

Project. Proxy herd with asyncio

[131] [home](#) > [Homework](#)]

Background

[Wikipedia](#) and related websites are based on the [Wikimedia server](#) platform, which is based on [Debian GNU/Linux](#), the [Apache](#) web server behind an [NGINX-based TLS termination proxy](#), the [Memcached](#) distributed memory object cache, the [MariaDB](#) relational database, the [Elasticsearch](#) search engine, the [Swift](#) distributed object store, and core application code written in [PHP+JavaScript](#), all using multiple, redundant web servers behind the [Linux Virtual Server load-balancing software](#), with two levels of caching proxy servers ([Varnish](#) and [Apache Traffic Server](#)) for reliability and performance. See [Wikimedia infrastructure](#) for diagrams of how this all hooks together.

While this 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 or HTTPS, and (3) clients will tend to be more mobile. In this new service the PHP+JavaScript 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 on the core clusters.

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 Python's [asyncio asynchronous](#) networking library 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 `asyncio`'s event-driven nature should allow an update to be processed and forwarded rapidly to other servers in the herd. However, she doesn't know how well `asyncio` will really work in practice. In particular, she wants to know how easy is it to write applications using `asyncio`, how maintainable and reliable those applications will be, and how well one can glue together new applications to existing ones; she's worried that Python's implementation of type checking, memory management, and multithreading may cause problems for larger applications. She wants you to dig beyond the hype and really understand the pros and cons of using `asyncio`. She suggests that you write a simple and parallelizable proxy for the [Google Places API](#), as an exercise.

Assignment

Do some research on `asyncio` as a potential framework for this kind of application. Your research should include an examination of the `asyncio` [source code](#) and [documentation](#), and a small prototype or example code of your own that demonstrates whether `asyncio` would be an effective way to implement an application server herd. Please base your research on `asyncio` for Python 3.11.2, even if a newer version comes out before the due date; that way we'll all be on the same page.

Your prototype should consist of five servers (with server IDs 'Bailey', 'Bona', 'Campbell', 'Clark', 'Jaquez') that communicate to each other (bidirectionally) with the following pattern:

- Clark talks with Jaquez and Bona.
- Campbell talks with everyone else but Clark.
- Bona talks with Bailey.

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 1621464827.959498503
```

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, `1621464827.959498503` stands for 2021-05-19 22:53:47.959498503 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 Clark +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1621464827.959498503
```

where AT is the name of the response, Clark 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 Clark +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1621464827.959498503
{
  "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" : "4d56f16ad3d8976d49143fa4fdfffb0a7ce8e39",
      "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 [Enabling APIs](#) for an intro on how to get an API key.

To query the Google Places API, you will need to send Google servers an HTTP request. However, `asyncio` only supports the TCP and SSL protocols. Your server will need to manually create and send an HTTP GET. You may use the [aiohttp](#) library to do so.

You may use as many `.py` files as you would like to prototype this server. However, your prototype must have a main file named `server.py`. `server.py` takes one command line argument containing the name of the server (e.g. `python3 server.py Clark`), and must start the server on one of your assigned ports. Contact the TAs to get your assigned ports.

Write a report that summarizes your research on whether `asyncio` 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 `asyncio` to that of [Node.js](#), with the understanding that students who have not worked with Node.js in earlier courses 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 `asyncio`-based programs that run and exploit server herds?
- What are the performance implications of using `asyncio`?
- How important is it to rely on [asyncio features of Python 3.9 or later](#), such as `asyncio.run` and `python3 -m asyncio`, or is it easy to get by with older versions of Python?

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 paper](#) (with an extra coverage you don't need) are available.

Your report is not expected to be just like that example paper! That was written by experts 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.

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.

Please use only the Python standard library and the libraries mentioned in this specification. If you would like to use a different library, please get the approval of the TA first. Also, please ensure that your server does not send any data other than what is mentioned in the spec over TCP, as we will grade your server based on its TCP output.

Frequently Asked Questions

- How do I find the ports assigned to me?*

Please check on CampusWire.

- What happens if I don't use my assigned ports?*

Your final submission should have your assigned ports hardcoded into it. If not, you will lose points.

- What libraries can I use for HTTP requests?*

You can use the `aiohttp` library to send HTTP requests asynchronously. Please do not use synchronous libraries like `urllib` or `requests`.

- How do I parse JSON?*

Use the `json` package.

- Is the 'talks to' relationship bidirectional or unidirectional?*

The relationship is bidirectional. If Clark talks to Jaquez, Jaquez also talks to Clark.

- How do I test my server?*

You may use the `telnet` or `nc` commands to test your server. However, we will be testing your servers using a python client implemented with `asyncio`, so please make sure your solution works with this library.

- How should the herd behave if a server goes down?*

When a server goes down and comes back up, the herd should talk to it as usual. It is not necessary to propagate old messages to a server when it comes back up.

- Can I output logging information to telnet?*

Output only the specific response data mentioned in the spec. You will lose points if your server output anything other than what is specified.

Submitting your work

Submit:

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

Reference

This is a Safari online book, so its contents should be [viewable by anybody with a UCLA campus IP address](#).

Luciano Ramalho, *[Fluent Python](#)*, 2nd edition. O'Reilly (2021), ISBN 9781492056355.