CS131: Proxy Herd with Asyncio

Emmett Cocke
2022-06-02

Abstract

There are many options when it comes to implementing web servers. Many major programming languages have libraries supplying developers with access to common web server functionalities. An interesting implementation of these functionalities comes with the asyncio library for Python. When tasked with efficiently handling frequent updates to multiple distributed servers, an application server herd architecture the built with the asyncio library effectively provides the needed functionalities along with the speed required to satisfy requests in a reasonably short response time. This paper will detail the implementation of a prototype application server herd along with comparisons to the many alternative frameworks available for other languages with a focus on node.js. Additionally, this paper will explore the difference in implementations of the application server herd with Python and Java, along with the benefits and drawbacks of each approach.

1 Introduction

Wikipedia, a popular free online encyclopedia, is built upon the Wikimedia server platform using a Linux, Apache, MySQL, PHP (LAMP) stack. emphasises scalability through separation of operations and servers. For example, in Wikimedia's architecture there is a separation of cheap and expensive operations into separate query groups such that the cheaper operations have priority when running to minimize delay of users. Additionally, spacial and temporal locality is utilized via caching of the most commonly visited sites so requests are fulfilled in a minimal amount of time.[1] While this architecture is effective for hosting content which is not frequently updated, an application where updates occur frequently along with the need to satisfy accesses via various protocols, not just HTTP and HTTPS would be slowed down by the very things that make Wikimedia effective at hosting Wikipedia. Because frequently accessed content is cached and updates need to be propagated to one of the central databases prior to server updates, a news service will be unable to utilize this feature as content will be regularly updated and changes will need to be propagated across servers regularly so users accessing separate servers will be supplied with the same, up to date, information. Additionally, the separation of operations would not be as effective for a news service as priority of updating and requesting information from the servers should be relatively equal priority thus it makes more sense to fulfill requests as soon as possible rather then complicating their completion with scheduling.

2 Prototype

To meet the demands of a news based service, an application server heard would be an ideal server architecture. An application server herd is a collection of servers which communicate with each other and clients in order to share client information. With an emphasis on inter-server communication, this is a good fit for applications where updates to data are frequent and clients request updated information from servers frequently.

[1] https://wikitech.wikimedia.org/wiki/Wikimedia$_i nfrastructure$

2.1 Server Herd

The prototype of this architecture consisted of five servers each with a list of adjacent servers to whom client data would be spread to and received from bi-bidirectionally. Each server actively waits to receive connections and commands from adjacent servers or clients. Once a connection is made and a command is received, the server verifies the validity of the command and responds accordingly. Additionally, if client information is sent to the server, the server will attempt to open a connection with adjacent servers and send the client data. When client data is sent between servers, a flooding effect is achieved by comparing the received client data with the cached data and an update to the cache will only occur if the server has not seen this client before or if the server has outdated client information cached.

2.2 Commands

An integral function of the prototype is to handle different commands which resemble the type of functions a news service would encounter. To mimic this behavior, commands are built around clients location data as updates are frequent due to the mobility of devices like phones and laptops.

2.2.1 IAMAT

One of the commands recognized by the server is the IAMAT command which records a position specified by the client along with a time and a username. The format is:

IAMAT $\langle username \rangle$ $\langle latitude \rangle$ $\langle longitude \rangle$ $\langle time \rangle$

The location information is parsed by the server as if it is following ISO 6709 notation and the time is expressed in POSIX time. When this command is received by a server, the location will be cached by the receiving server and a response will be sent back to the user in the form:

AT $\langle server \rangle$ $\langle timediff \rangle$ $\langle command \rangle$

Server is the name of the server who originally received the command, time diff is the difference in times of the client and server, and command is the original request sent. This identical response is also sent the the adjacent servers so that they have the most up to date location of the client in case their data is requested on another server.

2.2.2 WHATSAT

The other client command supported by the server is a command which lists nearby locations based on a clients location. The form for WHATSAT is:

WHATSAT $\langle username \rangle$ $\langle radius \rangle$ $\langle limit \rangle$

Radius is the limit of where to look for locations around the location tied with the username and limit is the number of locations to receive. The WHATSAT command uses the Google Places API to look up and receive location information within the specified radius of the client.

2.2.3 Invalid Commands

When commands are received from a client that are not formatted like the above commands, the commands are marked as invalid and sent back to the client following the format:

?$\langle command \rangle$

Command is the invalid command sent by the client.

3 Evaluating Asyncio

Asyncio is an asynchronous networking library for Python providing the functionality to handle TCP connections along with handling requests and sending responses to connections asynchronously. The core of asyncio is the event loop which abstracts away the details of how to handle running of asynchronous operations providing a high-level API for use. This high level API enables users to create and interact with the event loop by supplying it with coroutines and tasks.[2]

3.1 Suitability

Python with the asyncio library is suitable for meeting the requirements of the application server herd

[2] https://docs.python.org/3/library/asyncio-task.html

architecture. With the high level APIs provided, creating listening servers that respond to requests efficiently is relatively simple. Additionally, the functionalities provided by the API are perfect for mitigating the long latency associated with IO as these tasks can run asynchronously. This makes things like the google API call not eat up processing time idly as while waiting for a response, the server can handle other requests.

3.2 Performance Implications

As the server herd was built using an asynchronous library, the performance of asyncio is much better than a synchronous counterpart. Because servers are able to receive and handle multiple simultaneous connections and process incoming requests asynchronously, the main slow down in a synchronous framework which is waiting for responses from clients or other called APIs is a non issue. These delays are not visible when the server is running because of the event loop driven execution where coroutines (a subroutine that performs some task needed by the caller) are able to be awaited so they do not block when waiting for responses.

3.3 Ease of Use

Python is a very programmer friendly language in that it has many conveniences and asyncio is no different. Because all the functionalities needed for the server herd are provided by a high level API, the building of the architecture is very simple once the idea of asynchronous execution is understood.

4 Python vs Java

When compared to Java, Python has a very different aim. While Java is built more for speed, Python is built more for convenience. An example of this is in how they both manage memory. In Python, memory management is done via reference count and garbage collection which works by increment and decrement each objects reference count when a pointer to the object is created or deleted. When the reference count hits zero, the object's memory is freed by the garbage collector. Like Python, Java has a garbage collector but instead of keeping track of reference counts it instead runs a mark and sweep algorithm which will deallocate objects which are no longer accessible. Another major difference is that Java would implement the server herd using multithreading. This capability is out of Python's reach as it is implemented with C using mutexes in a way which prevents easy and efficient multithreading[3].

5 Asyncio vs node.js

Asycio and node.js share the trait of being asynchronous models. Because both serve very similar purposes, the real difference in use comes from the load of the tasks. Python, being a more general language has the benefit of being better able to efficiently handle large and expensive requests which can be undertaken by the many fast libraries written for Python. Node.js on the other hand was built specifically for hosting servers and it also benefits from using the same language as browsers[4]. When the functionalities of a web server are not intensive, node.js is a better choice then Python with asyncio[5].

[3] https://docs.oracle.com/en/java/
[4] https://nodejs.org/en/docs/
[5] https://eng.paxos.com/python-3s-killer-feature-asyncio