# Twisted Places Proxy Herd

Alvin Lim, 804011675
*University of California, Los Angeles*

## Abstract

In this paper we explore Twisted, an event-driven networking framework for Python, and use it to prototype a potential proxy server herd application. We will first examine the pros and cons of the design, implementation, and maintenance process in using the Twisted framework, then discuss its feasibility when compared to alternative solutions such as Node.js.

## 1. Introduction

In the LAMP platform based on GNU/Linux, Apache, MySQL, and PHP, we have a general-purpose web server which makes use of multiple, redundant servers behind a load-balancing virtual router for reliability and performance. With the adoption of new technologies and the modern shift in Internet architecture, however, we have to now consider a different environment which may pose problems for the LAMP platform such as: more updates to servers, access via various protocols, and the growth of mobile clients. This gives motivation to explore the viability of an application server herd as a solution to these new conditions.

### 1.1. Twisted

We will investigate the feasibility of implementing the application server herd using event-driven programming and Twisted. Twisted is an event-driven networking engine licensed under the MIT license. It is written in Python, running on Python 2 with a subset of functions working with Python 3 [1]. The Twisted framework allows for easy prototyping of a custom application layer protocol. It is built around a reactor, which processes service requests and events from concurrent clients, dispatching them to event handlers. The framework focuses on using single thread, due to the problems of efficiency and complexity of context switching and synchronization when it comes to using multiple threads.

## 2. Design

For the design of our server herd, each node of the herd plays a role as both a server and a client. Nodes act as servers when receiving queries and reports from clients, but also act as clients when flooding the herd with user location reports, as it has to propagate the information to its neighbors. Our server herd does not encapsulate the roles of server and client into separate modules – instead, each node listens to a specific port where it receives both client-server and server-server communications. The way it differentiates the different types of communications is through the command label of the data it receives. The client-server command types are (a) IAMAT, a client command to update its position and (b) WHATSAT, a client query for information about places near other clients' locations. The server-server command type is the AT message, which is used to forward client information (from IAMAT commands) within the server herd.

### 2.2 Inter-server Communication

When client queries are received, the server herd nodes have to keep the client locations synchronized, in order to reply with current data for corresponding clients. To do this, we (a) store the client location data at every server and (b) flood an update to the entire herd when new client data is received. While this design takes up more space (storing client data at each node), it makes up in speed, since every node will have updated states for client information when queried.

This inter-server communcation protocol allows for simple recovery from connection losses: since each server keeps a list of neighboring nodes and uses the stored TCP transport info to propagate data, when connections with neighbors are broken, servers would try to re-establish a TCP connection every time an update is attempted to be propagated to the neighbor.

## 3. Implementation

The bulk of the server implementation is in the ProxyServerProtocol class, which is instantiated for every connection by the factory created by the ProxyServer class. A rector invokes the factory whenever a request comes in on a specified port. The protocol uses the lineReceived method implemented from the LineReceiver protocol of the Twisted framework. lineReceived checks the incoming line for the type of

command and then calls the corresponding event handler. To run the server, the following call is used:

python chatserver.py <server_name>

## 3.1 Handling IAMAT Messages

Server nodes need to update client location data when receiving IAMAT commands. The server herd IAMAT handler first checks that the client message is a valid command. Once it determines so, it checks to make sure that the timestamp on the IAMAT message signifies updated data for that client. If so, it parses the message into tokens which are used to create a forwarding AT message to flood the server herd with. The IAMAT handler also sends back a formatted response to the client, either notifying the client of an improper command or confirmation of the command.

## 3.2 Handling AT Messages

When server nodes receive a forwarded AT message, the handler checks to see if the timestamp of the AT message signifies that this is an updated entry for a corresponding client. If the message is a duplicate or outdated, the server node does not take any further action; otherwise, the node updates the client information in its cache and continues forwarding the message to its neighbors. Since nodes stop forwarding messages when they are duplicates or outdated, this implementation avoids the issue of messages circulating indefinitely.

## 3.3 Handling WHATSAT Messages

For WHATSAT queries from clients, server nodes use the request to perform a call to the Google Places API for its results. The handler first finds the client location information in its cache, then uses it along with the information from the WHATSAT request in order to generate an API call with the getPage method. This generates a deferred object, which can be used with a callback method to process the response. The Python json module is used to load the json response and filter out the message to create a streamlined response without extraneous information, which is then returned to the client.

## 3.4 Logging Actions

We use Python's logging module in order to keep track of all notable events and errors, such as the number of connections, the types of messages sent/received, etc. The filename for the log files are of the form:

servername_YYYY-MM-DD_HH_MM_SS.log

Where the time is reported as the UTC time.

# 4. Analysis

## 4.1 Note on Data Free Implementation

Since Twisted operates on an asynchronous nature, it may be important to note whether or not this implementation is data race free. Since network transmissions have to be taken into account, execution sequence take importance when multiple events happen concurrently.

First off, we note that callback functions could be interrupted by one another, causing data race scenarios, but the event loop nature of the Twisted framework allows the callback functions to be atomic, even though they are asynchronous. This is because the callback functions are always invoked form the event loop by reactor.run(), which utilizes blocking and operates on a single thread [2]. This means that even when another event occurs during the execution of the previous event's callback, the second event will be queued.

In contrast, we can consider a scenario where two events occur at the same time, one where a client queries a server for location results based on a client X, while the server also receives updated information for that client X. In this case, the server may or may not serve results based on the updated location information, depending on the order that the events are executed. This type of issue is hard to replicate, however, and is a common issue faced by network and distributed systems.

For the purposes of our implementation, the second scenario is unlikely to happen and unavoidable, so we can generally conclude that our simple application server herd is data race free.

## 4.2 Prototyping

Earlier, it was discussed how Twisted allows for easy prototyping of application layer protocols. This is in comparison to the requirements in implementing network programs in other languages, such as C. In C, one must build the networking application directly on the socket API, using the system calls. Twisted provides encapsulation for its socket functions, and callbacks allow for easier implementation when programming.

## 4.3 Node.js

A good alternative to Twisted is the recently popular Node.js. Node.js is a JavaScript runtime – however, whereas JavaScript is usually executed within browsers, Node.js is a barebones, customizable server engine

runtime installed into the OS and executed from the console.

Similar to Twisted, Node.js implements an event-driven, non-blocking I/O model. Though it is newer and less mature than Twisted, Node.js has gained a lot of traction lately, with many large corporations using it. Since core functionality of Node.js is very small, many advocates say that this makes it easier to learn, while additional functionality can be added through libraries [3].

Node.js does have good performance because of the V8 JavaScript Engine it operates on. With the way modern technology is moving towards scalability and performance, the performance benefits of Node.js is a large advantage [4]. In addition, JavaScript and JSON have become largely popular due to the internet, allowing Node.js to fit into the internet architecture more naturally.

Where Node.js doesn't shine is in applications with intensive server-side computations, because Node.js is single threaded and such an implementation offsets the throughput advantages of its non-blocking I/O; in such applications, threaded platforms are better approaches.

If we were to implement our server herd using Node.js instead of Twisted, we would need to make significant design changes to our application. Since JavaScript is prototypical and doesn't have object oriented constructs such as classes, we would have to define our own prototypes for the client and servers and creating objects from those prototypes, rather than deriving them from classes like the ServerFactory and ServerProtocol of the Twisted framework. Additionally, in JavaScript, functions are objects, so they could be used more flexibly with other objects. For example, in our Twisted implementation, we used a lambda function for getPage: if we were to use Node.js instead, the function could have been created as a function object, which would have been able to interact with variables of the function it was nested in.

## 5. Conclusion

Twisted proved to be a simple and efficient framework for implementing our application server herd. Where it particularly shined was the fact that it allowed for easy prototype implementation. While the implementation of the server herd would be quite different under using Node.js, we concluded that Node.js is a strong alternative framework, and can understand the recent growth and adoption of Node.js by many large companies. Additionally, we saw that the implementation of the application server herd architecture is a viable option for the traditional web server architecture, whether it is implemented in Twisted or Node.js.

## References


[1] "Twisted." Twisted. Web. 03 Mar. 2016. <https://twistedmatrix.com/trac/>.

[2] "Twisted Documentation: Using Threads in Twisted." Twisted Matrix. Web. 04 Mar. 2016. <http://twistedmatrix.com/documents /13.1.0/core/howto/threading.html>

[3] "6 things you should know about Node.js." JavaWorld. Web. 05 Mar. 2016. <http://www.javaworld.com/article/2079190/scripting-jvm-languages/6-things-you-should-know-about-node-js.html>

[4] "The Switch: Python to Node.js." Web. 05. Mar. 2016. Paul's Journal. <https://journal.paul.querna.org/articles/2011/12/18/the-switch-python-to-node-js/>