CS131 Report

Abstract

In this paper, we will be looking into Python's asyncio asynchronous networking library and it's potential as a framework for application server herds and interserver communication. We will compare it to Node.js and Java-based approaches.

1. Introduction

Concurrency and parallelism are two key concepts in computing that deal with handling multiple tasks at once. For example with web servers concurrency allows the server to process multiple requests through context switching between them, while parallelism would involve processing requests on multiple cores at the exact same time.

In modern applications, concurrency models allow system resources to be used effectively which can improve performance, responsiveness, and reduce latency in applications like web servers and databases.

We will now introduce two concurrency models, Python's asyncio library and Java multithreading.

The asyncio library provides a set of high-level APIs to run Python coroutines concurrently and have full control over their execution as well as perform network IO and synchronize concurrent code.

Java on the other hand uses threads for concurrency. The Thread class allows multiple tasks to be run in parallel and enables maximum utilization of the CPU.

2. Overview of Each Model

The asyncio library is built around the event loop. Instead of using multiple threads, it uses a single-threaded event loop that manages multiple tasks by rapidly switching between them. Tasks are written as coroutines using the async/await tag that yield control when waiting for I/O.
Java's concurrency model is based on multiple threads running in parallel. Each thread can be scheduled on a separate CPU core which is beneficial for CPU-bound tasks.

3. Key Differences & Comparison

The execution model for asyncio is cooperative multitasking, which is where tasks/coroutines yield control with await so the event loop can run other tasks. This can be used for network I/O or interserver communication when waiting for a response.

The execution model for Java is preemptive threading. In this model, threads are able to be interrupted by the operating system in order for CPU time to be given to a different runnable thread to execute.

Each model is more suitable in scenarios. The asyncio library is suitable for I/O-bound tasks as stated earlier because it can minimize waiting time while ensuring work can still be done. Java, however, is suitable for CPU-bound tasks because it can take advantage of multiple cores to do work in parallel.

Let's compare the ease of use for both of these models.

Using asyncio can be done with just the await/async tags and calling asycn.run() on an async method. It is easy to write code with and not much thought needs to go into what methods should and should not be tagged with async. Debugging is made simpler because of this, but we do need to be careful to make sure we don't await in the wrong places and make tasks wait far longer than they need to.

With Java multithreading, we need to use low level parallelism primitives such as locks and semaphores. This allows for more precise control over CPU utilization between threads, but can easily become buggy due to unforeseen race conditions and things such as deadlocks which need to be look out for.

In regards to the scalability of the two models, asyncio can scale to a large number of tasks because there is not alot of overhead of yielding control. For Java the scalability of threads is limited by the number of cores and the fact that context switching has a lot of overhead. This means that more threads could be detrimental to the performance of the multithreaded application.

4. Pros & Cons of Each Approach

There are pros and cons to both models that should be discussed.

For asyncio, it has the pros of being able ot handle multiple I/O-bound tasks concurrently with minimal overhead, deal with events that require waiting without stopping the entire program, and simple concurrency due to only using a single thread. However, it has the cons of being weak at CPU-bound processing which could stop an event loop due to context switching, and needing other asynchronous libraries for tasks like HTTP networking.

For Java multithreading, it has the pros of true parallelism which is good for CPU-bound tasks and parallel processing as well as extensive support in java.utils.concurrent. But it's cons are high overhead from context switching, the risk of data races and deadlocks, as well as the need to use low-level primitives which can cause the second con to occur.

5. Use Cases, Recommendations, & Problems

As said earlier, asyncio is strongest when being used for tasks where waiting for an event like I/O or a network response is common, which means it is best for network and web-servers, database connection libraries, distributed task queues, etc. For Java multithreading, its ability to do parallel processing makes it more suited for computation-heavy applications and data processing such as simulations, machine learning and linear algebra.

6. Demonstration of Prototype

To demonstrate the potential of asyncio for interserver communication, I created a simple prototype of an application server herd using the Google Places API. The implementation has five servers with bi-directional communication and each server being able to only talk to certain ones directly. The servers are able to accept TCP connections from clients that emulate mobile devices which informs the server of their current location. This information would then be propagated to all other servers through a flooding algorithm so all of them have the same information. There is also the functionality for the client to request information about places near them in a defined radius.

I believe the prototype is a good demonstration of the potential of using asyncio. The await tags made it easy to structure the code in a way that minimizes server wait times and respond as fast as possible. Because it is single-threaded, I also don't have to worry about data races from using a simple flooding algorithm where a server can receive an update multiple times from different neighbors. In addition to this, it allows me to accurately do events such as informing all other servers when a specific server opens or drops a connection. There is no worry about memory management too, because no data structures are allocated, TCP messages are sent as strings and upon testing there has been no issue with sending a long response to clients.

For example, here is a log from one of the servers, Bailey:

```
Starting server: Bailey
Bona has opened a connection. Hello Bona.
Campbell has opened a connection. Hello Campbell.
Clark has opened a connection. Who asked?.
Jaquez has opened a connection. Who asked?.
Bailey received ? from CLIENT
Bailey received IAMAT from leon.cs.ucla.edu
Bailey received AT from Campbell
Server closed
```

Bailey was the first server to open a connection, so it is able to see when ny others become available if such a functionality is needed.

Here is another log from the server Jaquez:

```
Starting server: Jaquez
Jaquez received AT from Bailey
Jaquez received AT from Campbell
Jaquez received WHATSAT from nahidwin.cs.ucla.edu
Bailey has dropped their connection. GET OU-
Bona has dropped their connection. GET OU-
Campbell has dropped their connection. Bye Campbell.
Clark has dropped their connection. Bye Clark.
Server closed
```

As you can see, the library makes it easy to send simple TCP messages and log occurrences without having to worry about context switching messing up the timing of events and still being able to have good concurrency.

The asyncio library works very similar to Node.js, but in the case of Node.js, you need to manually implement sleep. This means if the waiting takes slightly longer than normal you could run into errors.

Because of this, I strongly recommend using asyncio for application server herds.

7. Conclusion

In conclusion, Python's asyncio provides a lightweight, single-threaded event loop model ideal for I/O-bound tasks and efficient interserver communication in application server herds. In contrast, Java multithreading enables true parallelism on multiple cores for CPU-bound tasks but comes with higher overhead and complexity due to synchronization and context switching. The prototype demonstrates that asyncio simplifies TCP message handling and logging without worrying about multithreading pitfalls, making it a strong recommendation for modern server applications.

8. References

asyncio Reference Page
https://docs.python.org/3/library/asyncio.html

Lynn Kwong. Asyncio in Python: A Guide
https://builtin.com/data-science/asyncio

Silvia Crafa. The role of concurrency in an evolutionary view of programming abstractions
https://www.sciencedirect.com/science/article/pii/S2352220815000784

Multithreading in Java
https://www.geeksforgeeks.org/multithreading-in-java/

Oracle Concurrency Documentation
https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html