CS131 Project: Proxy Server Herd with asyncio

Su Mon Thaw
University of California, Los Angeles

Abstract

Wikipedia is based off of a web architecture using the LAMP stack, but in the scenario where it were to become a news service, other options need to be considered. Since updates to articles will become more frequent and accesses need to be flexible for more protocols than just HTTP, the current application server will become a bottleneck in terms of response time. This report will examine the option of implementing an "application server herd" architecture through Python's asyncio library and compare this approach against other options such as writing it in Java or using Node.js.

Introduction

An "application server herd" is an architecture are where multiple servers are both able to communicate directly with each other as well as with caches and databases. The ability to intercommunicate with one another is the key to defeating the bottleneck. Ideally, information can be rapidly processed and passed on through others in the herd with this architecture. Python's asyncio library allows an event-driven style of networking which is potentially a viable implementation for our herd. However, Python's static type checking made it a bit difficult to complete our parallelizable proxy for the Google Places API.

1. Implementation

The prototype focuses on five servers--Alford, Ball, Hamilton, Holiday, and Welsh. When our client talks to one of these servers, relevant information would eventually propagate to all of them through interserver communications. We were also instructed to have other servers continue to operate even when one of them is down.

The program is thus split up into two files -- main.py which imports server.py. Within main.py, we instantiate the server objects, giving the option to either start one particular server by taking in a command-line argument or starting all of the servers using the multiprocessing library. Inside server.py, we actually define the server object that creates and starts the event loop along with variables and procedures that will handle its communications. For example, all the communications will first be directed towards the procedure *handle_connection* which breaks down the message (and disregarding the whitespace) using *split()*. It also makes use of the server's *msgDirectory* to pass the work onto its corresponding message handler. In addition, the directory lookup is wrapped in a try-except statement to account for invalid commands.

1.1. Handling IAMAT

With IAMAT messages, I was able to easily dissect the information that was passed into the *handle_IAMAT* procedure by simply indexing the list. The message is archived within a *messages* dictionary that is keyed by the client name because we want to sort our updates by the recent message it individually gives us.

When creating the timestamp for the returning AT message, I ran into a bit of trouble because I forgot to typecast the message's 'time' parameter from a string to an float then back to a string. Note that I was not able to catch this until runtime testing of this procedure.

Propagation of languages was handled within both *relay_messages* and *propagate.* The former is only passed the AT message string as an argument. The procedure then looks up the server object's name inside its *srv_directory* dictionary to find which other servers it needs to pass on the information to and creates an event loop for each with a for loop, each instance passing the next steps onto *propagate*. This procedure then actually sends the AT message to the other servers, also checking if it's working using a try-except statement. If the intended server is not working, it throws an exception and logs the failure.

1.2. Handling AT

Since our connections are bidirectional, we first need to check if the AT message is already inside the server's message dictionary. This prevents perpetual

propagation of the AT messages. If the AT message is yet stored in our server's messages, this procedure stores it and calls *relay_information* to make sure the other servers will receive it as well.

1.3. Handling WHATSAT

Similar to handling the other types of messages, *handle_WHATSAT* can easily index the message into parts to look up location information in the server object's *messages* dict. It then can easily construct the corresponding AT message. The procedure then passes all of this information to another procedure *get_location_info* and adds it to the event loop through asyncio's *ensure_future* proc.

With *get_location_info*, we are able to send an HTTP request to the Google Maps API. I was a bit worried that it would be difficult to construct the correct format for this request. but Python's string operations (*format* and *encode*) made it easy to communicate with the API and collect the corresponding JSON information. Luckily, Python also has a json library that helps parse json files by converting it to a dictionary. With this, the procedure can limit the number of results we get from the json response by simply using Python's *slice* and reverting its format by using the json library's *dump.*

2. Comparison to Java

While implementing the solution, I found that Python's static typing gave a false sense of security when testing the servers. At first, it was convenient to not have to think about types when declaring procedures and variables. All that was needed is a name to start using it[1], unlike Java. However, I realize that it was frustrating to catch bugs during runtime instead of compile time, a problem that Java has some protection against with its dynamic type checking. One wouldn't be able to catch an error with a particular procedure until it's actually used within the right context. With larger applications, this would become a problem, as it'd be harder to ensure that the program is robust and reliable. As new applications are potentially added on, there must be extra attention to how these would connect with existing ones and not break. The duck typing feature seems to ultimately push extra work onto the programmers in the long-term.

However, since our project also dealt with quite a bit of parsing, it was helpful that Python's language features was easy to pick up and use, making the code easier to write and also read. For example, my implementation passes in a multitude of arguments which could be hard to keep track of. I appreciated the fact that I was able to specify arguments out of order by putting *'argname=value'*. In addition, I was easily able to pass on functions as an argument, which I found to be pretty convenient. Python also had extensive string operations, such as *split* and *format* that were easy to pick up and use.[2] These language features allowed me to write and implement the program quickly and seamlessly. Despite the fact that coding in Python makes it hard to analyze what goes on at the lower level when an error occurs[3], it facilitates the development of smaller and simpler solutions which would otherwise be more complicated in a lower-level languages.

In terms of memory management, Python has an entirely heap based model, so the developer doesn't need to be concerned about manually allocating objects for the heap. Furthermore, while both Python and Java have a garbage collector, Python also uses reference counting. Since everything in Python is an object, everything has a reference count that is increased when put in a container or assigned to another name and deleted when it's reassigned or goes out of scope.[4] When the count hits zero, it's automatically disposed to conserve memory. This feature is especially useful for this project because it can immediately remove cycles that are no longer used to conserve memory without waiting for the garbage collector.

Python's event driven asyncio is also running single-threaded, compared to a Java approach which would make use of a multithreaded model. The latter's performance would have a dependence on how many processors were to be available while running the program. With the single-threaded model, the developer would have performance consistency throughout various machines, which is more important for our server herding.

3. Comparison to Node.js

Both Node.js and Python's asyncio are event-driven and uses single-threading. In addition, Javascript and Python both have simplistic syntax that's quick to develop with. However, Node.js is new and does not have the extensive library support that Python provides as a mature language[5]. Despite this, Node.js is becoming more popular amongst developers and has a significant amount of community support to provide new packages with npm, the package manager.

With Python's asyncio, a procedure can be passed as an argument to use as a callback function. In Node.js, it

similarly utilizes promises to retain certain features of control flow, function semantics, and exception handling that are lost through the asynchronous model[6]. This convenient feature gives a choice between two callbacks for when a function is resolved to examine what occurred with the asynchronous code. Overall, Node.js seems to be a substantive alternative for server-herding compared to Python.

Conclusion

Although Python's duck typing made it difficult to catch multiple errors before runtime testing and asyncio's single-threaded design has a risk of livelock, its language features facilitated quick development for server-herding. In addition, single-threading has the benefit that the programmer doesn't have to worry about cost of context switching between threads. Python's memory model also makes good use of reference counting and garbage collecting to reduce the programmer's responsibility to manage these details and focus on implementing the program instead. I would overall recommend the use of asyncio because of the abstraction that Python provides to cover low-level management because I value quick development with a moderate amount of robustness. In addition, I would also suggest researching more into using Node.js as an alternative for other web server application due to its increasing popularity and community support among developers.

References

[1] Klein, B (2011) *Python Tutorial*. Retrieved from https://www.python-course.eu/variables.php

[2] 7.1. string — Common string operations. In Python.org. (2017, December 3). Retrieved from https://docs.python.org/2/library/string.html

[3] Radcliffe, T (2016, January) *Python vs. Java: Duck Typing, Parsing on Whitespace and Other Cool Differences*. Retrieved from https://www.activestate.com/blog/2016/01/python-vs-java-duck-typing-parsing-whitespace-and-other-cool-differences

[4] Beazley, D (2009, July) *Python Essential Reference: Types and Objects.* Retrieved from http://www.informit.com/articles/article.aspx?p=1357182&seqNum=3

[5] Sehar, U (2016, August) *Node.js vs Python: Which is best option for your startup.* Retrieved from https://vizteck.com/blog/node-js-vs-python-best-option-startup/

[6] Harter, M (2016, August) *Promises in Node.js - An Alternative to Callbacks.* Retrieved from https://developer.ibm.com/node/2016/08/24/promises-in-node-js-an-alternative-to-callbacks/