Computer Science 131: Proxy Herd with Asyncio

Matthew Chan
UID: 805291212

Abstract

I was tasked with the assignment to investigate whether Python's asyncio library was fit to improve upon PHP-JavaScript in our new service for news. We utilized this library in the form of an "application server herd", where the servers communicate with each other as well as the core database and cache, which took the form of Google's Places API in this project. Here, the specifics of what we are trying to investigate are how Python's asyncio library fares against Java's Node.js approach.

Introduction

Originally, a Wikimedia style approach was used to implement our service. However, a few problems quickly presented themselves and forced us to seek better alternatives. We initially thought that this paradigm would be a good fit for our service, as updates happen frequently, users access the service from more than just HTTP, and those same users gravitated towards the mobile platform. The problems that quickly arose came from the fact that adding new servers dedicated to the mobile platform presented too much trouble. In addition, the response time gets throttled because of the general implementation of the service. In short, working in this manner is not ideal, causing us to look elsewhere for improvement.

My team is tasked with investigating the application server herd and to note its characteristics and compatibility with the goal we have in mind. This application server herd works by setting up various servers that communicate with each other, updating as they 'flood' each other. The idea is for a device to connect to one of those servers and make an update while that server updates the rest of the network. To achieve this, we chose Python's asyncio library to implement the server herd and Google's Places API as the core service and database for clients. Given Python's nature of single threaded concurrent programming, this coding paradigm should prove safe for us.

Server Implementation

To give Python a thorough test, our team designed a network of five servers with bidirectional communication channels. A client could connect to any of these servers and request the commands 'IAMAT' and 'WHATSAT'. The former simply aims to update the database of the client's ID, coordinates, time of access, and server originally accessed. The latter uses the Google API, requesting information on places in a specified circular area around a client's coordinates. The requester must also limit the search results, no more than 20 places may be given.

I. IAMAT

This command take input from clients in the form of 'IAMAT <client_id> <coordinates> <POSIX time>'. The server will take note of the client's ID and insert it as the key into a dictionary (empty to begin with) with server accessed, response time, as well as a copy of the command's information as the definition. If a client was to put in two valid 'IAMAT' commands with the same ID sequentially, then the one with the later time takes precedence, because how would overwriting new messages with old ones be considered an update?

Furthermore, the addition of a new client then forces an update on the other servers in the herd using a flooding algorithm. A common mistake here would be to implement an infinite loop, which I admittedly did at first; so, to combat this we designed the server to only flood other connected servers on the condition that it updated its client database. This is the only command a user can give that would cause the herd to flood, as the other does not use that feature at all.

There is an apparent weakness here, however. If say, some servers in the herd went down and it was the only server connecting two others in the herd, then any 'IAMAT' requests to those servers would not update the ones disconnected due to the offline servers being unable to propagate the information.

II. WHATSAT

This is the other command users will be able to request for our purposes of testing. This is the piece of code that utilizes the API and sends the information received back to the user. Commands take the form 'WHATSAT <existing_client> <radius> <result_limit>'. Users must make their requests using an existing client ID, otherwise nothing happens. Given a valid command, 'WHATSAT' will return a copy of the original server the client accessed as well as the other parameters used when originally

placing the ‘IAMAT’ request. In addition, the Google API will list out information about various places in the form of a json dump that we present the user with. The result is not very elegant, but it gets the job done. We had to make a developer account using Google’s free trial as well, but that presents no problems unless we were to test the code again months from now.

Python feature investigation

To delve deeper into which coding paradigm we should pick, we must also investigate the features that may present problems: dynamic type checking, memory management, and multithreading.

Type Checking

Python has a dynamic type checker. It is so powerful to the point that you don’t have to place types before declaring variables. Throughout a program, the type of a variable could change a few times and that would be okay. This makes things incredibly simple, though there still exists cases where you have to type cast to compare two variables or when appending. This allows for a more robust behaving program, though there is of course overhead cost to this.

Memory Management

Python’s garbage collector works in the way of a reference count. This is when an object has a zero reference count, the item gets deleted. In other words, when an object cannot be accessed anymore then it goes straight to the trash. As opposed to other languages having to free and malloc space for their objects, Python takes care of it for the developer.

A problem that arises, however, are circular data structures. Under this kind of memory management, those data structures will, unfortunately, not get deleted by the automatic garbage collector.

Multithreading (and asyncio)

Python does not actually support multithreading using its own language. However, you could use python as a proxy to access lower level languages like C++ and access multithreading that way. The fact remains, though, that Python itself does not support multithreading. This allows for much safer code and prevents possible race conditions. Everything is done on a single thread concurrently, not parallelism is involved.

The Python module asyncio is how Python implements asynchronous and concurrent code. It also uses the async/await syntax found throughout our team’s code. The documentation on the website even lists it as a perfect fit for I/O high level structured network code, which is exactly what we aim to achieve. The basis of Python’s asynchronous code is through the use of the event loop and queues. Tasks on the queue execute when the event loop gets to it and yields to another task in the queue when faced with operations like I/O. This allows for safe and fast programs. In short, because of all the above features Python presents, it becomes easy to run and exploit server herds. The developer must first become familiar with the modules available, however, which takes a little time. However, the tools presented to you make it ideal for our kind of project.

Java

On the other hand, there is Java and Node.js. Java features static type checking, and both a different type of memory management as well as multithreading approach. Static type checking makes the program more precise, however compared to Python’s dynamic type checker, the more robust has the advantage simply because of its adaptability.

Furthermore, Java’s memory management differs a bit, as it marks all objects referenced to something and deletes those that aren’t. It’s similar to Python’s methodology, though the implementation is clearly different. Java, however, does not share the same problem Python has with cyclic data structures. Since the garbage collector marks unreachable objects for collection, unreachable circular data structures will still be marked for collection. This should not pose too big of an advantage though since we avoid making circular data structures, though it would still be useful developing other pieces of code.

In addition, Java supports multithreading (this was HW3 lol). However, Node.js is actually single threaded, as per the documentation. It is also asynchronous and event driven, similar to Python’s asyncio. Node.js and asyncio are very comparable indeed, as Node.js offers faster response times but Python features much cleaner code. The bottom line is that Node.js does not work well with CPU intensive applications, which our program consistently deals with.

Conclusion

Node.js and asyncio offer various things for developers. While Node.js is incredibly fast and made for asynchronous activity, asyncio offers ease of access and superior handling of I/O applications. Therefore I would recommend using Python’s asyncio.