Proxy Herd with *asyncio*, Lab Report

CS 131, Programming languages

Zhouyang Xue

Abstract

The Goal of this lab project is to use *asyncio* to design a simple server herd where all servers share the same information and are equally accessible by an arbitrary client. The project utilizes the features of *asyncio* event loops and coroutines to achieve asynchronous behaviors of the herd, such as transmitting data, updating information, requesting and responding to connection, etc. This report goes over my design and implementation of the project, the challenges that I went into during the process, and a brief evaluation of *asyncio*'s performance on this particular application.

1. Introduction

asyncio is a Python library designed to implement asynchronous behaviors in applications. The goal of this project is to implement a server herd that each member is accessible to a client and also able to send request via Google Place API. asyncio's event-driven feature allows an update to be quickly made among the herd, thus is fully capable of the task. asyncio.Protocol is the base class for implementing streaming protocols, and is used as an infrastructure in the server implementations.

This project is based on Python 3.6.4, and imports aiohttp to use Google Place APIs.

2. Design

The server herd consists of five independent servers: Goloman, Hands, Holiday, Welsh, and Wilkes. Each server has its own client dictionary that keeps records of the information of clients of the whole herd. The related servers communicate through transport (sockets). The relation of the serves are as such:

"Goloman":["Hands", "Holiday", "Wilkes"]
"Hands":["Wilkes","Goloman"]
"Holiday":["Welsh", "Wilkes", "Goloman"]
"Welsh":["Holiday"]
"Wilkes":["Goloman", "Hands", "Holiday"]

Each server is also assigned with a port number:

"Goloman":16670
"Hands":16671
"Holiday":16672
"Welsh":16673
"Wilkes":16674

The herd listens to three categories of messages: IAMAT, AT, and WHATSAT, and respond correspondingly. The connection within the herd is activated once a client's information is updated. All the connection activities are logged separately for each server.

3. Implementation

3.1 Overview

The execution of the whole program is based on an event loop. *event_loop* is initialized in the main function, and is a global variable. Server construction and

connection are defined as coroutines to allow mutual communication under the *asyncio* environment.

Intuitively, I separated my program into two main parts. The first one is a protocol for connection and communication within the server herd. The second part is the server class, including all the methods of each server.

All protocols are based on asyncio.Protocol, which means each has three basic methods:

communication_made
communication_lost
data_received

communication_made is called when the object is connected, *communication_lost* is called the connection is shut down, and *data_received* is called when data is transmitted to the object from the socket.

Depending on the message received, servers, resource is passed over to three different methods: processIAMAT, processAT, and processWHATSAT, which would be explained in detail in the following.

3.2 processIAMAT()

processIAMAT() is in charge of parsing and processing IAMAT messages. An IAMAT message has 4 parts: [message type], [client], [location], and [issue time]. Note that in python3, objects are dynamically typed, I took advantage of this feature to calculate the time difference, where the issue_time is properly mapped to float.

I check if the IAMAT message is legal in processIAMAT(). If so, an AT message is generated, and passed to processAT(). Otherwise, an exception is triggered to log the illegal message and to print out the warning.
The AT message is also written back to the socket as a response to the client.

3.3 processAT()

processAT() is implemented to process AT messages. Its main job is to update the client information that is stored in each server within the herd.

An AT message contains 6 components: [message_type], [server], [process_delay], [client], [location], and [message_issue_time].

At the very beginning, I check if the AT message is legal. If not, I pass it to the illegal message exception. Otherwise, run the normal routine.

To update the information stored in each server, the function first checks if the current information of this client is missing or out-of-date. If not, there will be no need to update, and thus the function instantly returns. Otherwise, the information needs to be updated.

First, I update the client info stored in the current server. Then I go over each server that is associated with the current server, and propagate the AT message to trigger the their processAT() methods. I met a major challenge at this step, where via a two-way connection, the two servers infinitely propagate the AT message to each other. I overcame this problem by comparing the client information in advance. If the received message is the same as that stored in the client information, then neither should the server update the information, nor should it pass a new AT message to the servers it associates with.

To propagate messages within the herd, I generate a new TCP connection between the servers with a coroutine in the event_loop -- create_connection(). As soon as the AT message is received, I close the socket because message processing won't be using the socket.

3.4 processWHATSAT()

This method is, with no doubt, the trickiest among the three. Its goal is to parse the WHATSAT message, form the url to access Google Place API, and set up the connection to the API.

I first check if the WHATSAT message is a legal message. Note that I also made sure that the radius is no more than 50km, and the upper bond of places is no more than 20. After validating the message, I generate the url with the format given on the Google Place webpage.

Once generated the url, I used ClientSession() from the aiohttp library to set the connection to the Google Place server. The challenge is both processWHATSAT() and Google_place_connection() have to be defined as *async* functions. *asyncio* largely improved the performance of processWHATSAT(), where multiple input and output are processed by the server heard.

The output from the Google Place API is in json format. I defined and called parsing_json() to parse and generate the required json output and respond to the client.

4. Comparison

4.1 Python vs. Java

4.1.1 Type checking

Python is a dynamic and strong typed language. That is to say, variable names are bounded to objects rather than certain types. This feature has its own pros and cons. The advantage is that in this particular project where a significant number of messages are processed, dynamic typing makes life much easier for the programmer, because the messages can be easily parsed and casted into certain types when necessary. On the other hand, this advantage has its own flaw. That is, it makes debugging more difficult.

However, Python is also a strongly typed language. That being said, incompatible types cannot be in one operation. For example, "a" + 1 would return TypeError. Therefore, Python made type checking safer compared to weakly typed languages.

My conclusion is that I definitely recommend Python's asyncio for this project. It's easy to write with and the code is easy to understand. The only tricky part is that the programmer needs to be very careful about logical errors where the dynamic type checking cannot find during compile time.

4.1.2 Memory management

In terms of memory management, Python utilizes a mixture of reference count and mark-and-sweep garbage collection. This feature makes life much easier by deleting any unreferenced objects at once. In this case, once transports or internal connections are no longer referenced, they would be deleted by the system to prevent memory leak.

Java, on the other hand, does not guarantee such memory safety. Therefore,

programmers need to be very careful about allocating memory on the heap. Moreover, using Java's garbage collector comprises the performance of the server herd.

4.1.3 Multi-thread programming

The advantage of Python's asyncio library is that it enables multiplexing and concurrency operated on a single thread. It avoids the delay when CPU/routine waits for certain inputs to run. Instead, it's based on an event loop, within which one action gives up the resource and yields to others while waiting for inputs. This feature is critical to the server herd's performance, for a large amount of synchronization and update goes on after one member server receives a message from the client. On the contrary, for Java, which is designed to optimize multi-thread programming, asyncio is not a good option, because it's very likely to cause correctness issues based on dependency.

5. *Asyncio* vs. Node.js

Node.js is developed from Javascript. It uses "an event-driven, non-blocking I/O model that makes it lightweight and efficient." From this perspective, I would say it's similar to Python's asyncion. Both are designed to optimize single-thread program through concurrency. However, they have their own privileges and disadvantages over one another.

Some people find Node.js appealing because Node.js is very similar to Javascript, which is a popular scripting language for web developing. Thus, to implement a web-related program that involves API requests and socket programming, Node.js seems to be a fair choice for them.

However, compared with Python, Node.js is much less stable. That being said, the language is evolving quickly, and it takes a risk that the older version code might not be compatible any more in the near future.

Overall, I would recommend Python's asyncio framework to implement hardcore networking APIs, as well as server herds.

6. Conclusion

All languages have their pros and cons, so is Python's asyncio framework. It's not perfect, but absolutely capable of the implementing the server herd. It's easy to write, and has great performance over single-thread operation server herd. Although it makes debugging very hard, and includes tricky features like "event_loop" and "coroutin", I still find it a great option to implement the server herd.

7. Reference

[1] Asynchronous I/O, event loop, coroutines and tasks
https://docs.python.org

[2] Node.js
https://nodejs.org/en/

[3] Synchronous and Asynchrounou I/O
https://msdn.microsoft.com/en-us/library/windows/desktop/aa365683(v=vs.85).aspx