CS 131 Proxy Server Herd Application Project Fall 2016

Jahan Kuruvilla Cherian - *University of California, Los Angeles*

Abstract

There are many stacks upon which web based architectures work, with the LAMP stack powering the Wikimedia functionality. However, for some kinds of data operations that require quick and in-sync updates, such an architecture with database communication may just not be worth the lack of efficiency.

With this need comes the potential of using an application server herd, for which the servers act as intermediary proxies between each other, pushing updates for every inter-server communication for which the Twisted event-driven framework might be useful.

Introduction

Given the task of implementing an application server herd so as to be able to propagate server updates quickly to all servers so that they might all be in-sync began as a seemingly daunting task, but choosing to use the Twisted framework within Python made it such that the task didn't end up being too difficult. This is where Python as a language truly shines, and this report will continue to highlight my design choices made in creating the server herd implementation and then continue onto compare Python's dynamic duck typing, reference count based memory management and the event-driven nature to the analogous alternatives present in Java. The final comparison made will be with reference to a Node.js server, and how the efficiencies and implementations would differ between the Twisted model. We will conclude on whether choosing to go with Python and Twisted was a good and valid decision for the company in question based on its current needs.

1. Application Design

The application was completely written within a single *servers.py* file using the Twisted's event driven networking through the use of a reactor. This reactor runs a continual loop within which a Server Factory builds the server protocol. However, because the server needs to constantly communicate with it's neighbors, it must also have the ability to behave like a simple client, which is why the file also implements a Client Factory with a similar client protocol to build per instance.

The bulk of the work is done within the server's protocol, wherein the server has to handle the data it receives accordingly, making sure the data is checked against containing any errors, and upon producing the associated result, said result is correctly and safely propagated to all the server's neighbors.

The factory in a high level overview is involved in generating the server protocols, starting it and creating a logging file to which all updates are pushed for further inspection later on.

The client side of the server is much simpler in that all it needs to do is transport the data through the TCP socket.

The entire application is linked through a main function that makes sure the input parameters to the application are valid and that the server factory is set up to listen on it's respective port through TCP sockets, for reliable inter server data communication.

Throughout the entire application, my key aim was to make sure the implementation was as modularized as possible, and thus I finely divided the functionality of the server into three key components:

1.) Handling IAMAT messages

2.) Handling WHATSAT messages

3.) Handling AT messages through propagation

In order to handle these messages effectively and ensure valid propagation of data to neighboring servers, each instance of the server was given a cache – so that the server can act as a proxy – to store the key-value pairing of client IDs to the messages associated with them.

1.1. Handling IAMAT Messages

IAMAT messages when sent from a client contain the client ID, GPS location and the approximate client timestamp in UTC format. Given these three key parameters within an IAMAT message, it is imperative to check that the message is a valid one.

I broke this down into ensuring that the message had 3 parameters, and that the types of the parameters were valid. Because Python is a dynamically typed language, each object is bound to a type at runtime and this is

done through a methodology done known as *duck typing* wherein the Python interpreter checks to see if the operation upon the object makes sense with the object's bound type. Because of this, it is important to see if the input parameters would be able to be cast to their respective types.

This can be seen when checking the client timestamps, the latitude and the longitude. In order to make sure all these parameters are valid, we first break up their string representations by splitting the data up into fields delimited by whitespace. The combined GPS result is then broken up into its latitude and longitudinal coordinates by iterating to find the unary positive and negative operators. Upon doing so, the latitude and longitude are compared to reasonable ISO 6709 standard based bounds to make sure they fall within the range, and are then finally tested to see if they are valid floats.

This explicit casting of the time and coordinates is employing to some degree a static type check to ensure safety – something a static typed compiler would check against such as in Java.

For Error Handling within any of these situations, we log the error and output a ? followed by the data containing the invalid parameters.

Given a valid input, we proceed to handle the IAMAT message, by creating an AT message. This message is constructed so that the additional parameter of a time difference can be prepended with the server name to the original IAMAT fields. Because this AT message is a location update of sorts, we need to make sure that not only does the client receive this through the TCP socket, but also that the information is added to the local cache and is propagated to all the neighboring servers. However, before saving the message and propagating it, we add an extra, custom parameter to the message, which is the server's current timestamp. This then uniquely identifies the message as having originated from the current server, so that in the future we don't get stuck in a propagation cycle.

It is also worth looking into the caching update policy when discussing the local save of information. It only makes sense for the server to store the most recent location update, which can be indicated through the client's timestamp. That is if the server already contains an entry for the given client, then a new location is only added if the client timestamp on the new message is greater (newer) than the stored timestamp. Otherwise the message is considered void, and the cache is not updated. The decision to still transport the message is still an important one as it means the client will receive a response from the server ensuring its functionality. Note that the information propagated is whatever is within the cache, which as mentioned before might not be the latest sent message, but rather the most up to date location based on the most recent client timestamp.

The propagation worked by using the Client Factory discussed above, to generate clients emulating the ports of the neighboring servers, to which a TCP connection could be made to sent data through.

1.2. Handling WHATSAT Messages

The key functionality behind the WHATSAT message response was to make the Google Places API request to find the nearby places within the given radius, limiting the results to that specified in the messaged.

As with the IAMAT message, there was an initial message parameter check that ensured the correct number of arguments were given, and that the radius and the limit were valid integer values, capped at an upper bound of 50 and 20 respectively. And just as before, the error handling would take any invalid input and through a ? with the respective data that caused the error.

We begin by generating the valid API endpoint by referencing the latitude and longitude values stored in the server's instance, using the given radius (converted into meters by a multiplication by a 1000) and the API Key stored in the configuration file. Once the endpoint URL is created, the WHATSAT Handler uses Twisted's web client module to make an asynchronous call to get the page at the specified URL as text in a JSON format (as per the endpoint).

In other languages such as Java, most calls could be made through the use of a another thread, wherein we could achieve parallelism through the use of a multiprocessor system where one core can take the burden of downloading the webpage while the main thread executes other tasks, and upon completion, Java's concurrency modules would send an interrupt to signal the completion of the download. This ideology goes against Python's implementation, since Python was designed under the influence of a *Global Interpreter Lock*[1]. This in essence means that any Python program runs within a single thread, with the design mindset being that if the interpreter rarely makes calls outside of the interpreter for long periods of time, then there is minimal speedup to be gained on a multiprocessor system. This is why the Twisted API relies more heavily on the use an event driven loop, which continues to be the most common way of running asynchronous servers, and so in this instance, Python's lack of concurrent programming models as compared to Java, is not a con, but rather a pro, as it allows us to perform non-blocking based parallelism more effectively.

Through this method, any explicit asynchronous call requires a callback to attached to the deferred object returned from the call. In this instance we add two callbacks to the response from the getPage call. One is for explicit error handling, while the other is to process the response that we got back. Because the response is in JSON format with no limit on the number of results, we first set the results of the response to be limited to the value given by the WHATSAT message. We then formulate and transport the final message as the concatenation of the the AT message stored for the given client ID within the local server cache along with the limited JSON response, indented accordingly.

1.3. Handling AT Messages

The final set of messages that require explicit handling, are the AT messages. While these messages aren't explicitly sent by the client itself, they are the messages that are propagated to all the neighboring servers, and thus must be appropriately handled.

As mentioned earlier, these messages are the basic AT messages created as a response to an IAMAT message, but with the inclusion of an extra parameter – namely the timestamp from the originating server. As with all the message handlers, the first step in handling these messages is to perform a validation, ensuring that there are in total 7 parameters and that the parameters are of the correct type and format.

Upon validation, the handler begins by updating the server cache. Note that as discussed in Section 1.1, the update cache method ensures that the cache is only updated if the message it receives is newer than the stored messages. The final step is to check if the propagation should continue, or whether it should come to an end. This is determined by the final unique timestamp within the message. If this timestamp is the same as that which is stored in the server cache, then we have come back to the server that first sent the message, thus ensuring that we have updated all neighboring proxy servers to keep the information in sync, and that we can end the propagation cycle. If the values were not the same, then this message would be propagated to the neighboring servers.

2. Python and Java Comparison

To address the concerns about whether Twisted is a suitable framework to use in practice, we first look at the areas concerned around Python's type checking, memory management and multithreading. Note that throughout this discussion, the assumption is based on Python running on top of the *CPython* interpreter.

Twisted is simple to use, stable and mature [2] and above all else, is an excellent abstraction to several low level implementations that would otherwise be harder to grasp. Because of this, it makes development of such an application server herd extremely quick, easy and robust. Because Python is a dynamically typed language, the developer need not worry about figuring out the types of objects they need to use, and can essentially write the server implementation as if it was as simple as writing pseudo code. This makes it extremely quick to build out applications. In comparison to Java's statically typed nature, it makes the program very readable, but suffers from some lack of clarity. Because Python's interpreter will employ duck typing to figure out whether what type the object should be based on its contextual usage, it is hard for an external reader to fully grasp the details of the application, such as what object is being passed around and operated on. In essence, the application becomes easier to read and understand on a high level with Python, but might prove to be a little more cumbersome when trying to understand the details of the application.

Python's memory management system makes using the Twisted framework a pro. Python creates references to the objects it creates and then uses a mixture of *reference counting* and *non-moving mark-and-sweep garbage collection* as compared to the standard *mark-and-sweep garbage collector* [3]. The reason Python works well for this application is because of its reference counting, which essentially stores all counts of references to objects, and when said reference falls to zero, the object gets deleted. The use of the garbage collector in Python is to delete cycles that can arise from cyclic references. But this essentially ensures that within our application server herd all the objects we create are quickly deleted when they are no longer referenced, rather than having to wait upon the garbage collector to free up the memory. Because Python is essentially purely on a heap based memory model, with no constraints, the application becomes easier to develop as the programmer need not worry about allocating objects on the heap as in Java, with no use for the keyword *new*.

Finally, the Twisted framework is brilliant in that it uses a single threaded, event driven asynchronous model with async I/O to perform networking operations, and this is mainly because of Python's single threaded implementation as mentioned earlier in section 1.2. Java on the other hand would use a multithreaded approach, which would rely more heavily on the system

the server is running on, with a direct correlation to the number of processors available. This means that creating an application of similar stature in Java would be faster and slower based on the machine architecture, while with Python's Twisted framework, the reactor loop will ensure that performance is the same across all types of machines, and thus can benefit a lot more through horizontal scaling, as is the need for a server herd.

3. Twisted and Node.js

In essence Python's Twisted framework is very similar to Node.js in that both are asynchronous, single threaded event driven frameworks for networking. The key distinctions lie in the age differences and the languages themselves.

Twisted is an extremely mature and well experimented and used API that is extremely well abstracted and due to Python's simple syntax and conciseness, is quick to develop with. However, Node.js is a very new framework built on top of JavaScript, which is also a runtime interpreted language, built explicitly for the web. Even though Node.js is newer than Twisted, it is in rapid development and is supported through a massive community. It's core functionality is a lot simpler and lightweight than that of Twisted, but can be easily appended onto using *NPM* which is the package manager to add community supported packages to your application.

Another positive factor for Node.js is that is runs atop Google's V8 JavaScript Engine, which makes it extremely easy to integrate with pre-existing web applications that require a server. Because Twisted is built in Python, which is more of a scripting language, it's integration into web based servers is a little more convoluted.

Both Node.js and Python Twisted based applications are extremely reader friendly as they use simple and concise syntax, with Node.js being a little more on the strong typed side in comparison to Python, which might make it easier to understand the details of the application faster with Node.js. However, until the recent introduction of ES6 and ES7, Node.js was not an object oriented framework which might make it harder to modularize and work with code in comparison to Python's Twisted framework.

Twisted uses the idea of deffered objects to be able to attach a chain of callback functions to, in order to provide asynchronous operations upon events. Node.js does something similar with the idea of *promises* [3] which are resolved once and their respective callbacks are executed only once – very similar to Twisted's method of doing things. However, Node.js also provides the ability to create closures in that states are referenced by functions and can then be called when an event is complete, allowing the chaining of closures to effectively act as stateless machines within themselves. This means the Node.js has a little more flexibility in terms of its mechanics.

Despite it's youth, Node.js has been massively popular in both small and large scale projects, and is thus a very enticing alternative to Twisted. The choice is very heavily biased on the purpose and intention behind the project. Node.js would serve as a better choice for web based server side applications, while Twisted would work well in implementing hardcore networking API's and server residing applications, such as the application server herd.

4. Conclusion

Twisted has proven to be an extremely simple and powerful framework running on top of Python which is a brilliant and easy language. It allows for extremely quick development and maintainability. The code is heavily modularized within Twisted and acts as a beautiful abstraction layer between the low levels of networking. I would very strongly recommend its use, and am extremely confident in its robustness. However, it is worthwhile looking into Node.js as an alternative to creating the application herd. Because Node.js is built on top of the asynchronous model, while Twisted incorporates it into a global lock based model, it is believed that the former can run slightly faster than the latter. This is worth testing through specific benchmark tests that could emulate the application server herd prototype for the Google Places API.

References

[1] Global interpreter lock. (2016, May 1). In Wikipedia, The Free Encyclopedia. Retrieved 23:34, May 1, 2016, from https://en.wikipedia.org/w/index.php?title=Global_interpreter_lock&oldid=718173703

[2] Tezer, O.S (2013, October) *Comparison of Web Servers for Python Based Web Applications*. Retrieved from https://www.digitalocean.com/community/tutorials/a-comparison-of-web-servers-for-python-based-web-applications

[3] Stott, N (2011, December) *Asynchronous Control Flow with Promises*. Retreived from https://howtonode.org/promises