Application Server Herd with `asyncio`

Jason Lai
ID: 204-995-126

University of California Los Angeles

Abstract

When implementing an application server, reliability and performance are key issues, especially for applications that involve complex architectures, a variety of transfer protocols, and rapidly-evolving data. However, one possible solution to these concerns is the "application server herd" architecture, which utilizes multiple servers to communicate information, but is less reliant on a core database to do so. Here, we explore one possibility in implementing this proposed architecture by using Python's `asyncio` library to devise an asynchronous, event-driven approach to the framework.

1 Introduction

For this assignment, we look at implementing a Wikimedia-style service architecture called an "application server herd", which utilizes multiple servers that communicate with one another to handle rapidly-evolving data, but maintain a stable database structure. In essence, when a client communicates new information to one of the servers, the servers will propagate this information, until all active servers have obtained and stored this information. This makes it so that multiple servers will have the same data without ever having to communicate with the database, and when a client requests information from any of these servers, the servers can provide said information for the client. This approach in application server architecture is designed as a solution to potential bottlenecks that are present in application servers that must process new information frequently, handle varying transfer protocols, and receive queries from mobile clients.

To consider different ways of implementing such an architecture, we look at Python's `asyncio` asynchronous networking library as a potential replacement for the Wikimedia platform. Note that this is Python 3.7.2. To decide whether our choice of Python and the `asyncio` library is a good candidate suited to our needs, we consider Python's API and implementation, as well as the `asyncio` library's API and implementation, and compare them to their Java-based counterparts: the Java language and the Node.js framework.

As an exercise in testing these aspects of our implementation, we design a simple proxy herd for the Google Places API. This proxy herd utilizes up to five different servers, which can each take two different commands: 'IAMAT' and 'WHATSAT', from clients.

2 `asyncio`

Python's `asyncio` library is the framework of focus for this exercise. The library implements several functions for the user to utilize "event loops" to write programs with asynchronous executions. In an event loop, the loop waits for incoming messages and performs executions according to the received messages; such executions are called "coroutines". In the event loop, since asynchronicity isn't true parallel programming, the loop has the ability to pause executions of coroutines, while waiting for events to occur or for other coroutines to finish their executions [1]. Functions that perform as coroutines are declared with the `async` keyword and can be "paused" using the `await` keyword. In the context of this exercise, the servers are implemented using event loops which wait upon incoming messages from clients to begin execution of coroutines. Each client with a connection open to a server can send requests simultaneously and the event loop will schedule and process each one. This allows servers to collect requests via the event loop and process each one as a coroutine in the loop, thereby reducing the downtime of the server doing useless work, giving the illusion of executions faster than those characteristic of single-threaded programs to client.

3 Implementation

To start with the implementation of the server herd, we consider key components of the herd's functionality. We must consider the structure in which servers communicate with one another, how each server communicates with the client, and how the servers process each command.

3.1 Structure of Server Herd

The structure of the server herd is comprised of five different servers with the IDs: 'Goloman', 'Hands', 'Holiday', 'Welsh', and 'Wilkes'. Each server is assigned a port and, for the purposes of this exercise, the servers communicate with each other bidirectionally in this configuration:

- Goloman talks with Hands, Holiday, and Wilkes
- Hands talks with Wilkes
- Holiday talks Welsh and Wilkes

A client can communicate via TCP connections with any of the five different servers so long as it is available, and depending on which server is given an 'IAMAT' command, we use a "flooding algorithm" to propagate information across all available servers in this configuration of connections.

The ports assigned to each server are hardcoded in the program using Python's dictionary feature with the server IDs as the *keys* and the port numbers as the *values*. To represent which servers a specific server can connect to, a dictionary is again used with the specific server ID as the *key* and a list of the server IDs of its adjacent servers as the *value*.

3.2 Event Loop Handler

In the main function of the server, we initialize the server using the `asyncio` function `start_server()` and pass in our handler function for reading and writing to clients, the host address, and the port number of the server taken from our dictionary of ports. Once the server is initialized, the event loop is established and we can now do seemingly simultaneous reads and writes to clients and other servers as coroutines.

The handler function `handle_echo()` is essentially an event loop which, upon every open connection to the server, the handler function awaits to read in single, end-of-file-terminated requests. These requests can either be an 'IAMAT' request in the form:

```
IAMAT [client ID] [geometric
  coordinates] [time sent]
```

or a 'WHATSAT' request in the form:

```
WHATSAT [client ID] [radius] [item
            count]
```

Once a command has been processed, the server then writes the response over the open connection and the the loop moves to the next coroutines that it receives. This allows for asynchronous processing of requests as each connection or received request can be handled concurrently through the event loop.

Besides reading and writing, the event loop also records the timestamps at which every message is received, and uses auxiliary functions to verify if a message is a valid command or not. Invalid commands are responded to with a specially formatted message of the form:

```
? [invalid message]
```

This response is also written back to the client if the client's 'WHATSAT' request could not be fulfilled on the account of the server not having the requisite data.

3.3 IAMAT Requests

Upon receiving an 'IAMAT' request, a server first verifies the two numeric operands: the geometric coordinates and the POSIX timestamp of the request. Both operands must be floating point values, however, the geometric coordinates are a string consisting of two signed floating points, representing latitude and longitude, and must be separated to be evaluated.

After the operands of the 'IAMAT' are validated, a separate auxiliary function then asynchronously follows the protocol to fulfill the request. In this case, the servers has three tasks it must do for an 'IAMAT' request; it must store the information if it's the most recent request, construct a response to the client in the form of an 'AT' message, and it must relay the information provided by the 'IAMAT' to all of its connected servers in the aforementioned configuration.

First, to decide whether to store the information or not, we check the client ID to see if it exists in our dictionary of clients. If it doesn't exist, then we store the information; if it does exist, then we compare the POSIX timestamps and store the information with the most recent timestamp. To construct the 'AT' message, we compute the difference between the received timestamp and sent timestamp of the request and, using the `.format()` function, we construct a string consisting of 'AT', the name of the server the client talked to, the time difference, and the operands of the 'IAMAT' as follows:

```
AT [server name] [time difference]
[client ID] [geometric coordinates]
            [time sent]
```

To propagate the information, we use a simple flooding algorithm that is reliant on the 'AT' response constructed by the recipient server. In the flooding algorithm, we essentially have the server that receives the 'IAMAT' act as a client and opens a connection to all servers available to it according to the connection dictionary. If a connection to a server is successful, we send the 'AT' message to that server. In every server, there is a separate protocol that handles 'AT' messages, wherein, we process the 'AT' message to obtain the pertinent information, similar to the protocol for processing 'IAMAT' messages. We decide whether or not to store the information, and an important distinction to make is that we intuitively only propagate the information further if the information does not already exist in the current server. In essence, this means everytime we store information, we propagate it. This follows the fact that, if a server already has the information, then all other servers likely have or will have the information at some point as well, barring any unforeseen issues. This prevents an infinite loop in the flooding algorithm and reduces the amount of redundant propagations of the information.

3.4 WHATSAT Requests

When a server receives a 'WHATSAT' request, the server first verifies the numeric operands similarly to how 'IAMAT' requests are verified, except the radius and information bound have their own bounds which must be verified. The radius must be at most 50 (in kilometers), and the information bound must be at most 20.

After the operands of the 'WHATSAT' are verified, an auxiliary function then asynchronously performs all tasks associated with the command. This entails retrieving information of the specified client's last known location and a JSON-formatted Nearby Search result from the Google Places.

To first get the information, we check if the client ID exists in the dictionary of clients; if it exists, then the server constructs an 'AT' response, however, if it doesn't exist, then a '?' error message regarding the 'WHATSAT' request is returned.

To retrieve the Google Places API, we first obtain an API key from Google; this key is then hardcoded in the server. We then specify the string formatting of a Google Places URL for a Nearby Search request, which requires following parameters: the API key, the latitude and longitude of the specified client, and a radius given by the 'WHATSAT' command. Once the URL has been formatted, we use the `aiohttp` and `json` modules to get the JSON on nearby landmarks from an aiohttp request, and append the JSON to end of our response on a new line as follows:

```
AT [server name] [time difference]
[client ID] [geometric coordinates]
            [time sent]
               [JSON]
```

4 Analysis

4.1 Python vs. Java-based: Type Checking

Unlike Java, which is a statically typed language, Python is dynamically typed, so Python performs type checking at runtime as opposed to compile time. Advantages of dynamic typing is Python's *duck typing* feature, which makes it so that certain variables' types don't have to be known by the compiler/interpreter until actual computation involving those variables. This allows functions to be written without strict typing rules imposed on the variables and return values; the user can freely define variables without worrying too much about an unknown function's return type. In combination with Python's built-in string, list, and dictionary processing features, the dynamic typing of variables allows for easy, short, and intuitive implementation of asynchronous functions.

However, Python's performance may fall behind Java by virtue of its interpreter and type checking. Since Java is statically typed, this places greater emphasis on security and stability of Java programs before they are compiled and ran; Java's JIT (Just-In-Time) compiler must affirm that the types of the program agree. However, once this compilation process is finished, a Java program may run faster than its Python counterpart, due to Python's interpreter having the overhead of assuming and checking types at runtime. And, for a system such as application server herd framework that we're interested in, static type checking gives an added layer in security as it ensures that the program is robust to a degree. Furthermore, due to the strict type rules imposed on variables in Java, this can make the program more readable, as viewers of the code do not have to infer the types of variables like they would for Python programs.

4.2 Python vs. Java-based: Memory Management

Both Python and Java have their own garbage collectors, placing less emphasis on the user's ability to account for all data structures stored on the heap. However, their implementations of garbage collection differ as Python uses *reference counts*, whereas Java uses *mark and sweep*, generational garbage collecting.

The advantage of Python's reference count garbage collection is that reference counts keep track of how many references to an object in memory there are, and once this count falls to zero, the object is deleted [2]. This allows for almost immediate reclamation of memory as soon as a variable is no longer necessary, however a separate data structure in memory must be stored in order to keep track of these reference counts. Furthermore, the reference count implementation suffers from issues with cyclic dependencies (where there is a cycle of object references), and if there are frequent allocations and deletions of objects (if an object oscillates between zero and one reference), greater external fragmentation of memory could occur.

The advantages and disadvantages of Java's mark and sweep implementation are essentially the inverse of Python. Java's garbage collector is not affected by cyclic dependencies; this is due to the mark and sweep method which utilizes data structures which store references to objects in data structures with a bit in each object's reference header representing a "mark" [3]. This mark tells the garbage collector if an object is good to delete, so, unlike reference counts, independently checking each object mitigates the issue of cyclic dependencies. However, Java's memory management will still suffer from external fragmentation to some degree, and reclamation of memory for an unnecessary object may take some time as the collector must sweep.

In terms of memory use, both Python and Java must use some memory in order to store data structures for their garbage collectors. Python must have a table for its reference counts, whereas Java requires multiple tables for free spaces, roots to objects, amongst other objects.

4.3 Python vs. Java-based: Multithreading

One of biggest differences between the Java and Python implementations is their support for multithreading; Java supports multithreading, whereas Python does not. Python's decision to not support multithreading stems from the inclusion of the *global interpreter lock* mechanism from the original CPython implementation, which a majority of Python implementations use. The idea behind the global interpreter lock is to ensure that built-in data structures such as dictionaries would not be affected by concurrent accesses [4]. It does so by locking the interpreter to a single thread, so processes running on multiple threads are forced to run one at a time, only when they can monopolize the interpreter. However, this design sacrifices the performance achieved by parallel programming on multithreaded machines that can be done in languages such as Java.

In the context of this exercise, despite a seemingly large performance gain due to asynchronicity, in reality, Python is still restricted to the performance of a single threaded machine; it's just that the event loop causes less waiting, but requests are still processed one at a time. In contrast, a multithreaded implementation of the server herd could process multiple requests at a time independently, which would drastically increase the throughput of the framework.

4.2 `asyncio` vs. Node.js

When comparing the Python `asyncio` library to Node.js, both can be used to implement a back-end server application. However, key differences in the implementation include performance, functionality, and API. In terms of performance, Node.js is in general faster than Python due to its powerful engine

based on Chrome's V8, leading to code being executed much faster [5]. In terms of functionality, Node.js is an asynchronous framework on its own, whereas Python must implement the `asyncio` library and its own event loop structure to support asynchronous programming [6]. In terms of their API, Python may be more development-friendly as it does include a variety of built-in functions alongside an extensive assortment of libraries that developers can use to suit their specific needs. Furthermore, despite Node.js being based on JavaScript, the framework has functions that were intended for asynchronous use, so developers using Node.js must have a better grasp on Node.js's specific functionalities to use the framework more effectively.

5 Recommendations/Conclusions

As far as a application server herd is concerned, whether or not `asyncio` is a good candidate to implement the framework will largely depend on the priorities of the developers. If an easy to use API is desired, then Python's `asyncio` library is a good choice for this application. Otherwise, I would not recommend `asyncio` as a good choice for this application, as considering other avenues such as the Java-based approach reveals that there is a significant performance gain opportunity that Python omits.

A multithreaded implementation of this framework in combination with optimizations from the JIT compiler at runtime for a Java-based approach could significantly increase the throughput and performance of this application. Since our main concerns about the original Wikimedia-style service were bottlenecks having to do with frequent client requests and data propagation, an implementation that takes advantage of performance gain may attack these problems better and, in general, scale better once the server herd architecture grows larger in number of clients and servers.

Aside from this, Python's dynamic type checking also raises some concern, as the program could have unforeseen bugs that may not be found until the actual implementation is used. In short, I believe other approaches to the server herd will outperform Python significantly, however, if asynchronous programming is the method of choice, then Node.js may present a better solution than Python still.

References

1. Cannon, B. (2016). *How the heck does async/await work in Python 3.5?*. [online] Tall, Snarky Canadian. Available at: https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/ [Accessed 5 Jun. 2019].
2. Docs.python.org. (n.d.). *1.10 Reference Counts*. [online] Available at: https://docs.python.org/2.0/ext/refcounts.html [Accessed 5 Jun. 2019].
3. Oracle.com. (n.d.). *Java Garbage Collection Basics*. [online] Available at: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html [Accessed 5 Jun. 2019].
4. Docs.python.org. (n.d.). *Glossary — Python 2.7.16 documentation*. [online] Available at: https://docs.python.org/2/glossary.html#term-global-interpreter-lock [Accessed 5 Jun. 2019].
5. Da-14.com. (2017). *Python vs Node.js: Which is Better for Your Project | DA-14*. [online] Available at: https://da-14.com/blog/python-vs-nodejs-which-better-your-project [Accessed 5 Jun. 2019].
6. Notna, A. (2019). *Intro to Async Concurrency in Python and Node.js*. [online] Medium. Available at: https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36 [Accessed 5 Jun. 2019].