Building an Application Server Herd with Python and Twisted

Katherine Bell
UCLA, Fall 2016

Abstract

This report explores the Twisted framework as a viable solution for implementing a fast, scalable and reliable server herd intended for a production setting. A simple server herd prototype was built for the purpose of discussing some of the benefits and drawbacks of this strategy. Both its implementation and these opinions will be discussed in further details in the pages that follow.

1. Introduction

The goal of my work was to find an alternative to the Wikimedia architecture, which relies on LAMP and as such a field of servers with identical functionality choreographed by a load balancer. While this approach guarantees a degree of reliability and overall performance, our needs require a design geared towards fast and frequent updates, as well as one that allowed for various protocols other than HTTP and a greater mobility for clients. Moreover, the application server in the Wikimedia strategy is seen as a global bottleneck—one that causes trouble both in terms of the difficulty in adding servers and the performance hit embedded in the need to route all requests through a single portal.

In the quest to answer these perceived issues, an event-driven networking engine called Twisted has been suggested as an alternative solution. A different architecture is also suggested: "application server herd" architecture. This combination suggests a more dynamic and flexible strategy fueled by Twisted's asynchronous capabilities and the server herd's distributed nature. In order to fully explore the benefits and drawbacks of these suggestions, we've decided to build a simple prototype.

2. The prototype

The prototype is a simple application that connects five servers in a server herd architecture. Each server listens to a unique port and a given client is able to communicate to any of the ports with a request. The servers are able to handle multiple different requests relating the locations of these various clients. A client is able to update its location (through the "IAMAT" command) as well as query nearby establishments to other clients (through the "WHATSAT" command). In the first case, the server receiving the request is then in charge of storing the location data in a local cache as well as distribute it to its neighboring servers. In the second, the contacted server queries the Google Places API for the relevant information and passes it along to the client in an asynchronous fashion.

This simple program covers a number of Twisted's most fundamental capabilities to be discussed further in the following sections.

2.1 Implementation

The implementation of our prototype was straightforward. This in part was due to Twisted's approachable interface to building server herds and clients. Each server in the herd contains both the application logic to query Google and propagate communicated data as well as the ability to store its own local version of these communications. In this way you've taken the role of the application server and distributed it across multiple nodes. Each one is able to operate completely autonomously and respond to clients regardless of the status of the other servers. Data is propagated from one server to another through a simple flooding algorithm.

The initial architecture of this program was built largely on top of Twisted's internet and protocols modules, which include the ClientFactory, ServerFactory and LineReceiver classes. These classes served as the foundation for the client/server interaction created. The Line Receiver class dictated how data was received, processed and transmitted. Within this class, Twisted documentation provides a barebones outline of functions to get you started, namely the ConenctionMade, ConnectionLost and LineReceived functions. Adding the functionality desired was relatively trivial once this framework had been established.

An interesting aspect of this work was that our application provided both the server and client implementations and the logic within the program interwove these

categories. Because each server was responsible to communicate new data to a select number of the others, it had to take on the role of a client as it transmitted data.

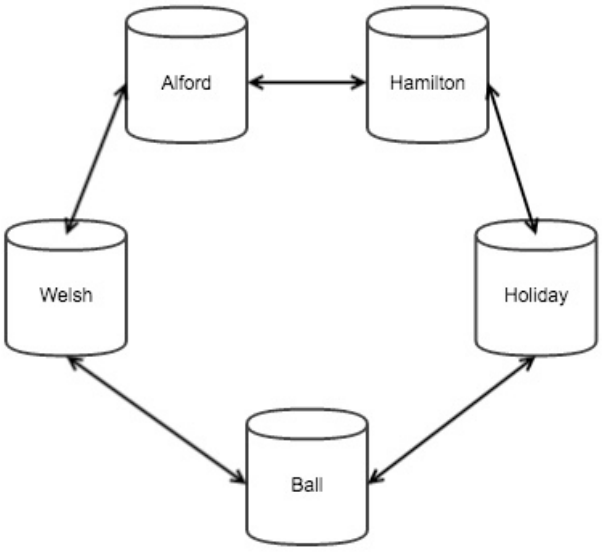

Figure 1: Communication between servers in the herd

2.2 The Flooding Algorithm

Figure 1 above depicts the communication channels between each server in our herd. The salient aspect of this network is that each server does not communicate with all others. Rather each has two "neighbors" to which it is responsible for passing data. However, a key goal of our architecture was fast and reliable communication across the cluster upon each atomic update. The way we were able to achieve this was through the implementation of a simple flooding algorithm. The idea behind this algorithm is that if each server contains a path (direct or indirect) to every other server, then full saturation of information is guaranteed. Welsh communicates to Ball and Alford which in turn communicate to Hamilton and Holiday respectively.

While effective, this algorithm naturally has a major flaw to consider: the possibility for an infinite cycle of the same information. In order to avoid this kind of redundancy two breaking points were implemented in the propagation of data: The first is that no server passes the same information back to the server that sent it and the second comes from checking the timestamp of the newly acquired data with that which was already stored. If the new data is either the same or older than the current data, the information cycle is stopped as there is no need to continue its propagation.

2.3 Limitations of the Design

This quick prototype was exactly that, quick. And in its straightforward implementation it lacks in a few key areas. The most significant of which is in the way that it stores data. While an undeniable performance benefit is gained from only locally storing data (i.e. not traveling back and forth to a database or another more persistent data store), this implementation easily looses data if a server is not kept running. This means that while the herd might otherwise be aware of a client's location a particular server may respond to an inquiry with an error if it had been down since the time of original reporting.

2.4 Logging

A short note on logging: Python's logging library is incredibly simple to implement and integrating it within this simple prototype was facile. Each server maintains it's own log file, which serves as a record for connections gained and lost, data received and transmitted, and any errors that the application may encounter. I chose a simple logging format, which included the relevant data in question, as well as the method being evoked and the time of said execution. This baseline logging strategy enabled fast development and debugging.

2.5 Running the Prototype

Running the prototype requires the 5 servers to be started. Individual servers can be started from the terminal with the following command "python project.py <server_name>" or all servers can be started at once using the runsevers.sh script which can be found in the project's directory. Clients can connect to these servers via their ports (once they are up) using a telnet command following this structure: "telnet localhost <server_port>." For a more comprehensive look at how the clients and servers communicate see testproject.sh, which both starts the servers and uses telnet to communicate with them.

3. Working with Twisted

Depending on the source, Twisted is either complex and overly featured or simple and easy to develop on. My experience was the later. Though the documentation can be obtuse and sparse, I found that Twisted's model of abstraction made creating clients and servers fast and intuitive. I was able to create a working prototype (including both a server factor and client implementation) in about 250 lines of code. As the adage goes, fewer lines of code mean fewer mistakes. Furthermore, though the Twisted library itself is extensive, I only needed to import a few modules to run the prototype, which meant that the runtime overhead was kept small.

3.1 A few Practical Considerations

Twisted is open source and open licensed so the ability to understand its inner workings and adjust them how you see fit is undeniable. One of the requirements for this project was that the chosen framework must make it easy to glue new applications (presumably using the framework) to existing ones. Because of the flexible and customizable nature of Twisted, the goal of backwards compatibility seems much more feasible then if one were to work with a black box program whose nuanced workings were made unavailable to the developer.

In terms of reliability, Twisted is built on top of Python and is therefore able to take advantage of and extend Python's comprehensive error handling. This means Twisted offers stability in its performance. Twisted has a large community of developers and is well maintained. Both of which are significant considerations when deciding on a technology that will become a core aspect of a company's product. Furthermore, Twisted's asynchronous programming approach allows for great efficiency under significant load.

3.2 Asynchronous Programming

A discussion of Twisted that did not include mention of its asynchronous programming style would be remiss. It is one of the hallmarks of Twisted's programming style and one of the core features that makes Twisted an interesting contender for the company in question. It allows Twisted to be a dynamic single-threaded program that makes up for its lack of parallelism through this event-driven strategy. Twisted doesn't wait for a connection to be made, it allows for the developer to instead schedule a block of code to execute when the response is completed. This is done through the Twisted deferred class.

A deferred object is essentially an object that when created represents something that will be created in the future (or error out). It's a kind of placeholder. The prototype uses this strategy when it issues a remote connection to the Google API and attaches a callback function to the method, which then only executes when the request to Google has been returned. This is all made possible through the reactor module, which functions essentially like a big loop that monitors the protocols and executes the callback functions when the necessary event occurs.

This event-driven strategy allows for Twisted to handle many connections per thread rather than having opening a single thread for each. This means greater flexibility and scalability in the end. The developer doesn't need to worry about managing the thousands of idle threads. Furthermore, issues like data race conditions can be more easily simulated in a synchronous environment. This allows for easier debugging and greater overall stability. So while Twisted certainly doesn't guarantee data races will not occur, it cuts down on the time it takes to catch these kind of issues and develop solutions to address them.

However, it must be mentioned that there are necessarily drawbacks with a single-threaded method. And while Twisted does technically support threading and parallelism that's really not what it was designed to do. All of it's core I/O functionality is single-threaded and for a reason (including some of what was discussed above). So using Twisted in a natural way means there is still no true parallelism in the Twisted Framework and depending on the application considerations that could be a major drawback. Synchronous programs are also generally more complex than their multi-threaded counterparts, so maintaining this kind of code base can prove more difficult, but again one must consider the needs of the individual application. In our case, I do not believe this to be a significant hurdle.

3.3 Dynamic Typing in Python

Running on Python, Twisted necessarily implements a dynamic typing approach. In considering that we intend our code to run in a fast-paced, highly dynamic environment it's worth considering whether or not giving up

the security of a statically-typed language is worth it. In our case, the input server's received and processed was highly controlled and specific. If it wasn't then the code simply responded with a question mark and the flawed command. Storing our data in Python dictionaries and arrays and making use of Python's methods on this data types means that their was little flexibility in terms of the types our data could even inhabit in the first place—most bits of information were stored and propagated as strings. In this sort of implementation the security risk of dynamic typing becomes less relevant.

Furthermore, with Twisted's extensive library of unit tests and Python's facile logging mechanism, detecting errors at runtime is fast and easily addressable. With this in mind Python's type inference allows for the kind of fast paced development cycle critical for an application whose product is inherently timely.

4. Node.js

Node.js is an alternative event-driven framework to Twisted. It's gaining popularity for its concise source code, approachable coding style and performance. While not a JavaScript framework technically, much of its underlying modules are written in JavaScript and as such developers are able to write new modules in JavaScript as well. It thus shares many of its syntactic styling. Whether or not this is a benefit or a drawback largely depends on your position on Javascript. Some see JavaScript's freewheeling style as a recipe for convoluted code and a subsequent nightmare in terms of maintenance. Others extol the virtue of sharing a codebase between the front and back ends, allowing more developers to make more significant contributions.

Like Twisted, Node.js implements an asynchronous programming pattern. And like Twisted it is ideal for lightweight, fast functions that can be executed in isolation and deferred if necessary. However, unlike Twisted, Node.js cannot easily be extended to support more computationally heavy requirements. While there are thousands of libraries and counting built on Node.js it is in some ways a more isolated platform than Twisted. Node.js is truly single-threaded, whereas there is the option to extend Twisted to support parallel efforts. So to some extent the Node.js is a less flexible option. It may also be the case that it would be harder in the end to work Node.js into existing architectures for this reason.

Still, there are advantages to the Node.js path. For one, as I mentioned, it's fast. It runs on Google's V8 engine, which compiles the native JavaScript source code to machine code rather than interpret it real time and the performance awards are noted to be substantial. Whether it is faster than Twisted, I couldn't definitively say, but performance is something it is known for. Like Twisted, it is also widely supported. Its developer community has seen remarkable growth over the last few years and there is an ever-evolving pool of useful libraries to extend and ease the use of Node.js as such.

5. Conclusion

Twisted takes much of the headache out of building simple networking solutions. It supports many common transport and application layer protocols and is known for being easily extensible. Its event-driven architecture makes it ideal for the "server herd" architecture we have been exploring and the rapid cycle of a news distribution network. This asynchronous design, as well its capacity for substantial error handling and simple logging make it a secure bet for an industry product.

References:

http://web.cs.ucla.edu/classes/fall16/cs131/hw/pr.html

https://twistedmatrix.com/pipermail/twisted-web/2008-April/003754.html

https://twistedmatrix.com/trac/wiki/TwistedAdvantage

http://twistedmatrix.com/~exarkun/pycon-presentation.html#slide142

http://twistedmatrix.com/trac/wiki/Documentation

https://journal.paul.querna.org/articles/2011/12/18/the-switch-python-to-node-js/

http://substack.net/node_aesthetic

http://notes.ericjiang.com/posts/751

http://www.linuxjournal.com/article/7871

http://www.aosabook.org/en/twisted.html

http://wiki.c2.com/?BenefitsOfDynamicTyping

http://jessenoller.com/blog/2009/02/11/twisted-hello-asynchronous-programming

https://en.wikipedia.org/wiki/Node.js

http://www.dotnettricks.com/learn/nodejs/advantages-and-limitations-of-nodejs