CS 131 Python proxy herd via asyncio

University of California, Los Angeles

Abstract

Python's `asyncio` module was introduced into the standard language package to address asynchronous co-operative code execution, which works especially well for developing servers and socket communication applications. This report discusses the implementation of a proxy herd over TCP/IP without using a central database and its pros and cons versus Node.js and Java.

1 Introduction

As applications users and clients switch towards a more mobile mode of computing, the currently predominant server architecture of a central server and database is becoming more and more of a bottleneck. With state changes happening frequently, users using other non-HTTP protocols, and the mobility of users a central server and database quickly becomes overloaded.

This project explores the idea of using a server herd, where a set of non-centralized servers communicate and propagate new changes via flooding. We wish to assess the suitability of Python and its `asyncio` module for dealing with this type of load.

In this vein, several of the key things we plan to investigate are the reliability of `asyncio` and the maintainability of it. Some theoretical advantages include the ability to assign asynchronous callbacks for clients, making the development and maintaining of the code easier. However, Python's choices regarding execution time type checking, multithreading, and memory management could lead to potential issues.

To investigate this issue further, we developed an example proxy herd in Python using `asyncio`. It supports five separate server instances. Each server is connected to a specific subset of the other servers, and they can all receive client information. Upon receiving client information, each server checks to see if the information is new, relevant information and updates and floods the update to all neighboring servers if it is. Clients send updates over TCP via messages starting with the phrase 'IAMAT', followed by several arguments pertaining to the client information. Clients can request information about a place via messages starting with the phrase 'WHATSAT'. Servers respond and propagate updates to each other with messaages starting with the phrase 'AT'. Additionally, the servers support logging to help with maintainability and reliability.

2 asyncio

`asyncio` was extremely easy to use to develop the application, and the use of asynchronous execution with cooperative scheduling allows the higher latency calls such as accessing the Google Places API to run efficiently and not waste CPU time. Additionally, this allows for a single serer to easily handle many clients by switching execution between them when awaiting on an `awaitable/coroutine`. This allows for single-threadaed concurrency, as long as the coroutines are written well and give up the thread cooperatively.

2.1 Advantages

Without a doubt, one of the strongest advantages of `asyncio` is the simplicity of the code. It was easy to grasp the execution flow and understand how to write asynchronous code for Python. The choice from Python to use the simple keywords `async` and `await` and keep the rest of the functions feeling the same made writing asynchronous code extremely easy. Additionally, the functionality provided by the `asyncio` library specifically for writing and developing TCP servers on top of Python's simple but extremely powerful string manipulation made starting a TCP server and parsing requests quite easy. Additionally, the `aiohttp` library and many other asynchronous Python libraries are built on top of the `asyncio` paradigm and code, allowing for seamless integration even when making HTTP requests.

Another advantage of Python over many similar technologies is the efficiency of the `asyncio` coroutine structure.

Other technologies often leverage the OS's built in concurrency via functions similar to `fork` in C, making new processes. However, this causes the OS to have to manage concurrency and context switching, abstracting it away from the Python interpreter and slowing it down.

2.2 Disadvantages

The most prominent and obvious disadvantage of `asyncio` is Python's inherent single-threadedness, and therefore the single-threadedness of `asyncio`. By using a global interpreter lock, Python and `asyncio` are unable to natively support multithreading. If the load on the servers is extremely high, it will be extremely difficult for a single core to handle everything, even with low overhead. This could be alleviated by taking advantage of running many small instances of the server, allowing any specific server to only have to handle a few clients at a time. However, with the current flooding system the actual load on each server will not be low as every server will eventually learn about every client update. In general, however, the single-threadedness of Python will make it unable to scale to extremely high load applications.

3 Python versus Java

A primary focus of this experiment was to compare the effectiveness of Python versus Java. In general, Python takes a strong advantage in its ease of writing. However, Java maintains a strong advantage in its multithreading capabilities.

One area where either could win is type checking; Python's runtime type-checking and style of exception propagation can make the server more robust to exceptions, but Java's compile time type-checking can help mitigate bugs before they manifest.

3.1 Type Checking

Because Python is dynamically typed, it only checks variable types at runtime while Java type-checks during compile time. One of the most obvious advantages of a statically typed language is that the language can report inconsistencies and catch programmer errors/bugs before running, which can help avoid costly production errors. However, this can add additional work to the programmers, especially when intentionally changing types.

This additionally affects Python's actual execution performance versus Java's; Pythons's execution environment has to type check during execution while Java code does not have to during execution.

Both languages are strongly typed, which can also reduce the convenience to the programmer since the programmer cannot take advantage of implicit type conversions but increases the robustness of the code and can help avoid expensive, hard to track down bugs.

3.2 Memory Management

Unlike languages like C or C++, Python and Java do not require manual memory management. Instead, they leverage garbage collectors to manage memory. However, the way that Python and Java accomplish this differs.

In Python, the garbage collector records the number of references to each memory location at any given point in time. If the number of references reaches zero, then the Python interpreter releases the memory. In contrast, Java uses a generational garbage collector, which traces each memory allocation and allows the interpreter to know when memory is no longer being used. The advantage to Java's generational garbage collector is that it can easily resolve cyclic dependencies (i.e. when two objects share references to each other). In contrast, a pure reference garbage collector would be unable to handle this situation.

Of course, Python has workarounds for this, but in general Java's approach is more robust and more stringent. However, Python's approach is extremely simple, allowing it to be used during runtime fairly cheaply.

Regardless, both should work in both our trivial application and in a real-life setting. Java may be more efficient with memory, but will likely encounter a bit of additional overhead when managing memory.

3.3 Multithreading

Python is inherently designed to be single threaded. The use of a global interpreter lock makes single threaded execution simple and efficient, but prevents multiple threads from executing simultaneously under normal circumstances. Java, on the other hand, supports true multithreading natively and even provides default keywords (such as `synchronized`) in the language itself.

The advantages to using a global interpreter lock are that single threaded execution has very little overhead, and can be synchronized very efficiently. This means libraries like `asyncio` can do what they want (i.e. run asynchronous code and context switch) very effectively, albeit in only one thread. However, well written multithreaded code would allow Java to handle a much higher load, especially with the embarrassingly parallelizable nature of handling multiple clients. This would allow a Java implementation of the same server to scale well with a more powerful server and handle a much higher load.

4 asyncio vs Node.js

`asyncio` and Node.js are both asynchronous event-driven models, which suits them extremely well for the task at hand. However, one thing that immediately comes to mind is the fact that for Python, this functionality exists as an extension of a language and framework not directly built or intended for this modality of execution. Node.js, on the other hand, was developed solely for this mode of execution. It is likely Node.js may be more efficient because of this, and it is definitely more feature-rich than `asyncio` within Python.

However, one important factor to think about is the language itself. Python, unlike Javascript, is strongly typed which may be very attractive as it can reduce the likelihood of hard to find bugs. Additionally, programmers may simply prefer Python due to familiarity or other language decisions between Javascript and Python (as opposed to Node.js vs `asyncio`).

5 Recommendation

Overall, in my opinion `asyncio` is a very good way to prototype and test a proxy herd, but it may not be a particularly good choice for full-scale production due to Python's single-threaded limitations and the language's fairly young support for `asyncio` (it was only added in Python 3.4). As the package matures and execution becomes more efficient, it may be possible for this package to eventually become a strong candidate for production scale servers. For the time being, `asyncio` may fall behind and be unable deal with the high amount of constantly changing state.

It should be noted that if the actual computation done on each server is trivial (like it was with our proxy herd example) and there are a fair number of servers, it may still be possible to use it as a full scale production server implementation.

6 Concluding Thoughts

The results found in this report show that `asyncio` definitely has its strong points and shines in its ease of use, maintainability, and robustness. However, it may struggle in high load production environments due to several language limitations including the global interpreter lock (which prevents multithreading) and the young support for event-driven asynchronous execution (the immaturity of `asnycio`). For the purpose of this prototype, `asyncio` is undoubtedly a very good choice, and it should be for any prototyping and to help test server designs.