CS131 Project Report

Maggie Zhu
Lab 1B

Abstract

In this paper, we will examine Python's asyncio asynchronous networking library when used to implement a proxy server herd application. We will evaluate the efficacy and efficiency of asyncio library in creating such applications. We will compare the Python approach using asyncio with an alternative approach that is Java-based, and compare the entire approach with Node.js.

1 Introduction

Wikipedia is a free content online encyclopedia written and maintained by a community of volunteers through open collaboration . It is the largest reference work that is most-read in the history of encyclopedia. It receives between 25,000 to 60,000-page requests every second on average.

Wikipedia and other related websites currently run on platforms based on Dabian GNU/Linux, the Apache web server behind an NGINX-based TLS termination proxy, the Memcached distributed memory object cache, the MariaDB relational database, the Elasticsearch search engine, the Swift distributed object store, and core application code written in PHP+JavaScript.

They all use multiple servers behind the Linux Virtual Server load-balancing software, with two-level caching proxy servers, Varnish and Apache Traffic Server. The requests are first passed to a front-end layer of Varnish caching servers then back-end layer caching is done using Apache Traffic Server. Requests which could not have been served from the Varnish cache are sent to load-balancing servers which run the Linux Virtual Server software.

While this solution works well for Wikipedia, we are building a new Wikimedia-styled service designed for news in the given project task. With the following differences, it is important to explore and search for an alternative method that is more suitable:
(1) updates to articles happen more often
(2) access will be required using multiple protocol, not limited to just HTTP or HTTPS
(3) clients will tend to be more mobile and will be making requests from more diversified areas

In the new Wikimedia-styled service, PHP+JavaScript application server does not perform as well. It will become a bottleneck that lowers the performance of the service. From a software standpoint, the application will put too much effort into attempting to add new servers, due to reasons including the significant amount of cellphone client constantly broadcasting their GPS locations to the servers. From a system standpoint, the response time will also increase since the Wikimedia application server becomes a central bottleneck on the core clusters.

A possible solution to the stated issue is the implementation of an application server herd. In such a solution, multiple application servers are able to communicate directly to each other as well as to the central databases and caches. The inter-server communications are designed for fast-changing data. That encompass smaller data including, the focus of this project, GPS locations and to larger data including ephemeral video data. The original data base server will still be available for use, mainly for more stable data that is less often accessed, or that requires transactional semantics.

To implement such an architecture, we will be examining Python's asyncio networking library. We will be evaluating its strength and weaknesses by looking at a prototype application which proxies for the Google Places API.

2 Python and Java: Comparison

2.1 Type Checking

One of the biggest difference between Python and Java is at type checking. Python variables are dynamically typed and Java variables are statically typed. In Python, type checking is deterred until runtime, thus there is no need to declare a variable type in advance in the assignment statement. Java variables are type checked during compilation. A Java variable's name and type need to be declared during the assignment statement.

Dynamic type checking allows Python code to be written with more clarity and simplicity. It greatly enhances the productivity of programmers. However, users of Python need to be extra careful when declaring variable since errors will only be reported during runtime; whereas Java and other programming languages that uses static type checking are able to catch type error during compile time, hence increasing the runtime efficiency. A downside of static type checking is that it is nearly impossible to manually raise a type error in the code even when the code block will hardly ever be called – the type checker will be able to find a situation to raise the type error and prevent you from successfully compiling the program. The programs mostly crash during compile time.

2.2 Memory Management

In terms of memory management, Python and Java share a common strategy that their management systems are both automatic: Python memory manager and Java Virtual Machine. The memory management systems both are able to automatically allocate and deallocate pieces of memory per program needs. The main difference between the two languages is at their garbage collection strategies. Garbage collection is a form of automatic memory management. The garbage collectors will attempt to reclaim memory no longer in use that was allocated by the program, AKA garbage.

Python has two ways of deleting unused object from memory. The first is by reference counting. Each Python object is stored in memory with names and references. Whenever the reference count of an object becomes zero, the garbage collector will erase the object from memory. However, it becomes a problem when facing cyclic reference. For instance, appending a list to itself will result in cyclic reference. The solution to this problem is the second way of garbage collection – generational garbage collection. It is a trace-based method which is able to break cyclic references and it is able to delete unused objects even when they are referred to by themselves. A drawback of Python garbage collection method is that it is bad for memory itself since the reference counting algorithm is constantly running, counting and storing the reference counts in memory in order to keep the memory clean and make sure the program runs effectively.

Java is a memory safe language. Garbage collection is a feature which helps in collecting the resources that are free and released in the program. It uses concurrent mark sweep collector. It is a form of tracing garbage collection algorithm that involves tracing which objects are reachable from one or multiple roots to find unreachable memory locations. Unlike reference counting in Python, concurrent mark sweep garbage collector is not constantly running in the background, but it can run at any point in the program lifespan.

In comparison, for the new application server herd, it is better for the architecture to use Python's reference counting garbage collection method considering the large amount of memory the application will need.

2.3 Multithreading

In Java's JVM, it natively support multithreading running on multiple cores. Therefore, multithreading in Java can achieve true parallel execution. This will improve the performance for CPU-intensive jobs. If too many threads are executing at the same time in comparison to the number of CPU cores, context switch will bring too much overhead to the program which will in turn decrease the performance for CPU-intensive programs, which only makes sense when the program purpose is IO-related servers.

On the other hand, Python does not support concurrent execution on different CPU cores. It has a global interpreter lock (GIL). For any thread to run, it must hold the GIL in order to access the shared memory space. This was implemented to prevent race conditions in order to preserve the correctness of the reference counting mechanism. Unfortunately, Python is implemented using C library which is not thread-safe, and the Python interpreter does not support fine-grained locking mechanism like JVM does. Thus, parallel execution in Python is simply context switching. It is not effective when executing parallel programs since it does not allow true parallelization. Considering the nature of our application, we should be utilizing a language that support large amount of IO-tasks, since the in the project, servers often receive, propagate messages and conduct API calls.

3 Asyncio and Node.js

3.1 Introduction

Aysncio is a language-agnostic paradigm that has implementations across a variety of programming languages. It has received dedicated support in Python and developed rapidly from Python 3.4 to 3.7 and it is still widely used til

now. It is a single-threaded, single process design which uses cooperative multitasking. It allows tasks to voluntarily run and pause. This is well suited for the need of the new application.

3.2 Implementation

The asyncio library in Python is essential implemented as an event loop. Each thread task voluntarily runs, pauses and yield control. This execution paradigm, where code control moves front and back between different tasks, waking them up at the point where they left off, is called "coroutine calling". Asyncio brings this execution paradigm to Python which facilitated CPU usage since it is idle less of the time.

Functions and codes can be declared as coroutines by using the keywords "async" and "await". Keyword "async" allows the code to be able to pause its execution and yield its control to other coroutines. The "await" keyword allows the code to pause its execution until the awaited function returns. This approach works well for IO-bound code where long pauses are needed to wait for a response to a request.

In addition, asycnio works well for the new application we are trying to make. The asyncio streams in the asyncio library is designed to work with network connections as streams allow sending and receiving of data without using callbacks or low-level protocols and tranports.

3.3 Asyncio in Server Herds

In our project, we are asked to design a server herd that can synchronize data through inter-server communication and communicate with client applications using the asyncio library. The server herd should behave like a parallelizable proxy for the Google Places API.

Asyncio provides convenient and easy-to-use functions when creating the server herd. The asyncio.start_server() function starts a TCP server by simply supplying it with the server IP address and a port number. Each server is provided with a reader, writer pair object that can be used to read data from client and send data back to client. We can add tasks like read into the asyncio event loop using keyword await and async. The codes will be able to run concurrently, read and parse messages asynchronously. The function serve_forever() is able to send and receive messages continuously without having to wait for a response before executing another task. In the project, the await and async keywords are also used when we are propagating messages between servers.

The five servers are named: Bernard, Campbell, Clark, Jaquez and Juzang. They have bidirectional communication as described below:
Clark talks with Jaquez and Juzang.
Bernard talks with everyone else but Clark.
Juzang talks with Campbell.
Using the Asyncio await and async keywords, the five servers should be able to pass each other messages starting with "AT".

On the other hand, like Python, Asyncio is single-threaded and it means that it will have limited ability to maintain the parallelism and it lacks support for multithreaded programs. As stated above, Python has the Global Interpreter Lock (GIL) which enforces single-threaded execution. Hence, the performance of asyncio in Python is less ideal than it would be in other languages that support multithreading.

A potential problem might come up when the number of servers in the herd significantly increase. With asynchronous functions in the asyncio library, it is highly likely that our code will execute out of order. And due to the nature of asyncio, it is very difficult to determine the actual order of tasks received. A possible situation is that when an "IAMAT" request arrives within close time proximity with an "WHATSAT" request, if executed out of order, the "WHATSAT" request will not be successful, giving the wrong result back to the client.

3.4 Asyncio and Node.js

Python and Javascript are both dynamic languages which is not designed to execute code in parallel. Python's global interpreter lock and Javascript's intentional single-threaded design both hinder these two languages to run threads concurrently. Running multithreaded code is equivalent to running multiple programs while maintaining inter-process communications between them. The process of it is both time and memory consuming. Python's asyncio and Javascript's Node.js provide asynchronous programming solution.

The asyncio library included with Python provides the tools to use async for processing disks or network I/O without making other code pieces to wait. Node.js, on the other hand, is designed with the goal of achieving asynchronization.

Essentially, asyncio can be thought of as a framework that is built off of Node.js. Async/await is a new way of writing asynchronous code; previous alternative in Node.js for asynchronous code are callbacks and promises. Async/await is, like promises, non blocking. Async/await makes asynchronous code look and behave a little more like synchronous code. This is where all its power lies.

3.5 Performance Implication

For our project, the task being carried out mainly focused on the IO operations which is not very CPU-intensive. Thus asyncio is capable of carrying out the task. For any larger operational tasks that is CPU-intensive, for example, running heavy calculation-focused algorithms, will cause the asyncio to become a bottleneck since it is essentially single-threaded. If the project was to be implemented by a language which allows for true parallelization, the performance of the application could be improved significantly.

3.6 Reliance on Python 3.9

Due to significant security concerns, the reuse_address parameter of asyncio.loop.create_datagram_endpoint() is no longer supported. A new coroutine shutdown_default_executor() is added that schedules a shutdown for the default executor which waits on the ThreadPoopExecutor to finish closing. asyncio.run() has also been updated to use the new coroutine. Beside these major changes, some minor updates were made in Python 3.9; the developers should be able to use older versions of Python to implement this project.

4 Conclusion

Based on above research in comparing the asyncio library in Python compared with other alternatives including Java-based program and Node.js, we can conclude that asyncio in Python is well-suited for proxy server herd application like this project, provided that there are no CPU-intensive tasks and the amount of servers are within manageable range. Asycncio introduces event loop and coroutine to Python which allowed tasks to be asynchronously performed by letting tasks to voluntarily pause itself and yield its control to the next task in the event loop. When compared with Java, Python lacks certain features or is less powerful in certain functionalities including the garbage collection model. Overall, I would suggest the use of Asyncio in Python used in the new Wikimedia-styled services.

5 Reference

What's New In Python 3.9
https://docs.python.org/3/whatsnew/3.9.htmlasyncio

Node.js Documentation
https://nodejs.org/en/docs/

Asyncio Reference Page
https://docs.python.org/3/library/asyncio.html