Event Driven Programming: asyncio

Graham Wing

Henry Samueli School of Applied Science and Engineering, UCLA

gewing@ucla.edu

Abstract

The following report is an investigation of the asyncio library and its application to modern problems. This text examines the basic concepts of asynchronous IO as well as the amenability of Python's asyncio library. As an example, a prototype and its use case are summarized. The report concludes with a comparison to a Java multithreaded approach and Node.js followed by a recommended use for asyncio.

1 Introduction

The current paradigm is that of distributed computing. By deploying massive herds of inexpensive computers, models like MapReduce can accel at performing large tasks. This horizontal scaling has taken the throne from vertical scaling. A single machine with powerful hardware is more subject to network bottlenecks and IO requests than a distributed system with many machines.

In keeping with this current paradigm, Python's asyncio library is a single threaded solution to the distributed, asynchronous environment. The following is an examination of the library, including a prototype and comparisons to alternatives.

2 Basics

The asyncio library provides functions for event driven, asynchronous programming. By abstracting away various IO methods behind asynchronous functions, the library allows for programs to perform IO without blocking. This prevents a program from being context switched, allowing it to utilize more CPU time for computations. Additionally, the library provides event loops and functions for interacting with event loops. These functions include asyncio.ensure_future(), which takes a coroutine (an asynchronous function) and schedules it for execution. When an asynchronous function yields control of the program (using await to wait for some future), the event loop schedules the execution of another function that has received/is awaiting the result of a future.

This powerful concept enables concurrency within a single thread. Similar in concept to a multithreaded program running on a single core (no true concurrent execution), asyncio is safer than multithreading due to its avoidance of data races. If preempted by the operating system mid-operation, when the program is resumed the same operation will be resumed. This is the benefit of using yields within a single thread to enable concurrency. The programmer decides when and where context switches can occur. asyncio avoids much of the headache involved with multithreading.

3 Ease Of Use

As is typical with Python, learning to use Asnycio is straight forward and well documented. Compared to multithreading, there are far fewer concerns when writing an asyncio program. Rather than thinking about whether to lock an object or if deadlock may occur, the programmer can concern themselves with writing code pertinent to the task at hand. For the most part, the programmer can write a conventional, event driven program. To then increase speed and efficient use of CPU time, blocking IO can be replaced with non-blocking calls that yield from the current function. By making use of await, asyncio.ensure_future(), and asyncio.gather(), functions can be parallelized in tiers. asyncio.ensure_future() creates tasks on the event loop and asyncio.gather() waits for a set of tasks to complete. Together, they allow functions to dispatch other functions and wait for their completion. Thus, asyncio makes it easy to write programs that listen for input and dispatch various concurrent functions to handle input.

However, this is not to say that asyncio is error free. It is possible for a programmer to get in trouble if they show little concern for the state of variables when yielding from a function. For instance, if correct result requires multiple operations to be performed on a set of variables in sequence,

yielding in the middle of the said operations could allow another asynchronous function to interact with said variables, leaving the program in an inconsistent state. The troubles of multithreading programming are lessened but can still be a problem for a careless programmer.

In general, Python and its library support are very well understood and maintained. The current popularity of Python has led to the creation of a plethora of tutorials and documents describing Python. This makes maintaining a Python program much simpler. Understanding a Python paradigm or function is simple due to the documentation and large community support. Additionally, the large number of functions provided by the Python's standard library prevent programmers from getting bogged down writing low-level code. This leads to fewer bugs in code and more reliable code.

Connecting with other programs can also be achieved with little trouble. To communicate with other programs, asyncio and a pipe or socket based approach would work well. asyncio provides abstractions for both, allowing for easy integration. Additionally, modules like sqlaclhemy allow for easy communication between Python programs and SQL based databases. Thus, a program written using asyncio would be relatively easy to maintain.

4 Prototype Program

To test the ease of use of this library, I wrote a prototype that deploys a server herd and interfaces with the Google places API. The prototype supports three forms of input: IAMAT statements indicating where a client is located, WHATSAT queries from a client asking for locations near their most recent lcoation, and AT statements for communicating between servers. To handle incoming connnections, I used asyncio.start_server(), which takes a function, ip address and port to listen on, and a loop to add tasks to (see Figure 1). When the server hears an incoming connection, the server routine function is called and added to the event loop. The event loop is capable of handling multiple incoming connections, all of which are run concurrently. When a client indicates their position, the server must propagate the client's location to its neighbour servers. The server routine acheives this by calling propagate_message(). As seen in the Figure 2, the function uses ensure_future() to add tasks to the event loop. These tasks then concurrently send messages to the appropriate servers. Because a task is added for each individual cant_stop_the_signal() call, communication with a slow server can be preempted in favor of a more responsive server.

If another server/client connects to the current server, connection with the current client is not lost. This is due to asyncio starting a new server routine for each incoming connection. Thus, multiple asynchronous inputs can be handled. However, because the server routine waits the coroutines it calls and said asynchronous calls make usage of asyncio.gather(), a tree like structure of runnable tasks forms. In this tree, parents wait for the completion of a set of children. Therefore, while a server is propagating data from a client, it will be nonresponsive to that client. Despite this downside, this does prevent the creation tasks that may step on each other. For instance, if a client indicates their

```
server_port = get_port_num(self.name)
routine = asyncio.start_server(self.server_routine, \
          '127.0.0.1', server_port, loop=self.loop)
self.server = self.loop.run_until_complete(routine)
try:
    self.loop.run_forever()
except KeyboardInterrupt:
    pass
```

Figure 1: Code Snippet. Starting server with function server_routine(). server listens for connections, on each connection a server_routine() coroutine is added to the event loop for execution.

```
for friend in talks_with(self.name):
    friend_port = get_port_num(friend)
    if friend not in up_stream:
        propagation_tasks.append(\
            asyncio.ensure_future(\
                self.cant_stop_the_signal(\
                    friend, friend_port, message_to_friends)))
await asyncio.gather(*propagation_tasks)
```

Figure 2: Code snippet from propagate_message(). Notice that once all the tasks have been added to the list, the routine waits for all of the tasks to complete in no particular order.

location has changed before the propagation of their old location has been completely propagated, neighboring servers may get information out of order. Due to timestamped locations, this would not be an issue, but it exemplifies the benefits of the structuring

asynchronous tasks such that children finish before parents are runnable.

4.1 Logging

To monitor the servers and examine the ordering of execution, every server logs to a file <server_name>.log. The logging scheme is as follows:

- Message propagated to available servers.
- Recipient: <neighbouring_server> Output: <AT_message>
- Connected To <neighbouring_server>
- Lost Connection To <neighbouring_server>
- To Client <ip_address_and_port_number>: <escaped_message>
- From Google: <json_returned_by_google_places>

As seen in Figure 3, the logging output indicates that a new connection can be made before the completion of propagating to other servers. This is due to the individual propagations being coroutines themselves. Not only does this allow propagation to occur concurrently, it also means that output does not inhibit input to a separate server routine from occurring.

Figure 3: Logging example from Goloman. Notice the order of execution, before it finishes propagating it receives a message and begins propagating said message.

5 Python Vs Java Approach

In general, the Python approach will not be able to obtain the speed that a comparable Java approach would manage. Python employs dynamic strong typing. A benefit of which is that when types do not agree, correct coercion will be made through python builtins or an error will be thrown. This makes Python programs easier to debug with respect to types. However, dynamic typing makes Python programs slower than an equivalent statically typed. Additionally, type errors (beyond syntactic) will not be caught until an exception is thrown at run time, slowing the debugging process. On the other hand, Java is statically strong typed. Therefore, Java will catch type errors at compile-time (to byte code that is), preventing bugs from going unseen.

A benefit of both Java and Python is their garbage collecting memory management. Though specifics can vary between implementations, both Java and Python reduce chance of error by removing the programmer from memory management. Either option reduces the chance of memory leaks or dangling pointers by abstracting such problems away from the user.

Both languages are interpreted to an extent. Java's employment of running byte code on a virtual machine with JIT compilation when applicable make it faster than Python's straight forward interpretation. However, both can be completely compiled to machine code if desired. A downside of the asyncio approach is its limitation to a single thread. While this is a benefit for simplifying code, this reduces the possible speed of the program. The Java multithread approach would utilize more of the available CPU cores and be able to truly parallelize requests. This makes the ceiling of performance much higher for the Java approach. Of course, one must consider the downside of the Java approach: race conditions. This added complexity will make coding, debugging, and maintanence more difficult. Coordination within an event loop requires less complexity than coordination between threads.

The server herd use case described indicates the precense of many servers, so the load borne by each server is ideally minimal. This allows the asyncio approach to scale well for a growing number of inexpensive servers. Within a single machine, asyncio would not scale well to the number of cores available. In general, asyncio is not thread safe and is meant to be used within a single thread.

For an environment employing a smaller number of very powerful servers, Java's approach would provide better performance. In an environment utilizing a large cluster of inexpensive machines, Python's asyncio method is very simple to implement and performs very well.

6 Comparison to Node.js

Node.js and Python's asyncio employ very similar concepts. Node.js runs an event loop and handles concurrency using callbacks. Similarly, Python's event loop employs futures and tasks to achieve the same end. The basic concept to both is to use a single thread and schedule functions concurrently with that thread. Like asyncio, Node.js is not designed for a multithreaded environment. Instead the ideal deployment is using many lightweight servers rather than heavy processes on a single server. It is up to programmer preference as to which package is more desirable.

7 Conclusion

Python’s asnycio library is easy to learn and simple to implement programs in. Its use of event loops mitigates the problems of concurrency found in multithreading while achieving high performance by avoiding blocking. Its recommended use case is in a horizontally scaled environment in which the number of cores on a single machine is small. For a vertically scaled environment with few servers and a large number of cores per server, a multithreaded approach is recommended over asyncio. The modern paradigm makes asyncio very appealing.

8 References

- *Node.js Documentation,* https://nodejs.org/en/docs/
- *asyncio source code,* https://github.com/python/cpython/tree/3.6/Lib/asyncio/
- *Python3 Documentation,* https://docs.python.org/3/
- Webber, Adam Brooks. Modern Programming Languages: *a Practical Introduction*. Franklin, Beedle & Associates, 2011