Project Report - Proxy Herd with ‘asyncio’

Raymond Lin
CS131 Programming Languages - Fall 2019

Abstract

For this project, we were in charge of developing a server program that serves as a node in a server herd, using a Python library called ‘asyncio’, a library that allows for code to be run asynchronously and for multiple connections to be maintained concurrently. We were also asked to compare and contrast the asynchronous technologies, Node.js and Python ‘asyncio’, as well as the programming languages Python and Java.

1. Introduction

Wikipedia and its related sites use the Wikimedia architecture, which is implemented using a LAMP platform. LAMP stands for Linux (Operating System), Apache (server), MySQL (database), and PHP (server side language). This type of architecture is a traditional method of launching and running a typical web application. However, this type of architecture is unsustainable if there are a lot of requests for updating articles. The application server will become a bottleneck, as all the incoming requests will have to go through that single node.

A solution to this issue would be to have multiple application servers running at any given time. Clients can connect with any of the nodes, and depending on the type of request or information sent to the server, the server will either serve the request or propagate necessary information to all other nodes. This type of architecture is called an “application server herd”. Each node in the server herd can only contact certain other nodes, clients that connect to them, as well as the database. We assume that there is always at least one path from one node to any other node in the server herd (or else messages would not be able to be propagated to the entire herd).

Furthermore, we would like each node in the server herd to act as if it were on its own too. Thus, each node in the server herd should be able to support concurrent connections from multiple clients. This is where the Python ‘asyncio’ library comes in handy, as it contains functions that allow for concurrent connections, as well as the definition of coroutines that can be run concurrently with other code.

By implementing a node of a server herd using ‘asyncio’, we can learn about its advantages and disadvantages and compare it against other technologies that can fulfill this task, such as Node.js. and Java.

2. Python ‘asyncio’ Library

The ‘asyncio’ library allows us to write concurrent code. According to the documentation, the library is used for many purposes, such as web-servers, database connection libraries, and distributed task queues. It contains a set of high-level APIs that run coroutines concurrently, establish connections between processes for interprocess communication, controlling sub processes etc… It also contains a set of low-level APIs for library and framework developers to manage event loops and utilize transports for efficient protocols. It is important to note that ‘asyncio’ was designed to run on a single thread.

2.1 Syntax

The ‘asyncio’ library changes the syntax of the code, mainly by including the usage of two new keywords: ‘async’ and ‘await’.

The ‘async’ keyword is typically placed in front of function definitions, in order to specify that that particular function is the be run as a coroutine. The function then cannot be called regularly, but instead, it must be added as a task to some event loop or called using ‘await’. Then, it runs concurrently with the code that is sequentially next.

The ‘await’ keyword is used to call coroutines. If we call a coroutine like a regular function, it will create a coroutine object, but it will not run it. However, if we add the keyword ‘await’ to the call, then it will run it. Then, it will wait for the coroutine to finish running and extract its return value. More generally, this

keyword can be used to call any awaitable object, of which a coroutine is one of them. Other functions in the 'asyncio' library return awaitable objects, and thus can also be used with 'await'.

2.2 Main Features

When using the 'asyncio' library, the running program has an event loop, which is an object that serves as the central execution device. It can be used for many things, such as scheduling or cancelling tasks and coroutines, creating client and server transports etc…

We can specify coroutines, which are basically methods that are able to be run concurrently with other code. Additionally, coroutines can be suspended by 'await'-ing the completion of another coroutine.

Tasks are very similar to coroutines, except they are to be used with the main event loop. We can add tasks to the event loop, scheduling them to be run in the future.

Finally, there are two different APIs that can be used to implement sockets and interprocess communication. The first is to use streams, or in other words, a coroutine-based API. The second is to use transports and protocols, or in other words, a callback-based API. Their differences are self-explanatory, as the coroutine-based API uses coroutines to maintain concurrent connections for a server, while the callback-based API uses concurrent callbacks upon connections from multiple clients.

2.3 Advantages of 'asyncio'

The library is powerful, as it abstracts much of the concurrency issues away. Thus, developers can write concurrent code using this library quite easily. While not all concurrency issues are solved, it is much easier to identify them, due to the presence of new keywords, such as 'async' and 'await', in the code.

An upside to concurrency is that we do not have to wait for remote calls, such as HTTP requests or database queries. It allows us to run other operations while the above complete. Another benefit is that we can serve multiple clients at once.

2.4 Disadvantages of 'asyncio'

While we can achieve concurrency using the library, we will never be able to achieve full parallelism. The 'asyncio' library was not designed to work multithreaded. If the program were run in a multithreaded environment, we would likely run into issues, such as race conditions, because many of the functions in the library are not thread-safe.

Furthermore, learning concurrent programming is difficult. Debugging a concurrent program is difficult, as the developer does not know the order in which the events occurred. Operations from one task could be interleaved with operations from another. In short, the learning curve for concurrent programming is steep.

2.5 Compatibility of 'asyncio' and Server Herds

The 'asyncio' library works well to implement a server herd. If we want to support multiple connections, it only makes sense to choose a library that implements concurrency. The library contains many functions that allow for concurrency. For example, we can use 'open_connection()' in a coroutine in order to open multiple connections with other processes. We can specify the retrieval of data from a database or another server as a coroutine, so the program can run other operations while it waits for the remote call to return. Finally, we can propagate messages to other nodes in our herd and do other operations concurrently. In short, all of the server node's responsibilities don't have any concurrency issues associated with them. Thus, 'asyncio' is a suitable library for implementing a server herd.

3. Experiment - Using 'asyncio' to Implement a Server Proxy Herd

The overall approach to implementing a server proxy node was to use Python in an object-oriented style. I wrote a Server class, whose class variables maintained the name of the server (Hands, Goloman, Wilkes, Welsh, or Holiday), the host on which it was running, and the port number. It also held dictionaries to hold the clients' locations that it had received, and the connections that it held.

On creation, the server object runs a function that retrieves the event loop from 'asyncio' and continuously runs a coroutine. This coroutine, named 'interpret_message()' was run whenever the server established a TCP connection with any other process, whether it be a neighboring server or a client. I selected a coroutine, because we must allow multiple client connections concurrently. If we used a regular function, a client would have to wait for any other

connected clients to finish their tasks.

There are three types of messages. First there is the 'IAMAT' message, which is sent by the client to any server node in the herd. This tells the node the current location information of the client. Thus, it is the receiving node's responsibility to propagate or flood the information of this message to the entire server herd. This was achieved by defining another coroutine called 'flood', which only propagated if the node did not have that client's location information, or if the node's information about the client was stale. I implemented this function in a coroutine because it is possible that the propagation of messages causes delays. Thus, it should not be done sequentially, in case there are other clients that want to connect to the server node. Location information was only sent to the servers that had a link to the current server node.

These server-to-server location information messages had a different format, however. They were sent in an 'AT' message, which specified the name of the server node that received the 'IAMAT' message from the client. Once they reached a neighboring node, the neighboring node would once again propagate the message on the same conditions as the original node - if the node did not have the client's location information or if the node's information about the client was stale. The server node also sent the 'AT' message as a response to the client.

Finally, there is the 'WHATSAT' message, which is sent by clients to inquire about the nearby location. This message was not propagated to the server herd. However, it was used by the contacted server node to retrieve the nearby places of the client's location, which was saved earlier by the server node. If a client's location is not sent to the server herd beforehand, an error is raised. The nearby places were retrieved by sending an HTTP request to the Google Places API. I used the 'aiohttp' library to make this request. Once again, this operation was encapsulated in a coroutine, because the HTTP request has the potential for delays. If there is a delay, the coroutine allows the server program to run other operations in the meantime. The server also responded to the client with the same 'AT' message sent to the herd, as well as the result of the HTTP request.

In short, there were five server nodes in the server herd. While each node was not connected to every other node in the herd, there still existed a path from each node to every other node, in order for messages to be successfully flooded. From clients, each node could receive 'IAMAT' location information, and 'WHATSAT' requests for nearby places based on their locations. From neighboring servers, each node could receive 'AT' location messages that are flooded to the entire server herd.

4. Analysis of Results

Implementing a server herd using 'asyncio' went smoothly. The initial prototype had few, and easy-to-find bugs. After fixing the bugs, the final implementation ran smoothly without errors. It could do all that was required of it in the spec, including propagating messages to the server herd, processing client requests and saving client location information. The most difficult part of the implementation was figuring out how to use the 'asyncio' library and what each function did exactly.

4.1 A Comparison of Python and Java

Another alternative to using Python for this task, would be Java. There are many similarities as well as differences between the two languages. For example, Python is an interpreted language, whereas Java is a compiled language. Typically, this means that Java runs faster than Python. They are similar in that both are imperative, object-oriented languages. We explored the two languages in order to decide which one would be the better option for this type of project.

4.1.1 Type Checking

Python is a strongly typed, dynamically typed language. Strongly-typed means that there is little implicit type conversion. We cannot interpret a number as a character, like in C or C++. Dynamically typed means that the compiler does not have to know the type of a variable before the program is run. It only does this during runtime.

Java is also a strongly typed language. However, it is a statically typed language. This means that we must declare all variables' types before running the program. In other words, types are checked at compile time.

Due to these differences, we can conclude that both languages have their own strengths and weaknesses. Python, a dynamically typed language, is easier to write. However, it may be more prone to error during runtime and difficult to debug. Java, a statically typed

language, is more difficult to write, as there are more restrictions with types at compile time. However, it may be more secure, robust and have a faster runtime.

4.1.2 Memory Management

Both Python and Java manage memory in a similar way, similar to most other mainstream programming languages - with a stack and heap. The main difference between their memory management systems is the way they garbage collect.

Python conducts garbage collection using a method known as reference counting. Python keeps track of how many references there are towards a particular object or piece of memory. As soon as the reference count hits zero, Python knows that it can no longer be accessed and thus will clean it up. The advantages to reference counting is that it can immediately clean up a non-referenced object instead of delaying it to a later time. It is also more space efficient. However, the downsides of reference counting is that it is not perfect. For example, reference counting cannot detect cycles. If there are two objects and each object has a reference to the other, but no external references to the cycle exist, then they won't be garbage collected. An additional cycle detector is needed for tasks like that.

Java conducts garbage collection using a method called tracing. Java traces the roots of the memory, such as the stack references, in order to find which objects are still referenceable. Those that are not will be garbage collected. Tracing works better than reference counting, and so it will change the program to a cleaner state. Furthermore, as long as there is enough memory, tracing will work faster than reference counting. However, if there isn't enough memory, tracing can result in freezes in the program.

4.1.3 Multithreading

Java and Python differ greatly when it comes to multithreading. Python was not designed to support multithreading, even though there may be ways to get around that problem these days. Python has a global interpreter lock, more commonly known as GIL, which ensures that only a single thread is running the program at a given time. In contrast, Java was designed to be multithreaded. There are built-in keywords, such as 'synchronized', that allow the developer to avoid multithreaded programming issues, such as race conditions.

4.1.4 The Verdict

Choosing between Java and Python as a programming language for this task is not so black and white. Each language offers strengths that the other lacks. Python is easier to write due to its dynamically-typed nature. However, Java has a better garbage collection mechanism, as well as the ability to support multithreaded implementations. It comes down to the purpose of the application or project. Let's use our current server proxy herd as an example. If it is simply an early prototype of the server herd that is used to prove a concept, it would make more sense to write it in Python, as it can be written much faster. However, if it were the production code for the server herd, Java would be a better alternative due to its robustness, cleaner garbage collection and ability to support multithreading, in case we need to speed up the program.

4.2 A Comparison of 'asyncio' and Node.js

While the choice of language is important, we can also take a look at the different libraries, frameworks or technologies available to implement a server proxy herd. Python has the 'asyncio' library, while JavaScript has the Node.js framework. Both of these technologies use event-driven asynchronous models, and thus would be suitable for this project.

Node.js was originally designed to be asynchronous in the first place, while Python was not. Even with the 'asyncio' library, Python may not perform as well as Node.js. Thus Node.js is faster.

While there are few technical differences between the two technologies, there are many intangible factors regarding Node.js and Python 'asyncio'. Node.js developers have mentioned the community to be convoluted and that other developers cannot seem to agree on standards. The learning curve is easy for both languages, but difficult for asynchronous programming. Since most people know JavaScript for its frontend capabilities, switching over to Node.js in order to use JavaScript for a backend purpose should not be difficult. For Python, if we only consider the programming language, it is quite easy for beginners to pick up. In both cases, the difficulty for new developers comes with learn asynchronous programming and the event-driven paradigm.

5. Conclusion

In conclusion, Python's 'asyncio' library is a good choice for prototyping a server proxy herd. It is easy to write Python programs due to its interpreted and dynamically-typed nature. It also has a large community with strong support, as well as versatile libraries for many purposes. Although it may not be the most reliable language in terms of memory management and multithreading, Python is a great language for prototyping purposes. Code written for production should use a more reliable language, such as Java. Asynchronous code, in particular, should be written in a language designed to be written asynchronously, such as Node.js.