

Project. Twisted Places proxy herd

Background

[Wikipedia](#) and its related sites are based on the Wikimedia Architecture, which uses a [LAMP](#) platform based on [GNU/Linux](#), [Apache](#), [MySQL](#), and [PHP](#), using multiple, redundant web servers behind a load-balancing virtual router for reliability and performance. For a brief introduction to the Wikimedia Architecture, please see Mark Bergsma, [Wikimedia architecture](#) (2007). For a more extensive discussion, please see Domas Mituzas, [Wikipedia: Site internals, configuration, code examples and management issues \(the workbook\)](#), MySQL Users Conference 2007.

While LAMP works fairly well for Wikipedia, let's assume that we are building a new Wikimedia-style service designed for news, where (1) updates to articles 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. In this new service the application server looks like it will be a bottleneck. From a software point of view our application will turn into too much of a pain to add newer servers (e.g., for access via cell phones, where the cell phones are frequently broadcasting their [GPS](#) locations). From a systems point of view the response time looks like it will too slow because the Wikimedia application server is a central bottleneck.

Your team has been asked to look into a different architecture called an "application server herd", where the multiple application servers communicate directly to each other as well as via the core database and caches. The interserver communications are designed for rapidly-evolving data (ranging from small stuff such as GPS-based locations to larger stuff such as ephemeral video data) whereas the database server will still be used for more stable data that is less-often accessed or that requires transactional semantics. For example, you might have three application servers A, B, C such that A talks with B and C but B and C do not talk to each other. However, the idea is that if a user's cell phone posts its GPS location to any one of the application servers then the other servers will learn of the location after one or two interserver transmissions, without having to talk to the database.

You've been delegated to look into the [Twisted](#) event-driven networking framework as a candidate for replacing part or all of the Wikimedia platform for your application. Your boss thinks that this might be a good match for the problem, since Twisted's event-driven nature should allow an update to be processed and forwarded rapidly to other servers in the herd. However, he doesn't know how well Twisted will really work in practice. In particular, he wants to know how easy is it to write applications using Twisted, how maintainable and reliable those applications will be, and how well one can glue together new applications to existing ones; he's worried that Python's implementation of type checking, memory management, and multithreading may cause problems for larger applications. He wants you to dig beyond the hype and really understand the pros and cons of using Twisted. He suggests that you write a simple and parallelizable proxy for the [Google Places API](#), as an exercise.

Assignment

Do some research on Twisted as a potential framework for this kind of application. Your research should include an examination of the Twisted source code and documentation, and a small prototype or example code of your own that demonstrates whether Twisted would be an effective way to implement an application server herd. Please base your research on Twisted 16.5.0 (dated 2016-10-28), even if a newer version comes out before the due date; that way we'll all be on the same page. (We suggest using Python 2.7.12; much of Twisted 16.5.0 works with Python 3 but some still assumes Python 2.)

Your prototype should consist of five servers (with server IDs 'Alford', 'Ball', 'Hamilton', 'Holiday', 'Welsh') that communicate to each other with the following pattern:

1. Alford talks with Hamilton and Welsh.
2. Ball talks with Holiday and Welsh.
3. Hamilton talks with Holiday.

Each server should accept [TCP](#) connections from clients that emulate mobile devices with IP addresses and DNS names. A client should be able to send its location to the server by sending a message using this format:

```
IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1479413884.392014450
```

The first field `IAMAT` is name of the command where the client tells the server where it is. Its operands are the client ID (in this case, `kiwi.cs.ucla.edu`), the latitude and longitude in decimal degrees using [ISO 6709](#) notation, and the client's idea of when it sent the message, expressed in [POSIX time](#), which consists of seconds and nanoseconds since 1970-01-01 00:00:00 UTC, ignoring leap seconds; for example, `1479413884.392014450` stands for 2016-11-17 20:18:04.392014450 UTC. A client ID may be any string of non-white-space characters. (A white space character is space, tab, carriage return, newline, formfeed, or vertical tab.) Fields are separated by one or more white space characters and do not contain white space; ignore any leading or trailing white space on the line.

The server should respond to clients with a message using this format:

```
AT Alford +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1479413884.392014450
```

where `AT` is the name of the response, `Alford` is the ID of the server that got the message from the client, `+0.263873386` is the difference between the server's idea of when it got the message from the client and the client's time stamp, and the remaining fields are a copy of the `IAMAT` data. In this example (the normal case), the server's time stamp is greater than the client's so the difference is positive, but it might be negative if there was enough [clock skew](#) in that direction.

Clients can also query for information about places near other clients' locations, with a query using this format:

```
WHATSAT kiwi.cs.ucla.edu 10 5
```

The arguments to a `WHATSAT` message are the name of another client (*e.g.*, `kiwi.cs.ucla.edu`), a radius (in kilometers) from the client (*e.g.*, 10), and an upper bound on the amount of information to receive from Places data within that radius of the client (*e.g.*, 5). The radius must be at most 50 km, and the information bound must be at most 20 items, since that's all that the Places API supports conveniently.

The server responds with a `AT` message in the same format as before, giving the most recent location reported by the client, along with the server that it talked to and the time the server did the talking. Following the `AT` message is a [JSON](#)-format message, exactly in the same format that Google Places gives for a Nearby Search request (except that any sequence of two or more adjacent newlines is replaced by a single newline and that all trailing newlines are removed), followed by two newlines. Here is an example (with some details omitted and replaced with "...").

```
AT Alford +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1479413884.392014450
{
  "html_attributions" : [],
  "next_page_token" : "CvQ...L2E",
  "results" : [
    {
      "geometry" : {
        "location" : {
          "lat" : 34.068921,
          "lng" : -118.445181
        }
      },
      "icon" : "http://maps.gstatic.com/mapfiles/place_api/icons/university-71.png",
      "id" : "4d56f16ad3d8976d49143fa4fdffbc0a7ce8e39",
      "name" : "University of California, Los Angeles",
      "photos" : [
        {
          "height" : 1200,
          "html_attributions" : [ "From a Google User" ],
          "photo_reference" : "CnR...4dY",
          "width" : 1600
        }
      ],
      "rating" : 4.5,
      "reference" : "CpQ...r5Y",
      "types" : [ "university", "establishment" ],
      "vicinity" : "Los Angeles"
    },
    ...
  ],
  "status" : "OK"
}
```

(The example ends with an empty line, since it is terminated with two newlines.)

Servers should respond to invalid commands with a line that contains a question mark (?), a space, and then a copy of the invalid command.

Servers communicate to each other too, using AT messages (or some variant of your design) to implement a simple [flooding algorithm](#) to propagate location updates to each other. Servers should not propagate place information to each other, only locations; when asked for place information, a server should contact Google Places directly for it. **Servers should continue to operate if their neighboring servers go down**, that is, drop a connection and then reopen a connection later.

Each server should log its input and output into a file, using a format of your design. The logs should also contain notices of new and dropped connections from other servers. You can use the logs' data in your reports.

You'll need an API key to use Google Places. You can get a free key that will let you do a limited amount of testing per day, so don't overdo it, and remember that it's not a good idea to publish the key. See [Enable an API](#) (requires authentication) to get an API key.

Write a report that summarizes your research on whether Twisted is a suitable framework for this kind of application. Your report should make a recommendation pro or con, and justify your recommendation. Describe any problems you ran into. Your report should directly address your boss's worries about Python's type checking, memory management, and multithreading, compared to a Java-based approach to this problem. Your report should also briefly compare the overall approach of Twisted to that of [Node.js](#), with the understanding that you probably won't have time to look deeply into Node.js before finishing this project.

Your research and report should focus on language-related issues. For example, how easy is it to write Twisted-based programs that run and exploit server herds? What are the performance implications of using Twisted? Don't worry about nontechnical issues like licensing, or about management issues like software support and retraining programmers.

Style issues

Please see [Resources for oral presentations and written reports](#) for advice on generating high-quality reports.

Your report should use standard technical academic style, and should have a title, abstract, introduction, body, recommendations/conclusions, references, and any other sections necessary. Limit your report to at most five pages. Use the [USENIX style](#), which uses a two-column format with 10-point font for most of the text, on an 8½"×11" page; an [example of the output format](#) and an [example student paper](#) are available.

Your report is not expected to be just like that example student paper! That was written by a graduate student, she was writing a conference paper describing months of full-time research, and the paper is too long for us. It's merely an example of technical style and layout.

Research mechanics

If you need to run servers on SEASnet to do your research, please let the TA know how many TCP ports you need, and we will allocate them for you. Please do not use TCP ports at random, as that might collide with other students' uses.

You can grab a copy of the Twisted source code from the Twisted web site. You can also find a copy of the Twisted source code in the directory `~eggert/src/Twisted-16.5.0` on SEASnet; a distribution compressed tarball is in the file `/u/cs/fac/eggert/src/tarpit/Twisted-16.5.0.tar.bz2`.

You can run Twisted on SEASnet by prepending `/usr/local/cs/bin` to your `PATH` in the usual way. For example, you can run the chat server (taken from the [Twisted code examples](#)) as follows:

```
# Replace '12000' with your TCP port number as allocated by the TA.
sed 's/1025/12000/' /u/cs/fac/eggert/src/Twisted-16.5.0/docs/core/examples/chatserver.py >chatserver.py

# We recommend the -n option to forestall runaway servers.
twistd -n -y chatserver.py

# Now you can telnet to port 12000, from another session
```

```
# on the same host, to exercise the chat server.  
# A log should appear on your screen.
```

```
# Type control-C to terminate the chat server.
```

If you can think of some similar project you'd like to do, that resembles the current project but is cooler, ask the T.A. for permission to do the similar project, and then go for it.

Submitting your work

Submit:

1. a file named `report.pdf` containing a copy of your paper in PDF form
2. any other supporting work (e.g., your source code) in a gzipped tar file named `project.tgz`.

References

This is a Safari online book, so its contents should be viewable by anybody on the UCLA campus.

Jessica McKellar & Abe Fettig, [*Twisted network programming essentials*](#), 2nd edition, O'Reilly (2013), ISBN 978-1-4493-2611-1.

© 2008–2016 [Paul Eggert](#). See [copying rules](#).

\$Id: pr.html,v 1.45 2016/11/18 22:00:25 eggert Exp eggert \$