CS131 Python Project Report: Proxy Herd with Asyncio

Juan Bai

Abstract

The primary objective of this project is to look into a different architecture called an "application server herd", where the multiple application servers communicate directly to each other as well as via the core database and caches. The interserver communications are designed for rapidly-evolving data (ranging from small data such as GPS-based locations to larger data such as ephemeral video data); whereas, the database server will still be used for more stable data that is less-often accessed or that requires transactional semantics.

In this project we are mainly looking into implementing a Proxy herd using the Python asyncio asynchronous networking library as a candidate for replacing part or all of the Wikimedia platform for our application. The project assesses the Python's type checking, memory management, and multithreading compared to Java-based approach along with comparison with Node.js.

1. Introduction

Wikipedia and its related sites are based on the Wikimedia server platform, which is based on GNU/Linux, Apache, MariaDB, and PHP+JavaScript, using multiple, redundant web servers behind a load-balancing virtual router and caching proxy servers for reliability and performance. While this scheme works fairly well for Wikipedia, it is not necessarily the best scheme for a Wikimedia-style service designed for news, where (1) updates to articles will happen far more often, (2) access will be required via various protocols, not just HTTP, and (3) clients will tend to be more mobile.

With these requirements, the Wikimedia server architecture is not suitable given new servers to need to be easily added and response times need to be faster than can be provided by the Wikimedia PHP+JavaScript application servers. Instead of just connecting to a single server, mobile clients can connect to any server in the herd, and the data will be propagated to the nearby servers without having to access back to the main database.

2.0 Implementation of Server Herd Prototype

In this project, we use the Python asyncio library to implement the interserver communications across five servers in the server herd, i.e. Hill, Jaquez, Smith, Campbell, and Singleton along with their ports: 11625, 11626, 11627, 11628, and 11629 respectively. These five servers communicate with each other as shown below in Figure 1: Server Herd Communications.

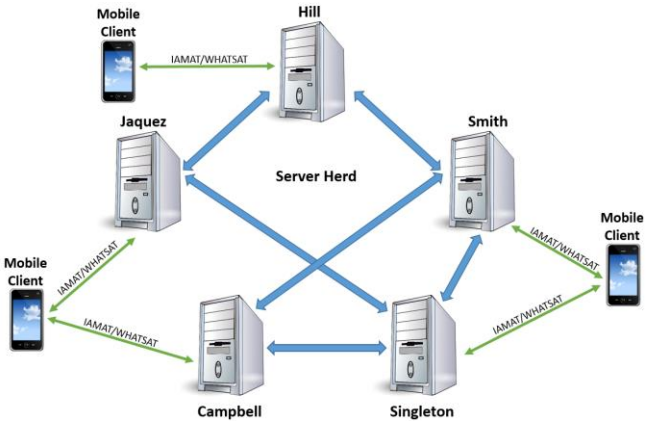

Figure 1: Server Herd Communications

Each server should accept TCP connections from clients that emulate mobile devices with IP addresses and DNS names. Clients can connect with any of the five servers sending either IAMAT or WHATSAT requests. Servers will respond to any other invalid request with ? and appending a copy of the invalid request. Each new server is added with the built-in start_server function in the asyncio library that starts a socket server and returns a pair of (reader, writer) objects that the server can us to communicate with the client.

2.1 IAMAT

IAMAT requests are sent from the clients to update the servers about its current latitude and longitude decimal degrees using ISO 6709 notation along with the client's current timestamp in POSIX time. The IAMAT request format is as follows:

IAMAT <Client ID> <Coordinates> <Timestamp>

Once a server receives an IAMAT request, the server stores the information and propagates the client infor-

mation to its other connected servers through a simple flooding algorithm. To prevent looping, a server checks whether the received message information as already been received by checking against stored client information.

The server responds to client with a message using the following format:

AT <Server Name> <Time Difference> <Client ID> <Coordinates> <Timestamp>

Time difference is the difference between the server's time when it received the message from the client's timestamp.

2.3 WHATSAT

Clients can also query from a server for information about places near other clients' locations, with a query using this format:

WHATSAT <Client ID> <Radius> <Max Number of Results>

The server receiving the WHATSAT will consider the request invalid if the server does not have any information on the client specified in the Client ID either through a direct IAMAT request or client data flooded to it from another server. Since servers in the herd flood client data between servers, a client can send a WHATSAT to any server in the herd even if the client had originally sent the original IAMAT request to a different server; hence, no flooding is needed for WHATAT requests.

The server responds with a AT message in the same format as in response to the IAMAT request, giving the most recent location reported by the client, along with the server that it talked to and the time the server did the talking. Following the AT message is a JSON-format message, exactly in the same format that Google Places gives for a Nearby Search request.

The server that receives the WHATAT requests calls the Google Places API using the aiohttp library calls. Each successful call returns a JSON result of nearby locations.

2.4 Interserver Communications

Servers propagate client location data to their connected servers (as shown in Figure 1: Server Herd Communications) upon receiving a valid IAMAT request or a valid flood propagated message from another server. Using the asyncio library method open_connection, a server can create reader and writer objects to communicate with another designated server. Connections are closed after information is flooding, and messages are only propagated if not already received by a server.

3. Python Asyncio Library vs Java

Migrating from Java to a Python based implementation requires addressing differences in type checking, memory management, and multithreading abilities.

3.1 Type Checking

Python uses dynamic type checking; whereas, Java uses static type checking. Dynamic type checking means the Python interpreter conducts checking while the code is running which allows variables to be typed differently throughout the program. Dynamic type checking allows for simpler more concise code where variables/objects can be passed between modules without having to fully declare their types before compilation. This allows for more flexibility when setting up the server without having to know static type in advance. On the other hand, Java's static type checking at compilation allows for errors to be caught before run-time. For our server herd, dynamic type checking provides more flexibility, but in large scale projects, static type checking may be more appealing to prevent hard to debug run-time errors.

3.2 Memory Management

Python and Java both use garbage collection to allocate memory but with key differences. Python uses a simple method based on the idea of reference counts in which each object has a count that depends on the number of times that the object is being referenced by another object. This object can only be deconstructed/de-allocated when its count reaches zero. This approach is inefficient with circular references resulting in the count never reaching zero. The server herd in our application does not present circular references; hence, Python's memory management works well. Java's garbage collection uses a mark and sweep method where the memory manager sweeps through objects and marks unreferenced objects for deletion on the next sweep. This two-step process is slower than Python.

3.3. Multithreading

Python implements a Global Interpreter Lock (GIL) which guarantees only one thread who holds the lock can be executed at one time. This prevents multiple threads from executing on an object at the same time, i.e. no paralleling threading in Python. Java does use multithreading with support for concurrency and parallelism. For our server herd application with a single-threaded program, the Python GIL is acceptable given the servers are not doing intensive computations that could benefit from multithreading.

4.0 Asyncio compared to Node.js

Both Python and Javascript provide asynchronizatin and event looping, i.e. Asyncio for Python and Node.js for Javascript. Python used coroutines; wherease, Node.js uses the concept of callbacks. Node.js automatically handles communications asynchronously compared to Python handles communications asynchronously when using the asyncio library. This may make Node.js easier for implementing our server herd application, but still Python has stronger support for more applications.

5.0 Pros and Cons for using Asyncio

Python and the Asyncio library make it easy to build a server herd application especially given that Asyncio's coroutines allow servers to handle multiple requests. Asyncio provides many built-in methods that are tailor made to implement the server herd, e.g. start_server, open_connection. Start_server for server to client communications is called by simply passing the IP address, port and a coroutine function for the server to run. Establishing this type of TCP connection in other languages is much more complicated. Open_connection is used with similar ease for server-to-server communications. Extending the server herd beyond our current five servers would be easy using the asyncio framework. Additionally Python provides libraries like aiohttp which makes handling HTTP requests easily.

As a con, Asyncio does not support multithreading which may be required for the herd servers if they need to launch and run intensive multiple tasks. Asyncio also uses a cooperative yielding of tasks. Tasks that do not yield gracefully can lock out other tasks and over consume server CPU time. Preemptive multitasking can resolve this issue, but Asyncio does not provide this capability. Lastly, asynchronous methods can create situations where the order of received requests and flooding can result in an out of order handling of IAMAT and WHATAT requests. If the WHATAT request is processed before a valid IAMAT request, the server will send back an invalid respond to the client which is an error.

Conclusion

This project assessed implementing a simple server herd using Python as the programming language and the Asyncio library has the communication framework. In general Python and Asyncio allow for easy implementation given coroutines and event loops in Asyncio. Additionally client/server and server/server communications are supported with built-in methods. This approach does lack in multithreaded capabilities and memory management for handling circular references, but given our server herd application with the need for simple single threaded communications handle client-server connections and server-server connections to flood client data, the Python and Asyncio library are well suited to implement our new Wikimedia-style service.