Project: Twisted Places Proxy Herd

Vu Cu - University of California, Los Angeles

ABSTRACT

LAMP platform are based on GNU/Linux, Apache, MySQL, and PHP, using multiple, redundant web servers behind a load-balancing virtual router for good reliability and performance. Wikipedia and its related sites are using it. However, LAMP isn't suitable for applications that needs fast and in-sync updates, because it is designed to be reliable more than to be fast.

For such applications, application server herd is a better choice, because multiple application servers communicate directly to each other, so it is much faster. Twisted is such an application. It is an event-driven networking framework built atop of Python. We will look at its design and implementation, then we will compare it with Node.js. We will discuss the advantages and disadvantages.

Introduction

We need to implement an application server, with those assumptions: (1) updates 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.

Twisted make the design of this project quite simple, because of the Python framework. The server herd application is about 270 lines of code, contained in a single server.py file. If it is a framework based on, say, Java, the project will become much more difficult. We will compare Python's reference count based memory management and the event-driven nature with Java, and explore how it affects the choice of design.

Twisted Overview

One of the main features of Twisted is the reactor design pattern and deferred objects. The reactor is the core of the event loop within Twisted. The event loop is a programming construct that waits for and dispatches events or messages in a program from clients. It calls the event providers, which generally blocks until an event has arrived, and then dispatches it. Twisted is single thread. Single thread is more efficient than multiple thread, because multiple thread has a lot of overhead from context switching, race condition, and complexity in writing multithreaded programs.
Because the reactor is single threaded, it is not blocked. Instead it pushes the call to the deferred objects, or a promise. We need to add a callback function to the promise, which will be resolved by when the deferred object returns with some value.

Implementation

ServerProtocol

ServerProtocol is inherited from LineReceiver. It handles the events sent to the factory. Every time it received a message, it called the lineReceived function. The function calls the appropriate subroutine depends on the message format it received. It first checks the parameters. If the parameters are correctly formated, within bounds, and containing no errors, then it processes the message. After that, it propagates the results to the neighbor.

Server

Server is inherited from ServerFactory. It is a service to generate the ServerProtocol, start the server, and maintain a log files.

The log file is necessary to record all activities in the server, so that if the server crashes or encounter some bugs, we can investigate the log files and find where the error is.

Each server listens to messages through the TCP socket on a specific port. It calls on the ServerProtocol to handle requests from clients.

Client, ClientProtocol

Because the servers are constantly communicating with each other (through propagating), they also need to behave like clients. Hence the application needs both the Server component and the Client component. However the Client component is relatively simple, only needs to contain a factory.

Process_IAMAT

IAMAT is the client telling the server where it is. It stores the client's position in server.

When the client sends an IAMAT message to the server, the first step is to divide the message into tokens.

Then the server checks if it is a valid IAMAT message. A valid IAMAT message must have

- Exactly 4 arguments
- Correctly formatted position: First sanity check is to see if there is a "+" and "-" in the string, then it breaks the string into two components: the longtitude and the latitude. If both the longtitude and the latitude are valid floats, it passes the test. This test is not complete, however. A user can send an invalid longtitude and latitude pair by making them larger than 180, for example. So this application only check validity of the position, it doesn't check the bound of the position.
- Correctly formatted time: The application uses a simple sanity check. That is, if the time is a float or not. It doesn't check the bounds, for example if the time is negative.

If any of the condition above is not satisfied, the server sends a message error to the client, and it also write it to the log file.

Then the application processes the IAMAT message. It created an AT message. It also stores the information of the client and the client's position in a dictionary, so that the processing WHATSAT procedure and the processing AT procedure can use this information.

Then it propagates the result message to the neighbors. The propagation works by using the Client discussed above. During the propagation, the server acts as the client. It establishes connections to the neighboring servers using TCP port. Then it sends the AT message to the neighboring servers.

The log illustrate the whole process:

```
INFO:root:Line received: IAMAT
kiwi.cs.ucla.edu +34.068930-
118.445127 1479413884.392014450

INFO:root:IAMAT begin

INFO:root:New client:
kiwi.cs.ucla.edu

INFO:root:Response: AT Alford
+10044024.7116 IAMAT kiwi.cs.ucla.edu
+34.068930-118.445127
1479413884.392014450

INFO:root:Propagate to neighbors

INFO:root:Success! Alford propagated
to Hamilton

INFO:root:Success! Alford propagated
to Welsh

INFO:root:Done!
```

Process_WHATSAT

WHATSAT is the server finding nearby places of a client. It performs a Google Places API call to find nearby places with a given radius limiting by an upperbound number of results.

Same as IAMAT processing, it checks the parameters. A valid WHATSAT message must have:

- Exactly 4 arguments
- Correctly formatted radius: The radius must be a float. The application also check the bounds. The radius must be a positive number, capped at 50.
- Correctly formatted limit: The radius must be an integer. The application also check the bounds. The radius must be a positive number, capped at 20.

Then the server retrieves the client information in the cache. If the client isn't there, that means that client hasn't sent its position to the server yet. The server will send an error message to the client sending the WHATSAT message.

The server uses the client's location in its cache, the radius and the limit in the client's message to form the API parameters. Then the server perform an asynchronous request to Google Places, by using getPage. It returns a deferred object, so a callback function is added to the object. It performs the callback function when the API returns the result.

The callback function, print_json, uses Python json module to process the response and to filter out unnecessary information. The final result is a string in json format.

The log illustrate the whole process:

```
INFO:root:Line received: WHATSAT
kiwi.cs.ucla.edu 10 5

INFO:root:WHATSAT begin

INFO:root:Cache response: AT Alford
+10044024.7116 IAMAT kiwi.cs.ucla.edu
+34.068930-118.445127
1479413884.392014450
```

```
INFO:root:Google Place request:
https://maps.googleapis.com/maps/api/
place/nearbysearch/json?location=+34.
068930,-
118.445127&radius=10&sensor=false&key
=AIzaSyCnI8rdETbHR_UNbw1sEkPzDdPRdRMZ
dBI

INFO:root:Done!

INFO:root:Google Place response: {

    "status": "OK",

......
```

Process_AT

AT messages are not originated from clients. It is propagated from servers as the result of IAMAT location update.

Since it isn't from clients, parameter checks might not be necessary. Because in the case of IAMAT or WHATSAT messages, clients can send all kinds of strings to the server. But we have full control of what kind of AT messages the server sent. Nevertheless, sometimes a client might send an AT messages, and there is really no way to check for that. Thus, we still needs to check the parameters, to avoid a server crash should the client send a bad AT message.

However, we need to check for duplicate propagation, to avoid propagation cycle. If the client is already in the cache, and the timestamp is earlier than the timestamp in the cache, the message is old and it is a duplicate. The server doesn't propage such message to avoid an infinite loop.

The log illustrate such a case:

```
INFO:root:Log start

INFO:root:Server Hamilton

INFO:root:Port 12122

.....

INFO:root:Line received: AT Alford
+10044024.7116 IAMAT kiwi.cs.ucla.edu
+34.068930-118.445127
1479413884.392014450

INFO:root:AT begin

INFO:root:Duplicate from Alford

INFO:root:Done!
```

If the message is new, the server simply propagates it to the neighboring servers, using the same process as when it processes IAMAT message.

The log illustrate the whole process:

```
INFO:root:Server Ball

INFO:root:Port 12121

.....

INFO:root:Line received: AT Alford
+10044024.7116 IAMAT kiwi.cs.ucla.edu
+34.068930-118.445127
1479413884.392014450

INFO:root:AT begin

INFO:root:(AT) Update from new
client: kiwi.cs.ucla.edu

INFO:root:Added kiwi.cs.ucla.edu : AT
Alford +10044024.7116 IAMAT
kiwi.cs.ucla.edu +34.068930-
118.445127 1479413884.392014450

INFO:root:Propagate to neighbors

INFO:root:Success! Ball propagated to
Holiday

INFO:root:Success! Ball propagated to
Welsh

INFO:root:Done!
```

Python's Type Checking vs Java's

Python checks its type dynamically, while Java checks its type statically.

Thanks to the dynamic type checking in Python, the developer doesn't have to specify the type, and he can write the server herd application quickly like a pseudo code. We only need 270 lines of code in a single Python file to write this application. If we have to write it in Java, we probably have to specify the classes, and it will be in multiple files with much longer code.

Also, it makes the Python program more readable and make more intuitive sense. However, the Python types are ambiguous. We don't know its type, that's why in our program we have to use routines such as isFloat(), isdigit() to check the parameter types.

Generally, Python program is easier to understand on a higher level, but harder to understand when we need to examine the details.

Python's Memory Management vs Java's

It depends on the Python implementation, but for now we assume the CPython implementation.

Python memory management is a mixture between reference counting, mark-and-sweep, and garbage collection. Java using generational garbage collection.

The reference counting features is suitable for this server herd application. For reference counting, when the reference count drops to zero, it garbage collects the object. In this application, we use a lot of temporary objects, and they need to be free quickly once we don't need them anymore.

In Java, we will need to allocate a lot of objects with the keyword new, and the generational garbage collection will delay the deletion, so the temporary objects will remain in memory long after their usefulness. Reference counts, on the other hand, immediately removes an object when it is no longer useful to us.

Multithreading

Python is mainly designed for single thread application, and it doesn't have a lot of support for multithreaded application. Java, on the other hand, has many standard libraries for multithreading.

Twisted is using single threaded, with asynchronous event handling to perform networking operations. In our application, the only place for it is when we call getPage in process_WHATSAT, and we attach a callback function. If we write it in Java, we probably have to build a multithreaded application, which will be slower because of context-switching cost.

Twisted vs Node.js

Node.js, written in Javascript, is a server-side application, similar to Twisted. Both Twisted and Node.js uses asynchronous events for networking. Both of them are also single threaded.

However, Node.js is relatively newer and less mature than Twist. But in a short few years, it has been gain tremendous popularity. The author of this paper is more familiar with Node.js before he discover Twisted.

The reason is unity. Python is used for backend technologies like Twisted or Django, but Javascript is usable for both backend technologies (Node.js) and frontend technology (Angular.js, React.js).

Also Javascript, with JSON, is a language writing especially for the web, it is runs atop of V8 JavaScript Engine, so it is easily integrating with web applications that need a server.

That's explain the rising popularity of Node.js. It is good for most purpose, however, since it is newer, applications with heavy server-side computation should still use Twisted, because Twisted is more mature, well experimented, and well documented.

Conclusion

Twisted is a simple yet powerful framwork for implementing server herd. Python is a easy-to-use languages, thus the development of this application is quick. Also the short code makes maintain the code easy. The application is heavily modularized within Twisted, and the framwork allows abstraction of the lower networking layer. I would strongly recommend to use it in networking application. Also, Node.js is also an attractive alternative, because of the simple framework and the Javascript language.

References

[1] "Developer Guides" Developer Guides - Twisted 16.5.0. Documentation http://twistedmatrix.com/documents/current/core/howto/index.html

[2] Python and Java comparison. https://pythonconquerstheuniverse.wordpress.com/2009/10/03/python-java-a-side-by-side-comparison/

[3] Why I'm Switching From Python To NodeJS. https://blog.geekforbrains.com/why-im-switching-from-python-to-nodejs-1fbc17dc797a#.q93n9683r