Suitability of *asyncio* for implementation of server herds

Aashita Patwari – University of California, Los Angeles

Abstract

There are many different approaches to implementing the architecture of an application server. One example would be LAMP, an approach that the Wikimedia Architecture is based on, that uses multiple, redundant web servers behind a load-balancing virtual router for reliability and performance. However, that is not suitable for this particular application that requires frequent updates, access via various protocols apart from HTTP and mobile clients. This is because the central Wikimedia application server will become a bottleneck. Hence, we are investigating the implementation of a different architecture – an application server head, where multiple servers communicate with each other directly and via core database and caches. The prototype is implemented via Python's asyncio asynchronous networking library to discuss its suitability as a replacement for the Wikimedia platform.

Introduction

The architecture of an application server head allows it to be a better replacement for the Wikimedia platform as it reduces the central bottleneck. With an application server head, there is both direct and indirect communication. Direct communication between servers is used for rapidly-evolving data ensuring that the core database is not accessed, and indirect communication through the core database and caches is used for more stable data.

In this paper, I will primarily focus on how I have used Python to implement a prototype for the application server head and handle different types of messages, and the advantages and disadvantages of it from a language point of view. Following that, I will also compare using Python against using Java and asyncio against the overall approach of Node.js.

1. Design

The prototype consists of five servers with a stipulated bidirectional communication pattern, that is specified under the *servers* dictionary in server.py. Each server accepts a TCP connection from clients. The functions that are used by the servers are also implemented in server.py where a main *process_command* function is called when the server is started. The server is looped to run forever until there is a keyboard interrupt to close the server. I initially used another approach where I created a ServerProtocol class that was implemented when a server was created, and all the required functions were contained within this class. However, I felt that it complicated the implementation and that the built-in methods such as *connection_made* and *data_received* were not necessary. Hence, I decided to remove the class.

process_command subsequently reads the data and calls another function to process it. There are 3 types of messages that the data could contain – AT, IAMAT, and WHATSAT – handled by *process_at, process_iamat,* and *process_whatsat* respectively. Other important functions include *error_message,* which handles invalid messages, as well as *flood,* which implements my flooding algorithm to send the message through every outgoing link except the one it arrived on.

IAMAT message processing

IAMAT messages are used by a client to send its location to the server. The format of it is as follows: IAMAT <client-id> <coordinates> <send-time>. A client ID can be a string consisting of any non-white-space characters. The coordinates are in decimal degrees using ISO 6709 notation and the time is expressed in POSIX time.

The server responds to this with an AT message of the following format: AT <server-id> <time-difference> <client-id> <coordinates> <send-time>. The server-id is the ID of the server that got the message from the client, the time-difference is the difference in time between when the server got the message and the 'send-time' according to the client. The following 4 fields are identical to those received by the IAMAT message.

In order to implement this, the *process_input* function checks if the format is correct – that there are 4 arguments, the time is a float value and the coordinates are proper coordinates with two values that can be positive or negative. If these conditions are fulfilled, the *process_iamat* function is called, else the *error_message* function is called.

The *process_iamat* function calculates the time difference and constructs the server's response using data received from the IAMAT message. This response is encoded and then written using the writer passed into *process_iamat* and is logged in the receiving server's log file. The coordinates, send_time and the server's response is also stored in a *clients* database that is in the form of a dictionary that is indexed using the client_id. Following this, the flood method is called to flood the server's response to other servers it communicates with.

WHATSAT message processing

WHATSAT messages are used by a client to query for information about places near other clients' locations using the Google Places API. The format of it is as follows: WHATSAT <client-id> <radius> <amount of information>. The radius is in kilometers and it must be at most 50km, and the upper bound on the amount of information it receives from Places data should be at most 20.

The server responds to this with an AT message of the following format: AT <server-id> <time-difference> <client-id> <coordinates> <send-time>. This is of the same format as the IAMAT message response. This response gives the most recent location reported by the client and the corresponding details of that report. The AT message is also followed by a JSON format message followed by two newlines. This JSON message is in exactly the same format as that given by Google Places when a Nearby Search request is made and includes information such as the name and icon.

In order to implement this, the *process_input* function checks if the format is correct – that there are 4 arguments, the radius and information bound are within the stipulated range, and if client-id that is enquired about is within the database. If these conditions are fulfilled, the *process_whatsat* function is called, else the *error_message* function is called.

The *process_whatsat* function constructs a url to query the Google Places API and receives the required json information by using a aiohttp.ClientSession. The amount of information is extracted from the result and then formatted. This final response is then written using the writer passed into *process_whatsat* and is logged.

AT message processing

AT messages are sent between servers when a server's response to an IAMAT message is being flooded to other servers. There is no response that is sent in this case, it is instead flooded to clients that have not received the message, or if the message has a send time greater than the one stored in the database, suggesting that it was sent again with newer information.

Once again, *process_input* makes sure the AT message is of a correct format as stated under IAMAT message processing, and calls *error_message* otherwise.

Flooding algorithm

The flooding algorithm I have implemented is fairly straightforward, where the *flood* function iterates through the server's peers by checking the *servers* dictionary mentioned earlier. After checking that the peer and the source is not the same, ensuring that the message is not flooded back to the link it arrived on, the function opens a connection and writes the encoded message. There is a message logged both when it is successful and when it is unable to flood when the server is not running. The checks inside *process_at* before the *flood* function is called ensures that messages are not sent indefinitely in a cycle.

Error message processing

If any of the conditions that were discussed earlier fails, this function is called where a question mark is prepended to the received message and written back. A log message is also recorded.

2. Evaluation of *asyncio*

From the prototype we can see that the framework provided by asyncio greatly eases the process of running servers that can handle requests asynchronously. The asynchronous nature allows the server to process multiple requests at a time as it does not have to wait for one request to complete before processing another. The event loop keeps track of the different I/O events and switches between them to ensure time is not wasted on tasks that are not ready. This allows for improved performance.

A disadvantage of using asynchronous programming is that messages are not necessarily processed in the order that they arrive. This is demonstrated when the program is run with the flooding algorithm where the message is received twice even though it is the same message. This would not occur if the messages were sent between servers sequentially. However, in this application, this does not affect the overall result and hence is acceptable.

3. Python and Java comparison

The comparison is done for 3 main categories – type checking, memory management and multithreading.

Type-checking

Java is statically typed whereas Python uses dynamic type checking. While static typing allows for more clarity in understanding the implementation, dynamic type checking helps to abstract the details and makes it easier to make code that compiles properly, thus easing the process of building the program. For example, the programmer does not have to know the type returned by *asyncio.start_server* and can simply pass it in as an argument to another function. However, it makes it trickier to debug as an incorrect output might be due to a wrong type being passed in. For example, in my implementation, forgetting to convert the send time into a string while formulating the server response lead to the program still compiling but not writing the server response as required. In this specific implementation however, the ease of implementation due to abstracting the details of asynchronous programming is preferred over ease of debugging.

Memory management

Java and Python use different garbage collecting systems. Java uses a mark and sweep method while Python primarily uses reference counts. In mark and sweep, the memory is swept from the root to detect reachable blocks and marked. Non-marked blocks are then removed. In reference counts, each object has a count associated with it, which is incremented when there is an additional reference to it and decremented when the reference is removed. If this count reaches 0, the object is removed.

The advantage of using reference counts over mark and sweep is that garbage collection is more immediate as the program does not have to wait for a mark and sweep before any unreferenced object is deleted from memory and is a less expensive operation. This implementation however, causes assignments to be more expensive due to an additional updating operation. This could be a potential issue in this application where a lot of variables are being created.

Multithreading

Python uses a Global Interpreter Lock, a mutex that protects access to Python objects in concurrent execution, for multithreading. Java, on the other hand, has strong multithreading support that allows for great improvements in performance as compared to Python due to maximizing parallelism.

4. asyncio and Node.js comparison

Both Node.js and asyncio are event-driven asynchronous approaches that use an event loop. JavaScript is also dynamically typed like Python. However, Node.js is built on the V8 engine which makes it have fast performance and make it more scalable. It is also built to handle asynchronous I/O whereas in Python, asyncio is a library that an exception to how Python usually runs.

Node.js may also be preferred by developers because the front end of applications are often built using JavaScript and hence building both the front-end and back-end using JavaScript will make it easier to integrate.

Lastly, Node.js is a newer language and hence is able to correct mistakes made by older languages without backwards-compatibility concerns.

5. Conclusion

From the above analysis, we can see the asyncio is a powerful framework that is relatively simple to use and implement to build an application server herd. The asynchronous nature makes it suitable for this specific use as we want to be able to process several requests without a performance bottleneck. However, a brief investigation into Node.js shows that it might be a better alternative to Python as it is inherently asynchronous and could perform better. It is also inherently more practical from a developer's point of view as both the backend and frontend could be built using the same language. However, in order to make a more informed judgement, we have to build a similar prototype using Node.js and test and compare the performance of both prototypes.

References

asyncio – Asynchronous I/O, Retrieved Mar 10, 2019 from https://docs.python.org/3/library/asyncio.html

About Node.js, Retrieved Mar 10, 2019 from https://nodejs.org/en/about/

Intro to Async Concurrency in Python vs. Node.js, Retrieved Mar 10, 2019 from https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36