UCLA Computer Science 131 ProxyHerd Project Report

Qingwei Lan
University of California, Los Angeles

Abstract

Twisted's event-driven nature allows an update to be processed and forwarded rapidly to other servers in a herd, making it a great candidate for building a application server herd, where the multiple application servers communicate directly to each other as well as via the core database and caches. In the following report, I will explain my implementation of `ProxyHerd` using Twisted and Python and compare Twisted with Node.js.

1 Introduction

We are building a new Wikimedia-style service designed for news, where (1) updates to articles will happen far more often, (2) access will be required via various protocols, not just HTTP, and (3) clients will tend to be more mobile. From a software point of view our application will turn into too much of a pain to add newer servers. From a systems point of view the response time looks like it will too slow because the Wikimedia application server is a central bottleneck.

2 Setup and Testing

2.1 Environment

Python Version : `Python 2.7.9`
Twisted Version : `Twisted-15.0.0`

2.2 Running `ProxyHerd` and Tests

To start a server named `<server-name>`, simply execute the following shell command

```
python chat.py <server-name>
```

A few complimentary scripts have been supplied to lessen the burden of starting and closing the servers, among which are

`run.sh` : starting all the servers
`kill.sh` : stopping all the servers
`clean.sh` : cleaning up all logs from servers
`test.sh` : script for running all tests

Note that running `test.sh` will automatically start and stop all servers, so it would be unnecessary to run the scripts `run.sh` and `kill.sh`.

3 Twisted and Python

Twisted is an event-driven framework that eliminates the burden of using multiple threads for multiple accesses to the main server, keeps the performance of multi-threaded model, but also maintains the simplicity of a single-threaded networking server.

3.1 Twisted

At the core of Twisted is the reactor, which can be viewed as the main thread of the server. It handles all events and requests that comes into the server, and handles all responses that are sent from the server.

Where does the beauty of Twisted lie? It lies in the even-driven nature. We can compare it to other two approaches: multi-threaded, and single-threaded.

3.1.1 Single-Threaded Server

For a single-threaded server, we process the in-coming requests procedurally. This approach maintains simplicity and is easy to debug. No matter how many requests we get from clients, we keep only one thread, and we eliminate all issues we have to deal with in multi-threading (e.g. race conditions, deadlock). However, the problem is also inherent in this nature. The server can only serve a single client. If a client blocks up the server with a long request or infinite loop, other requests from

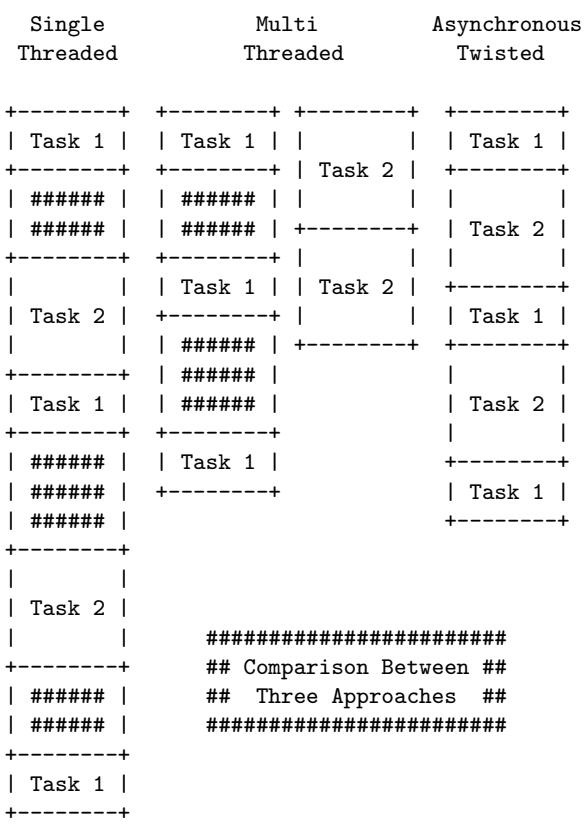
other clients would have to wait, which would be a waste of CPU time.

3.1.2 Multi-Threaded Server

For a multi-threaded server, we create a new thread for each incoming client, then we process all requests from that client on the new thread. When a client disconnects, we kill the thread. This approach solves the problem of waiting on a single request by using threads to process the requests, therefore efficiently using CPU time. However, problems arise as we add complexity to the system with multiple threads (e.g. race conditions, deadlocks).

3.1.3 Twisted

Twisted combines the advantages of the two approaches by using a single thread to eliminate complexity, while also making good use of CPU time by using an event-driven framework. Each event has a callback, so whenever an event arrives, the main thread processes the event. If the event hangs, such as waiting for I/O, then the reactor mentioned above will put the event aside and poll for other events. When the I/O arrives, then the reactor will issue a callback associated with the event, therefore awaking it and finishing the processing. The following diagram shows the comparison between the three, where '######' stands for waiting time (e.g. waiting for I/O)



3.2 Python

Since Twisted is written in Python, it has a concise syntax, which is demonstrated by the prototype that I wrote, which took roughly 200 lines of code.

Python is mainly an interpreted language which means that type-checking will be done at run-time instead of compile-time. Python is also dynamically-typed, unlike C/C++, which is strongly-typed. Although this may seem unsafe, it is also yields more flexibility. Run-time type errors can be manually handled by the user with the `try-catch` pattern.

Python has a garbage collector, which means that memory leaks will happen less often. The garbage collector also makes sure that memory is used efficiently, which is a great advantage for running servers where memory is a scarce resource.

There is inherently no multi-threading in Twisted, but Twisted handles multiple requests with a event-driven nature. It works with event loops, call-back functions, and multiple processes instead of multiple threads. This lifts a lot of pressure off the programmer but the downside to this approach is that if the a single event crashes, the whole server goes down with it, whereas in a multi-threaded approach, a single error will only crash a single thread.

4 Implementation of `ProxyHerd`

The implementation of `ProxyHerd` is mainly an extension of `LineReceiver` imported from `twisted.protocols.basic`. As specified by the protocol, we have to implement the `connectionMade`, `connectionLost`, `lineReceived` functions for the `ProxyHerdProtocol`.

4.1 Logging

For this implementation, we log all events for a server named `<server-name>` into a file called `<server-name>.log`. Events include establishing connections, closing connections, errors, all requests and responses, etc. Each log shall be preceded with the server's name, for example, a server named `Alford` should be preceded with `[Alford]`. We record all this data so that we can refer to it in the future.

4.2 Message Handlers

The server accepts three kinds of messages, `IAMAT`, `WHATSAT`, and `AT`. The following explain how a certain message is handled.

4.2.1 IAMAT

This message has 4 arguments, in the form

```
IAMAT <client-ID> <coordinates> <time>
```

where <client-ID> is the ID of the client, which is a string; <coordinates> is the reported location, in the form LatLng; <time> is the POSIX time reported.

The server first checks the <time> argument to make sure it is positive, as defined by POSIX time. Then it computes the time difference between the server's timestamp and the client's timestamp, and converts it into a string with a preceding '+' or '-' if the difference is positive or negative correspondingly. It then creates a string in the form

```
AT <server-name> <time-diff> <client-ID>
          <coordinates> <time>
```

where <server-name> is the name of the server; <time-diff> is the difference between the server's idea of when it got the message from the client and the client's time stamp; and the rest are simply copies of the message's arguments.

The server then checks if the <client-ID> exists in the database. If it does not exist, the server simply adds a new entry to the database and executes the flooding algorithm to propagate the new client to its neighbors. If the <client-ID> exists, then it checks the time stamp to see if it is a newer update and if it is, the server updates the database and propagates the new data.

4.2.2 WHATSAT

This message has 4 arguments, in the form

```
WHATSAT <client-ID> <radius> <limit>
```

where <client-ID> is the ID of the client, which is a string; <radius> is the radius (in kilometers) from the location of the client in which to query Google Places; <limit> is the maximum number of results.

The server first checks to see if <client-ID> exists in the database. If it does not exist, then the server reports this and returns. If it does exists, then the server fetches the stored message from the database and gets the <coordinates> parameter. It then inserts a ',' between the latitude and longitude. Then the server builds the query URL and calls the getPage method to perform the query, also setting a callback lambda function. When the response arrives, it prints it out using a callback function.

4.2.3 AT

This message has 6 arguments, in the form

```
AT <server-name> <time-diff> <client-ID>
          <coordinates> <time>
```

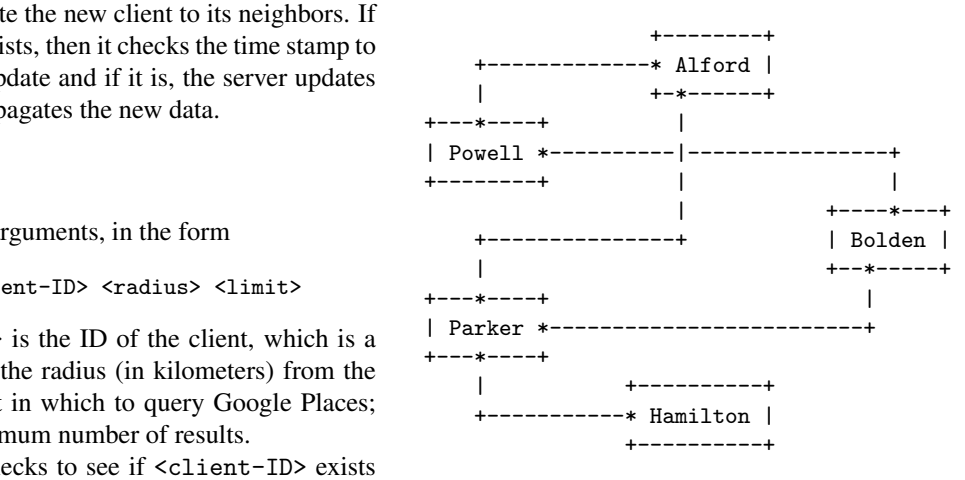
where <server-name> is the name of the server; <time-diff> is the difference between the server's idea of when it got the message from the client and the client's time stamp; <client-ID> is the ID of the client, which is a string; <coordinates> is the reported location, in the form LatLng; <time> is the POSIX time reported.

If the <client-ID> exists in the database and its timestamp is newer, the server will report this and return. Otherwise, if the <client-ID> does not exist, then we will add it to the database. If the <client-ID> exists, then we will update the database. In both cases, the server will propagate the new data to its neighbors.

4.3 Flooding Algorithm

As the name shows, the flooding algorithm will propagate its received data to its neighbors. In this case, if a server receives a new update, then it will send the same update to its neighbors. Similarly, its neighbors will send the update to their neighbors, therefore 'flooding' the database.

Our server herd has the following structure, where the names are the names of the servers and the lines connecting them are their communication lines.



Now suppose we get an update at server 'Alford', it will send the update to 'Powell' and 'Parker'. 'Powell' will send updates to 'Bolden' and 'Alford', and 'Parker' will send updates to 'Alford', 'Bolden', and 'Hamilton'. Therefore all servers will get the update, thus 'flooding' the database.

However, there is also an obvious drawback in this design. We get multiple updates, and also cyclic updates. When we get an update at server 'Alford', we send the update to 'Powell'. However, 'Powell' will 'resend' the update back to 'Alford', which will be a waste. Although it will not result in an infinite loop because of the

timestamps, it will waste a lot of network data and since network data is a scarce resource, we wouldn't want that.

My solution for this is that when sending an update to a neighbor, we also send the name of the sender in the message. In this case, when '`Alford`' sends an update it will notify its neighbor that '`Alford`' is the sender so that the neighbor can check and not send the message back to '`Alford`'.

5 Analysis of Implementation

Twisted has been documented well and has a wide-supported community, making it easy to write a working application from scratch by following the well-documented example code. Also, since it follows a simple pattern, it does not requrie the programmer to be too familiar with the Python language.

Python is an interpreted language with dynamic typing, making it easy to work with when involving servers. It also has a garbage collector to manage its own memory, therefore lessening the burden on the programmer's shoulders and allows them to focus on more important implementations.

Since Twisted uses callback functions, it may be appear strange to new users to this functionality but once the programmer gets the idea, it is a simple and elegant solution. People who have experience programming in iOS should identify that this is identical to Objective-C 'blocks', a form of callback functions commonly used in iOS programming to deal with multi-threading.

6 Comparison with Node.js

Node.js, released in 2009, a JavaScript runtime that follows the same event-driven paradigm, is a relatively new framework compared with Twisted, which was released in 2001. Despite its short history, Node.js has gained popularity and growth these years. Node.js is primarily written in JavaScript, a language that is popularly supported worldwide and integrates well with server-side development.

JavaScript is also an interpreted language with dynamic typing (although you would have to declare variables before using them). It supports multiple paradigms with first-class functions. Support for lambda functions makes it possible to support callback functions in an event-driven framework.

Typically these two frameworks work similarly with minor differences between language issues. Twisted has a longer history and can be considered as a stabler framework. Node.js is a boosting framework that has gained popularity rapidly recently. Node.js is also light-weight compared to Twisted. Any additional functionalities would have to imported from third-party libraries. Node.js has greater developer support and has been widely used in large companies and startups worldwide. Node.js also has better performance since it has been built atop Chrome's V8 JavaScript engine.

7 Availability

Twisted is an event-driven networking engine written in Python and licensed under the open source MIT license. Twisted runs on Python 2 and an ever growing subset also works with Python 3. Information about Twisted can be found at

`https://twistedmatrix.com/`

Information about Python can be found at

`https://www.python.org/`

Information about Node.js can be found at

`https://nodejs.org/`

8 Conclusion

Twisted is a reasonable candidate for the type of service we are building. Its event-driven nature and implementation in Python makes it simple to create an application (as we did in 200 lines of code). Callback lambda functions make it easy to implement the event-driven paradigm in Python, which lies at the core of this implementation.

Similarly the fast-rising Node.js is also a great alternate candidate for this type of application. Its wide popularity and support may be able to make it the most popular framework that is used.

References

[1] Jiang, Eric. "The emperors new clothes were built with Node.js", http://notes.ericjiang.com/posts/751, 30 Nov. 2015.

[2] James, Mike. "What Is Asynchronous Programming?", http://www.i-programmer.info/programming/theory/6040-what-is-asynchronous-programming.html, 31 May 2014.