Twisted Places Proxy Herd

Haonan Zhou – University of California, Los Angeles

Abstract: In this project, I built a simple proxy herd server with the Twisted network model on Python. After some research and experiment, I found the event-driven Twisted to be an ideal platform for such an application.

Introduction

Websites such as Wikipedia runs on a LAMP platform, which uses multiple servers behind a central router to balance network traffic. This approach would significantly slow down when data are updated at a high enough rate in the network. The goal of this project is to build a network architecture called an "application server herd," in which servers directly communicate with each other, instead going through a central router, to deal with the rapidly changing data.

Twisted

Twisted is an event-driven networking engine. The most important part of a Twisted program is the reactor event loop. The reactor loops forever waiting for external events. Upon the arrival of an event, the reactor decides which method or callback should be triggered. These callbacks are registered with the listenTCP() and connectTCP() methods, before the reactor starts running.

Another important class in a Twisted network is a protocol factory, which maintains each connection as an instance of the protocol class. When listenTCP() or connectTCP() methods are called, we have to specify to the reactor which protocol factory to use. The protocol factory includes a buildProtocol() method, which simply instantiates a new instance of the protocol class. The reactor can then register methods in this protocol as callbacks.

Implementation

proxyinfo.py

This file keeps some basic information and topology of the network, including port number of each server, communication between servers, together with a URL and a key for Google Places API. It also defines a find_mst() method, which returns a minimum spanning tree given a server as the source node. More details of this method is given below.

find_mst()

This method returns a minimum spanning tree given the source node of the tree. More specifically, it gives a dictionary that indicates where each server should send the update message to. The method is implemented with a depth-first search algorithm. Since there are only 5 servers in our simply network, construction of a MST is probably an overkill for this project itself. An alternative would be just to inspect the topology and determine the tree in person. However, such an algorithm would be very helpful in a network of large scale. One thing worth noticing is that our network contains a cycle. Therefore, if our server simply "floods" the message to all its neighbors, the update would flow through the cycle and the network would never stabilize.

Also, in a real-world application, this method is more efficient if implemented with breath-first search algorithm. But none of these issues should be of concern, given the limited size of this problem.

proxyserver.py

ProxyServerFactory

__init__(self, name)

This method is called to instantiate a protocol factory. Information of the server is initialized in

this method, including server name and port number. In addition, logging of the server is also configured when the factory is created.

buildProtocol(self, addr)

This subtle method simply returns an instance of the server protocol class.

ProxyServerProtocol

__init__(self, factory)

Since the protocol factory maintains the server information, each server connection should be aware of which factory this protocol comes from. Therefore, this subtle method simply initialize a factory within each connection.

lineReceived(self, line)

This method is triggered whenever the connection received a line of data. In our protocol, the first word of the received line identifies the functionality of the command. Therefore, this method first split the line into word, and inspect the first word to decide which action it should take.

Different command would trigger different method in the protocol, which would be explained later.

IAMATreceived(self, line)

The method first calls split() to separate the received line into words, and match the needed client information with corresponding fields. Then, it calculates the time difference between server and client, and formats the response message to client which starts with "AT." This message is sent to the client with the sendLine() method. The server also writes this response into a local dictionary for later reference.

The server is also responsible for sending the client location to other servers in the network. It invokes the find_mst() method to compute whom it should communicate with, and send them the response with connectTCP() method.

ATreceived(self, line)

If the server receives an AT message, it should simply log the message in local dictionary, and send the message to other servers, using the approach described above.

WHATSATreceived(self, line)

In this method, the server should first check whether it has information of the specified client. If the server never received any update message regarding the client, it should respond the WHATSAT with some sort of error message. Otherwise, the server opens an URL that contains parameters required for the Google Places API. The response by Google Places is a JSON-formatted message that consists of place information in a given area. Of course, if the client specifies a limit on the amount of information, the server only keeps the first several places and discards the rest. Lastly, the server sends the json message to the client.

ProxyClientProtocol

__init__(self, factory)

This method is similar to the initializer of ProxyServerProtocol. It simply copies the factory information into the protocol.

connectionMade(self)

This method is invoked once the client successfully established a connection with the server. In our client protocol, once the connection is made, the client immediately sends the query stored in its factory to the server.

ProxyClientFactory

__init__(self, query)

We need to override this method because we want to initialize it with the query message.

buildProtocol(self, addr)

This is a subtle method similar to the one in ProxyServerFactory. It returns an instance of ProxyClientProtocol.

Testing

In testing scenario, the five servers are run on five terminals. Then, in another terminal, the following commands are sent to the specified server in this order:

1. Alford: IAMAT ucla +34.0722-118.4441 1432696700.426443843
2. Hamilton: IAMAT nyu +40.7300-73.9950 1432702730.426443843
3. Parker: WHATSAT nyu 5 5
4. Bolden: WHATSAT ucla 10 3

After these queries are sent, logs of the servers contains these contents (some details are replaced with "…"):

```
# Alford.log:
INFO:root:Alford> Server started
INFO:root:Alford> Line Received: IAMAT ucla +34.0722-118.4441 1432696700.426443843
INFO:root:Alford> Sent response to client: ucla
INFO:root:Alford> Location update sent to Parker
INFO:root:Alford> Line Received: AT Hamilton 2809.16293025 nyu +40.7300-73.9950 1432705539.59
INFO:root:Alford> Location update sent to Powell

# Bolden.log:
INFO:root:Bolden> Server started
INFO:root:Bolden> Line Received: AT Alford 8809.99099827 ucla +34.0722-118.4441 1432705510.42
INFO:root:Bolden> Location update sent to Powell
INFO:root:Bolden> Line Received: AT Hamilton 2809.16293025 nyu +40.7300-73.9950 1432705539.59
INFO:root:Bolden> Line Received: WHATSAT ucla 10 3
INFO:root:Bolden> Response sent to client:
{
   "status": "OK",
   "next_page_token": "CvQ…",
   "html_attributions": [],
   "results": [ … ]
}

# Hamilton.log:
INFO:root:Hamilton> Server started
INFO:root:Hamilton> Line Received: AT Alford 8809.99099827 ucla +34.0722-118.4441 1432705510.42
INFO:root:Hamilton> Line Received: IAMAT nyu +40.7300-73.9950 1432702730.426443843
INFO:root:Hamilton> Sent response to client: nyu
INFO:root:Hamilton> Location update sent to Parker

# Parker.log:
INFO:root:Parker> Server started
INFO:root:Parker> Line Received: AT Alford 8809.99099827 ucla +34.0722-118.4441 1432705510.42
INFO:root:Parker> Location update sent to Bolden
INFO:root:Parker> Location update sent to Hamilton
INFO:root:Parker> Line Received: AT Hamilton 2809.16293025 nyu +40.7300-73.9950 1432705539.59
INFO:root:Parker> Location update sent to Alford
INFO:root:Parker> Line Received: WHATSAT nyu 5 5
INFO:root:Parker> Response sent to client:
{
   "status": "OK",
   "next_page_token": "CvQB7A…",
   "html_attributions": [],
   "results": [ … ]
}

# Powell.log:
INFO:root:Powell> Server started
INFO:root:Powell> Line Received: AT Alford 8809.99099827 ucla +34.0722-118.4441 1432705510.42
INFO:root:Powell> Line Received: AT Hamilton 2809.16293025 nyu +40.7300-73.9950 1432705539.59
INFO:root:Powell> Location update sent to Bolden
```

Node.js:

Node.js is a network platform built on Chrome's JavaScript. Similar to Twisted, it uses an event-driven model for reliability and performance. In that sense, it is an ideal platform for applications where information is updated rapidly. It would be an equally efficient solution as Twisted in our proxy herd problem.

A major advantage of Node.js is that it is built on JavaScript, which makes it much more portable in web browsers than Python. Therefore, if we want to develop an application that runs on web pages, Node.js is probably the preferred choice. However, we may want to utilize some of the built-in features of Python in some applications. For example, we may like our application to easily crawl data from the web. In such a case, we would probably choose Twisted as our platform.

Conclusion.

After some researches and experiments, I found Twisted to be an ideal solution for the proxy herd problem. Its asynchronous, I/O non-blocking feature improves performance while keeping the robustness of a synchronous model. We do not have to worry too much about memory management since Python's garbage collector takes care of that. Also, in an event-driven network, a single-threaded program has about the same performance than a multithreaded program in a synchronous model. If we use Java, for

example, we most likely have to use multithreading in order to accomplish such a speed, since I/O blocking can have a large cost.