Asyncio for Server Herd Architecture

Abstract

Due to rapidly-evolving data, the LAMP architecture is no longer suitable for a new Wikimedia-style service. We thus want to design a server herd architecture to replace the LAMP architecture. In implementing the server herd architecture, I use flooding algorithm to allow for an new update to one service will be forwarded rapidly to the other servers and thus reduce the need of central database and its bottleneck. Additionally, I also examine how type checking, memory management and multithreading compared to the Java's approach play an important role in my choosing Python and Asyncio to implement this server herd architecture.

Introduction

The goal of this project is to implement a server for handling frequent requests to update news articles via various transport protocols. Also, clients will be assumed to be more mobile. Using the LAMP platform to handle the above services will be a bottleneck because the response time of Wikimedia database server is too slow to handle constantly updating data. Moreover, database server is not suitable to handle rapidly-evolving data since it's primarily used for more stable and infrequently accessed data. Therefore, to avoid this kind of bottleneck, we want to design an application server herd architecture, where multiple application servers directly communicate with each other without having to talk to the database so that rapidly-evolving data can be processed efficiently. Thus, we decide to utilize Python's asyncio module to help us achieve a better server performance and reliability because of its asynchronous feature. Furthermore, it has high-level APIs to simplify asynchronous code and make it more as readable as synchronous code. Compared to the LAMP platform which relies on a load-balancing virtual router for reliability and performance, asyncio is definitely a better solution used to handle rapidly-evolving data in terms of simplicity, reliability and maintainability. In addition to Python's asyncio suitability for the new-style application, we also want to compare Python with Java in terms of type checking, memory management and multithreading as well as the comparison between asyncio and Node.js.

Advantages and Disadvantage of Using Asyncio

Due to the constantly mutating data the servers have to deal with, Asyncio is a suitable framework for this kind of application because of one of its key element in its asynchronous feature: coroutines, also known as co-operative routines. It is the coroutines, and their co-operative nature, that enable yielding control of the event loop when they are in idle state so that other coroutines can start to work instead of waiting for the previous task to finish. Coroutines cooperate with each other so well that they not only increase the performance but also the reliability of our program. Coroutines can pause itself while waiting for its output and so other awaited coroutines can run immediately, and then resume to where it previously left off or its saved state and then continue to work on the rest of the task. This makes sure that the time period when a coroutine is waiting for its dependent task to finish is being fully used, and hence greatly increase performance. Furthermore, the event loop is constantly managing and looking for the potential time gap and allow the awaited coroutines to go in the loop and run for that period of time gap. In this way, while the time is being efficiently used, the tasks run after one another and thus only one task is running all the time without conflicting with each other, ensuring the reliability of the program. This behavior is reasonably ideal for high traffic server since both performance and reliability can be satisfied at the same time.

When we are implementing a simple and parallelizable proxy for the Google Places API, the servers need to handle the requests and process them so it can decide what to do next. If the request is simply a location update, such as requests that begin with IAMAT, the server will check if the client ID is already in its dictionary. If it doesn't, the server will store the information to its dictionary. If it does, the server will then check if this is a repeated message or an updating message. In other words, if the server already has an exactly the same copy of the message in its dictionary, it will ignore it in order to avoid infinite loop when propagating messages to other servers; however, if only the client ID is the same while location and time information are different, then the server will treat it as a request for information update from the clients and update the message in its dictionary. On the other hand, if the requests beginning with WHATSAT command, the server will parse the message and directly contact Google Places for the information and send back to its client. With the help of coroutines in asyncio, these processes can be easily implemented. We can define async def function to do messages propagation. While the servers are checking if the messages are repeatedly sent from other servers or simply a location update from the clients, or even when they are parsing the received messages, servers can use this period of time to do other tasks such as processing more requests and sending back messages that have already been processed. Therefore, asyncio indeed can prevent bottleneck as much as possible as it is able to handle frequent inputs and outputs reasonably well.

In addition, during this process, no central database is involved since using async def to define an asynchronous function for messages propagation enables all the servers to share the same updated information. Like I mentioned before, coroutines make full use of the time, where one coroutine is waiting for its currently-processing results, and thus dramatically increase the performance, whereas the LAMP platform which heavily relies on central database to store all the updated information, will be a bottleneck in such situation. In my implementation of the servers herd architecture, one server receiving a request or command from the clients means all servers will also receive that message because I utilize asyncio's event-driven nature which allows an update to be process and forwarded rapidly to other servers in the herd. And then the servers will parse the commands to decide their next actions, including prevent infinite loops. The servers in the server herd architecture almost act like they have "telepathy". Therefore, there is no need to contact the central database for updated information. That's also why implementing server herd architecture with asyncio is such a big advantage over LAMP platform.

Moreover, while contacting Google Places for places nearby using aiohttp module, asyncio's asynchronous nature allows the servers that when getting the results from Google Places API, the servers can do other tasks. All in all, asyncio is a great fit for implementing server herd architecture since the co-operative nature of asyncio and the coherent and collaborative nature of server herd architecture line up perfectly. However, there is a disadvantage of using asyncio. Most of the time, coroutines are waiting for slow operations to finish; and asyncio can speed this up by allowing the servers to overlap the times the coroutines are waiting rather than doing them sequentially. But in a CPU-bound problem, there is no more waiting. The CPU works as fast as it could to finish the task; and setting up coroutines for tasks will only make the CPU do extra work. Worse still, coroutines are useless in such situation since there is no waiting. Although this disadvantage of asyncio is unpleasant, luckily we are implementing a server herd architecture which requires a lot of waiting. Therefore, for implementing asynchronous server herd architecture, I would recommend using asyncio. Obviously, asyncio outperforms the LAMP approach in the new-style application.

Trouble and Problem

The problem that I ran into while implementing server herd architecture is that there is infinite loop during messages propagation in the server herd. I later fixed this by checking repeated sent messages and then everything worked as expected.

Python vs Java

Type Checking

The biggest difference between Python and Java is that Java is statically typed while Python is strongly but dynamically typed. This difference is almost enough to make Python easier to use than Java. In Python, we don't have to provide a type when declare a variable but the variable must be defined before they are used. In other words, we spend less time debugging syntax and semantic errors so we can spend more time on debugging logic errors, which reduces some unnecessary distractions when implementing a server. On the other hand, Java's statically type checking has many type constraints but at the same time offers a better run time performance and compiler optimization. At the compile time, we can find out what errors causes the program to crash and hence increase reliability of the code. Therefore, in the case of implementing a simple project, Python is a better choice since we focus more on ease of use and the logic to implementing the server herd architecture. Type constraints may distract us in some way and impede our progress while if using Python, we don't have to be concerned too much about the type.

Memory Management

Python internally has a private heap containing all Python objects and data structures managed by memory manager. For memory allocation, Python has two strategies: reference counting and garbage collection. For the reference counting approach, it works by counting the number of times an object is referenced by other objects. When references to an object are removed, the reference count for that object will be decremented. When the reference count drops to zero, that object will be deallocated. However, if there is a reference cycle, some objects may never be freed. Therefore, Python has a second strategy for that. Garbage collection is a scheduled activity. If the number of allocated objects minus the number of deallocated objects is not zero, the garbage collector is run.

On the other hand, Java also has a built-in garbage collection, which uses a mark-and-sweep-algorithm. The algorithm traverses all the object references, starting from the garbage-collection root, and marks every object that has not been deallocated. And then all of the heap memory that is not occupied by those marked objects will be reclaimed.

Although both Python and Java have their own garbage collector, they work quite differently. However, in the case of my implementation server herd architecture, I don't think the memory management issue matter that much since I mostly define local variables and some global variables. Python's two strategies for garbage collection will handle them well.

Multithreading vs Asyncio

Python's asyncio use a single thread to facilitate concurrent code and increase better performance while Java has many libraries that support concurrent programming and multithreading. This means that on a single CPU core, Asyncio's cooperative nature will probably outperform Java's concurrent feature since multithreading running on a single CPU core will be slower than a single-thread version. Moreover, Java's concurrency suffers from race condition, while Asyncio always makes sure that only one task run at a time. However, nowadays, most of the computers equipped with multi-core CPU; and thus Java will be a better choice since the threads can be fully exploited. But considering the reliability issue, Python's asyncio is overall a better approach, especially in the implementation of the server herd architecture.

Asyncio vs Node.js

Both Asyncio and Node.js use asynchronous programming and support only one thread. Moreover, their event-based architecture and non-blocking I/O allow for maximizing the usage of a single CPU core and computer memory, making servers faster and more productive. Therefore, they are pretty similar when building a server. However, when it comes to the ease of use and convenience, Asyncio becomes a better choice. When using the async await in Python's asyncio, we don't need to work with the low-level stuff, such as futures and event loop. Asyncio has both high-level and low-level APIs. On the other hand, in Node.js, async await is built on top of Promises just like Python's async is built on top of coroutines, and futures, except that when working with async await, the lower level stuff has to be taken care of too all the time. In other words, with Asyncio, we can always stay on the high level and finish the job, whereas with Node.js, we need to deal with both high level and low level methods. Although with Node.js, both backend and frontend can be written in Javascript, users do not have to learn other new technologies, our implementation of server herd architecture is a complete server side project without any frontend requirement, so Asyncio in this case is a better fit and capable enough for us to implement this project.

Conclusion

Python and asyncio not only provides ease of use and thus easy to maintain but also reasonably good performance and reliability. To implement a simple project liker server herd architecture, Python and asyncio surpass many of their counterparts. Moreover, Python has high-level syntax, which is fairly readable and writable, as well as many powerful modules like Asyncio. Therefore, for this specific kind of project, server herd architecture, I would say Python and Asyncio are a perfect fit for it and thus truly recommend them to implement the project.

References

Statically typed vs dynamically typed languages, Apr 29, 2017, Available: https://hackernoon.com/statically-typed-vs-dynamically-typed-languages-e4778e1ca55

Java Memory Management, Available:https://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works/

Garbage Collection In Python, Available: https://www.geeksforgeeks.org/garbage-collection-python/

Node.js vs Python Comparison, Jan 30, 2018, Available: https://www.netguru.com/blog/node.js-vs-python-comparison-which-solution-to-choose-for-your-next-project

Intro to Async Concurrency in Python vs. Node.js, Feb 5, Available: https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36