Application Server Herd with Python's asyncio

Abstract

In an investigation to to avoid a potential bottleneck of the LAMP architecture, we explore the concept of an application server herd. Implemented in Python using its `asyncio` library, we access the suitability of this new architecture to better handle (1) more frequent updates, (2) accesses happening on protocols other than HTTP and (3) when clients are more mobile for a Wikipedia-like news service. After prototyping such herd, we give a more detailed discussion on the language/technology choices to consider when solving the problem.

1 Introduction

Currently, there are many different web-stack architectures to choose from, and each must be carefully chosen to solve their corresponding expected problems. In the case of Wikipedia, it uses the LAMP platform which consists of Linux, Apache, MySQL, and PHP while utilizing multiple, redundant web servers behind a load-balancing virtual router for reliability and performance. The load-balancing router would serve as a bottleneck and cause for many issues for the LAMP architecture, so instead we investigate the server herd architecture.

A server herd architecture is a group of servers which propagate information to each to other as it is received, so that when queried for such information, we can bypass fetching from a database. This paper explores a prototype of such a herd. Additionally, we discuss the advantages and disadvantages of designing such a herd using Python and asyncio, in comparison to other languages or frameworks such as Java or Javascript's Node.js

2 Design of Server Herd

The prototype consists of five servers which respond to requests from clients. Clients can send their location information to any of the servers, and this data is propagated based on predefined relationships between the servers. If a client queries for location information of what is around a particular client id, then any server should be able to respond quickly with the relevant information. The implementation of this process is discussed in more detail below.

2.1 Structure of Server Herd

The five servers used in the prototype are named after current UCLA basketball players, Goloman, Hands, Holiday, Hands, and Wilkes. Instead of passing basketballs, they are passing messages. The following figure illustrates the realionships in which they communicate with each other:

Figure 1.

Server Name	Talks with
Goloman	Hands, Holiday, Wilkes
Hands	Goloman, Welsh
Holiday	Goloman, Welsh, Wilkes
Welsh	Holiday
Wilkes	Goloman, Hands, Holiday

Using Python's `asycio` library, the servers are able to receive and handle TCP connections from clients asynchronously using coroutines. Coroutines define a structure of control where control is passed in an cooperative manner between two or more routines without returning. Asyncio uses a constantly running event loop which patiently waits for events to occur so that it can trigger subtasks. This way, servers are able to handle many connections at the same time without a

drop in performance.

Using `asyncio`'s transports and protocols, I was able to implement the server herd. Transports are simply communication channels, and protocols are classes which can be defined to broadly define network protocols in how data is received, communicated, and processed.

2.2 Protocols Used

I define 3 protocols for implementing the application server herd.

The main protocol used is ServerClientProtocol, and it represents how the server will handle a variety of different messages. The protocol is put into process by asyncio's event loop which allows for a server to be created. The server is easily able to listen to whatever messages are sent to the specified port corresponding with the correct server name. Section 2.3 discusses in more detail how such messages are handled by its internal methods.

The next protocol is the HTTPClientProtocol which allows a server to connect to the Google Places API and act as a client as it sends an HTTP GET Request to obtain information. When it receives data from Google, it cleans and processed the data before sending it back to the client.

The final protocol is the ServerToServerProtocol which simply defines how data is propagated from server to server. This section serves as a brief introduction to the protocols, but more detail will be discussed over the next sections.

2.3 Messages

Clients can communicate with servers using either an IAMAT or WHATSAT message.

Valid IAMAT messages are processed by the ServerClientProtocol which will update a specified client's stamp of information. IAMAT messages use the following format:

IAMAT <client id> <location> <time>

Location is specified using longitude and latitude in ISO 6709 notation while time is specified in POSIX format. When the ServerClientProtocol updates the client's information, the server will then propagate the information using the ClientProtocol by flooding the information to the other servers based on the relationships defined in Figure 1. Lastly, the ServerClientProtocol responds back to the client with an AT message using the following format:

AT <server specified> <time diff> <IAMAT content>

Valid WHATSAT messages are sent from clients to any of the five servers to gather information about what places are around a specified client id. WHATSAT messages use the following format:

WHATSAT <client id> <radius> <num results>

The ServerClientProtocol will fetch the previous time stamp of the specified client and will gather a valid HTTP request message but then will pass the remaining duties to a newly created connection based on the HTTPClientProtocol. The HTTPClientProtocol takes the built request information and uses two transports to write the data. First, it uses a transport gathered from simply connecting to Google Places API to make an HTTP request. Then, it uses the original transport specified in ServerClientProtocol to write the message back to the client in a nice clean format.

Servers communicate with each other by spreading AT messages to each other. When an IAMAT message is originally processed by the ServerClientProtocol, and the time stamp needs to be propagated, a new connection is created based on the ServerToServerProtocol. These messages also contain which servers have already been visited or has had client stamp info already spread to them in order to avoid being redundant.

If an unknown or invalid message is detected, then the message is sent back to the client by the ServerClientProtocol with a '?' prepended to it.

3 Asyncio for the Problem

In this section, I detail how asyncio fared in creating this application server herd prototype and discuss strengths/weaknesses. Lastly, it will discuss the particularity of the Python language for this problem.

3.1 Asyncio's Advantages

Asyncio is able to solve the problem of multiple incoming requests being handled in a quick and reliable fashion. Each server has an event loop running in the background ready to handle or create new asynchronous connections. It is ready to handle requests from clients,

while also receiving information from the other servers. Due to its asynchronous nature, messages no longer need to wait for a connection to complete before handling another process. A synchronous server herd would halt, which would never be acceptable for the Wikipedia-like news service.

Additionally, in the circumstance where servers are receiving data from other servers, the asynchronous nature allows the herd to receive the data without sacrificing the ability to respond to any current requests. Thus, in the face of the client, the servers are always ready.

Writing and developing a server herd is a difficult task, but many of pains that would intuitively come with it are minimized or erased using the asyncio library. Servers are able to run using the same code, and as long as there are ports available, adding new servers becomes highly scalable. In comparison to using a multithreaded program, asyncio makes the nature in which events are handled more understandable for the programmer.

3.2 Asyncio's Disadvantages

By definition, asynchronous means the order in which processes are handled is not known ahead of time. So, that means that requests are not necessarily processed in the order they arrive. This means that a potential WHATSAT message could be handled before an IAMAT request for a particular client id, leading to errors or bugs. Thus, reliability is worse in comparison to a synchronous model, but performance is gained. Since servers all depend on the same code, having to make edits to them will shut all of them down at the same time, which can potenitally be a major issue.

4 Python for the Problem

In this section, I discuss potential worries about Python's type checking, memory management, and multithreading in comparison to a Java-based approach to the problem.

4.1 Type Checking

Python uses dynamic type checking, meaning that is checks for correct types at runtime. This could lead to potential errors at runtime that would have otherwise been caught by a statically typed language like Java. An advantage of this type of dynamic checking is its ease of development for the Python programmer in comparison for the difficulty of development for a Java programmer which must continually check for this. An adjantage of static type checking is that is allows for easier reading and possible documentation as the types and relationships between variables and functions are well defined. While dynamic type checking can lead to some possible runtime errors, there are many reliable frameworks based on "Duck-typed" languages such as Django (Python) and Ruby on Rails (Ruby). The ease of development proves to be the biggest advantage as more can get done in a shorter period of time.

4.2 Memory Management

Both Java and Python use a heap for containing all objects and data structures. However, these two languages differ in terms of their garbage collection. Most importantly, Java's garbage collector will delete objects which have not been in use for an extended period of time. Python's garbage collector uses a different method, using reference counts to decide whether objects should be deleted. For Python, objects are stored with an additional data field specifying the number of times is referenced in memory. It is incremented when the object is referenced by another object and is decremented when the other object does not reference it. Python will use this reference count to determine the objects which are no longer being used and should be deleted.

Memory management would be more efficient with Python since objects are immediately erased as soon as they are no longer referenced. For a large asynchronous application which uses lots of memory, this memory management model will prove most efficient. In comparison with Java, objects are based on time for deletion, and will lose out to a model which has an assurance for reliability for deleting an object immediately as soon as it is longer referenced.

4.3 Multithreading

Python is not a strong language for multithreading in comparison with Java. Java uses multicore processors while using multithreading to get more work done in the same amount of time. At Python's core, it has an global interpreter lock to synchronize threads so that only one thread can run at a time, even on multicore processors. However, since Java depends on these multi-core processors for its strength in using threads, it is system dependent. Python has horizontal scaling in the fact that is will run the same across all different systems. If Java has the correct system, it will win out and is more scalable in terms of performance.

5 A Comparison of asyncio and Node.js

This section briefly compares the overall approach of `asyncio` to that of Node.js. Node.js is an event-driven, asynchronous framework for writing code on the server side written in the language of Javascript. Just like asyncio, Node.js functions are scheduled as callbacks, though they are called "Promises", on the underlying event loop and may be executed until a yield is given.

Node.js is based on Chrome's V8 engine, and allows for the same language to be used on both the frontend and backend. In terms of performance, Node is significantly faster[1] as it is based on the very fast and powerful engine. However, Node.js is not the best option for processor intensive applications and with an application such as the prototype, which has to handle lots of connections efficiently, this is an important drawback. Overall, both Python and Node.js are dynamically typed allowing for easy development on both sides. Node.js should be experimented with for implementing the server herd, especially if a web application is the desired end product. For this particular server based application, asyncio proves most suitable.

6 Conclusion

In this paper, we discussed a prototype of a server herd written in Python using its `asyncio` library. Asyncio proved extremely suitable with this choice of architecture as it was able to receive, handle, and process many requests simultaneously. It improved upon the LAMP architecture in terms of performance, as data no longer needed to be fetched from a MySQL database to obtain client information since client time stamps were propagated throughout the herd. We then assured how Python proved to be a suitable language, with its strength being in the ease of development that dynamically type checking offers along with its more efficient memory management model in comparison to Java. Lastly, we drew a brief comparison between Node.js and asyncio, discussing their own similarities and potential drawbacks. In the end, asyncio proves the best technology for this particular problem but Node.js should be further explored if a web application is the final destination of the product.

7 References

[1] Python vs. Node.js: Which is better for your project. https://da-14.com/blog/python-vs-nodejs-which-better-your-project