Analysis of the using Twisted to Created Python Server Herds

Omar Ozgur, *UCLA*

Abstract

Twisted is an event-driven networking engine that provides support for custom network applications to be written in Python. This paper examines the process of creating an application server herd with the Twisted framework, and analyzes the performance in respect to other popular server architectures.

1. Introduction

In the Wikimedia Architecture, service is provided through the use of a LAMP platform, as well as multiple redundant web servers behind a load-balancing virtual router. However, the application server can become a bottleneck for applications that have more mobile clients, more updates to database, and use various protocols. While this architecture works well for sites like Wikipedia, it is not suitable for many sites with different needs.

For services with data needs to be moved between clients and servers quickly, an application server herd architecture may be more suitable. These architectures allow for servers to directly communicate with each other to send and receive data, which helps to remove traditional bottlenecks, and allows for servers to maintain relevant data without frequently accessing a central database. For this project, the Twisted framework was used to demonstrate the use of such an architecture.

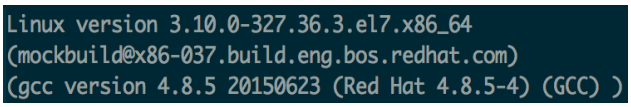
In order to determine the effectiveness of the Twisted framework, a sample parallelizable proxy was created for the Google Places API. The system includes 5 servers that each take location information from clients and respond with relevant information. These servers also propagate information to neighboring servers using a simple flooding algorithm. This allows for information to quickly spread throughout the server herd, which adds the ability for clients to request information from different servers without losing service, and helps to remove reliance on a centralized database.

2. Project Setup

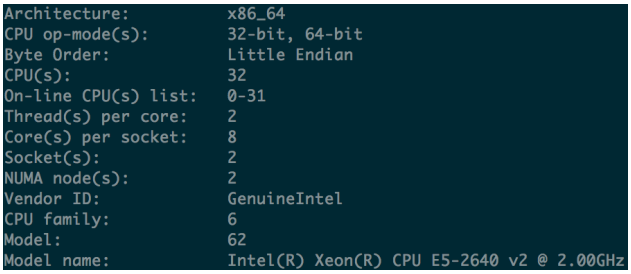
The main logic for each server can be found in the file "server.py". Server configuration details can be found in the file "conf.py", which contains information such as server names, server port numbers, neighbors for each server, and API information. In order to protect personal information, the Google Places API key was removed from the configuration file. Logs for each server can be found in the "logs" folder. A server log is created as soon as the server is started. To start a single server manually, the command "python server.py server_name" can be used. In order to start all 5 servers seen in the configuration file, the "start" script can be run with the command "./start.sh". Similarly, the "stop" script can be run with the command "./stop.sh" in order to stop all of the servers.

3. System Specifications

This project was created and tested on a Red Hat Linux machine (version 4.8.5-4). Further operating system details are provided in the image below. The information was found by running the command "cat /proc/version".



In terms of system hardware, some details were found by using the command "lscpu". A few relevant details are shown in the image below.



Although these system properties are not all necessary in order to run the project, some commands may behave differently if run with different configurations.

4. Implementation

4.1 Overview

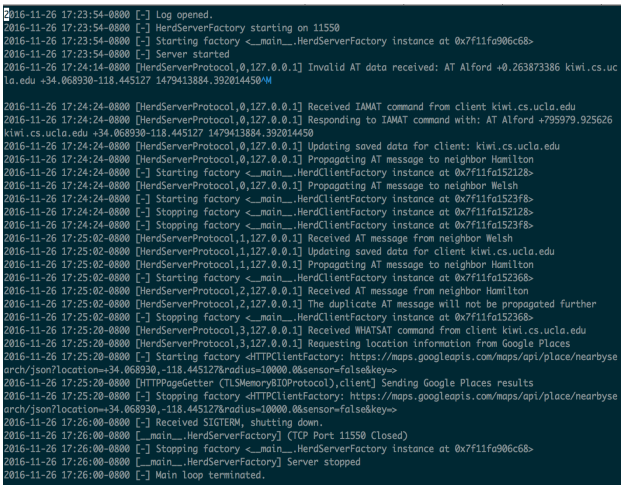
For this project, a single file was created for the server implementation. Since each server has the same requirements, they can all benefit from the same logic while using different details from the separate configuration file. When a server is started, relevant data from the configuration file and command-line arguments are parsed. Afterwards, a HerdServerFactory object is created, which inherits from twisted.internet.protocol.ServerFactory. This factory sets up the server's local database, as well as logging information. Whenever a client connects to the server, a HerdServerProtocol object is created, which

inherits from twisted.internet.protocol.Protocol. The "dataReceived" method within the protocol is used to deal with data that is sent from the client.

In order for data to be sent to be manually sent to a server, the "telnet" command can be used to create the connection. For this project, port numbers 11550 – 11554 were used to create 5 servers. Therefore, to connect to the first server on the same network, the command "telnet localhost 11550" could be used. While a connection is open, IAMAT and WHATSAT messages can be sent to the server. The command parameters and generated responses are explained further in sections 4.3 and 4.5.

4.2 Logging

When a server factory is created, a log file is opened in the local "logs" folder through the use of the twisted.python.log class. Afterwards, some relevant log information such as factory initialization and deletion, are automatically populated in the file. To manually add a log entry, the line log.msg('log_entry_message') is used. This is mainly used to log error messages when invalid data is received by the server, or to show details about data that is sent and received. A sample log file for server "Alford" is shown below.



```
2016-11-26 17:23:54-0800 [-] Log opened.
2016-11-26 17:23:54-0800 [-] HerdServerFactory starting on 11550
2016-11-26 17:23:54-0800 [-] Starting factory <__main__.HerdServerFactory instance at 0x7f11fa906c68>
2016-11-26 17:23:54-0800 [-] Server started
2016-11-26 17:24:14-0800 [HerdServerProtocol,0,127.0.0.1] Invalid AT data received: AT Alford +0.263873386 kiwi.cs.uc
la.edu +34.068930-118.445127 1479413884.392014450^M

2016-11-26 17:24:24-0800 [HerdServerProtocol,0,127.0.0.1] Received IAMAT command from client kiwi.cs.ucla.edu
2016-11-26 17:24:24-0800 [HerdServerProtocol,0,127.0.0.1] Responding to IAMAT command with: AT Alford +795979.925626
kiwi.cs.ucla.edu +34.068930-118.445127 1479413884.392014450
2016-11-26 17:24:24-0800 [HerdServerProtocol,0,127.0.0.1] Updating saved data for client: kiwi.cs.ucla.edu
2016-11-26 17:24:24-0800 [HerdServerProtocol,0,127.0.0.1] Propagating AT message to neighbor Hamilton
2016-11-26 17:24:24-0800 [-] Starting factory <__main__.HerdClientFactory instance at 0x7f11fa152128>
2016-11-26 17:24:24-0800 [HerdServerProtocol,0,127.0.0.1] Propagating AT message to neighbor Welsh
2016-11-26 17:24:24-0800 [-] Starting factory <__main__.HerdClientFactory instance at 0x7f11fa1523f8>
2016-11-26 17:24:24-0800 [-] Stopping factory <__main__.HerdClientFactory instance at 0x7f11fa152128>
2016-11-26 17:24:24-0800 [-] Stopping factory <__main__.HerdClientFactory instance at 0x7f11fa1523f8>
2016-11-26 17:25:02-0800 [HerdServerProtocol,1,127.0.0.1] Received AT message from neighbor Welsh
2016-11-26 17:25:02-0800 [HerdServerProtocol,1,127.0.0.1] Updating saved data for client kiwi.cs.ucla.edu
2016-11-26 17:25:02-0800 [HerdServerProtocol,1,127.0.0.1] Propagating AT message to neighbor Hamilton
2016-11-26 17:25:02-0800 [-] Starting factory <__main__.HerdClientFactory instance at 0x7f11fa152368>
2016-11-26 17:25:02-0800 [HerdServerProtocol,2,127.0.0.1] Received AT message from neighbor Hamilton
2016-11-26 17:25:02-0800 [HerdServerProtocol,2,127.0.0.1] The duplicate AT message will not be propagated further
2016-11-26 17:25:02-0800 [-] Stopping factory <__main__.HerdClientFactory instance at 0x7f11fa152368>
2016-11-26 17:25:20-0800 [HerdServerProtocol,3,127.0.0.1] Received WHATSAT command from client kiwi.cs.ucla.edu
2016-11-26 17:25:20-0800 [HerdServerProtocol,3,127.0.0.1] Requesting location information from Google Places
2016-11-26 17:25:20-0800 [-] Starting factory <HTTPClientFactory: https://maps.googleapis.com/maps/api/place/nearbyse
arch/json?location=+34.068930,-118.445127&radius=10000.0&sensor=false&key=>
2016-11-26 17:25:20-0800 [HTTPPageGetter (TLSMemoryBIOProtocol),client] Sending Google Places results
2016-11-26 17:25:20-0800 [-] Stopping factory <HTTPClientFactory: https://maps.googleapis.com/maps/api/place/nearbyse
arch/json?location=+34.068930,-118.445127&radius=10000.0&sensor=false&key=>
2016-11-26 17:26:00-0800 [-] Received SIGTERM, shutting down.
2016-11-26 17:26:00-0800 [__main__.HerdServerFactory] (TCP Port 11550 Closed)
2016-11-26 17:26:00-0800 [-] Stopping factory <__main__.HerdServerFactory instance at 0x7f11fa906c68>
2016-11-26 17:26:00-0800 [__main__.HerdServerFactory] Server stopped
2016-11-26 17:26:00-0800 [-] Main loop terminated.
```

In the log file shown above, the first 4 lines are provided to show the starting of the server. Afterwards, it is noted that an invalid command was received by the server. When a valid IAMAT command is received, the log provides details about the response that is sent to the client, the connections that are opened and closed between neighboring servers, and the data that is propagated to those neighbors. When a valid WHATSAT message is received, the Google Places page request details are printed to the log. The last 5 lines of the log file show the steps that are taken to stop the server factory and reactor. This sample log file shows that logging useful information can give important insight into the inner workings of a server.

4.3 IAMAT Commands and AT Messages

A valid IAMAT command includes the keyword "IAMAT", followed by a client ID, a location based on latitude and longitude, and a client timestamp. An example of a valid command, which was provided in the project specifications, is "IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1479413884.392014450". When a server receives a valid command, it stores the data locally and responds with an AT message, which includes the keyword "AT", followed by the server name, the difference in server and client timestamps, the location in terms of latitude and longitude, and the saved client timestamp. For example, if the previously stated IAMAT command was sent to the server "Alford", the response would be "AT Alford +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1479413884.392014450".

After responding to the client, the server also propagates the AT message to neighbor servers. In this project implementation, it was assumed that only the newest data in the local storage would be propagated to other servers. In order to do this, a TCP connection is created between the server and each neighbor. This causes a HerdClientFactory object to be created, which inherits from the twisted.internet.protocol.ClientFactory class. This object stores the message that will be propagated, and creates a HerdClientProtocol object, which inherits from twisted.internet.protocol.Protocol. Once the protocol is created, and the connection to a neighboring server is made, the server sends the AT message to the connected neighbor. Once the data is sent, the connection is closed.

4.4 Flooding Algorithm

In order to propagate messages to all of the servers, a simple flooding algorithm is used. Along with the AT message, the first server to propagate the information includes a server timestamp. If this timestamp is not equal to the timestamp in the data that a server has in storage, the server updates its local data and sends the data to its neighbors, excluding the server that sent it the data. If a server receives a timestamp that is equal to the timestamp in its local storage, it knows that it already propagated the data, and doesn't need to send it again. This method ensures that all running servers eventually receive the data without continuously sending data in an infinite cycle.

This propagation algorithm can be further explained by the log messages shown in section 4.2. When server "Alford" gets a valid IAMAT message, it propagates data to

neighbors “Hamilton” and “Welsh”. In another instance, the server gets an AT message from “Welsh” and propagates the information to “Hamilton”. When a duplicate message is received from “Hamilton”, the server correctly refrains from propagating the message again.

4.5 WHATSAT Commands

A valid WHATSAT command includes the keyword “WHATSAT”, followed by a client ID, a radius (in kilometers) that is at most 50km, and an upper bound on the number of responses to retrieve from the Google Places API, which is at most 20. When a valid command is received by a server, the server searches its local storage for location data related to the specified client. This information is used to send the client a corresponding AT message, as well as to send a Google Places request. A callback method is used to process the response when it is received. This callback decodes the received JSON data, removes responses based on the specified upper bound value, re-encodes the data, and formats the data to replace consecutive newlines with single newlines, and to ensure that the data ends with two newline characters. The formatted data is sent back to the client.

5. Testing

In order to test the program, all 5 servers shown in the conf.py file were started at the same time. Afterwards, telnet was used to connect and send valid IAMAT commands with unique client IDs to each server separately. Examination of the logs showed that the data in each test was successfully sent to all other servers without getting stuck in infinite loops. Afterwards, telnet was used to send valid WHATSAT commands to each server separately based on the information that was previously sent through IAMAT commands. Each command caused valid data to be received by the client in JSON format based on the results of the requests to the Google Places API. Many invalid messages were also sent to servers to ensure that errors were handled correctly.

6. Analysis

6.1 Python Language

Creating this project in Python was fairly straight-forward. The use of powerful libraries allowed for complex ideas to be implemented in small amounts of lines. For example, the twisted.python.log class allowed for log files to be created in 1 line, and allowed for detailed log messages to be added quickly at any line of code. It was also convenient to be able to import variables from the configuration file with a single import statement without having to manually parse data.

The simplicity of the syntax allowed for code to be written quickly without constantly referring to documentation to learn about the specifics of the language. Dealing with data structures such as dictionaries and lists was also very easy based on the built-in libraries. Many features of the syntax are similar to those in other popular languages such as C++ and Java, which made it easy to create useful code without much background in the language.

Instead of static type checking, which is used by languages such as C++ and Java, Python uses dynamic type checking and duck typing. This allows for variables to be written without specifying the types explicitly. Duck typing causes type errors to be caught only when operations fail. This allows for the system to be lenient with certain variables and data structures. For example, different types of variables can be stored in lists without any complaints. This usually allows for code to be written faster, but it also introduces more opportunities for errors to occur. It may be harder to catch type errors in Python than in other languages because errors can often go undetected until they actually occur while running a program. One common way to help prevent errors is to use exception handling to catch and handle errors gracefully during program execution.

In terms of memory management, the python implementation used for this project (CPython) uses a private heap to manage all objects and data structures. Python and C API functions are used automatically by the interpreter to modify the size of the heap. The CPython implementation also includes garbage collection to free memory. Reference counts are used to keep track of memory that is in use. Once a reference count is changed to 0, the memory can be reclaimed. A cycle detection algorithm is also periodically used to determine if cyclic dependencies are causing objects to not be garbage-collected. CPython also includes heuristics based on the fact that older data is less likely to be garbage-collected. Many of these memory-management concepts are similar to those in languages such as Java and Ruby. However, in Java, reference counting is not used. Instead, a mark and sweep algorithm is used to free unused memory. Automatic memory management may cause slight reductions in performance, but it allows for developers to worry less about the specifics of internal memory. This is consistent with other aspects of Python that favor ease and efficiency of the development process over performance.

6.2 Twisted Event-Driven Approach

The Twisted library made it very easy to create a herd of servers for this project. Very few lines of code were needed to create a server factory and protocol, and to start the reactor. Pre-defined class methods allowed for TCP connections to be quickly implemented, and helped to send and receive data.

Instead of using multiple threads, the Twisted architecture uses a "reactor" as the core of a single-threaded event-driven loop. The main idea is that when requests come in, callbacks are added to deferred objects, which are mainly handled asynchronously. This allows for the event-loop thread to quickly become available for other incoming requests. Once the deferred object calls the last specified callback function, the response can be processed and sent to the specified location.

This event-driven approach allows for many requests to be handled quickly without worrying about the traditional complications of thread management and synchronization. However, while multi-threaded applications do not have to worry about running as many asynchronous requests, this becomes a large concern in event-driven architectures.

In the Twisted Python library, synchronous requests can delay the main event loop, which inhibits the ability to respond to requests. For this reason, extra care needs to be taken in order to ensure that the event loop is not blocked. In terms of this project, I found this approach to be more convenient than dealing with thread creation and management for each request and response. If asynchronous requests are handled efficiently in the event-driven approach, performance can be comparable to that of a multi-threaded implementation.

6.3 Node.js Comparison

Node.js is an event-driven architecture built on top of Chrome's V8 JavaScript engine. In many ways Node.js is similar to Twisted. Both use an event-driven single-threaded approach to handle incoming requests, which allows for great potential for speed and scalability. These requests are usually moved to asynchronous event handlers that use callbacks when I/O is complete. Although Twisted is older and generally has more built-in features, Node.js has rapidly grown in popularity due its simplicity and its integration with JavaScript. Since JavaScript is widely used in many web development stacks, Node.js is usually a good choice when working on web applications. This has also led to widespread community support, web hosting support, and package extensions through the NPM package manager. The V8 core is also constantly being improved, which leads to better Node.js performance.

Node.js and Twisted both allow for fast event-driven servers to be built efficiently. Both have great community background and have extensive features to incorporate into projects. Although either would be a good choice for a project in need of a related architecture, it usually makes sense to use the one that fits better in a project's tech stack. The close association of Node.js with JavaScript makes it a good candidate for many web development projects. Similarly, Twisted's association with Python may make it favorable for projects that incorporate the Python language. The extensive built-in features provided by Twisted and the numerous Node.js extensions that are available should also be considered based on the needs of the project.

7. Conclusion

Based on the project results, the Twisted architecture was found to be an effective method of creating a server herd architecture. It was easy to create servers with separate reactors, factories, and protocols in relatively small amounts of code. The flood algorithm that was implemented was seen to be effective in propagating information throughout the herd. This allows for clients to talk with different servers and receive the same results as they would if they just talked to a single server. The Python language made it relatively easy to create the required architecture, but did make it more difficult to infer types and catch errors. Overall, the event-drive approach through the use of the Twisted architecture was a powerful way to make a herd of servers that communicated with clients and efficiently propagated information amongst each other.

8. References

[1] Kinder, Ken. "Event-Driven Programming with Twisted and Python." *Linux Journal*, 26 Jan. 2005, http://www.linuxjournal.com/article/7871

[2] Node.js Foundation. "Node.js." Web. 27 Nov. 2016, https://nodejs.org/en/

[3] Norris, Trevor. "Understanding the Node.js Event Loop." *Nodesource*, 20 Jan. 2015, https://nodesource.com/blog/understanding-the-nodejs-event-loop/

[4] Python Software Foundation. "Memory Management." Web. 27 Nov. 2016, https://docs.python.org/2/c-api/memory.html

[5] Twisted Matrix Labs. "Reactor Overview." Web. 27 Nov. 2016, http://twistedmatrix.com/documents/8.2.0/core/howto/reactor-basics.html

[6] Twisted Matrix Labs. "What is Twisted?" Web. 27 Nov. 2016, https://twistedmatrix.com/trac/