Viability of Asyncio and Python for Server Herds

[redacted], *UCLA Computer Science*

Abstract

Asyncio and Python have good potential to be used as a part of a backend web server hosting many client data sessions. Asyncio's very high level makes code understandable, while asynchronous tasks make switching between contexts while awaiting TCP communication fast. Also, as demonstrated with a sample server herd, communication between servers can be asynchronously facilitated in order to propagate information forward and eliminate the need for a database, if some data loss is permitted. Scaling out to increasing server counts requires only minor code changes, making the setup simple to scale. Despite the benefits, limitations also exist and may favor languages such as Java or JavaScript, especially as the size and complexity of the code-base and the number of clients increases.

1. Python and Asyncio: A Background

Python is a dynamically typed, interpreted language originally released in 1991. As such, it supports duck typing, and the syntax is generally simpler than many C-style languages, making the language more accessible. One benefit of Python is the large number of libraries built into the language, as well as others that can be easily imported using pip, the Python package installer. One such built-in language, in Python 3.7.2, is Asyncio, a framework for doing asynchronous network calls. The coroutines provided by Asyncio give a large performance boost, as blocking IO tasks can be awaited, allowing other meaningful computation or other IO tasks to be done at the same time, without the need to create multiple threads, a more complex and heavy-weight solution.

Python in general, and specifically Asyncio benefit from operating at a high level. Many concepts of asynchronous programming have been abstracted to simple functions that can be called easily and are not difficult to integrate into existing programs. In addition, documentation is abundantly available for most Python libraries, making the language and aspects like Asyncio accessible to both veteran and new programmers.

1.1. Asyncio Accessibility

Asyncio exists at both a high level, including coroutines, streams and subprocesses, and at a low level, with primitives like event loops and futures. This duality of the Asyncio library is beneficial. The high level concepts allow for easy prototyping of asynchronous code, while the lower level primitives let an experienced programmer take more fine-grained control over their code. We will focus on the high-level operations. In examining the Python source code for Asyncio, we inspected the streams library, which will be the focus for this server-herd project. The streams library is located at cpython/Lib/asyncio/streams.py in the cpython main repository. Looking at this library, two very useful functions have been defined, open_connection() and start_server(). These two functions serve as top level wrappers, providing easy access to their underlying functionality, stream readers and writers. The StreamReader and StreamWriter classes are both also implemented in the same streams.py file, but exist at a lower level. These two classes act with lower level writes and reads, but also have functionality to be placed into a paused state, and attempt to resume from that state. These additional fields of the class are what allow them to be implemented asynchronously, where the read and write tasks can be paused or interrupted.

1.2. Documentation of Asyncio

Asyncio is heavily documented, along with most of Python, and is written at a very accessible level, including the most basic examples and scaling to highly advanced usage. For Asyncio, we will focus on the documentation of the Streams functionality, as that is the primary sub-library utilized for a server herd. The documentation provides a simple example of how to open a TCP connection with a reader and writer, and then send and receive data from the stream, including awaiting data from the other end of the connection. The documentation also shows how to start a server, which we utilized to quickly bring up a server herd. The documentation also shows how to use the underlying StreamWriter and StreamReader, although these weren't investigated as a part of this project.

2. A Five Server Herd

In order to test the basic functionality and minimal scalability of a server herd using the Asyncio library, we wrote a basic script, server.py, which would register and handle requests on a given port, based on the name of the server and the ports that were allocated on the SEASnet server. Each server could communicate with at least one other server bi-directionally, but not all servers were in communication with each other, which would help in a scaling situation to reduce connections between servers.

2.1. Communication Strategy

The source code for the server.py script included a dictionary which held the mapping for each individual server. Based on the server names, each server would have a list of others that it communicated with, setting up a network between each server when running simultaneously. Names of the servers were also included in another dictionary that mapped names to port numbers, allowing any server to locate and send a message to any of the other servers it was in communication with. This strategy of hard-coding the maps between the servers has a couple of downsides. Whenever a new server wants to be added to the herd, it needs to be manually added to the dictionary, and any server it communicates with also has to have its list of communication servers updated. This makes it easy to create a unidirectional server by accident. A better solution would be to store server communications in another format, such as JSON with the ability of a new server to assert itself as existing to whichever other servers it desired, and then the data could be dynamically updated. In addition, only hard-coding one direction of communication would make missing bidirectionality less common, and would make server addition easier.

2.2. Handling TCP connections

To handle TCP connections, the server.py program relies heavily on the Asyncio function of start_server(). This function takes an asynchronous function as one of the arguments, and that function is called every time a TCP connection is opened with the server. Within that function, the heavy lifting of message analysis and response is completed. Every message received is decoded and split along whitespace to produce a list of strings. Python's large built-in library proved extremely useful to make otherwise complicated tasks like string manipulation simple to execute. From there, the first value of the split string was checked against IAMAT, the command to identify the location of a client, and WHATSAT, the command to find what exists at a given client location via a Google Places API query. All checking and parsing of an incoming message was handled within a try/catch block, allowing any error to keep the server running, rather than crashing one of the servers. In addition, this allowed for an easy way to exit if badly formatted input was provided, simply by raising an exception indicating this error and then handling that in the catch portion of the block. Once the message was produced, it was sent to the client and the stream closed. If there was any follow-up work to be done, this was completed after the client's stream was closed, allowing the client to continue without worrying about internal server updates.

2.3. IAMAT propagation

When an individual server receives an IAMAT message, it generates a response to the client, but also must propagate the location and other information about the client to the other servers it is in connection with, as well as store some data about the client for future requests. To store the information, a dictionary is used to keep track of all clients that interacted with the server. This is populated for each server from both directly received messages from clients and those forwarded by other servers. In order to propagate messages across other servers, a forward message is generated, with a specific identifier of CLIENTAT, intended only for other servers, and this message is sent to all others that the given server is in bidirectional communication with. Upon receiving one of these messages tagged CLIENTAT, a server will turn around and forward it to all servers it is in communication with, thereby propagating the message via a flooding algorithm to all other servers. In order to prevent endless loops, each message is tagged with an ostensibly unique UUID, generated by the server which receives the first communication from the client. This UUID is added as a portion of the CLIENTAT message. Every server maintains a set of UUIDs corresponding to the messages they have received. Therefore, if a server gets a CLIENTAT message and the UUID associated with the message already appears in the server's set of UUIDs, that message is not forwarded, thereby ending the infinite loop of flooding. Other improvements are possible to increase the efficiency and reduce cross-server communication. For example, when a server receives a new client update, it immediately turns around and sends the message back to the sending server. However, a less naive algorithm would reduce this additional communication step, saving computation and communication time.

2.4. WHATSAT querying

When a server receives a TCP message with the query of WHATSAT, it must return specific information from the Google Places API about locations near where the queried client's location is. To do this, the server finds the requested client in the dictionary of clients that have communicated with the server, gathers their location, and uses that as a parameter for an asynchronous HTTP request that is sent to the Google Places API. The response to this query is then formatted and sent back to the client, closing the connection. One major issue with the WHATSAT command is inconsistent data across servers. In this server setup, there is no central database, and data from clients is propagated between servers and stored on ephemeral storage, which is erased if a server goes down. Because of this potential data inconsistency, it is possible for one server to have knowledge about a client that another server requested. In order to solve this problem without resorting to a central repository of clients, when a WHATSAT command is received, if the requested client is not in the dictionary for the handling server, this server floods the network requesting the client. In a similar fashion to the IAMAT propagation, a unique UUID is generated to prevent duplicate efforts. For each queried server, if the client is in their dictionary, they return the client information, and otherwise query all their connected servers for the client information. In this manner, even if a single server goes down, once it recovers it will still be able to access information about clients that existed before the crash as well as those that communicated their location during the downtime. This adds additional reliability to the server herd, and makes scaling more beneficial, as more servers increase the data integrity. However, there are some long-term issues with data protection. Because servers that went down don't recover all the data they lost once they return, continually crashing servers means that eventually some data will be lost. While this is not a problem for the current application of clients posting GPS coordinates, this data storage solution should not be used for more valuable data.

2.5. Logging

Python provides a useful built-in logging library, supporting various logging levels, output to a file, log formatting and more. We utilized the logging library to track incoming and outgoing TCP connections, including every client request, and forwarded information from other servers. The logs were unique to each server, and each log would append continually, with a logged value for server start-up to denote when each server came into operation. Logs were also timestamped, assisting in analysis of the logs, especially after a crash or other issue.

3. Python: Pitfalls and Promise

Through the experience of working with Python and the Asyncio library, we conclude that these are a good choice for rapid prototyping and quickly getting a server-herd up and running, but may not be the best choice for a large, enterprise-scale project with multiple programmers.

3.1. Python's Plusses

Python and Asyncio have many benefits that help for fast prototyping. Python is dynamically typed and supports the duck typing paradigm, which helps to quickly write programs without having to worry about explicitly declaring types and type checking. In addition, Python's syntax is simple and quick to write, making it easy to turn out large quantities of code in a short period of time. Libraries like Asyncio also have extensive documentation, making them more accessible. Python also has benefits over older language like C and C++, such as managed memory and garbage collection done through reference counts, which makes programs written in Python less likely to crash and dump core. Multithreading is also supported in Python, although its usage was not investigated as part of this report. A big benefit Python provides for this project is Asyncio, with its ease of use. Asyncio's abstracted asynchronous functions make asynchronous code easy to write and understand in Python, which is beneficial for the environment of a server herd. Python also encourages efficient data structures, like dictionaries and sets, which help with performance and ease of use. Overall, there are many benefits to both Python and the Asyncio library, as explored here and in the sample project, making Python a good choice for creating server herds.

3.2. Python's Pitfalls

While Python may be a good choice for a server herd project, and may be a totally acceptable language to go with, there are some potential pitfalls which need to be clarified. Firstly, while dynamic typing and duck typing are useful for small projects and can get programs quickly off the ground, more issues arise with the scale of the program. Without specified types, it can be difficult to keep track of variables and function return types in a larger and larger program, especially with multiple programmers working on it. This requires careful naming of variables, and very good notes on variable types and function return types. Another issue with Python is that is runs atop the Python interpreter, which is generally slower than a Just-In-Time compiled language, like Java or JavaScript, and is much slower than a compiled language like C or C++. This speed difference may not be apparent at small loads, but as the server herd becomes larger, it may increase latency for users and potentially increase server costs over time. In addition, there is an issue with multithreading. While this project investigated asynchronous functions as an alternative to multithreading, it remains true that multithreading provides more computational power, especially on newer machines, by utilizing many cores, allowing instructions to be handled faster. While Python does have support for multithreading, it is less robust than the support of a language like Java, where multithreading is more central to the language.

3.3. Alternative Languages

While Python and Asyncio may be a good choice to create a server herd, two good alternatives to the problem exist, namely Java and JavaScript, or Node.JS. Java has the benefit of being a statically typed language, which makes scaling to larger programs easier and more self-documenting, when compared to Python. The strict object-oriented nature of Java also helps for maintaining organization in larger programs, while objects are optional in Python. In addition, Java was designed from the start to be a server-side language, meaning that it has strong support for server features like multithreading. Another alternate is JavaScript, specifically Node.JS. Node was designed as a back-end JavaScript, allowing full stack development in just one language. There are many benefits to Node as compared to Python. For example, JavaScript can be just-in-time compiled, much like Java, leading to greater efficiencies. In addition, the syntax of JavaScript is more similar to that of a C style than Python, which is beneficial for programmers coming from C-family languages. In addition, asynchronous programming is built into the heart of Node, and is almost the default mode of programming. JavaScript also has a much more well-defined system of "promises" which can be in several states and add a further granularity to asynchronous programming, especially as compared to the much more basic coroutines available with Asyncio. Node has the benefit that it was built for network calls, and that is its primary paradigm, whereas Asyncio is just a library atop Python, and Python itself is a much more multi-paradigm language.

4. Conclusions

Based on the research into Python and Asyncio, as well as the sample program written to create a small server herd, we can recommend these to build up a larger project. While there are definite downsides to Python and Asyncio, as denoted above, we believe the benefits of rapid prototyping, easily understandable syntax, high level asynchronous tasks, and a wide array of libraries make this a good choice. Other languages such as Java and Node.JS should be further considered, but overall Python and Asyncio are more than sufficient to build a server herd to communicate with clients and internally asynchronously.

5. References

1. Python 3 Documentation for the asyncio library, 2019 https://docs.python.org/3/library/asyncio.html
2. Cpython source code, GitHub, 2019
   https://github.com/python/cpython
3. NodeJS About page, 2019
   https://nodejs.org/en/about/
4. Aiohttp Documentation, 2019
   https://aiohttp.readthedocs.io/en/stable/