Available online www.jsaer.com

Journal of Scientific and Engineering Research, 2021, 8(1):225-229

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Highly Performant Python Services using gRPC and AsyncIO

Nilesh Jagnik

Mountain View, USA
nileshjagnik@gmail.com

Abstract Asynchronous programming of applications has several performance benefits. This is due to making optimal use of compute resources leading to scalable and efficient systems. In this paper we discuss the use of Python's AsyncIO library along with Google's gRPC framework to build a highly performant RPC server.

Keywords concurrency, parallelism, multithreading, multi-processing, performance, scalability

1. Introduction

Python is a popular choice for building services especially for applications in Data Science and Machine Learning. This is due to many libraries and frameworks for supporting these being available in Python. But there are scalability concerns when a traditional Python server is used to serve a high throughput of traffic. One way to address these concerns is the use of an asynchronous programming framework. AsyncIO was first released in Python 3.4, providing an asynchronous programming framework for Python. Starting with gRPC version 1.32, AsyncIO can be used along with gRPC framework for easily building highly performant and scalable services. In this paper, we discuss the workings and benefits of AsyncIO and gRPC, and how to build scalable and performant servers using these frameworks.

2. Optimizing Resource Utilization

A. I/O Bound vs CPU Bound Workloads

An I/O bound workload is one which has a lot of dependencies on external systems. This means an I/O bound application has to wait for Output from/Input to external sources. Examples of I/O are reading a file from disk, making a Remote Procedural Call (RPC) to an external service, database lookups, etc. A typical I/O bound application may fetch data from an external source, apply lightweight transformations and then send the result back to another external source.

In comparison to I/O bound workloads, CPU bound workloads have very minimal I/O. These types of workloads require make continuous use of CPU resources.

B. Threads and Processes

Processes are independently run instances of a program. A process has a dedicated memory space. The Operating System (OS) manages processes by allocating a memory space which contains program text and data. Threads are executable units within processes that share its memory space. The OS manages thread execution by allocating resource to track its execution state (program counter, stack, registers, etc.)

C. Blocking I/O

Synchronous programming involves writing code that blocks OS threads while waiting for I/O, referred to as blocking I/O. During this time the OS thread is being occupied but no work is done inside it, leading to wasted CPU cycles and low resource efficiency.

D. Multithreading

One way to solve this issue is by creating multiple OS threads. The OS executes these threads concurrently on CPU core(s). When blocking I/O is detected on a thread, it is suspended and the OS switches it out and schedules another thread that is unblocked. The problem however is that, as discussed previously, OS threads are relatively heavyweight because the OS has to allocate resources for creating and managing these threads. So there is effectively an upper limit on how many threads can be managed. An I/O bound server application will eventually have all its threads blocked by I/O, and waste CPU resources all while being unable to accept more requests (due to all threads being exhausted).

E. Cooperative Multitasking

A better way to solve this problem is by using an asynchronous programming framework like AsyncIO. AsyncIO works by creating lightweight tasks inside a single thread. The framework requires developers to clearly mark statements in code which have I/O. The framework handles suspension of tasks when these statements are reached. Due to the code developer specifying when a task should be suspended, the management overhead of tasks is lower making them lightweight. Since these tasks are lightweight in comparison to threads, the overhead of managing these is much lower in comparison to multithreading. AsyncIO allows I/O bound applications to utilize CPU resources in the most efficient way, leading to highly scalable and performant services.

F. A case study of Multithreading vs Cooperative Multitasking

Let us say we have an RPC server that has a pool of 100 threads for serving requests. During normal execution, this server has an incoming QPS of 20 requests/sec. Each request normally takes 2 seconds to process. The server is able to easily handle this QPS. Now let us say that an external dependency of this RPC starts having some latency issues. Now each request takes 10 seconds to process instead of 2. Within the first second, 20 threads are immediately blocked for 10 seconds. This means we blocked 20% of the thread pool for the next 20 seconds. This number grows by 20% every second, until after 5 seconds, when all of the server threads have been blocked. The server can not accept new requests at this time. This problem could have been completely avoided with the use of an asynchronous framework, which doesn’t block threads. The scalability and performance of the server would be drastically improved.

G. Why Multithreading CPU Bound Workloads is Ineffective

In other languages like Java and C++, multithreading can be used to effectively to perform work in parallel. This is because the OS executes threads across the CPU cores, utilizing all cores effectively. However, Python has the Global Interpreter Lock (GIL) that prevents two threads from the same process to be executed together. GIL is necessary because Python’s memory management is not thread-safe. GIL is a mutex that prevents multiple threads from accessing Python objects. Due to the presence of GIL, multithreading for CPU bound workloads is not recommended. Note that GIL doesn’t restrict I/O workloads as much since multiple threads can still run concurrently (though not in parallel).

H. Multiprocessing

In Python, CPU bound applications benefit from the use of a technique called multiprocessing. This involves creating multiple processes (as opposed to threads in multithreading) that can perform work in parallel. In contrast to multithreading, multiprocessing creates new processes and schedules work on them. These processes can run across multiple CPU cores in parallel thus improving performance. This technique is even more beneficial in cases where work can be divided into smaller chunks and executed independently.

I. Limitation of Multiprocessing

Multiprocessing works great when multiple processes do not need to share objects in memory. For cases where sharing memory objects is required, multiprocessing is difficult to achieve. It is not recommended to use Python for CPU bound workloads that need memory sharing.

This limitation can be somewhat overcome by either writing CPU bound code libraries in C/C++ or using a package like NumPy which utilizes multi-threading but is written using C/C++.

3. AsyncIO

As discussed previously, AsyncIO uses cooperative multitasking. As a result, code should specify where I/O occurs.

A. Awaitables

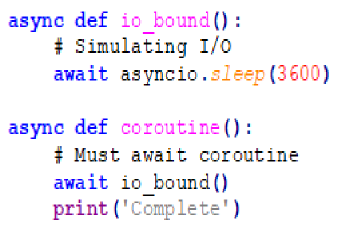
Awaitables are a category of objects that perform I/O and return. Awaitables are of three types – Coroutines, Tasks and Futures.

B. Awaiting

The await keyword is used to signal to the framework for suspending execution until the result of an awaitable is available.

C. Coroutines

The most common type of awaitables is Coroutines. Code which triggers I/O is written inside methods marked by the async qualifier. These methods are called coroutines.

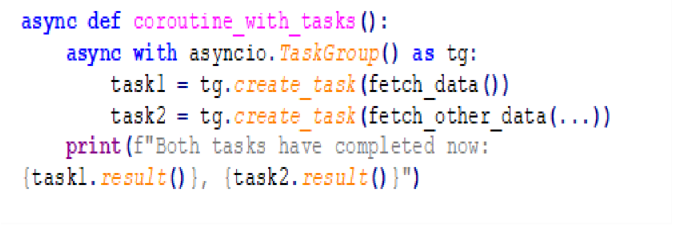


```
async def io_bound():
    # Simulating I/O
    await asyncio.sleep(3600)

async def coroutine():
    # Must await coroutine
    await io_bound()
    print('Complete')
```

D. Tasks and Task Groups

In addition to directly awaiting coroutines, one can fire off tasks without immediately awaiting on them. This allows executing work concurrently by firing off multiple tasks in a group and awaiting for all of them to finish.



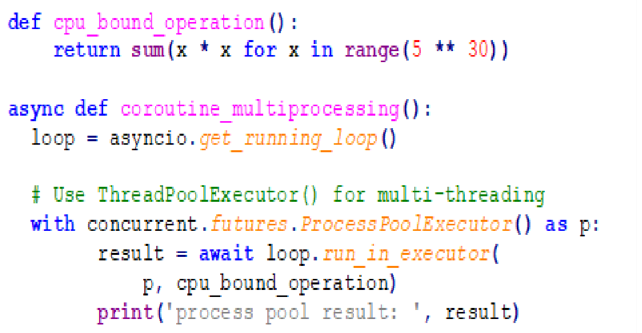
```
async def coroutine_with_tasks():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(fetch_data())
        task2 = tg.create_task(fetch_other_data(...))
    print(f"Both tasks have completed now:
{task1.result()}, {task2.result()}")
```

E. Event Loop

Event loop is the manager responsible for running asynchronous tasks. It exposes several low-level APIs that provide finer control over its behavior. It is generally not recommended to interact directly with the event loop except in cases where fine tuning its behavior is required.

F. Futures

In addition to Coroutines and Tasks, AsyncIO also provides an awaitable Future. The Future interface can be used for utilizing multithreading and multiprocessing in parts of code while the remaining uses AsyncIO.



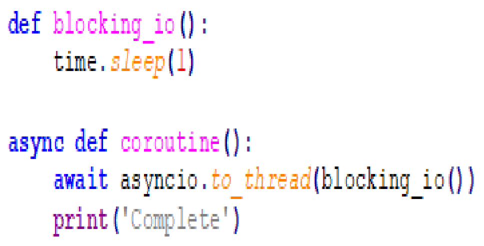
```
def cpu_bound_operation():
    return sum(x * x for x in range(5 ** 30))

async def coroutine_multiprocessing():
  loop = asyncio.get_running_loop()

  # Use ThreadPoolExecutor() for multi-threading
  with concurrent.futures.ProcessPoolExecutor() as p:
        result = await loop.run_in_executor(
            p, cpu_bound_operation)
        print('process pool result: ', result)
```

G. Blocking I/O in Coroutines

Blocking I/O in AsyncIO coroutines blocks the entire thread and the event loop. No other tasks can run while waiting for the I/O to finish. This could lead to serious performance bottlenecks and must be avoided. Blocking I/O can be scheduled on another thread to avoid blocking the event loop.



```
def blocking_io():
    time.sleep(1)

async def coroutine():
    await asyncio.to_thread(blocking_io())
    print('Complete')
```

4. gRPC Framework

gRPC is an open source Remote Procedural Call (RPC) framework. There are several benefits of using gRPC for client-server communications over plain HTTP.

A. Protocol Buffers

gRPC uses protocol buffers (also called protos) for both interface definition and client-server communication format.

Protocol buffers are a language-neutral, platform-neutral and extensible format for serializing data. This is similar to XML and JSON but with optimizations so that messages are encoded while on the wire for smaller footprints leading to faster transport. Protocol buffers provides auto generated classes for many supported languages and APIs to work with them. They are quite extensible and allow users to define messages of custom types.

B. Multi Language Support

gRPC servers and clients can be built in multiple languages including but not limited to Go, C++, Java and Python. This allows for flexibility in choosing the right language for different applications.

C. Proto based RPC Interface

The RPC interface is defined using protocol buffers. Autogenerated classes and interfaces make remote calls work similar to calling a local object. This leads to simpler code on client side.

D. Bi-directional Streaming Support

In many real-world applications, a lot of data is sent back and forth between client and server. Packaging this data as a single unit leads to high amount of memory buffering and processing delay since the receiving side needs to load all data before it can process it. In gRPC, clients and servers can send/receive data in smaller chunks. They can start processing these data chunks as they received without waiting for the entire message to arrive. This can improve end to end performance of systems. Note that streaming implementation in gRPC is much slower as compared to unary (non-streaming) RPCs. This performance can be improved by using AsyncIO streams.

E. Flow Control

In addition to streaming there is also support for flow control. This means that during streaming one side can signal to the other that it isn't ready to process more data yet. This prompts the sending side to stop sending data. The sender side can then halt/suspend the task that is generating the data. This leads to smoother and more efficient data flow through systems. By default, flow control is handled by gRPC. Some languages allow changing the default behavior but this is not supported in Python yet.

F. AsyncIO Support

gRPC provides AsyncIO support enabling cooperative multitasking. Using AsyncIO implies that server threads do not need to be blocked by I/O. This significantly boosts server performance.

G. Continuous Performance Monitoring

Monitoring metrics are vital for visibility into the state and performance of a system. gRPC provides an OpenTelemetry plugin which can be used to setup monitoring metric and send alerts when these metrics are low. These metrics can be used to create dashboards to display real time performance of systems. In addition to metrics, this plugin also collects traces which are sample logs of RPC interactions. These allow for troubleshooting issues in the system.

H. Load Balancing

Production systems comprise of multiple server replicas serving the same RPC. The requests from clients are distributed or load balanced across these servers. This prevents one server from having to serve an imbalanced number of requests leading to degraded performance. gRPC provides support for custom load balancing strategies.

I. Other Features

gRPC has several other critical features like retries, authentication, health checks, error handling, request cancellation, request hedging etc. These features make gRPC a good choice for RPC servers and clients.

5. Conclusion

Knowing about the type of workloads that a server/application has and techniques that can be used for optimizing them is essential for building scalable services. Using frameworks that allow easy access to these techniques makes it easier to build scalable systems. In addition to using the right tool for the job, visibility into performance metrics are vital in understanding the behavior of services.

References

[1]. Lei Mao, “Multiprocessing VS Threading VS AsyncIO in Python (Nov 2020),” https://leimao.github.io/blog/Python-Concurrency-High-Level
[2]. “GlobalInterpreterLock (Dec 2020),” https://wiki.python.org/moin/GlobalInterpreterLock
[3]. Aleks, “Programs, Processes, and Threads — What Are They, and How Do Computers Run Them? (Nov 2020),” https://medium.com/@al.eks/programs-processes-and-threads-what-are-they-and-how-do-computers-run-them-68f56ce023aa
[4]. Brad Solomon, “Async IO in Python: A Complete Walkthrough (Jan 2019),” https://realpython.com/async-io-python/
[5]. “asyncio — Asynchronous I/O (Oct 2020),” https://docs.python.org/3.9/library/asyncio.html
[6]. “gRPC AsyncIO API (Sep 2020),” https://grpc.github.io/grpc/python/grpc_asyncio.html
[7]. “gRPC | Guides (Sep 2020),” https://grpc.io/docs/guides/