Proxy Server Herd Application Project

CS 131
Bingxin Zhu
UID:704845969

Abstract

We want to build a new Wikimedia-style service designed for news, where updates to articles will happen far more often, access will be required via various protocols, not just HTTP and clients will be more mobile. Thus, we are asked to explore an architecture called an "application server herd". In this architecture, multiple application servers communicate directly to each other via the core database and caches. In this paper, I will use the asyncio Python library to implement the structure.

Introduction

We have to implement an application server herd and also make it being able to propagate so that they might all be in-sync. Since asyncio event-driven nature should allow an update to be processed and forwarded rapidly to other servers in the herd, so choosing to use the Asyncio framework with Python made it such that the task didn't end up being so difficult. This report will summarize my research on asyncio and justify my recommendation on the this framework. I will also mentioned the problems that I ran into and also address the about Python's type checking, memory management, and multithreading, compared to a Java-based approach to implement this framework. Finally, my report will also briefly compare the overall approach of asyncio to that of Node.js.

Application Design

Overall Implementation:

The application was written in server.py file using the Asyncio which is a Python library for asynchronous computation. It is used to handle events that don't come in any particular order (like TCP connections). The server is build around event-driven concurrency. The event loop provided by asyncio allows my program to sleep when there is no data processing and wake up when there is a task. The event loop behaves like a scheduler to schedule all routines. Asyncio provides approaches to provide good performances as well as avoiding its data races by transferring control at some specific points. One is that it could use scheduler to schedule executions of routines, this is called callback-based Protocols. Another one is called coroutine-based Streams, which ignores coroutines when they are waiting for data for IO. My project uses the first one. Servers are able to handle the data it receives, check the correctness and if it is valid, server will propagate data to neighbors. The loop is a high level to generate the server protocols. The client side of the server is much more simpler, it only needs to send the data through the TCP socket. Specifically, throughout the implementation, I should implement 3 functionalities:
1 Handling IAMAT
2 Handling WHATSAT
3 Handing AT messages which are used for propagation

1 Handling IAMAT messages

example IAMAT message:
IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1479413884.392014450
IAMAT messages should be sent from a client and the format should be "IAMAT {client identifier} {location} {client timestamp in UTC format})"

Firstly, I split this message into three parts and then make sure it is valid. Since Python is dynamically typed, object type is checked at run time. Here comes a concept of duck typing. Python will check to see if the operation is valid corresponding to the object's type. I use this to check client timestamps, and locations including latitude and the longitude. They should all be valid floats in this case.
This type of explicit casting of the time and coordinates is different as the java static type checking.
If the IAMAT message is correct, then we are supposed to translate it to AT message. An example of AT message should be:
AT Alford +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1479413884.392014450
AT message adds a time difference and server name to the original IAMAT fields. Server will transport this message if it makes sure that this is not an infinite loop, which means the message is a new message and it should also update its knowledge base. The propagation is done by a new protocol and thus generate a TCP connection to sent data.

2 Handing WHATSAT Messages

WHATSAT messages are used to query the nearby places within the given radius by using Google Places API. An example of WHATSAT message is
WHATSAT kiwi.cs.ucla.edu 10 5
Error check:

(1) Make sure the parameter is correct number
(2) Radius are in valid bound

The main idea is to implement GET on TCP. I use the Python json library. The main steps are :

(1) Parse the url, separate the url into URI and hostname
(2) put it together into a correct format into an HTTP GET request
(3) send this request using asyncio
(4) get the response with asyncio
(5) identify the response body
(6) handle that jso

If we are doing the similar thing in Java, it will use a another thread, and thus achieve it through the use of a multiprocessor system. But this is not what Python wants to do. Python was designed to run within a single thread, it lacks of concurrent programing models compared to Java. That is why we use event loop to schedule events. However, it is actually an advantage since we are able to use non-blocking based parallelism more effectively in this case.
We then cache the AT message and JSON response into local server cache.

3 Handling AT messages

AT messages are first created by server to respond to a client's IAMAT message. After that, those messages will be propagated to the server's neighborhood and in order to not form an infinite loop, we have to carefully check if the AT message is in the server's local cache before it is updated into local cache. I was originally come up with the idea which is to change the AT message itself but I failed because I noticed that I should also propagate it to other servers, so it is bad to change the message.
Finally, I used a cache and check the message which is determined by the timestamp within the message. If the value if not the same, it still need to be propagated.

Conclusions

I will briefly address:

(1) Summarize asyncio
(2) Compare Python's type checking, memory management, and multithreading, compared to a Java-based approach to this problem.
(3) Compared to Node.js

(1) Summarize asyncio

Asyncio provides infrastructure for writing single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives. It is actually an advantage since we are able to use non-blocking based parallelism more effectively in this case.

(2) Compare Python's with Java

(i) Type checking:
Because Python is dynamically typed language, programmer don't need to worry about the type of objects that they want to use and thus it is easy to implement. Java is statically implemented, compared to Python, it will be less readable but more clear. Since Python's interpreter will use duck typing to figure out what type is the object based on its contextual usage. To conclude, it is easy to code and read if using Python but will be more confuse to understand the detail.
(ii) Memory management:
Python uses a mixture of reference counting and non-moving mark-and-sweep garbage collection. Thus, in this project, this feature of Python makes sure that once the server herd we create are no longer reference, they will be deleted as soon as possible. While programmer who using Java in this case have to worry about allocating objects on heap.
(iii) Multithreading:
Asyncio is better fit for this project is because Python's single threaded implementation which I already explained in conclusion part 1. Java uses multithreaded and relied on the machine. If you implement this project in Java, you may have to worry about the performance while running on different machine architecture, but Python gives you the same performance so that it can benefit a lot more through horizontal scaling which is what we want for a server herd.

(3) Compared Python to Node.js

Both Node.js and Python's Asyncio framework are asynchronous, single-threaded event driven frameworks for networking. They are all easy to read and write.
Node.js is built on top of Javascript and is supported in massie community. Thus, Node.js provides a lot of useful libraries and is simpler which achieving its core functionality. But Node.js is not an object oriented framework since ES6 and ES7, so it is hard to modularize compared to Asyncio framework. Also, Node.js is still evolving rapidly so it is less reliable compared to Python.
Node.js is a better choice for web based server side applications, while Asyncio is better implementing hardcore networking API's and server residing applications. Thus, for application server herd, I recommend using Python.

References

[1] https://docs.python.org/3/library/asyncio.html

[2] https://www.guru99.com/node-js-vs-python.html

[3] https://www.quora.com/What-are-the-differences-between-Python-and-Java-memory-management