Server Herd using *asyncio*

Brendan Alger

Abstract

There are multiple possible architectures to consider. Wikimedia for example uses the LAMP platform. Each architecture has different strengths and weaknesses. In a an environment where clients will be more mobile and servers will be updated more frequently, implementing the LAMP platform like Wikimedia might cause bottlenecks. In this project, a server herd was used instead, implemented using Python's asyncio module.

About

A server herd is a group of servers where each server can handle client requests. What is unique about a server herd is that there is no central server, since each server stores all the information needed to complete a client request. Servers also communicate to other servers as well, sending important information on any updates made to any server information.

How the server is Implemented

Structure of the Herd

The server herd consists of 5 servers: Goloman, Hands, Holiday, Welsh, and Wilkes. Each server is limited in contact with other servers. In other words, each server cannot contact all 5 servers. In order for a server to contact all servers, they will need to go through another server to get to it. For example, Welsh only connects to Holiday, so in order for it to get in touch with Goloman it needs to connect to Holiday and then have Holiday connect to Goloman. Each server can communicate with all the servers using a flood fill algorithm.

Figure 1: Graph of who can communicate with who

Server	**Can talk to**
Goloman	Hands, Holiday, Wilkes
Hands	Goloman, Wilkes
Holiday	Welsh, Wilkes, Goloman
Welsh	Holiday
Wilkes	Goloman, Hands, Holiday

Servers and clients communicate with each other using TCP. Clients can send to servers an IAMAT or WHATSAT request in order to update, add, or access client information. Servers also communicate using a slightly modified IAMAT to update user information.

Implementing IAMAT

IAMAT is sent from the client to any server and has this format:

IAMAT <client> <location> <time>

<client> is the name of the client sending the request, <location> is the longitude and latitude using ISO 6709 notation, and <time> is the time the client sent the request using POSIX time. The server will read the information and store it into a dictionary list. It will then check other servers and see if the new information has been added yet. This is done by sending the IAMAT message to other servers with a CHANGE keyword, indicating for the server to check if the client information has been added or modified. The server will know whether or

not to make an update by checking the time that the server's iteration of the data was added. If the time is different, then it has not been updated and should get changed. This prevents the servers from going into an infinite loop that hangs the system.

After updating the client information, the server confirms the request by sending:

AT <server> <time_difference> <client> <location> <time>

<server> refers to the name of the server that recieved it, <time_difference> is the difference in time between client and server, and the last 3 are a reprint of the parameters of the original request.

Implementing WHATSAT

WHATSAT is sent from the client to the server and will send information on the area around the requested user. WHATSAT follows this format:

WHATSAT <client> <radius> <num_items>

<client> is the name of the client of whose information we're requesting, <radius> is the size of the area to look in, and <num _items> is the number of results the client desires.

To gather this information, WHATSAT uses the Google Places API to gather the information. Because of this, there are limitations on the radius and num_items parameters. Radius cannot exceed 50km and num_items cannot exceed 20 items. If the client violates this, the server will return the message, marking it as invalid.

Receiving a valid request from the client will output the same AT output as IAMAT, along with a JSON formatted list of the data requested.

Regarding invalid messages

The server checks all input before executing any code. If the client sends any invalid messages, then the server will return the message sent with a '?' appended to the front. For example, if the client sends a random string 'asdfghjkl' by accident, the server will respond by sending '? asdfghjkl' back.

Asyncio pros/cons

Using asyncio makes it very easy to run servers handling requests asynchronously. Each server can run the same code, and initializing the server is as easy as running the python script.

Functionally, requests aren't processed immediately when they are received, allowing the server to process lots of requests at the same time. This works very well for server herds, allowing them to process information and adding connections at the same time. Additional python framework aiohttp also allows for http requests to run asynchronously, and the json library can easily process JSON objects, which allows easy implementation of the Google Places API into the servers.

While allowing processes to run asynchronously is beneficial, it can also be detrimental. Since processes don't run the the same order they're sent, it gets difficult to debug and errors that wouldn't show up in a synchronous implementation shou up here. For example, if a client sends an IAMAT message to the server and a WHATSAT to the same client is sent after, it is not guaranteed that the IAMAT will be processed first. This could cause a potential

error despite the commands being called in the correct order.
Another issue is the lack of multithreading. Scaling the project up, it might be beneficial to split incoming tasks to another thread and run it parallel to increase performance. Asyncio works well for this project, but might be detrimental the larger the project gets.

A Java approach

As mentioned previously, scaling could be an issue when using asyncio. And Python's use of type checking, memory management, and multithreading has different effects on performance compared to using a language such as Java. Python uses dynamic type checking while Java uses static type checking. Python's dynamic type-checking makes it easier to use the asyncio methods since the programmer doesn't need to know the exact types. This makes it easier to start writing the code. However, the disadvantage would be rereading the code and trying to understand it. The advantage of Java's static type checking is its ability to understand the project later on.
As far as memory management, Python uses reference count while Java uses mark and sweep. Python's reference count method allows objects to be immediately deleted after use, but with the disadvantage of its large overhead. However for the server herd, since variables are only used briefly it doesn't affect performance.
The main disadvantage of Python that was mentioned earlier is multithreading support and scaling. Python uses a Global Interpreter Lock which only allows one thread to execute at a time. Java will allow use of multiple threads, which allows for more work in the same time. So as more clients use the servers, it will become easier to scale Java by adding more threads while Python will have difficulty.

Compared to Node.js

Node.js and asyncio are both quite similar. They are both dynamically typed, designed for server code, and are asynchronous. Both asyncio and Node.js are meant for server side code. Node.js is typically much better than Python for memory intensive programs, however luckily with server herds, they are not very memory intensive. Both also use asynchronous design, but asyncio uses a coroutine, which allows execution to be halted and new information to be entered. This ability gives more flexibility to asyncio.
A big disadvantages of Python however is its prevalence in front-end development. Javascript is widely used in front-end, so using it in the server herd would make sense as to keep the languages the same.

Conclusion

Asyncio is very suitable for server herd implementation. It allows for a quick implementation compared to Java, and thanks to the nature of a server herd, the disadvantages of using python over Java or Node.js can be mostly overlooked. Although asyncio works well for a small scale prject such as this, further research is needed to see if it will be equally effective compared to Java and Node.js once the server increases in scale.