CS 131 Project Report

Shunning Ma

Abstract

This report is a summary of my research into the ways of implementation of an application designed for a news system, where updates to articles are much more frequently than a traditional app using the Wikimedia architecture, like Wikipedia. The choice of implementation must avoid the problem of network bottlenecks. The workaround for this is to use a new architecture called application server herd. The main focus in this study is around the suitability of Python's asyncio library on this architecture, including the experience I had in its application into a real project, a proxy for the Google Place API. Its suitability is also compared with a Java implementation and a Node.js implementation. The conclusion is that this library works ideally for a small-scale single-threaded application like this specific project, but a larger project that requires multi-threaded support might need another solution like Java.

1 Introduction

Wikimedia architecture is built with the LAMP-stack, which is based on GNU/Linux, Apache, MySQL, and PHP. It uses multiple, redundant web servers behind a load-balancing virtual router for reliability and performance. This architecture works well with application with relatively static update flow, but this contradicts with the high-volume-update nature of the team's objective news application. Besides frequent updates to the articles, the users are highly mobile and are constantly broadcasting their GPS location. With original design the central bottleneck will be easily reached since every update needs to access the central database server. The team looks into the alternative design—the application server herd—where each of the servers are able to communicate with each other directly, and data that are frequently uploaded to a server will be shared immediately via interserver transmissions. This eliminates the need for high frequency accesses to the central database server and prevents bottleneck problem to a large extent. For implementing these servers, a proposed solution is the asyncio asynchronous library of Python.

2 Overview of asyncio Implementation

2.1 Advantages

asyncio is a comprehensive library designed for writing concurrent code in Python. It comes with both high-level APIs to run coroutines using async/await syntax and low-level APIs that create and manage event loops. The idea of setting up coroutines for each of the tasks are essentially grant the OS the ability to dynamically allocate resources for multiple tasks running at the same time. For example, when one task is waiting on establishing new connections, it's a waste of CPU time to simply make this task spin in the background. To save the resource, the library can make this task yield so other CPU intensive tasks can be granted more resources. This idea of concurrency is well-suited in a scenario with high traffic since it greatly reduces the average time needed to process each of the tasks.

The asyncio library is implemented with built in support of TCP connections by treating each connection as a coroutine object as a callback handled by an event-loop in the low level. Each TCP connection to a client server is essentially an asyncio Task. The library treats multiple connections to different clients as multiple concurrent Tasks and allocates resources among them dynamically to optimize performance. This is particularly helpful in this project since interserver transmissions using TCP protocol are established very frequently, so the performance gain is huge.

The module is also built with high-level enough APIs to facilitate development. In the recently released version, the asyncio library has abstracted out the necessity to create an event loop manually. Creating TCP connections is also straightforward--the user simply need to call the coroutine open_connection. Besides simplicity, the asyncio library is also extendable. To implement further functionalities, the only thing need to do is to prefixing the async

keyword before the declaration of a coroutine function that will do the job. To call the coroutine is also the same as calling a normal procedure in Python except prefixing an await keyword so that the future return value of the coroutine is handled and stored. This simplicity of code is another plus in the implementation of the application since shorter code means shorter time for development, enhanced readability, and ease of debugging.

While the asyncio library only supports limited types of protocols such as TCP and SSL, there are other modules that can make requests of other types and is compatible with asyncio. In the assigned project, the library called aiohttp is used for making HTTP GET requests to the Google Place API to get the JSON format data of places nearby the location represented by the passed longitude and latitude in the request. These two libraries work seamlessly since aiohttp is also asynchronous, so the performance gained from the asyncio library is not hindered. When one task is waiting for a response from the Google API server, the dynamic yield mechanism still works.

The combination of simplicity, extendibility, and powerful concurrency optimization is the main strength of the asyncio library.

2.2 Disadvantages

The asyncio library, as discussed before, is suited for an architecture like the application server herd, but only in a small scale. This is because the way asynchronous code increases program efficiency is run multiple tasks concurrently on a single core. This means that when the network flow is high enough, the problem of bottleneck will still occur once the processor's capacity limit has been reached. The only way to increase efficiency fundamentally is to utilize multiple cores and parallelize the tasks on multiple threads. However, the asyncio library does not work with parallelization. This will definitely put an upper bound on the capacity for the entire system. Thus, this library comes with the limitation that it is only capable of handling a relatively small amount of requests.

2.3 Problems during Implementation

When I implement the server herd for the assigned project, the main challenge I have faced is the flooding part. It is tricky to find an algorithm to flood the information from the incoming IAMAT requests from the client to all other servers that are reachable without causing an infinite loop of flooding messages repeatedly. The way I tackled this problem is generating a BFS tree on the tree structure of vicinities of all the servers, constraints on the availability of each server. The children of the server's node in the tree is passed as an argument of the command sent to the next flooded server. Benefitted from the nature of a BFS tree, infinite loop never occurs.

3 Comparison between Python and Java

Java is a popular language for implementing web application servers, and the following study served as a comprehensive comparison between Python and Java in implementing this task.

3.1 Type Checking

The ways Python and Java do type checking differ a lot, and each way has its own advantages and disadvantages. Python, though being a strongly typed language, uses dynamic type checking--the type of an object is resolved during run time. As an example, defining every variable in Python need not specify its type by prefixing the declaration of the variable a type name. One also do not have to specify what type of a parameter or the return value of a function is. This feature grants Python great flexibility and enhances productivity of program prototyping. But this flexibility comes with a cost of a decrease in reliability. Since the type is resolved at run time, the type related error will only be noticed during the execution, which leads to many unreliable programs.

In Java, this kind of problem is avoided by using static type checking--each object's type is resolved during compile time. When there is a type incompatibility, the Java compiler will immediately complain about it and throw an error, and the buggy program will never run at all. This does prevent possible type related issues, but it also comes with a cost that more effort is needed to write type-consistent Java codes, and this is usually a pain during development.

3.2 Memory Management

Both Python and Java's objects live within the heap. They implement their garbage collection mechanism, which cleans unused object and free the memory inside the heap, in different ways.

Java uses two mechanisms in its garbage collection--the mark-and-sweep algorithm and the generation-based copying collectors. The mark-and-sweep mechanism is a subroutine of every allocation of a new object. It uses mark bits to mark all of the objects in the heap via the roots pointed to them, recursively, and reclaimed the spaces of all unmarked objects. This algorithm is proved to be effective in reclaiming heap space and is in fact adopted by many other major languages. Another mechanism, the generation-based copying collectors, essentially uses the trend that most of the older objects are pointed to more frequently, so this algorithm will focus on cleaning those objects in younger generations. It uses pointers to keep track of the heap space occupied in this newest generation block, while cleaning and rearranging the objects to create new spaces within the block. Both mechanisms work effectively and can be run with reasonable time complexity.

Python uses a different but controversial approach. The necessity of cleaning an object's space is determined by the number of variables that are referring it, which is called the link count. Each assignment to another variable of an object will increase its link count. When the link counts go to zero, the object is deemed as garbage and its allocated space will be reclaimed later on. This approach is very straightforward and fast--the need of traversing the roots of objects or the generation blocks are eliminated. The only thing that need to be kept track of is each object's link count, which can be easily implemented. However, there are some potential problems. First, a reference cycle might appear when amateurish programmers assign variables such that they form a loop of reference. When this happens, and once all of the objects have lost all the referential connection outside the loop, these objects in the cycle will never have their link count decreases to zero--they become garbage forever. This type of problem could be solved by careful development, but this approach of link count fundamentally does not work with parallelization, which will be discussed in the next section.

3.3 Multithreading

As discussed above, due to Python's approach in garbage collection, the language does not work well in multithreading application. To maintain consistency of link count of a variable across multiple threads, a global lock needs to be applied and the performance gain from multithreading significantly decreases. In this case, since the asyncio library is single threaded, the problem regarding multithreading can be ignored, but the potential performance gain from parallelization is lost.

On the other hand, Java has comprehensive libraries that support multi-threaded application pretty well. With a server herd written in Java, the performance granted by powerful multithreading code is definitely capable of empowering high-volume requests from multiple clients.

3.4 Summary

In the implementation of the application server herd, the choice between Python and Java is essentially several tradeoffs. While Python gives more flexibility and simplicity, Java gives more reliability, performance, and scalability. For a small-scale prototyping application, like the assigned project, Python should be a better choice since its great simplicity decreases developing time a lot. Java, on the other hand, is better for larger applications.

4 Comparison between asyncio and Node.js

Node.js is a rising solution for web applications based on Chrome's v8 web engine. It has become popular within a short period since it grants many of the front-end JavaScript programmers the capability to write server-side code using the same language they are used to. Besides its portability in web development, it has many other great features that are similar to those Python's asyncio provides.

The general semantics of Python's asyncio and Node.js are very similar in writing asynchronous code. Node.js also put coroutines as callback functions on an event loop and dynamically yields the tasks when waiting on asynchronous calls. The Future class of asyncio is the analogy of the Promise class of Node.js. Starting from JavaScript ES2017, it also has the keywords async/await implemented which works exactly the same as those in Python's asyncio. They both have the performance gain from the asynchronous scheduling.

Though Node.js has many things that asyncio provides, applications written in Python still have their advantage in Python's comprehensive libraries, which make programs in Python more extendable.

Also, Python is more suitable in processor intensive tasks than Node.js. Thus, Node.js is better for a fast and light-weight application whereas Python performs better on an application that requires support from various libraries and puts more pressure on the processor.

Conclusion

Based on the analysis above, it can be shown that Python's asyncio has many problems when implementing a large-scale application server herd where the bottleneck problem cannot be resolved by asynchronous task scheduling. However, for a small application like the assigned project, Python is a great fit since its short development cycle and nice performance in small-scale programs. Thus, the conclusion is to use asyncio for a smaller prototyping server herd and use other languages like Java for larger scale applications.

References

[1] "asyncio — Asynchronous I/O" - *Python 3 Documentation*
https://docs.python.org/3/library/asyncio.html#module-asyncio