CS131 Project. Proxy Herd with asyncio Report

Abstract

The primary objective of this project was to utilize the Python asynchronous networking library asyncio to implement the application server herd architecture, as well as analyze the tradeoffs of choosing asyncio as compared to Java and Node.js.

0. Introduction

Wikipedia is implemented via the Wikimedia Architecture, which uses the LAMP web stack composed of Linux, Apache, MySQL, and PHP, and multiple redundant web servers behind a load-balancing virtual router for reliability and performance. However, the application server becomes the bottleneck for the scope of this project in light of new changes: (1) updates to articles happen more often, (2) access will be required via various protocols (i.e. HTTP, TCP, SSL), and (3) clients tend to be more mobile. These changes present bottlenecks from both a software point of view (i.e. added pain to add newer servers), as well as a systems point of view (i.e. response time is too slow).

Therefore, the project seeks to implement a new architecture known as the "application server herd", in which multiple application servers communicate directly with one another and the core database and cache. Inter-server transmissions can occur without having to take the time to go to the central database to get client GPS location updates, which improves performance. Additionally, the use of asynchronous programming via asyncio will improve performance in the face of large number of client connections, network requests to the Google Places API from multiple clients, as well as constant I/O between server and client and server and other servers.

1. Server Implementation

The "application server herd" consists of five servers (Goloman, Hands, Holiday, Welsh, and Wilkes) that communicate to each other bi-directionally, and accept TCP connections from clients. Clients send two types of commands: (1) IAMAT and (2) WHATSAT. Servers respond to IAMAT messages with an AT message, and respond to WHATSAT messages with an AT message, as well as data received from an HTTP GET request to the Google Places API (which is queried via the asynchronous aiohttp library). In addition, servers respond to invalid commands with a question mark (?) and a copy of the invalid command. Lastly, servers log information about their input and output, as well as new and dropped connections with clients and other servers.

1.1 IAMAT Message

Clients send IAMAT messages in order to inform the server of their current location. IAMAT messages consist of four fields: (1) the name of the command (i.e. IAMAT), (2) the client ID, (3) the latitude and longitude in decimal degrees using ISO 6709, and (4) the time that the client sent the message in POSIX time. Servers respond back to the client with an acknowledge message in the form of an AT message. AT message consist of six fields: (1) the name of the command (i.e. AT), (2) the server that got the message from the client, (3) the time difference from when the server received the client message from when the client sent the message, (4) the client ID, (5) the latitude and longitude, and (6) the time the client sent the message. In addition to responding to the client with an AT message; the server that received the message propagates the client location update to the other servers in the herd.

1.2 WHATSAT Message

Clients send WHATSAT messages in order to query for information about places near other clients' location. WHATSAT messages consist of four fields: (1) the name of the command (i.e. WHATSAT), (2) the client ID they want to query, (3) the radius (in kilometers) from the client, and (4) the number of results to receive from the Google Places API data within the radius from the client.

1.3 AT Message

Servers send AT messages to clients (in response to IAMAT and WHATSAT messages), as well as to other servers in order to inform them of client location updates. Servers are able to update the other servers in the herd by use of a simple networking-flooding algorithm to propagate the information from server to server.

2. asyncio

The "application server herd" architecture was implemented using asyncio, a Python asynchronous networking library. asyncio provides the infrastructure necessary to support asynchronous server-client and server-server TCP connections, as well as asynchronous I/O between the connections.

asyncio is desired for implementation of this architecture as the project consists of many processes that would be too slow to execute synchronously (i.e. establishing a large number of TCP connections, Google Places API queries, frequent client/server I/O, and propagation of client updates to other servers), and should be handled asynchronously to achieve sufficient performance. Using asyncio, time-intensive operations can be "paused", allowing other tasks to be scheduled in the meantime to better utilize idle time.

The main concepts of the asyncio library are the event loop, tasks, and coroutines. The event loop is a simple loop, in which only one task runs at any given time. A given task executes on the event loop until it executes some time-intensive operation (e.g. query to Google Places API), in which it "pauses" itself via the await keyword, and passes control back to the event loop to schedule another coroutine to run until the "paused" coroutine "resumes". Coroutines are subroutines that are scheduled on the event loop, and consist of "pause"- and "resume"-like methods to allow other coroutines to run, while performing some time-consuming task. Tasks are responsible for executing a coroutine object on the event loop, and wrap them in a future.

2.1 Usage in Project

There are two predominant APIs that were suitable to implement asyncio for this project: (1) callback-based API via transports and protocols, and (2) coroutine-based API via streams. I chose to implement the server via the callback-based API, but also utilized some of the coroutine-based APIs to communicate with the other servers. The callback-based API consists of three main callbacks: (1) connection_made (called when a new connection is made), (2) connection_lost (called when a connection is lost), and (3) data_received (called when the server receives some data from one of its connections). The first two are used to handle TCP connections with clients and other servers, while the last callback is used to handle I/O between them. Lastly, when flooding client location updates to other servers, several streams API methods (e.g. open_connection, write) were used to open connections with the other servers and to write updates to them.

2.2 Suitability for Project

The asyncio library is suitable for the implementation of the "application server herd" for the reasons described in section 0 and section 2, as well as several benefits that come with the asyncio library. First of all, it was quite easy to utilize asyncio to implement the server herd. The asyncio framework provides multiple APIs to implement the server herd (i.e. transports/protocols via callbacks and streams via coroutines), as well as easy-to-understand documentation on how to set it up. Moreover, when it comes to performance, asyncio performs fast-enough due to its asynchronous manner to be a suitable choice for the server herd in the face of a large number of TCP connection requests, I/O, and Google Places API queries.

While asyncio is a suitable choice for implementation of this project, and has its fair share of advantages, there are some problems that were encountered while implementing the server herd. Asynchronous programming is much more difficult to implement than standard synchronous programming; and similarly, debugging is also more difficult. Debugging took considerably longer than sequential programming for some portions of my implementation, and consisted of a large number of print statements to debug the program. In addition, asyncio doesn't provide strong support for the implementation of multi-threading as it is designed to be implemented in single-threaded concurrent code with most objects being non-thread-safe.

3. Comparison of Python and Java

One of the main concerns when choosing if asyncio was a suitable choice for the implementation of the "application sever herd" was Python's implementation of memory management, multithreading, and type checking as compared to other languages such as a Java-based approach.

3.1 Memory Management

Python implements automatic memory management as opposed to some other languages such as C++ (where programmers must explicitly de-allocate dynamic memory). Python handles automatic memory management by utilizing reference counting and garbage collection. Reference counting consists of counting the number of times that an object is referenced by other objects in the system. When an object is referenced, its reference count is incremented, while when a reference to an object is removed, its reference count is decremented. When the reference count of an object hits zero, the object is de-allocated. In addition, garbage collection occurs at scheduled intervals to remove objects with reference counts of zero from memory.

Java implements automatic memory management as well, but doesn't use reference counting. Instead, it employs garbage collection that consists of two main steps: (1) sweep through objects, and mark objects that are unreferenced, and (2) sweep through again, and delete objects that were marked in the previous step.

Java's implementation of memory management is superior to that of Python's for several reasons. The first is that reference counting can result in a problem known as a reference cycle, in which case the reference count of an object never reaches a value of zero. This can happen, for example, when two objects refer to each other; and thus, the memory is never de-allocated. In this case, a cycle detector is required which hinders performance. Moreover, aside from the problems associated with reference counting, it results in slower execution as each time an object is referenced/dereferenced, its reference count needs to be updated. Furthermore, Java garbage collection can be done on a separate thread from program execution, which reduces overhead considerably.

3.2 Multi-threading

In Python, multi-threading is impeded by the Global Interpreter Lock (GIL), which is a mutex that protects access to Python objects in shared memory, preventing multiple threads from executing on the object at the same time. This prevents true parallelism from being achieved, as multiple threads cannot operate on shared memory.

On the other hand, multi-threading in Java is easily attainable and Java features a class to achieve true parallelism. Due to Python's multi-threading inefficiencies, Java is a better option when it comes to multi-threading.

3.3 Type Checking

Python is a dynamically typed language, in which type checking occurs during runtime, and type is allowed to change over its lifetime.

Java is a statically typed language, in which type checking occurs at compile time, and programmers are required to declare the type of variables.

Being dynamically typed results in errors that can occur at runtime and can be more difficult to debug. On the other hand, statically typed languages yield errors at static time, and type errors during execution are less likely to occur. The dynamic type checking of Python allows code to be more flexible, compact, and easier to read however, while Java code is more constrained and clunky. Moreover, Python's prototyping speed is far superior compared to Java, and implementing the application via asyncio was easy to get up and running.

4. Comparison of asyncio and Node.js

Both asyncio and Node.js are event-driven asynchronous I/O models that both make use of an event loop with callbacks to schedule and execute tasks. Python is not asynchronous by default and employs asyncio to handle asynchronous code. Therefore, Node has the upper hand as it was designed to be asynchronous from the start, and everything is handled and thought about in an asynchronous manner. Moreover, Node ensures the same language on the client and server side. However, Node is not suitable for processor-intensive tasks, and since the application requires lots of connections and processing of a large amount of I/O, it is not as well suited to the project as asyncio is.

5. Conclusion

Based on the research conducted, the ease of prototyping in Python, and the ease of use and implementation of asyncio, I recommend the use of asyncio for the server herd. Python, as well as asyncio, have their fair share of disadvantages as compared to other languages and networking libraries, but both are well suited to meet the requirements for this project.