Implementation of Server Herd Architecture with asyncio: Performance and Suitability.

Robathan Harries - University of California, Los Angeles

Abstract

In this study, I will be investigating the both the performance, maintainability and the general suitability of the Python asyncio asynchronous networking library for Wikimedia-style news systems which require the ability to accept frequent small updates from highly mobile clients. The main object of this study will be a custom implementation of a server herd which accepts location updates from clients and propagates them among the servers, allowing a client to query any server for information related to its location. In this study I will also provide a brief comparison of the functionality of asyncio and Node.js, and present a more general comparison of the performances of Python and Java. After the following investigation, I recommend Python and asyncio as a viable solution for problems in this domain. This is largely because Python is readable and easy to prototype and maintain, while still providing comparative performance to Java and comparative functionality to Node.js.

Introduction

Many different web stack architectures exist, each with different ideal use cases, performance, and tradeoffs. As the relevant example, Wikimedia uses a LAMP-stack architecture, which is based on GNU/Linux, Apache, MySQL and PHP. It uses multiple redundant web servers with geographic load-balancing, automatically directing clients to their nearest server cluster[5]. However, this load-balancing becomes a bottleneck in Wikimedia-style services oriented towards news, in which articles are updated far more often and clients are much more mobile, connecting to different closer-proximity servers as they move.

This study investigates the feasibility of using the Python programming language and Python's asyncio asynchronous networking library to mitigate these issues.

My server-herd architecture implementation (which is the object of this study) attempts to work around this bottleneck of a load-balancing server by giving clients the ability to connect to any server. It also accommodates large numbers of client connections by asynchronously handling connections with asyncio. In addition, asyncio easily preserves reliability, as the server-herd is interconnected and can asynchronously spread client information throughout all the servers.

There are still minor problems with asyncio, such as it's lack of support for HTTP connections, but aiohttp, which interfaces well with asyncio.

While imperfect, this study concludes that Python's asyncio library provides extensive support for this server-herd architecture. In addition, developing in Python comes with many benefits itself, like automatic memory management and duck-typing. These drastically reduce the required development time, and incur relatively limited performance costs.

1. Program Design

For this study, I implemented a server-herd architecture which stores client locations and communicates with the Google Places API to provide the client with JSON-formatted location information near it's given location.

The implementation leverages co-routines and event loops to accept asynchronous connections and process input and output without blocking. This technique allows large performance improvements, especially when client connections are frequent and low-weight (they don't require much processing).

To communicate with the server-herd, the client must open a TCP connection with any of the servers in the herd. The server handles the connection as a coroutine on its event loop. It processes one coroutine at a time, temporarily yielding to other coroutines when waiting for I/O.

In this small test implementation, there are five

this is easily solved with external libraries such as

different servers, each run by a different instance of the same program. The servers have specific ID's: Goloman, Wilkes, Hands, Holiday, and Welsh. These ID's are important because there is a preset, asymmetric set of allowed communication channels between these servers. Each communication channel is bidirectional, and the asymmetry of the communication network demonstrates that the implementation's server herd architecture is viable for real life situations in which it isn't feasible for a server to directly communicate with every other server in order to spread an update. Figure 1 shows a diagram of the communication network.

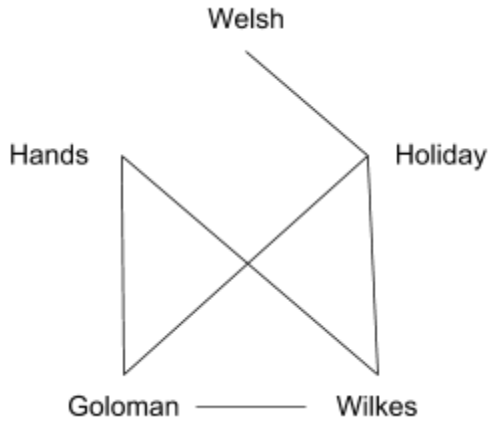

Figure 1. Diagram of preset communication channels between the five servers. Each line represents a communication channel, and each channel allows bidirectional communication.

After a client connects to one of the servers, the server waits for the client to send a message in one of two formats: AIMAT and WHATSAT.

1.1 IAMAT Command

The IAMAT command format is how clients inform the application of their current location. The format is as follows:

IAMAT *<client ID> <ISO 6709 latitude and longitude> <POSIX time when message was sent>*

Each field, including the command ID, is separated by whitespace, and none of the fields can include any whitespace or the message will be considered invalid (see 1.5: Invalid Commands).

server will update its information on the client's location, propagate this new information to the other servers (see 1.4: Server Flooding), and send a response to the client in the form of an AT command (see 1.3: AT Command).

1.2 WHATSAT Command

The WHATSAT Command is how clients initiate a Google Places API query. The format is as follows:

WHATSAT *<client ID> <radius> <max results>*

As with the IAMAT command, the fields are all separated by whitespace, and the fields must be strings of strictly non-whitespace characters, or the message will be invalid (see 1.5: Invalid Commands).

The *<radius>* field is the integer radius of the Google Places API search (in kilometers). It must be in the range [0, 50]. The *<max results>* field is the maximum number of location results the server should return that are within the given radius. This must be an integer in the range [0, 20].

When the server receives this command, it will send an http request to the Google Places API, and respond to the client with an AT command (see 1.3: AT Command) followed by the JSON response the server received from the Google Places API. If the server has no record of the client's information, the command is considered invalid (see 1.5: Invalid Commands).

1.3 AT Command

This command format is different from the IAMAT and WHATSAT commands in that it is sent by the server, rather than the client. It is used to respond to client commands and to propagate client information to other servers. The format is as follows:

AT *<server ID> <clock skew time difference> <client ID> <ISO 6709 latitude and longitude> <POSIX time when message was sent>*

As with the other commands, the fields are strings of non-whitespace characters separated by

Upon receiving a valid IAMAT command, the whitespace. Malformed AT commands are treated as invalid (see 1.5: Invalid Commands).

The <*server ID*> field refers to the original server the message was sent from, which isn't necessarily the same server that sent the command (see 1.4: Server Flooding).

1.4 Server Flooding

The server-herd spreads client information using a flooding algorithm: After a server receives an IAMAT command and updates client information, it takes the AT command that it sent back to the client, which is tagged with its own server ID, the new client ID, and a time stamp, and sends the command to each server it is connected to (see Figure 1). When a server receives an AT command from another server, it checks this new client data against its current records. If the new data is more recent, it updates its records and sends the unchanged (still tagged with the originator's server ID) AT command on to all connected servers, minus the originator of the command.

In this way, when a server receives new information it sends it to all connected servers which may not have received it yet. If a server receives information it already has, it simply ignores it.

1.5 Invalid Commands

If the client sends an invalid command to any server, the server responds with a message in this format:

? <*message*>

The <*message*> message field is simply the exact invalid command that the client sent.

General invalid commands include:

- Incorrect number of fields (triggered by fields with whitespace characters)
- Unrecognized command ID

IAMAT invalid commands include:

- Incorrectly formatted ISO 6709 formatted location
- Incorrectly formatted POSIX UTC time
- <*radius*> or <*max results*> not an integer
- <*radius*> or <*max results*> out of range

2. Suitability of asyncio Application

The asyncio library is extremely well-suited for exploiting server-herd architecture, because its main contents includes a pluggable event loop, built-in support for TCP, UDP, and SSL network communication, subprocess pipes and delayed calls, and coroutines and tasks built around yielding, just as this approach requires[1].

2.1 Event Loop, Coroutines, and Tasks

To develop server-side applications with asyncio, we must first set up the event loop, which runs asyncio coroutines as asyncio Tasks. Any asynchronous functions which we want the event loop to run are defined as coroutines and wrapped in Task objects, which get placed on the event loop. When wrapped in an Task, the asyncio library guarantees that the coroutine will be run in the future. It will run until it completes, and whenever it yields it will be rescheduled on the event loop[1].

In Python, a coroutine is a cooperative, non-preemptive multitasking generator function which is scheduled by an event loop. Coroutines decide when they yield execution, and can accept caller information when they do so. This allows communication reminiscent of inter-process communication, except entirely internal to a process [2]. In the server-herd application, a client connection is encapsulated as a coroutine, which runs until the connection closes, yielding whenever it needs to do I/O.

As a result, the asyncio event loop can communicate with clients through asynchronous coroutines, simultaneously passing input data from multiple open TCP connections to multiple client coroutines. This method allows asyncio to maintain separate message buffers for each client connection, guaranteeing messages are communicated reliably.

Since the server-herd application is so well-suited for the asyncio library's intended functionality, implementing it with asyncio provides both

WHATSAT invalid commands include:

- Unrecognized client ID

3. Python vs Java for Server-Herds

Beyond asyncio's library-specific benefits, it remains in question whether Python is the correct choice of programming language. Though Python allows increased development speed and ease of maintenance, the underlying mechanics of these improvements may have negative implications for its performance in larger applications.

In this section, I will focus on how Python deals with memory management (3.1), type-checking (3.2), and multithreading (3.3) and compare it to how Java approaches these problems. In doing so, I will assume a CPython implementation.

3.1 Garbage Collection

Python's garbage collection strategy is simple and effective, but has a major drawback. For the most part, Python uses reference counts to keep track of allocated memory. Every object stored in memory has an additional field which tracks how many different pointers to the object exist. When an object is first created, along with a label acting as a pointer, its reference count is set to one. Whenever a variable is created or its value changed, Python checks two things. If the new value is a pointer to an object, Python increments the object's reference count. If the variable used to point to a specific object and no longer does, then Python decrements the object's reference count.

Periodically, Python automatically performs garbage collection, sweeping over allocated memory and deallocating any objects with a reference count of zero. Since this process is completely automatic, a developer can for the most part safely ignore the issue of allocating and deallocating memory.

However, there is one large drawback to this method. Reference-count-based garbage collection can't deallocate objects in reference cycles. Since objects in memory often contain pointers to other objects in memory, a memory object may point to another memory object, which points back to the first memory object. Even after the variables/labels originally pointing to these objects have been reassigned, the performance benefits and ease of prototyping.

Java, on the other hand, performs garbage collection using a more traditional method: mark-and-sweep. Periodically, Java sweeps through the heap, starting at the root blocks of memory. Whenever it finds a pointer to some other area of memory, it 'marks' it, and also searches through it for more pointers to other memory areas. When it has 'marked' every accessible memory block, Java sweeps back over the heap, deallocating every non-'marked' block of memory.

Java augments this garbage collection method with generational garbage collection, which prioritizes mark-and-sweep on newer memory. The more waves of mark-and-sweep that a block of allocated memory has survived, the higher it's generation. Higher generations are treated as more persistent memory, and are less likely to be garbage collected.

Unlike in Python, Java's garbage collection runs on a separate thread, allowing it to operate concurrently with the running program using efficient built-in synchronization primitives. This makes Java's garbage collection both more efficient and more complete than Python's, as it doesn't suffer from the reference cycle problem.

3.2 Multithreading

In this domain, Java once again has a significant advantage over Python. Python has a Global Interpreter Lock, which is a mutex protecting Python objects from concurrent access by multiple threads[3]. This significantly limits the effectiveness of multithreading in Python, as threads cannot operate on shared memory. This slows down Python's garbage collection compared to Java's, and limits the ability to apply multithreading to the Wikimedia server-herd application, as concurrent client connections could never update shared memory concurrently, even when they don't actually access the same data in an object.

Because of the Global Interpreter Lock, Java's multithreading capabilities are far better than Python's. Since multithreading affects many aspects of performance (ex: garbage collection), this is significant.

reference count for each of these objects remains nonzero, and neither can be deallocated.

3.3 Type Checking

Though Python's performance lags behind Java's, it boasts of significant advantages in development speed. A large part of this is due to how Python type-checks using duck typing. Duck typing refers to how Python's variable containers are extremely flexible, and can hold primitives and objects of different types entirely implicitly. In Java, which uses static type-checking, variable types are strict and explicit.

Though Java's static type-checking can ease the programming of large programs with many interacting modules by explicitly declaring data formats, Python's flexibility makes it far superior when prototyping, though data mismatches will cause runtime crashes which Java would find at compile-time.

This effect is compounded by Python's general compactness and readability. In general, when implementing the same functionality, Python programs will be smaller and more readable than Java programs.

4. Node.js vs asyncio for Server Herds

Node.js and asyncio are incredibly similar in functionality. Both are event-driven, asynchronous and non-blocking. Just like in asyncio, Node.js functions are scheduled as callbacks onto the event loop, are processed until they yield, and are then rescheduled[4]. Both string chains of callbacks together to achieve asynchronous functionality.

Node.js is much newer than Python, and thus is more up to date and better maintained, with frequent updates to meet the latest standards. In addition, Node.js was built on the V8 web engine to develop fast, lightweight and scalable web apps, whereas Python was developed as a scripting language, though it has obviously evolved significantly. Node.js also has the advantage that when used to develop web applications, both the frontend and backend will be in the same language, reducing language incompatibilities and creating more reliable applications. As such, many companies prefer it over Python[4].

herd must be able to provide high throughput for handling many simultaneous connections, Python's asyncio library may be more efficient, and thus be a better fit.

5. Implementation Issues with asyncio

Though implementing a server herd with the asyncio library was relatively painless, it came with a few drawbacks. Firstly, as with any asynchronous event-driven approach, debugging becomes much more difficult, as stepping through the interpreter is largely useless. Instead, debugging requires extensive console logging.

6. Conclusion

After this investigation, I have found that using Python and the asyncio library comes with multiple performance issues which would disqualify it from other use cases. However, the ideal use case for asyncio fits with this server-herd application, which requires high processor throughput with many asynchronous network connections. In addition, they allow much faster development, and have no significant performance downsides compared to other approaches. As such, this I recommend the use of Python and asyncio to implement the Wikimedia-style server-herd application.

References

[1] "18.5. Asyncio - Asynchronous I/O, Event Loop, Coroutines and Tasks." *Python 3.6.7 Documentation*, Python Software Foundation, 29 Nov. 2018, docs.python.org/3/library/asyncio.html
[2]Diaz, Yeray. "AsyncIO for the Working Python Developer." *Hacker Noon*, 20 Feb. 2016, hackernoon.com/asyncio-for-the-working-python-developer-5c468e6e2e8e.
[3]"Global Interpreter Lock." *Python Wiki*, Python, 2 Aug. 2017, wiki.python.org/moin/GlobalInterpreterLock.
[4]Saba, Sahand. "Understanding Asynchronous IO With Python's Asyncio And Node.js." *Math Code by Sahand Saba Full Atom*, 10 Oct. 2014, sahandsaba.com/understanding-asyncio-node-js-python-3-4.html.
[5]"Wikimedia Servers." *Wikimedia Meta-Wiki*,

Python, on the other hand, is more suitable for processor intensive applications. Since a server

Wikimedia Foundation, Inc., 16 Nov. 2018, meta.wikimedia.org/wiki/Wikimedia_servers.