CS131 Project Report

Chaoran Lin, University of California, Los Angeles

ABSTRACT

This report examines the usefulness of the asyncio asynchronous networking library in Python with regards to implementing an application server herd architecture for a new Wikimedia-style news service. First, it offers a brief high-level overview of the implementation of this architecture using Python, and any difficulties that were met aloing the way. Then it discusses the features of Python, such as type checking, memory management, and multithreading, compared to Java. It also describes the process of building the prototype that was used to test out this Python-based approach, along with any problems that were encountered during the process. Finally, it compares asyncio to that of Node.js, and provides a final recommendation of the plausibility of using asyncio.

1 Introduction

1.1 Application Server Herd Architecture

An application server herd architecture is a model that involves multiple application servers communicating directly with each other as well as a core database. The data communicated between the servers will be of many different categories and sizes, while the database will be reserved to store more stable, less often accessed data. When a client communicates with these servers, it will only need to post its data to any one of the application servers, and the other servers will also receive the data through interserver communication, without needing to look at the database.

1.2 Python and asyncio

Python was used as the programming language to implement the application server herd architecture in this project, specifically Python 3.6.4. The testing server was lnxsrv07.seas.ucla.edu. The specs of the testing server were:

- CPU: Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
- RAM: 64GB

For communication between servers and clients, asyncio was utilized and examin. asyncio is a Python library that takes an event-driven approach towards supporting asynchronous I/O over sockets and running network clients and servers. An asynchronous event, such as a TCP connection, which is what our server architecture will be communicating via, means that it does not occur sequentially along with other asynchronous events.

1.3 Google Places

Google Places is an application developed by Google Inc. that allows a client to search for destinations within a certain radius nearby among other functionalities. In this project, an API key to Google Places was used so that the server could perform calls to the Google Places API to act as a middleman between the client and Google Places.

2 Implementation and difficulties

2.1 Overview

The project consisted of two files: server.py and client.py. The server.py file contains a design for a server within an application server herd architecture. In particular, it is able to receive connections opened by a client, receive information coming from that client, and based on that information, propagate a response to not only the client but also other servers that it can communicate to. The client.py file contains a simple prototype of a client that sends and receives one command, and is not meant to represent a fully implemented client with the ability to issue multiple commands. Rather, its primary function is to emulate the testing environment this project will be put in by testing the compatibility of the server with a Python-based client, implemented with asyncio. In implementing these two files, the asyncio library in Python provided a quick working template to implement such functionalities, as its event-driven nature was perfectly suited for this context.

2.2 Server Design

I chose to contain the server implementation in a class in order to maintain encapsulation of the server data, and also provide an interface that was easy to use. The server.py consists of a Server class that accepted several parameters: a server id, a hash table named port_of_server that mapped server ids to port numbers, and a hash table named can_communicate_with that mapped server ids to lists of server ids they could communicate with. The two hash tables were defined within server.py as follows:

- port_of_server = { "Goloman": 17345, "Hands": 17346, "Holiday": 17347, "Welsh": 17348, "Wilkes": 17349 }
- can_communicate_with = { "Goloman": ["Hands", "Holiday", "Wilkes"], "Hands": ["Goloman", "Wilkes"], "Holiday": ["Welsh", "Wilkes"], "Welsh": ["Holiday"], "Wilkes": ["Goloman", "Hands", "Holiday"] }

The Server class took these parameters and stored them in class variables. It also initialized an internal hash table named communicated_clients, in which it can look up a client id and find all the information of when the client at that client id talked to one of the servers in the network. Finally, it creates a local file to write to in the future.

The Server class contains a "master" handle_echo method that serves as the entry point of the server. It receives a reader and writer that are passed into its parameters when the server is started via the asyncio.start_server call. It awaits for the reader to read a message from the client, parses it, and checks whether the first word is either a "IAMAT", "WHATSAT", "AT", or invalid type of message. Based on what type of message it is, it delegates the handling of these message to other class methods, to be explained below.

Once a Server class was instantiated, I ran the server on a loop while serving requests until a KeyboardInterrupt is detected.

2.3 Responding to IAMAT messages

The server handled these messages through the respond_to_IAMAT method. The method takes in a client_message and a writer as parameters, and parses the client_message, obtaining the key information: the client id, the latitude and longitude in decimal degrees using ISO 6709 notation of the client, and the ISO time at which the client sent the message. It responds with an "AT" message which contains the server id, the difference between the time the server received the message and the time the client sent the message, and the client information, and logs this response. When it receives the client information, it stores it in the communicated_clients dictionary mentioned earlier. It also sends the same "AT" message to the servers it can talk to.

2.4 Responding to WHATSAT messages

The server handled these messages through the respond_to_WHATSAT method. The method takes in a client_message and a writer as parameters, and parses the client_message, obtaining the key information: the client id, a radius (in kilometers) from the client, and an upper bound on the amount of information to receive from Places data within that radius of the client. It responds with an "AT" message which contains the server id and the client information last received by checking the dictionary previously mentioned, as well as using the information provided by the client to make an HTTP request to the Google Places API (using the aiohttp library), and sends the response in JSON format back to the client. Note that if the server never received previous information about this client (by checking the dictionary), or if the upper bound of results specified by the client is above 20, then it will treat the WHATSAT message as an invalid message and delegate to the method for handling such messages. Otherwise, it sends the message back to the client and logs this information, as well as propagating this information back to the other servers, but without the JSON.

2.5 Responding to AT messages

Servers should also handle AT messages. These are typically messages that are broadcasted between servers, and represent propagated information that should be stored. Since a server only needs to update its internal dictionary with this information and does not need to perform any asynchronous I/O, the respond_to_AT method that handles these messages was not declared as a coroutine with an async keyword.

2.6 Responding to invalid messages

The server handled these messages through the respond_default method. It simply sent back to the client a message beginning with "? " and followed by the original client message, and logs this information. It does not propagate this information to the servers as there are no location updates to be shared.

2.7 Logging

The server logged what messages it sent in the format "Server SERVERNAME sent to client 'MESSAGE'" or "Server SERVERNAME1 sent to server SERVERNAME2 'MESSAGE'", what messages it received in the format "SERVERNAME received 'MESSAGE' from (HOST, PORT)". It also logged new connections using the format "New connection to server SERVERNAME was made" and dropped connections using the format "Connection to server SERVERNAME was dropped". It does this inside the write_to_log method.

2.8 Client Design

I also implemented a simple client using asyncio in the client.py file. It takes a server id as an argument, opens a connection to the port matching that server id, sends a message, and closes the connection upon receiving a response from the server. I based it off the template in the Python documentation, under "18.5.5.7.1. TCP echo client using streams". As mentioned previously, this client implementation is merely designed to represent a prototype for what would be a full client implemented with asyncio in the actual testing environment, and test how well server.py worked with it.

2.8 Testing

Testing was done locally as well as on the SEASNet servers. The typical setup was to open 5 windows in the terminal, in which each server was started up. I then opened a client in another window, either through running my client.py file or using the telnet Linux command, and sent/received messages to/from the server. Initially, before implementing the logging functionality, I had the server output what commands it sent and received with the print function. This helped me gain a clearer overview of what was happening between the client and server. Once I assured that there were no issues, I changed it to logging the messages in a file instead.

2.9 Difficulties

The documentation for the asyncio library provided several useful templates to quickly implement a basic version of the server, so it was relatively easy to hit the ground running. I based my server design off the template in "18.5.5.7.2. TCP echo server using streams". Then I decided to convert this design to a more object-oriented model using a class, which is when I ran into my first difficulty; I was facing errors of incorrect argument count, when interacting with the methods in the class. I fixed this by adding a self parameter to every method in the class. Although this difficulty was addressed quickly, it was representative of my initial rustiness and unfamiliarity with Python syntax and semantics.

Another difficulty I ran into was the problem of determining what indicated a dropped connection from the server. I solved this problem by observing what my client.py output when I forcefully shut down the server it connected to, and I found that a ConnectionRefusedError was what indicated it. I then incorporated this into a try-exception block in my code.

There was also the issue of how to limit the results returned in the API call to what the client specified. I was unable to find any reference in the Google Places API that mentioned the option to include a "limits" or "max-results" parameter that would limit the results, however I was able to check the condition that the max results fell under 20.

Finally, I also had some difficulty initially trying to figure out what functions should be declared async. Fortunately, I found some help on Piazza as a student and TA responded that whenever an async function was called, it would return a coroutine. I used this information to help determine what functions I should make async.

3 Python features compared to Java

3.1 Type checking

Python is a dynamically typed language, meaning that every variable is bound only to an object, at execution time. It is also possible to bind variables to objects of different types throughout the program. On the other hand, Java is a statically typed language, meaning every variable name is bound at compile time. Once a variable is bound to an object, it is assigned the type of that object, and cannot be bound to other types.

Python's dynamic typing feature certainly gives it a greater advantage in terms of flexibility over Java. It also decreases the likelihood of syntax and semantic errors, as most of the errors would probably occur as logic errors. In comparison, Java's static typing means that it offers better performance, as compilers can take advantage of it to perform optimizations, as well as stricter determination of variable types that makes it more attractive from a debugging standpoint.

Overall, there are tradeoffs to both languages when it comes to type checking. However, Python's flexibility means that a developer can implement a solution for this project in Python at a more convenient and quicker pace compared to Java, even if this may introduce some inconvenient errors. In our situation, this is a big plus as we want to scale our application quickly and conveniently. The fact that modern backend frameworks are also written in JavaScript (such as Node.js), another dynamically typed language, leads me to recommend Python in this situation.

3.2 Memory management

All objects and data structures in Python are managed through a heap. The heap memory is managed internally by a Python memory manager through garbage collection. In deciding what memory is cleaned, Python uses reference counting, so an object that has no more uses (references) will be destroyed and its memory freed by the Python memory manager.

Java also uses a garbage collector to manage its memory. The difference is that Java destroys objects that are no longer used at some specific time.

There are several tradeoffs that are open to discussion here. For one, Python's garbage collection method presents some drawbacks in performance, as the interpreter must check the reference count of an object whenever it is referenced/dereferenced (and remove it when the count reaches 0), slowing down execution. Java's garbage collection does not pose this problem.

On the other hand, Java's periodic collection of unused memory means that there will be certain instances where we have an excessive amount of unused memory waiting to be collected, thus slowing down the performance of the application. In this sense, Python's immediate removal of unused objects is much more beneficial.

In terms of memory management, both Python and Java offer performance optimization in some areas at the expense of other areas. So, neither can claim to have a significant advantage over the other, and it comes down to simply a preference for a particular design.

3.3 Multithreading

Python is able to spawn multiple threads via the threading module API. In terms of protection, in CPython, the reference implementation of the Python language, there exists a mutex global interpreter-level lock, or GIL, that protects access to Python objects. However, this comes at the inconvenience of other features having to depend on the guarantee that the GIL enforces. Furthermore, this effectively means that parallel execution of threads in Python becomes implausible, since any running thread must have acquired the GIL, and only a single thread can hold the GIL at any time. Thus, all threads would have to run interleaved with each other since each one needs to wait for the GIL to become available. This is not a serious issue on single-core machines, but on multi-core machines it presents a bottleneck. However, this can be resolved by having multiple interpreters, since GILs are interpreter-specific.

On the other hand, Java offers a more versatile set of tools to achieve multithreaded and concurrent programming, such as java.util.concurrent, java.util.locks, and java.util.atomic. Threads in Java are either implemented and handled by the OS or the Java Virtual Machine (JVM). This presents a sense of uncertainty as the developer can no longer completely control the performance of a multithreaded program; instead, that is left to the specific OS and machine that the program is run on.

Most other drawbacks of multithreading are common to both languages. In most cases, one would prefer Java for multithreading purposes due to its versatility and performance advantages, and the exception is when we have multiple interpreters available in Python.

4 Python asyncio vs. Node.js

Node.js is a JavaScript runtime built on Chrome V8's JavaScript engine. Much like Python's asyncio library, it is event-driven and allows for asynchronous, non-blocking I/O. However, there are several differences. For one, Node.js is implemented so that all code runs only on a single thread, compared to the multithreaded possibility of Python. This presents some tradeoffs: Node.js's single-threaded nature means that we can avoid the complexity of debugging multithreaded code, as well as concurrency issues such as deadlocks and race conditions. On the other hand, if we want to fully utilize more computationally advanced hardware, we should take advantage of Python. Furthermore, the asyncio library also provides event loops that emulate the single-threaded approach of Node.js, and it is also what I used to implement my server. Additionally, Node.js is not object-oriented, so the need for code modularization should be considered thoroughly before deciding which one to use.

5 Conclusion

Over the course of this report we have been able to examine several pros and cons of using Python and asyncio to implement the application server herd architecture. The flexibility and convenience offered by Python is certainly a huge advantage, as it allows the architecture to be quickly implemented and scaled. Some concerns I found were mainly concerning performance issues regarding multithreading performance in Python under some situations. However, these are relative to Java, a language that is rarely used to implement backend code currently, especially for lightweight ones like ours. When compared to Node.js, Python also has several important advantages. Overall, Python strikes a balance between our current needs for this architecture, and it is probably the best all-round solution. Therefore, I would recommend using Python and asyncio for this project.

6 References

[1] *Asyncio — Asynchronous I/O, event loop, coroutines and tasks,* https://docs.python.org/3/library/asyncio.html

[2] *Google Places API,* https://developers.google.com/places/

[3] *GlobalInterpreterLock,* https://wiki.python.org/moin/GlobalInterpreterLock

[4] *Memory Management,* https://docs.python.org/2/c-api/memory.html

[5] *Understanding Asynchronous IO With Python 3.4's Asyncio And Node.js,* http://sahandsaba.com/understanding-asyncio-node-js-python-3-4.html