# CS 131 Project: Twisted places proxy herd

Zhehao Wang, 404380075
*zhehao@cs.ucla.edu*

## Abstract

In this project we explored Twisted, an event-driven networking framework for Python, and used it to prototype a proxy server herd application. For a given network topology, a proxy server herd of 5 servers is created. The server herd handles location report and position queries from users. Each server disseminates user location reports to other servers in the herd by flooding. Our servers implemented simple error and failure handling, as well as extensive logging for the exchanged messages. The report first describes specific issues identified in the design and implementation process, then discusses other topics like the feasibility of using Twisted, and Twisted vs Node.JS.

## 1 Introduction

Twisted is an event-driven networking engine written in Python and licensed under the MIT license. Twisted runs on Python 2 with a subset of functions working with Python 3. [1]

Twisted allows easy prototyping of user's custom application layer protocol. Twisted encapsulates *ClientFactory*, *ServerFactory* and *LineReceiver*; and by deriving from these classes and implementing the corresponding callback functions, such as *connectionMade* and *lineReceived*, the user can build a custom protocol, and set up a network topology to see the protocol in action.

This report is organized as follows: section 2 covers networking related issues in the proxy herd design, section 3 describes programming language related issues in the implementation, and section 4 discusses the feasibility of using Twisted to prototype and protocol, how a Node.js implementation could be different.

## 2 Design

The protocol behaviors are mostly defined in the assignment description[2]. In this section we focus on the parts that are not specified, for example, the inter-server communication protocol.

### 2.1 Role of the nodes in the herd

In this application, each node in the herd plays the role of a server, as well as a client. For queries or reports received from the client, the node acts like a server; when propagating a user's location report to other nodes in the herd, this node could act like a client. Our work does not differentiate these two potentially different types of communication, which means the server node listens on only one port, accepts user-server connection as well as server-server connection on that port, and only differentiate these two types of communications by the command used.

### 2.2 Inter-server pattern of communication

When a client query is received, in order to be able to reply with the most recent data of queried client ID (which may or may not be connected with the server that receives the query), we need to keep the servers' notions of a client's location synchronized. Two approaches can be taken to achieve synchronization.

- Only store client location reports at the server that the client's connected to, and flood the user's query request on-demand. This approach will introduce a larger delay because relaying query requests takes time. However, the amount of data that each node should store becomes smaller, since each server only need to maintain the information of the clients that are directly connected to it. In a network where updates happen much more often than queries, this approach is likely to be more efficient.

- Store client location data at every server, and whenever a client publishes an update, flood the update so that the entire herd is aware of it. Contrary to the first approach, this one is likely trading space for time. And in a network where queries happen much more often than updates, this approach is likely to be more efficient.

The argument for both designs is very similar with the traditional networking problem of "proactive routing" vs "on-demand routing". Our design takes the second approach, because it's more likely to be a fit for the project's use case.

It's also worth mentioning that these two approaches have different impacts on the inter-server protocol overhead: in order to recover from a server failure (for example, a server crash), the second approach requires the crashed server to proactively synchronize with other nodes in the herd, while the first approach does not. Although our design does handle reconnections between servers should a server crash happen, the proactive synchronization after crash's not currently in the project's implementation.

The update flooding in the project follows a similar format as the server's reply to a client's IAMAT message, but with the propagating server's ID appended. An example looks like the following:

```
AT Alford +0.563873386 kiwi.cs.ucla.edu
+34.068930-118.445127 1400794699.108893381
                Bolden
```

We include the sender's server ID, so that the receiver of this message knows not to flood this message back to the sender. However, this alone cannot get rid of potential update loops in a topology where cycles with more than two nodes exist, thus we introduce the notion of a duplicate or out-dated update: the server only propagates the "AT", or "IAMAT" messages when the client timestamp in the message's larger than the local recorded timestamp for that client ID.

Realistically, it would be more effective to use a multicast tree based multicast protocol to disseminate a client's update message.

The inter-server communication protocol also features simple recovery from connection losses. Before loss of connectivity, each server keeps a list of proxy herd nodes that it's currently connected to, and uses the stored TCP transport info to propagate location updates. If a TCP connection with a neighbor's broken, the node would attemp to establish a new TCP connection each time an update need to be propagated to that neighbor.

## 2.3 Protocol definition addons

- WHATSAT with non-existent ID record. The assignment description did not describe the expected behavior when receiving a valid WHATSAT command, but with an ID that the server does not know. In this work, we treat the "WHATSAT" command as invalid: replying with the same message lead by "?".

- User input processing, and system security. User input in the system could be malicious in two ways: malformed data, or correctly formed but bogus data. In this work, we only handle simple malformed user inputs, for example, giving a string that cannot be parsed as an integer when an integer is expected. And we don't handle the potentially bogus data, since the security of the system is beyond the scope of this project. Also, we don't handle cases with duplicate server or client IDs.

  Realistically, a security model should be introduced into the system, and user inputs (client IDs, timestamps in the report) should not be trusted.

# 3 Implementation

With the design details fleshed out, this section covers a few issues we identified while implementing the project.

## 3.1 Cycling reference in action

In class we discussed the topic of object reference cycles, and how it might cause problems in a pure reference counting based garbage collector. In this project, we utilized a reference cycle in the following two ways in inter-server communication implementation.

The obvious one is that a client object instantiated from ClientFactory class keeps a reference to its protocol, and the protocol keeps a reference to its factory object. This implementation decision is made because of the flexibility it introduced: the client instance can proactively call sendLine of its protocol when needed, and the client protocol can modify the properties of its client instance based on what it receives.

The less obvious one is that the server instance keeps a dictionary of "other server ID" to "Twisted client object" mapping to represent its TCP connection with other nodes in the herd, and each "Twisted client object" keeps a reference to the server instance. Consider a server that wants to keep track of the state of its connection with other servers in the herd: connection made or lost is detected inside the client object, which will need to change the dictionary that's kept per server instance. Meanwhile, a server instance can call *sendLine* of the client objects

in order to propagate AT updates, or attempt to construct a new client object if a connection with another server's not already made. With this mapping, we can maintain the TCP connection status between servers in the proxy herd and only reconnects when a failure's detected, instead of recreating a TCP connection each time a server wants to propagate an AT message, and destroying that connection after the propagation finishes.

Fortunately, Python has a reference counting based garbage collector with reference cycle detection, and we don't need to worry about explicitly nullifying one node in the cycle when freeing in order to avoid memory leak.

### 3.2 Data race free implementation

Whether the implementation is data race free is an important question, and the answer's not obvious given Twisted's asynchronous nature. In this section we look inside Twisted source code, and decide that the implementation itself is data race free, but with network transmission taken into consideration, it's likely that the execution sequence matters when multiple events happen at the same time.

- Are the callback functions atomic.

  The first question is whether during the execution of one callback function, the execution of another could take over, interrupting the first one and creating data race scenarios. Our investigation shows that the answer's no, and callback functions are atomic because of Twisted's event loop mechanism.

  Inside Twisted, callbacks are asynchronous, but they are always invoked from the event loop started by the *reactor.run()* call, which is blocking, and single-threaded in nature[3]. If another event happens while the loop's executing a previous event's callback function, the later event will be queued for execution. This is a common pattern in Python libraries dealing with async IO, for example, trollius in Python 2, and the built-in asio in Python 3 both use similar event loops. Two general ways are usually provided to execute commands in the single-threaded event loop, one is to provide a callback, which gets invoked when a matching event happens, the other is to proactively "schedule" the execution of a function, for example, proactively calling Twisted protocol's *sendLine*, or *call_later* in trollius library.

- What happens if two events happen at exactly the same time.

  Consider this scenario, a client A queries server B for the location of client C, meanwhile, server B receives an update of the location of client C from server D. Would A get the updated version or not? Our investigation shows that both cases are possible, depending on the order of the execution queue. It's hard to replicate a scenario with two events happening at the exact same time (or one might argue it's not possible without Python Global Interpreter Lock tricks or multi-processing, since even multiple-threading in Python only uses one logical CPU) [4] Also, it turns out that this is mostly an unavoidable issue in distributed systems, or systems connected by network in general.

Thus, we conclude by stating that our simple system is data race free, though the sequence of execution still matters, when two network events happen at approximately the same time.

## 4 Discussion

In this section we cover the topics that are not directly related with the specific project's design or implementation.

### 4.1 Feasibility of prototyping with Twisted

We conclude that it's highly feasible and easy to prototype application layer protocols and exploit proxy server herds with Twisted. The conclusion's drawn from comparisons with our experiences writing networking programs.

In the past we've built applications on top of C's socket API, which directly wraps the system *read*, *write*, *listen*, *accept*, and *connect* calls. Twisted provides good encapsulation over the socket functions, and exposing callbacks to make the work of the programmer easier. For example, in Twisted we have *connectionLost* callback, while in C the programmer should first care if the IO calls are blocking or not, and after that, always manually handle cases like *read* returning 0 in length, and data reassembling if one call on *read* does not get all the data that's in the OS's buffer.

It's also worth noting that people have realized a robust network IO library is a commonly requested feature, so many other high level languages provide similar features through built-in libraries or third parties tools. For example, another networking library "xmpppy" in Python handles a specific protocol for chat[5], and provides similar callback interfaces as Twisted.

### 4.2 Python Twisted vs Node.JS

Node.JS is a JavaScript runtime that's gaining popularity over the past few years. As opposed to ordinary JavaScript that gets executed inside a browser, Node.JS

installs a runtime into the OS, and can be executed from console. We believe that for our application's purposes, using Node.JS would be as easy as Python Twisted after introducing a few major changes to the implementation.

First, Node.JS also uses an event-driven, non-blocking IO model, which is not much different from Twisted. Then, many Node.JS packages provide a similar API as Twisted, for example, the "net" package has *listen*, *connect*, *on* callbacks, etc. Finally, similar with Python "easy_install" or "pip", Node.JS comes with a package management tool "npm", which makes extending the functions of the application easy should the need arise. Deployment with Node.JS is convenient, too.

Speaking from our past experiences of deploying XMPP services in both languages, Node.JS is a good alternative for Python if the goal is easy prototyping of protocols. Scalability and execution efficiency of both remain to be tested, though there are claims that the fast evolving Node.JS is more scalable for larger deployments[6].

Should we choose to use Node.JS instead, a few major changes in the implementation need to be introduced, with the biggest one being that JS is prototypical, and traditional object oriented notions like class is not in JS (even though the keyword *class* is introduced in ECMAScript 6[7], it's just syntactic sugar for an underlying prototypical implementation). So in Node.JS, instead of deriving from classes like ServerFactory and ServerProtocol, we would be defining prototypes of a client and a server, and creating objects from such prototypes. Another feature of JS is that functions are function objects, thus unifying them with ordinary objects, and making them more flexible. One immediate usage in our current implementation would be the lambda function for Twisted *getPage*. In JavaScript, we would be directly declaring a function object there, which has access to the outer function's variables.

### 4.3 Lessons learnt

This section covers the non-technical topics of hurdles and lessons learnt from this project.

The project's not difficult, given that most of the design objectives are already specified.

The author is rather familiar with Python and JavaScript, and used both for network programming before. (Mostly in a Named Data Networking (NDN) context, using PyNDN, the NDN client library in Python, and ndn-js, the client in JavaScript [8], for NDN application development. It's worth noting that these libraries have similar abstractions like providing callbacks to Protocol and Factory objects. In these libraries it's called a face, an abstraction for a network interface in general). The author's also familiar with Google API, and used Youtube API extensively for a project in the School of Theater, Film, and Television one year ago (One thing that interested me about the Places API is that "maxResults", "pageToken" and "nextPageToken" fields are not present, so querying for a large number of results could require workarounds).

With these past experiences, the authored still learnt a lot from this assignment, by inspecting the implementation inside Twisted, and identifying the differences from the past projects. Should the time allow, the author would like to investigate a more practical proxy server herd, and prototype in a more practical context or application environment.

## 5 Conclusion

We used Twisted to prototype a proxy server herd application. The report first covers issues in the design and implementation, like inter-server communication protocol, data race free reasoning, etc, then discusses the feasibility of prototyping with Twisted, and how an implementation in Node.JS may be different, and finally concluded that it's easy to use Twisted, while Node.JS would be a feasible alternative, too.

## 6 References

## References


[1] Twisted website, `https://twistedmatrix.com/trac/`.

[2] Fall 2015 CS 131 project description, `http://web.cs.ucla.edu/classes/fall15/cs131/hw/pr.html`.

[3] Threads in Twisted, `http://Twistedmatrix.com/documents/13.1.0/core/howto/threading.html`.

[4] Python GIL, `https://wiki.python.org/moin/GlobalInterpreterLock`.

[5] RFC6120, XMPP, `https://tools.ietf.org/html/rfc6120`.

[6] Twisted to Node.JS, `https://journal.paul.querna.org/articles/2011/12/18/the-switch-python-to-node-js/`.

[7] ECMAScript 6 class, `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes`.

[8] NDN common client libraries, `http://named-data.net/codebase/platform/ndn-ccl/`.