CS 131: Project: Evaluating Effectiveness of Python's asyncio Module in Implementing Proxy Herd

Abstract:

When trying to scale a system, it is important to maintain reliability and correctness. We are trying to implement a server herd that will take multitude of requests from clients, while keeping track of clients from other servers as well. To do this, we need a library that allows for this reliability, scalability and correctness. In this paper, we consider Python's asyncio library.

asyncio is a Python library that allows for asynchronous nonblocking I/O programs through its event-loop design. It's ability to create these programs makes it a good candidate for our new program, and we want to test its ability to send and receive data by testing it on a small scale between a few servers. As a result of our analysis, we are able to make more conclusions about whether or not asyncio is fit for this job. We compare these results to Java and Javascript's Node.js to compare and draw the conclusion that Python is a good fit because it is very easy to understand, reliability when it comes to scaling, and maintains solid performance.

Introduction:

We are designing a news service that takes after the collaborative nature of Wikipedia. Compared to Wikipedia though, which uses a LAMP platform with multiple redundant web servers for reliability and performance, we would like a service that can (1) account for faster updates to articles, (2) allow for access from different protocols other than just HTTP, and (3) allow for mobile clients. Thus, using the same architecture as Wikipedia would be unwise because it would create bottlenecks in the central servers due to the constant updates.

We want to consider using an application server herd, which allows the multiple application servers to communicate directly with one another to keep each other up to date. It would be able to handle all the updates and changes happening through this service, allow for different protocols, and allow for mobile clients. In particular, this paper will judge the effectiveness of Python's asyncio module to implement this sort of server herd, and compare to some other libraries such as Java and Node.js to judge whether this module is the best way to implement this sort of service.

Requirements:

2.1 Python: asyncio and aiohttp

We will be using Python version 3.7.2 to implement both the servers and the test client, and will be heavily relying on the asyncio and aiohttp libraries to implement our servers.

The asyncio library was first introduced in Python 3.3 and integrated as a standard library in Python 3.4, coming about because Developers needed a way to create reliable and scalable code due to the increasing amount of Internet applications. It allows for asynchronous programs, thus the servers and client can handle multiple requests despite only having one thread. This is effective because it allows servers to handle the many requests and changes that clients will make to our new service, as well handle the requests that servers make to each other. This therefore eliminates some of the bottlenecks that were present in the use of the LAMP service for Wikipedia, and other synchronous architectures.

The aiohttp library is used to communicate with the Google Places API, which only takes in HTTP requests. The aiohttp library uses asyncio too.

2.2 Google Places API

Google Places API lets you search for place information based on proximity to a location or simply by text, returning information about each place. In this implementation we search using location, inputting a latitude longitude position using ISO 6709 notation. The API returns a JSON with places within proximity to that location.

Server Implementation:

3.1 Server Design

We tested this server herd implementation by creating five servers: Goloman, Hands, Holiday, Welsh and Wilkes. Each server has unique relationships with the others, so a single server doesn't communicate with all of the other servers. This way, it mimics the situation when this platform is scaled up: as more and more servers are needed, it becomes more and more expensive to connect each and every one of these servers to all the others. So by not connecting all the servers, we are testing the effectiveness of the servers to keep each other up to date with newest updates.

Additionally, each of the servers keeps track of which servers are down. This is kept track of by whether or not a server responds to a request.

3.2 IAMAT and AT messages

A client can send a message to any one of the servers in the form of:

IAMAT [client name] [location] [time]

The server that receives this message will log the client name and the location of the clients. The time is also kept for reference. After logging the client, the server responds to the client with a message in the form of:

AT [server name] [time diff] [client name] [client location] [original time]

Thus letting the client know that it has responded to the message. Additionally, the server will then propagate this message to the other servers that it has a relationship with. The other servers will then replace the server name with its own and send it out to all the servers that they have a relationship with. If there is no response from the other server, it is logged that the server is down, so that all servers can have a good idea of which of its neighboring servers are down.

When a server receives a message from one of its neighbors, it checks if it already has the client logged. If it is, but if the time stamp is the same, then it ignores it. This prevents a server from having to constantly update an entry.

3.3 WHATSAT message

A client can also send a message to any one of the servers in the form of:

WHATSAT [client name] [radius (km)] [max number of results]

Which prompts the servers to send a request to the Google Places API for a set amount of locations within the radius specified of the client name. The server then sends back the results in a form of a JSON.

Because the servers propagate the IAMAT message across the other servers, the same user doesn't have to send both the IAMAT message and the WHATSAT message to the same server. The two servers don't even have to be directly connected. A user can send the IAMAT message to Goloman and get results when sending the WHATSAT message to Holiday. Even though Goloman and Holiday aren't neighboring servers.

3.4 Error handling

Faulty requests are returned to the client in the form:

? [original message]

The can servers only catch syntactical errors – that is – the servers send this message back if the client sends a faulty time, or doesn't format the location properly. Errors can be made more specific in the future.

Notably though, the servers have no way of detecting other errors yet. That is – servers can't detect if somebody is giving a false location, or pretending to be a server by sending an AT message to a server as a client, thus corrupting an entry.

Discussion and Comparison:

4.1 Python: asyncio and aiohttp

Note that our implementation doesn't measure performance in any way yet, so it will be difficult to draw any concrete conclusions based on the test we have created. But despite that, this implementation still tells us things about the feasibility of Python's asyncio as a way to implement this server herd.

4.1.1 Ease of Use

It was surprisingly easy to implement and test this Python code. Errors were immediately shown, and I

could be assured that there would be no race conditions because of the asyncio library. Creating a multithreading program wouldn't have this advantage because errors are not obvious, possibly only revealing themselves when the program is scaled. But then it would be too late.

We also know that this application will need to handle frequent requests and must similarly send out at least an equal amount of requests too. But we can trust that asynchronous program will be able to handle these requests better than a synchronous program will be able to. Thus a first plus to using asyncio is that it is easily scalable.

4.1.2 Garbage collection

Python's garbage collection system helps the servers from not being slowed down by the eventual garbage collection that's necessary. Python uses reference counts to keep track of objects that are used, and objects that are referenced more than once are kept track of. So when the reference count goes to zero, the object is marked for deletion in real time. This is a big advantage.

For languages like C and C++, memory has to be manually freed, and can lead to error that only reveal themselves when the system is scaled up. By then it would again be too late. Using Python would be an advantage because it will keep the problem of garbage collection at a minimum, thus allowing the program to be scalable.

4.2 Java

Java is another language that we can consider. Java has it's own libraries that allows for asynchronous programming. We compare the differences between the two languages.

4.2.1 Type Checking

Type checking could become a problem for using Python because it uses dynamic typing. This will be less of a problem with Java because it uses static typing. Servers will be receiving all types of data and requests from different users. As I was implementing my test Python server, most of the error checking was checking whether the types matched up with what was sent in. Since requests came in as a string, I had to convert the string to a float or int. It was hard to keep track of when a variable was a string, and when it was a integer or float. This can be a problem later on, if there is an error with the checking, or if Python interprets a variable as a wrong type, thus leading to errors in the code. Especially when scaling up, servers will be not just receiving one or two types of messages, but possibly hundreds of different types. Having to check for types would lead to messier code and greater chance for error when scaling this code up.

Java will not have as much of a problem because of the static typing. It is trivial to see what the type of a variable is, making it much easier to keep track of everything. This would lead to less errors in the code and allow for it to be more scalable.

4.2.2 Garbage Collection

Java also uses a garbage collection method called "mark and sweep," that doesn't require constantly having to keep track all references to an object. It might have a slight advantage over Python because it takes more work to keep track of all the references, but there is no major advantage.

4.2.3 Java Virtual Machine

One big selling point of Java is its ability to run anywhere, making it a huge plus for scalability of the server herd. Being able to instantly run this code on a new machine without having to change the code to match the machine's specification saves time and leads to less errors in the code.

4.3 Node.js

Lastly, we look into the advantages and disadvantages of Node.js, a JavaScript library that supports asynchronous programming for JavaScript in the same way that asyncio does for Python.

4.3.1 Inherently Asynchronous

One difference between these two libraries is that Node.js is inherently asynchronous. This means that async and await don't have to be explicitly called, but Node.js already does it automatically. It also already has an event loop, so an event loop doesn't have to be called explicitly. This makes it easier to code in JavaScript. But on the other hand, it makes it less readable. Like mentioned earlier, Python is very readable code, and having these extra parameters to explicitly let readers know that this code is asynchronous may be helpful for them understand what is going on.

4.3.2 Support

Node.js is also popular language for server code, and has been well tested, while Python's asyncio is a fairly new library. Thus one advantage that Node.js has over Python is that support that it has. But this shouldn't be a major factor when deciding which language to use.

Further Research:

We have tested a small subset of a larger question on networking. Specifically, whether Python's asyncio library is fit to implement a server herd. Further research can still be done on the larger picture. More specifically, this program has brought up the importance of how the servers are connected to one another.

5.1 Server Failures

This project also tells us a little about the use of sever herds in general, and whether is a good idea to move away from the idea of a centralized servers, and into having multiple servers. Further research should be done to test the effectiveness of this model. A few problems should be addressed and looked into:

5.1.1 Server Failures

A big problem can be if not enough connections are made. Suppose a server only has one connection, and the other server goes down, then the server has no connections, and isn't updated with the current information, thus leading to problems. This problem may persist even when scaled up, and each server has thousands of connections. If a large sector of the servers go offline, it may lead to isolation of another group of servers. Therefore, it's not only important that each server is adequately in the loop with other servers, but that connections be made wisely.

5.1.2 Too many Servers

On the other hand, it may also be unwise for servers to be too connected. Having too many connections may lead to redundant calls from other servers, thus making it difficult for a server to handle its own request to its clients. A server that's bogged down with other servers trying to give it requests will be unable to best serve a client.

5.2 Flooding Algorithm

In this project, we used a simple flooding algorithm that forwards the message a server received to all of its neighboring servers. The server also tracks which messages it has already received, so if a redundant message is sent back, it won't process it again and create a cyclic loop. But having too many neighbors increases the chances the server will receive multiple redundant messages, thus slowing it down.

More research can be done on what the most effective flooding algorithm is, but while keeping in mind the possibility of server failure. A flooding algorithm should also be made so that a sever that goes offline can be kept up to date of the messages that it missed.

Conclusion:

In conclusions, we have found that Python's asyncio library is a very good fit for implementing a server herd. Being asynchronous, it is able to maintain reliability and scalability. Having multiple servers with a flooding algorithm lets clients make a request to any of the servers, and receive a response, thus making this system more mobile than the traditional method using LAMP. We can also access the servers using various protocols now, and not just HTTP. We can conclude that Python's asyncio is an excellent choice to implement it. Additionally, we have considered Java and JavaScript, and compared them to Python. Although further research and testing is required, we don't see any major drawbacks to using either one.

References

1. Asyncio, https://docs.python.org/3/library/asyncio.html
2. Aiohttp, https://aiohttp.readthedocs.io/en/stable/
3. Python PEP 492, https://www.python.org/dev/peps/pep-0492/
4. Jython, https://www.jython.org/
5. Node.js, https://nodejs.org/en/
6. Node.js Net Documentation, https://nodejs.org/api/net.html