with the function connect_to_other_server in order to connect to the other server asynchronously. In addition, if the server needs to call Google API, it will call asyncio.open_connection asynchronous.

The flooding works with a dictionary of clients and their last reported locations. If a server receives a SERVERMSGAT message with a client-time later than the client's client-time stored into its dictionary, the server will overwrite the old time with the new one and flood the message to the other serves which is communicates with. Alternatively, if the SERVERMSGAT message has a client-time earlier than the other it stored, the server will simply ignore that time. This prevents slow servers with earlier client-times from overwriting later client-times.

One benefit of Python is that Python's massive library makes string parsing quite easy. For example, with re allows regex string parsing and json allows json package parsing, so taking a Google Places response and parsing it to include only infoBound amount of items is simply five lines:

```
lines=lines.decode()
lines=lines[lines.find('{'):]
d=json.loads(lines)
d['results']=d['results'][:infoBound]
result=json.dumps(d,indent=1)
```

3. Major Difficulties

a. Connecting Between Servers

I spent half a dozen hours trying to set up a lasting connection between the two servers. In the process I ran into numerous problems including small syntax errors which took me half an hour to fix and larger logic errors which took me a few hours to fix.

Initially, my plan was for each server to setup a connection with the other servers it communicated to. That plan, however, didn't work because once A connected to B, B hanged and was unable to connect to A. Thus, I modified my plan so that only a single server was responsible for setting up the connection: once A connected to B, it sent B a message saying that A was a server. Then, both A and B saved their corresponding writers in a dictionary in order to speak when they wanted to.

This setup still didn't work, though, and it took me a long time to realize that B's messages were not going to the default message_handler because they were going to the reader which A received when it connected to B. Thus, I needed A to read from the particular reader. After surpassing these obstacles, I managed to set up a stable connection between the servers.

B. Connecting to Google Places

Since I was not familiar with http, I had trouble figuring out how I could create my own http request to access the Google Places API. I spent quite a long time researching online before I learned that I needed to open a connection to the google places url at port 443, and then submit a GET request with a query that following a very specific format.

4. Complex Asyncio Structure

Personally I had a lot of difficulty understanding Asyncio's logic. Asyncio felt quite intimidating because I needed to know so much to even get started. For a long time, I just experimented with async, await, event loops, and coroutines, without understanding how everything fit together. To understand Asyncio, a programmer must first understand event loops, coroutines, and tasks. The first major idea in Asyncio programming is loops. Event loops start the execution of everything else. An event loop is basically a call scheduler which can add calls to its queue, execute calls, or cancel calls. The second major idea in Asyncio is coroutines. Coroutines resemble generators in that they aren't automatically executed. Instead, coroutines will only execute when they are registered in the event loop and then executed by the event loop. If the coroutines hits an 'await', it will yield to the next event in the event loop until the awaited task is complete. Coroutines can await for other coroutines or futures in order to chain together asynchronous programming. Finally, the last major idea in Asyncio are futures. These are what the event loop plans to accomplish. Futures can be registered or canceled. For example, ensure_future(coroutine) will add the future event to the event loop. Futures is vital for non-blocking programming in Asyncio. These three ideas are only the very basic three for Asyncio programming. To truly master Asyncio programming requires a good grasp of not just the above ideas but also Streams, Transports, Protocols, and many other ideas. This means that Asyncio has a high learning curve and is not for programmers looking for a simple solution.

5. Too Many Variations

Since asyncio is a relatively new library, it has many different ways to accomplish nearly the same thing. I found this quite bothersome because it meant I learn multiple different syntaxes instead of just one. For example, the following two syntaxes both wrap a function into a coroutine, and they accomplish the same thing in 99% of scenarios.

```
@asyncio.coroutine
def methodA(x):
yield from a

async def methodB(x):
await b
```

They, however, behave differently when an attempt is made to iterate through the object they return. Furthermore, methodA is only available in Python3.5 and above, while methodB is available in the earlier versions of Python. Anyone programming in asyncio is bound to come across both syntax styles. This unnecessary variation and as well others make Asyncio difficult to learn. In addition, Asyncio seems to conflict with Python's original goal to give programmers exactly one way, the best way, to accomplish everything.

6. Python vs Java

Both Python and Java support asynchronous execution, but Java's concurrency libraries are a lot more mature than Python's. This means it's easier to find programmers with experience with Java's concurrency libraries.

In terms of speed, neither Python nor Java are supremely fast, but Java is able to take advantage of machine optimizations due to the JVM. Thus, Java is executed slightly faster than Python.

Furthermore, since Python uses type inference and duck typing while Java uses static type checking, it's faster to prototype projects in Python, and more efficient to develop last scale projects in Java.

Overall, I would recommend Java, unless if the project is simply a small prototype.

7. Asyncio vs Node.js

Node is an asynchronous event driven JavaScript runtime. From my experience with developing Node applications, I feel that Node focuses on polling and callbacks, but does not have as much of the advanced asynchronous features which Asyncio has.

Node is less powerful than Asyncio, but also significantly simpler than Asyncio specifically because it has fewer features and thus fewer ideas to learn. Developers will have an easier time learning Node than learning Asyncio.

While Node.js is easier to learn, it's harder to debug. Node seems to abstract much of the lower level ideas, making it difficult for debuggers to pinpoint the underlying problem. Thus, I've spent a lot of time trying to figure out minor typos in javascript which browser debuggers failed to catch. In that sense, it will be code to create large scale projects in Asyncio than in Node.

Lastly, Node.js is in javascript, so it can make use of amazing front end libraries including Angular and React for web development. Asyncio is in Python, and Python's front end libraries are less spectacular. Thus, if the project has a large UI aspect, Node.js is the solution. In general, I recommend Asyncio over Node.js, unless if the project is mostly front-end.

8. Conclusion

In conclusion, the async event-driven networking library works as an adequate candidate for the Wikimedia platform code. Although the asyncio library is difficult to grasp, it provides a solid foundation for asynchronous programming in Python 3.4 and above. One downside is that Asyncio runs on Python, a dynamically typed, interpreted language. This means it's harder to develop large applications in and faces slower execution speeds. Python's Asyncio is a great language for prototypes, but not for large scale products.