Proxy Herd with Python 3.9.2's `asyncio`

Abstract

This research was conducted in order to determine if Python's `asyncio` library is viable for implementing an application server herd. Using a local prototype to investigate the pros and cons of `asyncio`, application performance and usability were analyzed and compared to Java and Node.js counterparts. After consideration, it has been determined that `asyncio` is a perfectly reasonable framework to use to develop an application server herd.

1 Introduction

Wikipedia-style platforms are implemented on the Wikimedia server platform, a stack consisting of Debian GNU/Linux, Apache, Memcached, MariaDB, Elasticsearch, Swift, PHP + JavaScript, and redundant web servers.

1.1 Motivation

The Wikimedia server platform works fine for Wikipedia, however, if a new service is created that updates more frequently, requires more flexibility in protocols, and is tailored towards more mobile clients, the PHP + JavaScript in the existing server platform may throttle performance. As a result, this new application may favor being built on an application server herd architecture. This architecture centers around the servers' ability to communicate with one another, as well as communicating with the core database. These additional connections make the server herd very efficient in handling volatile data, as new data can be propagated between each server, removing the need for constant communication with the core database, making this architecture perfect for this hypothetical platform. This shift in architecture requires that an appropriate framework be found for it to be built on, which prompts the investigation into `asyncio`, a Python library allowing for the development of single- threaded, event-driven, concurrent code.

1.2 Server-side Prototyping

In order to better understand whether `asyncio` is viable for the purposes of this application, a small prototype was implemented, consisting of 5 servers: Riley, Jaquez, Juzang, Campbell, and Bernard. Each of these servers is capable of accepting TCP connections from clients, and was hosted locally on a set of 5 different ports. These servers were run using source code written in the file `server.py`, started by the `asyncio.run()` subroutine. These servers communicate bidirectionally, where Riley talks to Jaquez and Juzang, Bernard talks with Jaquez, Juzang, and Campbell, and Juzang talks with Campbell. This pattern allows a single request to any server to be propagated to each other server using a flooding algorithm built on AT messages and the `asyncio.open_connection()` subroutine. AT messages consist of the origin server's name, a time difference between sending and receiving, the domain name of the client, the client's geolocation, and a timestamp for when the message was sent. Each server is then responsible for processing AT messages and updating their client information appropriately to prevent infinite propagation loops. Every action these servers take is logged in a file, along with the timestamp that the action was performed at.

1.3 Client-side Prototyping

Each of the 5 servers may accept 1 of 2 messages from clients that are connected to them. The first of these messages is the IAMAT message, where the client provides a domain name, latitude and longitude, and a timestamp of when the message was sent. Upon receiving an IAMAT message, servers will propagate the location of the client to every other server using the flooding algorithm detailed above. This will then be followed by a response to the client in the same form as the AT message that the servers use to keep themselves updated. The other message is a WHATSAT message, where the client provides a domain name, a radius below 50 km, and a bound on the amount of information they want to access. This will

cause a server to initiate a call to the Google Places API, using the information provided in the message. This call to the API provides the server with location information in the form of JSON object, which is then returned to the user, following another AT message. Any message that the client sends that does not fall within these guidelines is replied to with an error message.

2 Python vs. Java

2.1 Type Checking

2.2 Memory Management

2.3 Multithreading

3 `asyncio` vs. Node.js

4 Conclusions

4.1 Ease of Use

4.2 Performance Implications

4.3 Features of Python 3.9+

4.4 Problems Encountered

4.5 Evaluation

5 References

"asyncio Source Code" Updated December 18, 2020 Available:
https://github.com/python/cpython/tree/master/Lib/asyncio

"asyncio - Asynchronous I/O" Updated March 6, 2021. Available:
https://docs.python.org/3/library/asyncio.html

P. Eggert "Project. proxy herd with asyncio" Updated February 24, 2021. Available:
https://web.cs.ucla.edu/classes/winter21/cs131/hw/pr.html

"What's New In Python 3.9" Updated March 6, 2021. Available:
https://docs.python.org/3/whatsnew/3.9.html