Evaluation of asyncio as Architecture for Application Server Herd

Jessica Cheng

Abstract

When designing a web server, different paradigms are available. For this assignment, we are assessing the efficacy of an application server herd architecture; specifically, we investigate building this framework using Python's asyncio module.

Of particular interest are language-related issues, such as the ease with which the program can be implemented, performance implications of the asyncio module, portability across different versions of Python, the language's type checking, memory management, and multithreading features compared to Java, as well as an overall comparison to Node.js.

Introduction

An application server herd is a server paradigm in which multiple application servers communicate directly to each other as well as via the core database and caches. Interserver communications are designed for rapidly-evolving data whereas the database server will still be used for more stable data that is less-oft accessed or requires additional operations.

We are trying to build a Wikimedia-style service designed for news. Currently, Wikipedia uses multiple, redundant web servers behind a load-balancing virtual router and caching proxy servers for reliability and performance. In our new product, updates to articles will happen far more often, access will be required via various protocols, not just HTTP, and clients tend to be more mobile.

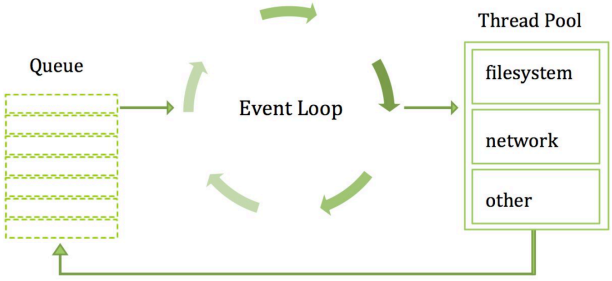
This project simulates a proxy for the Google Places API. In the instance of this project, this means that if a user changes location, any server can learn of this change without consulting the database, even if not all servers are connected to each other.

1. Design

1.1 Asyncio, Event Loops, Coroutines

Asyncio is a single-threaded, asynchronous framework to achieve concurrency in Python. Coroutines, which are functions that can suspend execution before returning, are able to 'pause' while waiting on other coroutines to run.

When servers are started up, they each create an event loop that can add co-routines. [1] The event loop runs tasks that are waiting to be executed. This is illustrated in the following diagram:

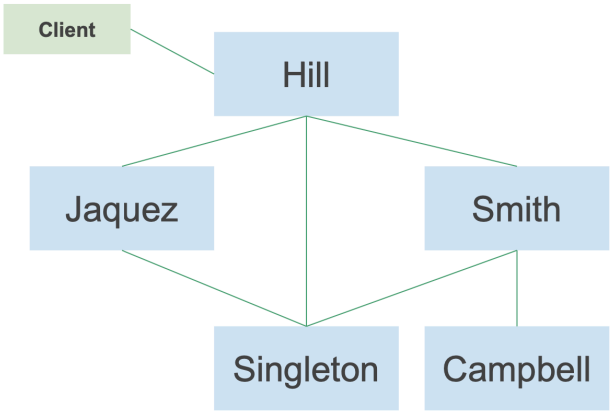


In the context of this project, co-routines are added as new connections are encountered. Due to asyncio's asynchronous nature, it can process many messages at once. This feature makes it a good candidate for the implementation of a server herd.

1.2 Server Herd Topology

There are five servers in this test, each of which do not have access to all other servers. Interserver communication is bidirectional. Clients are able to talk to any server they choose.

The relations are as follows:



1.3 Communication

IAMAT Messages:

Clients announce their location and name to a server using IAMAT messages. Messages are formatted as follows:

IAMAT <client name> <latitude,longitude> <time sent>

Where the time sent is reported according to the client's clock, expressed in seconds and nanoseconds since 1970-01-01 00:00:00 UTC (Python's epoch date).

Internally, servers update its logs of the client's location based on the time sent reported by the client. If there are

updates, this information is then flooded to other servers (implementation discussed in **§1.3**).

Servers then respond to the client with an AT message (discussed below) with the same information, plus the time difference between the time sent and the time on the server's own clock.

In terms of datatypes, the client name can be any string, latitude and longitude are real numbers with their signs denoted by the '+' or '-' symbol without spaces between them, and the time sent is a real number.

WHATSAT Messages:

WHATSAT messages are sent from the client to any server for information about a specified number of places that are near them within a specified radius, They are formatted as follows:

WHATSAT <client name> <radius> <results>

This message assumes that servers know the client's location based on their name, which is why an IAMAT message must be sent by the client beforehand to announce their location. The server then queries Google Places using an API provided by Google, and returns a number of places to the client.

An interesting feature is no matter the server that the client queries to, the replying server indicates that the message is from the one that the client *originally sent the IAMAT to*.

With regards to datatypes, the client name can be any string, radius is a real number, while the number of results is a an integer. Any radius above 50 or results above 20 will be queried with those numbers respectively.

AT Messages:

Servers communicate with each other using AT messages. They contain the following information:

AT <server name> <time difference> <client name> <latitude,longitude> <time sent> <server the client first announced its location to>

Where the time difference is the difference between the time sent and the time on the server's own clock; time sent is the time the client reported sending its location at.

In terms of datatypes, the server/client name can be any string, latitude and longitude are real numbers with their signs denoted by the '+' or '-' symbol without spaces between them, and the time difference/sent is a real number.

Invalid Messages:

Messages are deemed invalid if they do not fit the datatypes above. They are universally responded with a '?'. followed by a space, followed by the invalid message. Nothing else will be done on the server's part.

1.4 Implementation Details

Other design choices that were made:

1. A connection is considered established when a server successfully talks to another server, or if a message from a new server is received.
2. A connection is considered dropped when a server runs into any errors while attempting to communicate with another server.
3. **To prevent infinite loops in flooding**, servers do not forward messages to the server it received an AT message from, and keeps a cache of messages it has already seen—if the server has seen a message already, it will not forward it. This design was implemented with a set, with a constant lookup time, and is chosen for ease of implementation, since this test is for a small number of servers that will handle a limited number of test cases.
4. Internally, each server has a dictionary using the client's name as the key and value as a list that stores the client's location and time, and the server that first received a WHATSAT. An entry is updated when there is a new IAMAT message with a later timestamp than the one stored.
5. Finding whether a sender is from a valid list of servers is done through a lookup in a local dictionary.

2. Evaluating Asyncio

2.1 Ease of Programming

Asyncio makes programming a server herd fairly easy. All asyncio functionalities are built into the library without needing to much coding and configuration from the programmer, which translates to ease of programming.

Furthermore, since all servers run the same code, adding new servers is very easy. The only additional work that needs to be done would be to add new servers and their port numbers to every existing server, since this is stored locally in each server. Depending on the size of the server herd, this may require quite a bit of work.

2.2 Performance Implications

A problem with asyncio is its asynchronous nature. There is no guarantee that a IAMAT that is propagated through the servers through flooding will reach another server before a client sends a WHATSAT to it, resulting in an error. This problem is not limited to asynchronous frameworks, but

because events are run on a loop in asyncio, there can be a much greater delay, which would cause more of these errors. One particularly grim worst-case scenario is where a there is a bottleneck where two components of the network graph are connected only be one node—that node naturally undergoes a lot of traffic because it has to propagate AT messages across components, resulting many coroutines in its event loop. Then if a client moves from one component of the graph to the other, the IAMAT message has a high chance of not getting to the new component before the client sends a WHATSAT message. [2]

Another issue with performance would be code maintenance. Though writing the same code for each server was very simple, every minor update would require all servers to restart.

Also, if someone finds the format of the AT message along with the servers' names, it can easily fake user locations. A possible malicious use is to constantly update client information to cause congestion amongst networks, since servers have to flood. Additional information about each server must be stored so that servers can authenticate AT messages received, at the cost of needing to do more checks and needing to update more information on each server every time a new server is added.

2.3 Version of Python Used

Asyncio was introduced by Python version 3.4, and my server runs on both Python 3.7 and 3.8. However, I did encounter the following warning when running in Python 3.8:

```
DeprecationWarning: The loop argument
is deprecated since Python 3.8, and
scheduled for removal in Python 3.10.

  reader, writer = await asyn-
cio.open_connection(self.host, port,
loop=self.loop)
```

Originally, the loop argument is allowed to allow for a specific event loop that the programmer wants to utilize, but this is actually redundant. Though this does not impact performance, it does mean that if there is a reason to switch to Python 3.10 in the future, all servers will have to be updated. [3]

3. Comparison to Java

3.1 Type Checking

Python is an interpreted language, whereas Java is a compiled language. This means that type checking is done at runtime for Python, while it is done at compile time for Java.

A problem with both Python and Java datatypes is that decimal numbers have limited accuracy—what this translates to is IAMAT messages with reported times that are too close to each other (a difference by one in about the 9th decimal place for floats) may result in a later message being dropped. I circumvented this issue by storing the time as a string and using string comparison to determine precedence, but that is counterintuitive and exposes a deeper problem with Python. Other languages, such as Lisp or Scheme, are able to deal with decimal places with more accuracy due to their support of the bignum datatype, which can use as much memory as allowed to ensure decimal accuracy. For the scope of this project, that would be overkill (especially because real clients would not be able to move their location in such a short period of time), but some applications that may be extremely sensitive to the data being sent, and using any language that handles decimal numbers in the way Python does would lead to problems.

3.2 Memory Management

Both Java and Python have garbage collectors (as opposed to C++, where memory management has to be done manually by the programmer), but they are implemented in different ways.

Python uses reference counting, where every object gets a reference count which is incremented or decremented as references to it change and garbage collection is done when the count becomes zero. Because it relies on a count, it is unfit to handle cyclic references. [4]

Java uses mark-and-sweep as its method of garbage collection, where objects that are still being referenced are marked as 'alive', and then objects that are not alive are garbage collected. [4]

In the context of our server herd, servers create objects that are only used a few times before they become useless, and do not contain cyclic references. In this case, Python's reference counting method is more efficient because these useless variables can be destroyed as soon as they have no use when the reference count drops to zero, freeing up more space for the servers. Especially in the context of my implementation of the server, where messages received are cached to prevent flooding, more memory is a very good thing.

3.3 Multithreading

Python's asyncio is designed for single-threaded concurrency. A task has the exclusive use of the CPU until it gives control back to the event loop, even on multicore processors. On a large scale, this would lead to issues with performance speed, making asyncio not very scalable.

Meanwhile, programming in Java lets the server take advantage of the multiple cores on a CPU in its multithreading application. Because of this, a server can do more work and

process more messages using Java. As such, a server network with more servers will see much more benefit using Java than using Python, making Java more scalable than Python for this purpose.

4. Comparison to Node.js

This section places specific focus on comparing Python and Node.js's approaches to implementing a server herd.

In terms of speed, Node.js is known to be faster. For real-time applications, speed is the most important factor, so in this sense there is more advantage to using Node.js than Python. However, Node.js slows down in the face of CPU-intensive tasks, though this is not a major concern for our simple server herd. [5]

In terms of concurrency, both run asynchronous applications in a single-threaded manner. In a similar way as using Future object in Python, Node.js has a Promise data structure that allows a task to be done asynchronously in the future. [6]

In terms of barrier to entry and ease of application, both Python and Node.js are fairly duck-typed, so implementing a server heard in both languages should be fairly simple. However, JavaScript saw its beginnings in web development, so the developer environment and support for applications that require connectivity may be more robust for Node.js.

5. Conclusions and Recommendations

In all, implementing a server herd in Python using asyncio was very simple because of the robustness of the module and the fact that all servers can use the same code without any issues. However, scalability proves to be a problem on both code-maintaining and performance ends, where the advantages of other programming language such as Java and Node.js shines.

My understanding of the issue at hand brings me to recommend Node.js as the language to implement the server herd *if the number of servers is limited*, due to its similarity in implementation as Python, which means a fairly simple implementation, with the additional advantages of increased speed with respect to the operations we want our servers to perform and the more robust developer environment for creating applications that live online. Nonetheless, the problem of not being able to exploit multicore machines and needing to update code on each server when there are updates remains a hurdle to scalability.

If there is a large number of servers, scalability may easily become the most important factor to the success of the app, and it is then that a language such as Java may be more useful.

References

[1] https://eng.paxos.com/python-3s-killer-feature-asyncio, *Python 3's Killer Feature: Asyncio*, Michael Flaxman, 2017

[2] https://medium.com/analytics-vidhya/asyncio-threading-and-multiprocessing-in-python-4f5ff6ca75e8, *Asyncio, Threading, and Multiprocessing in Python,* Ajit Samudrala, 2019

[3] https://bugs.python.org/issue36373, *Deprecate explicit loop parameter in all public asyncio APIs*, Python Bug Tracker, 2019.

[4] https://dev.to/deepu105/demystifying-memory-management-in-modern-programming-languages-ddd, *Demystifying memory management in modern programming languages,* Deepu Sasidharan, 2020

[5] https://hackernoon.com/python-vs-nodejs-which-programming-language-to-choose-98721d6526f2, *Python vs Node.Js: Which Programming Language to Choose?,* Michael Yarbrough, 2019

[6] https://dev.to/neutrino2211/what-is-concurrency-in-node-js-3lgo, *What is Concurrency in Node Js?,* Mainasara, 2019.