Using asyncio for Application Server Herds

Jeffrey Xu
University of California, Los Angeles

Abstract

While using a single virtual router to coordinate and balance multiple web servers works for Wikipedia, it is not the most effective approach for all applications. Application server herds use multiple servers that can directly service requests while also directly sharing short-term data, which requires fast and asynchronous processing of requests to ensure that all servers quickly gain the most up-to-date data. In this paper, I examine the asyncio Python library for event-driven networking as a solution for this problem. This paper also investigates the potential issues and benefits caused by using Python for this application.

1 My Implementation

1.1 Structure

The server is built around event-driven concurrency. The event loop provided by asyncio allows my program to sleep when there is no data being read, processed, or written and automatically shifts to working on another task instead of blocking to await input or output, which provides the performance increases of classical multithreaded concurrency while avoiding its headaches and data races by transferring control only at specific, reliable points. asyncio provides two ways of doing this: callback-based Protocols and coroutine-based Streams. Protocols work by scheduling execution of other callbacks on the event loop, which can then switch to executing the callback while another method is blocked by input or output. Streams provide the ability to voluntarily suspend the execution of a coroutine while waiting for another one, which allows the coroutine to be ignored instead of holding up the CPU while waiting for data to be input or output.

1.2 Implementation details

My program uses Protocols to implement its concurrency. The main protocol awaits connections from clients or other servers, then processes their requests. Connections are only kept open as needed; even connections from servers to transfer updates on clients' data are closed as soon as each message is passed. The only case where a connection may be held open for an extended period of time is when the response requires querying the Google Places API, as the connection must remain open while waiting for the data from Google. However, the program is structured so the callback ends immediately after scheduling a connection to the Google Places API, so further requests can continue to be served while waiting.

As the main protocol's purpose is to serve requests from clients and updates from other servers, other protocols must be used to send messages to Google or other servers. While both protocols are similar in that their main purpose is simply to send a message as soon as the connection is opened, there are some minor differences that necessitate separate protocols. In particular, the protocol to query the Google Places API must close the connection itself, while the protocol to communicate with other servers can allow the server to close the connection for it. In addition, the Google Places API returns data in chunks, which must be buffered and gathered until the entire body is transfered, and then the data must be sent back to the main protocol to be returned to the client. While AbstractEventLoop.create_connection is used to connect to Google or other servers, this data must be returned across an already existing connection, so AbstractEventLoop.call_soon is used to asynchronously schedule a method contained in the instance of the main protocol that is connected to the client that sent the WHATSAT request.

In order to allow my implementation to accept and process requests delivered in chunks rather than all at once,

my program requires received messages to be terminated by a newline. The server buffers all input from a given connection until a newline or end of file is received, at which point it attempts to process all buffered data before the newline as a command. Errors that are checked for include an empty message, which causes no response and leaves the connection open, invalid commands, incorrect number of fields provided for a command, attempting a WHATSAT query for a client the server does not know about, or providing invalid values for the numeric fields, all of which cause the server to echo the failed message with a leading question mark and then close the connection.

To propagate updates throughout the server herd, I used a very simple flooding algorithm, where upon receiving an update from a client, a server produces a modified AT message that includes its own name twice. This modified message is sent to both servers that it is connected to. Because the given server configuration contains only one cycle, and the cycle visits all servers, each subsequent server only needs to ensure that it does not pass along the message to the server it received the message from or the server that originated the message. While this sends the message to each server twice, it also ensures that if any one server goes down, all the other servers would still receive the message, while only passing the message in one direction would be stopped before servers reachable in the other direction. While the information from each update is only retained if the timestamp is newer than that of the currently stored data, the update is propagated regardless in case another server went down and came back up in between the previously received message with a newer timestamp and the more recently received but already out of date message. This keeps each server as up-to-date as possible without needing to resend messages.

1.2.1 Notes for running the program

Because I performed much of my testing on my own machine, there are several things to note. First, in order to handle requests delivered only a character at a time before I built my test client, every request must be terminated with a newline. Second, the port numbers used for each of the five servers were chosen arbitrarily and were not allocated to me by SEASNET because my testing was done locally. The port numbers can be modified by changing the port_of dict on line 22 of server.py.

To create a server, use server.py with the desired server name as the sole command line argument.

2 Conclusions

2.1 asyncio

asyncio is more than capable of implementing an application server herd. Event-driven concurrency allows the server to continue running while waiting for input or output to be communicated across the network or between servers, and the encapsulation of each connection in its own Protocol or Stream means that this concurrency does not risk causing races because states are not shared, and in any event because the program is single-threaded, all operations are effectively atomic. asyncio uses an event loop to supervise execution of its concurrent code, so when input or output forces a callback in a Protocol or a coroutine in a Stream to wait, the event loop can take control to respond to other events. Furthermore, coroutines can yield to allow other functions to run or input or output arrive or be sent without blocking the rest of the server.

2.2 Python

Python is a good choice for this application. Its dynamic type checking allows request strings and JSON to be processed easily and efficiently, without needing to standardize or organize the types of the resulting data.

Memory management using reference counts works well, because objects are created for each connection, then are no longer needed when the connection is closed, and no references to them remain. This well-defined lifespan for its objects means that reference counts are a successful and simple way of performing garbage collecting. Garbage collecting is desirable to avoid the hassle and potential bugginess caused by manually freeing objects, but it risks memory leaks if it cannot successfully determine that an object is no longer needed, and can negatively affect performance by needing to scan through a large number of objects to check which ones are garbage, especially in a high-throughput server, where a very large number of transient objects may constantly be produced and discarded, meaning even a generation-based collector would have a lot of objects to process. Python's reference counts naturally avoid the issue of examining a large number of objects by naturally maintaining each object's status in the course of normal execution. The issues that may be caused by reference counts leaking memory when a reference is no longer needed but is not discarded are avoided in this application because the objects managing each connection that make up the majority of the objects created by the server, especially if the server is in high demand, are promptly destroyed when the corresponding connection ends. Thus, the objects managing each connection can be

immediately cleared from memory by Python's garbage collector, freeing it up for more connections.
Finally, Python's approach to multithreading is not an issue because as previously noted, event-driven concurrency on a single thread is more effective for this application, because multithreading does not offer much more while being much harder to execute [4] due to concerns about shared data, which necessitates locks, which in turn can cause deadlock.

2.3 Comparison to NodeJS

Much of what Python's asyncio library can do can also be done by NodeJS. Because both asyncio and NodeJS use a similar event-driven approach to concurrency, the decision of which one to use must be based on other features. NodeJS has some notable advantages: it has better performance in some applications [5], and the huge amount of libraries provided by npm dwarfs anything Python can put up [3]. Nevertheless, Python has its own advantages and should not be dismissed out of hand. As a young, very popular framework, NodeJS is still evolving rapidly, and the same huge amount of libraries provided by npm can cause bewildering dependency issues [1]. NodeJS also suffers from strange issues with error handling and callbacks, which is worsened by the current lack of standardization in JavaScript [2]. Thus, I recommend using Python for this application.

References

[1] David Haney, *NPM & left-pad: Have We Forgotten How To Program?*, Haney Codes .NET, http://www.haneycodes.net/npm-left-pad-have-we-forgotten-how-to-program

[2] Gavin Vickery, *After A Year Of Using NodeJS In Production*, Geek for Brains, https://blog.geekforbrains.com/after-a-year-of-using-nodejs-in-production-78eecef1f65a

[3] Gavin Vickery, *Why I'm Switching From Python To NodeJS*, Geek for Brains, https://blog.geekforbrains.com/why-im-switching-from-python-to-nodejs-1fbc17dc797a

[4] Thomas Claburn, *The future of Python: Concurrency devoured, Node.js next on menu*, The Register, https://www.theregister.co.uk/2017/08/16/python_future

[5] Uroosa Sehar, *Node.JS vs Python: Which is Best Option for Your Startup*, Vizteck Solutions, https://vizteck.com/blog/node-js-vs-python-best-option-startup