CS131 Python Project Proxy Herd with asyncio Report

UCLA

Abstract

In order to determine the viability of Python and the concurrency library asyncio for a Wikimedia-style news service based on “server herd architecture” a prototype was created. In addition to the prototype, we explore various features of the Python language—type checking, memory management and multithreading—as well as asyncio and other considerations in order to determine if the Python and asyncio library combination will aptly serve the intended purpose. The Python and asyncio library are further compared with its counterparts in Java and Node.js to further illustrate their viability. It is concluded that, with performance consideration in mind for a server herd that has a greatly increased number of IO requests, Python and asyncio are an appropriate choice.

1 Introduction

This paper aims to look into Python’s asyncio library in order to determine whether it is viable for the purposes of creating a Wikimedia-style service designed for news. The Wikimedia-style service is based on architecture that relies on multiple PHP and JavaScript application servers. For the news service in the Wikimedia style the usage and requirements are different so a different type of server architecture must be considered.

2 Background

The service will be designed around the “application server herd” architecture where application servers may communicate with each other in order to update data and information, while allowing client mobile devices to communicate to each of the respective servers. In a “application server herd” not all application servers may be able to communicate directly with the other and information is instead, propagated throughout the web of communicating servers. The “application server herd” will then communicate to the database for any further need of information. In order to build a Wikimedia-style server designed for news via server herd consideration must be given to three factors that differ for the purposes of this new service:

1. Updates to articles will happen far more often.
2. Access will be required via various protocols (not just HTTP and HTTPS).
3. Clients will tend to be more mobile.

In the consideration for Python and asyncio as a viable technology for implementation of this new service we have to consider the new tradeoffs that come from these three new factors. For example, how will performance be affected as there is considerably more IO being done from mobile devices updating their location to the servers and whether this design will allow access from various protocols.

3 Server Herd Prototype

A prototype of the “application server herd” was developed with Python’s asyncio library in order to determine viability of Python and asyncio for the task. The “application server herd” is created with five servers with IDs 'Bailey', 'Bona', 'Campbell', 'Clark' and 'Jaquez' where inter-server communication is limited to bidirectional lines: Clark talks with Jaquez and Bona, Campbell talks with everyone else but Clark, and Bona talks with Bailey.

The client may communicate with any of the servers within the herd via messages marked with command “IAMAT” to update the server on its location data and command “WHATSAT” to request location data of a client who previously sent location data to the server herd. The “IAMAT” command makes the servers propagate the client’s location data to its peer servers until the entire server herd is updated with the latest client location update via flooding algorithm. The “WHATSAT” command further retrieves Google Places JSON Data in order to show the ability of different protocol access via accessing a database.

For inter-server communication and response to the client the “AT” command is used. The “AT” command will be given to the client along with the client’s message data to indicate success of client message retrieval. For updating the peer servers of location data, the “AT” command will also be used where the source server (server where client originally gave “IAMAT” message) will propagate the message to the connected servers and they will do so as well and so on— each respectively stopping only when no new information is received.

4 Results

The "application server herd" was successfully implemented with asyncio and in terms of ease of writing and maintainability creating the server herd was straightforward as the asyncio library contains an API for network IO. Asyncio is based on the event loop and coroutines and functions only need to be labeled "async" and called with "await" within the event loop to be scheduled. Inter-server communication is also done with asyncio via start_server() and open_connection() and the programmer does not have to manage the sockets themselves. There are not much extra consideration or effort that the programmer has to take in implementing the server herd. Considering ease of implementation and code maintainability alone, Python and asyncio are definitely a good choice for writing a server herd.

Asycnio is also reliable— facing no problems in race conditions during the implementation— since coroutines run on a single thread and we allow the event loop to schedule the coroutines for us. In fact, since python (CPython) runs on a global interpreter lock—that is, only one thread is allowed to execute python bytecode at a time—it is ensured that there are no race conditions in reference counting. This combination of reliability and maintainability makes it easy for us to extend the program to other applications as necessary.

5 Python vs. Java

In this section we take a closer at Python, especially over the areas of type checking, memory management and multithreading and explore the tradeoffs of the language with respect to our application server herd. We also compare the python language features with those of Java in those previous listed areas.

5.1 Type Checking

Python verifies type safety at runtime (dynamically typed) and has strict rules on the type of variable expected in the type system (strongly typed). For example, python does not allow type coercion. Python uses the "duck typing" principle where the type of the object is less important than the properties and functions that may be used on the object. A method that can be used on an object can be used without consideration for what the type is exactly.

Java, on the other hand, is statically typed requiring variables to be declared before compiling. Java also has type coercion, implicitly converting data types as needed.

The dynamic typing of python allows programmers to write code without labeling types possibly reducing development time at the cost of some maintainability. There is an argument to be made that static typing has the better chance to catch errors as it checks for mistyping during compile time. A lack of type coercion reduces the chance for logical errors at the cost of having to be more rigid in programming practice, increasing development time. In the context of our large application server herd there is an arguably equal number of tradeoffs between the two typing systems.

5.2 Memory Management

In Python, memory is managed by the Python Memory Manager managing a private heap of Python objects and data structures. Management of memory on the heap is done by the memory manager without input from the user. Python uses a combination of reference counting—deallocating when the reference reaches zero—and periodic garbage collection.

Java, likewise, manages memory without input from the user. Both memory allocation and deallocation for the heap is handled by the Java Virtual Machine via the garbage collector.

For our purposes, both languages' automatic memory management means reduction of errors due to not having to manage memory personally in attempting to avoid memory leaks.

5.3 Multithreading

In Python or CPython (Used in the prototype program) there is a single global interpreter lock which allows only one thread to hold the lock and execute code at a time—this prevents true multithreading.

Java allows true multithreading and provides the user a threading API complete with classes like Thread and ThreadPoolExecutor to run multiple threads simultaneously.

Again, for our program, we would then lose out on the ability for parallelism and performance in simultaneous computation. However, we are more interested in having concurrency that allows multiple IO requests or network IO to time-share and run as needed since a majority of the IO requests' time will be spent waiting. Some of this waiting includes: waiting for connection, waiting to read a file, waiting to write to a file and waiting for the database. Since we can use asyncio for concurrency to allow IO requests to time-share and IO request waiting is the bottleneck of the concurrency problem, Python and asyncio serves our needs well. Even for larger applications, for our news service, the main issue for concurrency remains to be IO waiting time and not raw computation. There is no need for the multithreading provided by languages like Java.

6 Asyncio vs Node.js

Python's asyncio is based entirely on the event loop and coroutines scheduled on the event loop. We can define coroutines by marking functions with the keyword "async", wrapping the coroutine around a create_task() call, and awaiting the task within a top level entry point called by async.run(). The event loop will automatically manage the scheduling of the coroutine and has the ability to allow its execution to be suspended and resumed as needed—all without the input of the user. For Network IO we use start_server() to start a server on a certain host and port and open_connection() to connect to any started server. All of this, is likewise, scheduled and handled by the event loop.

Node.js uses a similar design handling concurrency with event driven asynchronous IO, also using an event loop to handle events with a single-threaded design. The major difference of Node.js is in its use of callbacks for concurrency and Network IO. Callbacks handle asynchronous code by being passed as an asynchronous operation's argument and only being executed when the operation is complete.

7 Asyncio 3.9 vs. Older Versions

Updating to asyncio 3.9 the only major change to the functions involving coroutines and Network IO is the change to asyncio.run() where it is updated to use loop.shutdown_default_executor(). This function will schedule the closing of the executor and wait for all the threads in the ThreadPoolExecutor. There are no major changes to the functions of interest that would impede the running of what is necessary in the "application server herd". It would seem that older versions of asyncio would be fine for use in development.

8 Recommendations and Conclusions

From the findings of building the "application server head" and the exploration of the various Python language features and asyncio library API it is recommended that Python and asyncio is used to write the Wikimedia-style news service.

It is found that the asyncio network IO API is simple and easy to use, only requiring the use of a few additional functions and the "async" and "await" keywords. This ease of use addresses the concerns of difficulty of building the server herd as well as concerns of maintaining the application.

The strongly-typed and dynamic type python, although may be more prone to error, causes an arguably equal tradeoff in that it is faster to develop without concern for necessity of type labeling. The automatic memory management further reduces development effort without need to concern with manual memory allocation and deallocation. Concern of Python's lack of true multithreading is irrelevant as what is necessary for our purposes is the ability to have IO requests time-share via concurrency which is provided by asyncio.

In addition, it is found that engineers may use older versions of asyncio without concern, with respect to the Network IO purposes of the server herd, since there are no updates throughout the versions from before 3.9 and onward that would cause our server herd implementation incompatibility errors.

References

[1] Python 3.11.2 documentation.

https://docs.python.org/3/

[2] Java Documentation.

https://docs.oracle.com/en/java/

[3] Wunder, Caudio. Asynchronous flow control.

https://nodejs.dev/en/learn/asynchronous-flow-control/