CS 131 Asyncio Proxy Server Herd Project Fall 2017

Artiom Arutiunov – University of California, Los Angeles

Abstract

The purpose of this assignment is to investigate whether the asyncio event-driven networking library is a good option for exploiting server herds. This report primarily focuses on the specific characteristics of asyncio, namely performance, memory management, type inference, and multithreading versus asynchronous behavior. After evaluating the benefits and drawbacks of asyncio, I conclude that is a good choice for designing a server that is lightweight and efficient.

Introduction

Asyncio is a Python library created to write asynchronous applications. The premise of the project was to determine whether asyncio is a good substitute for the LAMP architecture. The task of the project was to create a server that would listen to a client's requests, and upon the "IAMAT" request, propagate information about that client to other servers. Moreover, the server would make a GET request to the Google places API and send JSON to the client about their location. I worked on the project in Python 3.6.3, and used the asyncio, aiohhtp, and the async_timeout modules to create the server.

Design

Overview

I decided to modularize the project into three parts. The first file, "configuration.py", simply holds a dictionary of the two relationships in the project: First, a dictionary of server name keys associated with the names of the two other servers the key value server is allowed to talk to. The other dictionary is one that associates each server name with the port number provided to me by the TA. The second file, "client.py" is a class file containing a TCP echo client protocol skeleton. The file also has a main routine that creates an object of the client class and sends the corresponding "AT" request to the other servers. The last file, "server.py", holds all of my server code and logic. First, I imported modules needed for the project, including prewritten ones such as "json", "logging", "asyncio", and "sys." I also imported the other aforementioned modules. Next, I initialized all the global variables needed for the server, such as a dictionary, the server's name, and the URL needed to make the GET request. Next, I designed and implemented my server class, which is based off the TCP echo server skeleton. I added many new methods to it, including "parseJSON", "updateClientLocation", and "receiveInfoOtherServer" which I would use within the server design to execute the different API calls from the client. I also included a main routine that actually creates the server and keeps it running.

Server Flooding and Herding

My approach for the server herding was relatively simple. The way I differentiated between a client talking to a server and a server talking to a server is the type of request. Specifically, I programmed a server to handle an "IAMAT" request differently than an "AT." The server would call different methods depending on which request was made. When a server received an "IAMAT", it refers to the dictionary to find which servers to talk to. I then called the client module's main routine, which sent the message to other servers. Then, when a server receives an "AT" request, it checks its data to see whether the client's location is the same as what is passed in the "AT" request. If it is, then the server does not propagate to other servers. If the server's location is outdated, then it updates its own data and propagates. This ensured that propagation between the servers only happened once, with every server getting updated with a client's newest information only once. I wrapped up the whole process of propagating in an "asyncio.ensure_future" method. This assured that all of the servers would communicate with each other, and that the task would be added to the global loop each server has.

Google Places GET Request

I had a similar approach for the GET request as server flooding. I created three methods in my server class – "fetch", "parseJSON", and "formatLocation" – that were responsible for making the request. Specifically, after receiving a "WHATSAT" request from the client and checking to make sure it is valid, the server would call the "parseJSON" function. The "parseJSON" function would format the client's corresponding location via "formatLocation", and then make the get request using "fetch." Finally, "parseJSON" would format the returned JSON body and transport it to the client. As with server flooding, I wrapped the "parseJSON" method call in an "async_ensure_future" method.

Asyncio As a Server Framework

Type Checking

A couple of facets of Python that are important in evaluating asyncio are that Python is a dynamic and strongly typed language. Being a dynamically typed language entails Python binding a variable name only to an object and not to a type. This means names are bound to objects at execution via assignment statements. It is important to note that one can bind objects of different types to the same variable name during execution. This could make debugging code more difficult and make it more difficult to follow. In terms of being a strongly typed language, Python does not allow one to perform operations with different types if they are incompatible. In comparison, a weakly typed language will do some behind-the-scenes conversions to make a line of code valid. For example, Python will return a "TypeError" if one tries to do "'Hello' + 10 + 'Goodbye.'" In a weakly typed language, one potential output is "Hello10Goodbye", because strings support concatenation. As a result of the dynamic and strong typing of Python, unit testing is especially important: Bugs that aren't caught by the compiler are caught by unit tests. Ultimately, Python is a great language to create clear code – especially if one is in a hurry and doesn't want to worry about assigning variables types during development. However, I had to make sure that the content of client request was valid, such as having the valid location format. Python does not account for these errors during compile time, and there is therefore a larger burden on the individual to make sure the code is type-safe.

Memory Management

Python utilizes a garbage collector for memory management. A garbage collector is very useful for processing a lot of data, as it is automatically destroyed after leaving scope or program execution. It also makes memory handling on behalf of the programmer easier because garbage collection is done automatically via the Python memory manager. This is an important issue to consider in terms of Python programs interacting with or using libraries written in languages where memory and storage can be manually altered by the program. For example, C functions such as "malloc", "calloc", "realloc", and "free" would results in mixed calls between the C allocator and the Python memory manager. Furthermore, references to memory in Python programs can cause issues, because as soon as an object loses its reference, it is fair game for the garbage collector to delete.

Multithreading versus asynchronous

Asyncio avoids the time and memory penalty of context switching, as is done in multhithreading. Instead, Python uses coroutines – which can be thought of as a lighter version of threads – to handle asynchronous calls. This is significant for server performance: A task that relies on I/O could be devastating for program performance as context switching or locking could cause serious issues. With coroutines, asyncio ensures that data is always handled correctly. Specifically, asyncio creates a coroutine object that is then passed to the loop object to execute. This way, the programmer sees exactly when a certain task will be asynchronously called, which can help to avoid race conditions. One final thing to note about asyncio is the use of futures. A future is essentially an object that represents work that has not yet been completed. This gives event loops the ability to watch future objects or await them as well.

Node.js Versus Asyncio

According to nodejs.org, "is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient." As a result, asyncio and Node.js are actually fairly similar in nature in their attempt to make concurrency a medium for faster performance. One of the major reasons developers find Node.js so attractive is because it is written in Javascript, which is a very popular language to use for web development. Specifically, Node.js makes it very easy to communicate with a backend also written in Javascript called MongoDB. Moreover, Javascript is frequently utilized for frontend website development as well. As a result, Node.js is part of the world-renowned MEAN stack, which consist of MongoDB, Express.js, Angular.js, and Node.js. Moreover, writing a full-stack application in a single language is useful for debugging, clarity, and performance purposes. As a result, most people already familiar with Javascropt would chose to use Nodejs over the Python asyncio library.

Problems

There were several major roadblocks I experienced while working on this project. Having never taken a class on networks, getting a conceptual grasp on the topics relevant to this assignment took some time. After familiarizing myself with web services and understanding what a TCP is, I started to read asyncio documentation. The next major obstacle I faced had to do with looping. Specifically, I kept on trying to create a loop within a loop, which would generate the error that the loop is already running. After researching asyncio some more, I realized that I could use the "async_ensure_future" method to add tasks to a current loop. This resolved any loop error I encountered. Nevertheless, there was a particularly nasty bug I had to deal with over the course of this project: When a server would receive an "IAMAT" request, the servers would all communicate with each other infinitely. This was perplexing because my stopping case was that the server had the most recent data about a client. After a long period of debugging, I realized my issue lay in the fact that every time a client talked to another server, the server's dictionary would be reinitialized. This would consistently create an empty dictionary, and therefore the servers would never reach a stopping point and continue to propagate to other servers. I fixed this issue by taking the dictionary initialization out of the "connection_made" method.

Process

I would spin up a server by running the command "python3 server.py ___", where the underscore represents one of the five server names listed in the instructions. In order to run all five servers concurrently, I opened up five windows in my terminal and started all of them with the aforementioned command. Then, I opened up an additional terminal window to create my original client, which I did via the command "telnet localhost ___", where the underscore represents one of the port numbers of the servers.

Conclusion

Overall, asyncio is a good, albeit not perfect, choice for creating a server that can handle multitasking through asynchronous calls. If a server has to handle a lot of incoming traffic and perform various tasks, asyncio can optimize server speed and responsiveness. Asyncio incorporates a huge amount of tools one can use to optimize server performance, including event loops, awaitables, coroutine functions, tasks, protocols, executors, and streams. It can be a very useful tool for synchronizing code in terms of making sure a value is returned or a task is executed prior to another task execution. It was very useful in terms of making GET requests because the server would not freeze, and would continue to respond to the client while asynchronously fetching JSON data. However, I found asynchio very, very tricky to use. Event loops make the program prone to many errors, including loop overlap, and different coroutines stepping onto each other's toes. You can essentially run only one loop during a program, and have to use other methods such as "await" and "new_event_loop" to make sure the code works correctly. One also has to use the "async" keyword carefully to make sure any asynchronous calls are handled correctly. In retrospect, asyncio is great in terms of capability and performance and not as good in terms of ease of use.

Bibliography

"Interprocess Communication and Networking." *Asyncio — Asynchronous I/O, event loop, coroutines and tasks*, docs.python.org/3/library/asyncio-eventloop.html.

"Logging HOWTO." *Python 2.7.14 documentation*, docs.python.org/2/howto/logging.html.

"Python aiohttp.Request Examples." *Program Creek*, www.programcreek.com/python/example/81226/aiohttp.request.

"Python 3 – An Intro to asyncio." *The Mouse Vs. The Python*, 7ADAD, 2016, www.blog.pythonlibrary.org/2016/07/26/python-3-an-intro-to-asyncio/.

Ronacher, Armin. "I don't understand Python's Asyncio." *I don't understand Python's Asyncio | Armin Ronacher's Thoughts and Writings*, 30 Oct. 2016, lucumr.pocoo.org/2016/10/30/i-dont-understand-asyncio/.

Stinner, Victor. "Why use asyncio?" Asyncio Documentation, 2016, asyncio.readthedocs.io/en/latest/why_asyncio.html.

"Static vs. dynamic typing of programming languages." Python Conquers The Universe, 3 Oct. 2009, pythonconquerstheuniverse.wordpress.com/2009/10/03/static-vs-dynamic-typing-of-programming-languages/

"Why is Python a dynamic language and also a strongly typed language." Python Wiki, wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language.

"18.5. asyncio - Asynchronous I/O, event loop, coroutines and tasks." Asynchronous I/O, event loop, coroutines and tasks, Python Software Foundation , docs.python.org/3/library/asyncio.html