Report for CS131 Project

Zixuan Lu
304990072

2019-06-03

Abstract

This project allows us to explore Python asynchronous networking library asyncio to implement a web server herd. Currently, Wikipedia and its related sites based on Wikipedia architecture use LAMP, and work fairly well. This project, however, requires us to build a new Wikipedia-style service that has frequent updates, access from different protocols, and more mobile clients. This means servers are a bottleneck under this circumstance. As a result, a different approach using Python asyncio is considered. Also, we will compare the performance of this approach to that of a Java/Node.js based approach.

1 Introduction

Among the three major challenges we are facing in implementing this Wikipedia-style service, the high frequency of updates of article is the most significant reason that we choose to use asyncio for implementation. Currently the Wikipedia architecture used by Wikipedia-related sites uses LAMP platform, which based on GNU/Linux, Apache, MySQL, and PHP, using multiple, redundant web servers behind a load-balancing virtual router for reliability and performance. In the context of more frequent updates and more mobile clients, the LAMP platform will soon face bottleneck at the virtual router.

However, by using Python asyncio package, we increased the performance of the server herd by allowing handling multiple client requests simultaneously, as this solves the bottleneck of the virtual router. This will be discussed in detail in the next sections. Also, the report will also include comparison with Java in terms of memory management, multithreading, and garbage collection, and comparison of asyncio and Node.js.

2 Explanation of asyncio's advantages in this project

This section gives explanation of coroutines, event-loops, the project's design and implementation, minor disadvantages of asyncio, and a final conclusion.

2.1 Introduction to coroutines and event-loops

Python asyncio package offers coroutines, which technically, "are functions whose execution you can pause" [1]. This allows concurrent programming (multiple tasks have the ability to run in an overlapping manner[2]) to be possible. For example, when a coroutine is awaiting a task, the control can be given to other coroutines and tasks, thus improving performance. In the context of the server herd, when we are waiting for input/output, we can handle the control to data processing.

On the other hand, these concurrent features are all made possible by the event loop, which is like **while True** loop that is always looking for idling coroutines and tasks that can be completed in the meantime[2]. Taking this project as an example, when a client connects, an asynchronous function (handle_input in this

project) which is defined as a coroutine object handles the request and is put into the event loop. As a result, when there are multiple clients connecting, each connection is scheduled as a coroutine, and is handled asynchronously. Every connection can be handled simultaneously, and the implementation is free from mixing up the buffer for different clients because they are handled in different coroutines.

2.2 The project's design and implementation

As stated in the project spec, there are three types of requests a server can receive: IAMAT, WHATSAT, and AT. All of them are initially handled by a coroutine (handle_input) as stated in the previous subsection. While awaiting other asynchronous functions to validate the requests and generate outputs, the control can be handled to other coroutines. I will mention here one of the coroutines, which handles WHATSAT message. Since it needs to query Google Places API, and the aiohttp library (used to send HTTP GET requests) supports asynchronous HTTP requests, during the execution of this coroutine, control can be again handle to other coroutines while awaiting the response of Google Places API, enabling more concurrency. These implementations allows multiple clients to be served concurrently in an efficient way.

The implementation of this server herd using asyncio also solves the bottleneck of LAMP platform, which has a central virtual router. Instead of having a central server to store all the client information (which is needed in answering WHATSAT requests), each individual server store all the client information. Thus, they do not need to consult the central server for client information, eliminating the bottleneck. The only drawback of this feature is each server has to flood client information to all the other servers. I implement this functionality by writing an asynchronous function, which check whether the received IAMAT/AT message contains the most updated client information (by checking the time stamp). If yes, then flood the information to other neighbouring servers together with its own server id. This prevents infinite flooding loops, and is computationally cheap. Also, since the flooding function is asynchronous, while writing to the socket, the control can be handled to other coroutines, enable even more concurrency. Thus, we conclude aysncio is more suitable for the implementation of this server herd.

2.3 Limitation of the project, and solutions

Currently I have devoted a lot of words to describing why asyncio is a good architecture, however, it is not perfect. One of the un-addressed issue caused by concurrency is indeterminate execution order. For example, when a WHATSAT is sent right after an IAMAT message, in a synchronous model, the server will respond correctly with the information obtained from Google Places API. However, in the current asynchronous model, the WHATSAT message may be handled first, thus, the client may receive an error message which is an erroneous behaviour. However, this is a justifiable sacrifice for the higher performance of the server herd. Also, since handling the requests are not computationally expensive, it is very unlikely to receive two WHATSAT and IAMAT messages that are so close in time that will cause errors.

Another issue about current asyncio model of the server herd is that its concurrency only uses a single thread. This is wasteful especially in today's context: most CPU has multiple cores and threads, and this is especially true for server CPU's (consider the CPU's for linux servers at UCLA). With multi-threading, we can easily increase the throughput of each individual server by a few times. However, this issue is not hard to solve, and is not an actual problem of Python asyncio package. We can add new threads in our implementation of the server herd easily, and these will make the server better at handling multiple client requests.[3] In fact, we can actually create new threads for new event loops.

2.4 Problems that I ran into

The most significant problem for me is as a asynchronous programming beginner, it took me days to truly understand what it is, despite I read all the documentations, blogs and TA slides. Also, some problems regarding Python language arises during the process. As a scripting language, Python does not require compiling, thus some typos are only revealed when that line of code is actually executed. Also, since the server's event loop runs forever (without receiving an interrupt signal), I have to go into my logs to read the error messages. This makes me spend more time on debugging process.

2.5 Conclusion about Python asyncio package as the solution for the server herd

The asynchronous solution for the server herd not only solve the bottleneck problem faced by the LAMP architecture, it also increases the performance by allowing concurrency, as stated in the previous subsection. It has issues that is not addressed in this prototype, but certainly its benefits outweigh the problems, and hence it is an ideal solution for the server herd.

3 Comparison of languages and modules

As stated in the spec, the boss is worried about Python's type checking, memory management, and multithreading. Thus, I will compare these aspects with those of Java language. Also, I will compare the overall approach of asyncio to that of Node.js.

3.1 Comparison with Java

3.1.1 Type checking

Both Python and Java are strongly-typed languages, however the fundamental difference is: Python is dynamically typed, while Java is statically typed, just like C[4]. The advantage of Python's dynamic checking is easy to identify: it makes the project easier to write, more succinct, and looks closer to natural language. However, these advantages come with a cost. First and most importantly, dynamically checking means extra work during runtime, which reduces the efficiency of the server herd. This problem does not affect that much as of now when the server herd is small and client requests are relatively infrequent. However, when the herd scales, this problem will affect the performance in a clearer way as flooding will become far more frequent. Secondly, dynamic typing means less information about the source code. Static typing, in most projects, serves as a source of implicit documentation as it tells programs information about variables and return types[5]. Though in a mini prototype this does not help much, when the herd scales, type declaration makes work clearer. Also, no type declaration is a direct reason that makes me spend more time on debugging: variable names with typos are treated as new variable, and will only cause error during runtime.

3.1.2 Multi-threading

When talking about type-checking, both languages have their own advantage. However, Java's multi-threading performance is clearly better than python's. Python's inefficiency in multi-threading comes from its Global Interpreter Lock (GIL), which "is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once"[6]. While being safe, this means Python's parallelism is never a "true parallelism": taking the project as an example, it is never possible for an individual server to update its client database (implemented by a dictionary) simultaneously by multiple threads. Thus, Java is definitely the better choice for multi-threading applications as it allows "true parallelism".

3.1.3 Memory management

Both Python and Java have built-in memory managements, unlike languages like C. However, the two languages have different approaches, and I will

mainly talk about the garbage collector for Python and Java. Python uses reference counts while Java uses a mark and sweep algorithm.

Python's reference count approach is intuitive: each object has additional memory used to keep track of the number of time it is referenced. The reference count is incremented/decremented when a reference is added/removed. The object is ultimately deleted when the reference count becomes zero[7]. Though it is simple, it has the risk of creating cycles. Consider the following example:

```
def create_cycle():
    x = [ ]
    x.append(x)
```

x will not be free even after the function returns because it creates an object x that refers to itself.

Java, on the other hand, uses a mark and sweep algorithm. When an object is created, the mark bit is set to false. In the mark phase, the mark bit is set to true for all reachable objects. Then in the sweep phase, all objects with a false mark bit will be freed. This approach is free of cyclic references. However, when the garbage collection algorithm runs, normal program execution is suspended. Thus, program performance is affected[8].

In the case of this project, I only defined several simple variables to hold intermediate results of the output message, as well as client information. Thus, my code is free of cyclic references, and a simple Python garbage collector will suffice and achieve maximum efficiency. However, when the project scales, Java's approach can be used to prevent possible cyclic references.

3.1.4 Conclusion about comparison with Java

Both languages have their own advantages in memory management and type checking. Java is the winner in multi-threading performance comparison because it allows access to a shared memory simultaneously by two threads. In the context of this project, the ease of use of Python makes it the choice of implementation because the project is just a simple prototype. However, for the real server herd for the Wikipedia-like services, Java is probably the better choice due its more "rigorous" nature (static type-checking, cyclic-reference-free garbage collector, true parallelism).

3.2 Comparison with Node.js

Just like asyncio, Node.js is an event-driven and asynchronous open source server environment. I will compare these two modules/environments based on their performance and scalability.

Being based on Chrome's V8 which is a very fast and powerful engine, we can see that Node.js is significantly faster when comparing the performance of Node.js to that of Python asyncio. This makes Node.js the ideal choice for real-time applications, for example, those involving chat functionality[9].

Scalability is the ability of an application to serve the increasing number of requests with no compromise in performance. Node.js has good scalability in simple Web applications because of its single-threaded asynchronous architecture with I/O operations completed outside the thread[9]. However, when it comes to the development of complex applications with a lot of concurrent processes, using Node.js requires in-depth knowledge and research. Also, Node.js is not suitable for processor-intensive tasks which need to handle a large amount of I/O, just like this project. Thus, in the context of this prototype, Python asyncio is the better choice, especially with Python's succinct code structure and comprehensive documentation.

4 Conclusion

After analysing the features of Python asyncio module, comparing Python language with other language in the context of this simple prototype, I conclude Python asyncio module is the ideal choice. Asyncio's functionality, together with the ease of use and well-written documentation of Python, makes the task of

building the server herd more doable.

References

[1] How the Heck Does Async/await Work in Python 3.5? Brett Cannon - https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/

[2] Async Io in Python: A Complete Walkthrough Real Python - https://realpython.com/async-io-python/#the-event-loop-and-asynciorun

[3] Threaded Asynchronous Magic and How To Wield It Cristian Medina-Cristian Medina - https://hackernoon.com/threaded-asynchronous-magic-and-how-to-wield-it-bba9ed602c32

[4] Python Type Checking (guide) – Real Python Real Python - https://realpython.com/python-type-checking/

[5] Statically Typed Vs Dynamically Typed Languages 1558288245 - https://hackernoon.com/statically-typed-vs-dynamically-typed-languages-e4778e1ca55

[6] Python Wiki, https://wiki.python.org/moin/GlobalInterpreterLock

[7] Garbage Collection in Python https://www.geeksforgeeks.org/garbage-collection-python/

[8] Mark-and-sweep: Garbage Collection Algorithm https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/

[9] Python Vs Node.js: Which Is Better For Your Project https://da-14.com/blog/python-vs-nodejs-which-better-your-project