Evaluating Python's asyncio for Herd Server Architecture

Harry Qian
606003725

1 Introduction

This report investigates the pros and cons of using Python's asyncio library to implement server herds in an attempt to update our company's service architecture.

In our proposed model, several peripheral servers will handle client requests and frequently updated data, like GPS information from cell phones. These servers will talk to each other to quickly and efficiently update each other's records without the need to update a central database server. Clients should be able to query any server and get updated responses.

2 Overview of asyncio

`asyncio` is introduced in Python 3. It's a library for using a event-driven, coroutine-based approach to handle I/O bound applications. It's widely used in network I/O contexts. In this report, we will evaluate asyncio's availability for our project.

3 Comparison to Java

3.1 Type Checking:

Both Python and Java are Object-Oriented languages. However, their philosophy on typing is different.

Python is dynamically typed, which means type checking is done at runtime. In contrast, Java is statically typed, and type checking is performed at compile-time. Static typing can catch type-related errors earlier in the development cycle, but it can also be more verbose and rigid. Python's dynamic typing allows for more flexibility and rapid prototyping.

Python's type checking is performed by saving a "type" field in the heap objects themselves. Additionally, Python uses "duck typing," where "type checking" is performed by seeing if the object has an implementation for a named method or field.

Additionally, Python supports multiple inheritance, which means that a class can inherit attributes and methods from multiple parent classes. In Java, we would need to resort to interfaces to approximate this effect.

Another of Python's strengths is that not all objects need to conform entirely to the definitions of a class. At runtime, we can still add attributes and methods to a Python object, or modify its methods to do something else. This could be handy in creating server objects packaged in our own types, and later redefining what their behavior should be upon responding to client requests. In contrast, such fine-grained behavior is less straightforward to write and think about in Java; we can code several options of what the server should do, but not functionality that is generated on the fly.

3.2 Memory Management:

Python and Java handle the stack similarly, with the small distinction in compilation, if Java determines an object is only used by the callee and provably doesn't "escape" to other parts of the program's execution, it will allocate the object on the stack instead of the heap for quick access.

However, CPython (the standard Python interpreter implementation) does not perform escape analysis as an optimization technique. Escape analysis is a compiler optimization that determines which objects can be allocated on the stack frame rather than the heap, based on their "escape" behavior (whether they escape the current scope or function). It's worth noting that if our team wants the feature, the code written on servers can use PyPy, a Just-In-Time compiler for Python, do employ escape analysis, so this lack of feature is not a deal breaker for Python.

Both Python and Java uses Garbage Collection at runtime to cleanup unused objects. In an effort to simplify the mechanisms of the language, Python tracks the reference count: record of the number of references to that object that currently exist. When a reference goes out of scope, Python will go decrement the object's reference count. If it goes to zero, that object is deallocated, and all its member references go out of scope. This can potentially trigger a cascade of deallocation.

Glued on to this reference-counting approach is a garbage

collector. This garbage collector periodically scans for and breaks cycles of objects that are no longer reachable, allowing them to be reclaimed.

Java's Garbage Collector is more sophisticated and uses a Copying Collector. In this scheme, newly created objects are put into a "nursery" region in memory. Periodically, only objects that are still in use are copied into a new region, whereas the other objects are deallocated by abandoning the nursery. This leads to favorable cache contents and faster behavior, since we don't need to visit the objects that are deallocated.

3.3 Concurrency:

Python has the Global Interpreter Lock (GIL), which prevents true parallelism within a single process. However, Python's asyncio library and other concurrency libraries like multiprocessing allow for concurrent execution by using a green thread approach: coroutines are created and setup, and although only one of them is executed by the event loop at a time, while one is waiting, other coroutines can run.

In comparison, Java has multithreading support and many synchronization primitives. Our herd servers' tasks will mainly be sending updates to other herd servers and querying other parts of our application (like asking Google Places in our demo). Since most of our application should be wait-time bound instead of CPU-bound, the event-loop approach should be quite nice.

4 Ease of Use:

asyncio provides a high-level and user-friendly API for writing concurrent code. The async/await syntax makes asynchronous code look and feel similar to synchronous code, reducing the cognitive overhead associated with callback-based approaches. This can make asyncio easier to use and maintain compared to other concurrency models.

Adding new servers to the model is quite easy as well. The only thing needing update is the adjacency lists in existing servers, and given our system, a central server could send updates to the herd servers to announce the arrival of a new "friend." As written, the servers already get access to any incoming requests. They just need to recognize the new ports as from another herd server.

5 Performance:

asyncio is designed for high-performance I/O-bound applications. By using an event loop and cooperative multitasking, it can handle a large number of concurrent connections with relatively low overhead. However, for CPU-bound tasks, Python's GIL can limit performance, and other concurrency models like multiprocessing might be more appropriate.

6 Comparison to a Java Approach:

In Java, we could set up a main thread to manage a thread pool. Upon receiving a client request, the main thread can assign a vacant thread to handle the client request, update the hash table, and talk to friends. This would work pretty well, but if we set up locks on the hash table (or parts of the hash table) then the multithreaded application becomes wait-time bound again, offering no performance gains over the asyncio approach.

Java also has asynchronous and event-driven libraries, but Python's async-await syntax is very easy to write and comprehend. `open-connection()` and `start-server()` offer solutions to socket-IO in one line, for example.

One advantage of Java and a multithreaded approach may become apparent if we scale up to our company's production servers. If there are large numbers of clients, we may need to remove old records from our hash tables or handle them with other data structures that need periodic large-scale updates. In Java, we can use a dedicated thread to check and remove records in the hash table that are older than a specified timestamp. This will be fast, whereas removing old entries in Python will block the usage of the hash table for a while, leading to infrequent spikes in response time.

7 Comparison to Node.js

Both asyncio and Node.js use an event loop to handle concurrent operations. However, Node.js is built on top of the V8 JavaScript engine and uses Google's libuv library for its event loop implementation, while asyncio is built into Python's standard library.

Node.js traditionally used callback-based asynchronous programming, which can lead to callback hell and reduced code readability. Modern Node.js versions support async await syntax, similar to Python's asyncio, which can improve code clarity and maintainability.

Node.js has a large and active ecosystem, with numerous third-party libraries and tools available. Python's asyncio is part of the standard library and has a growing community, but the Node.js ecosystem is generally more mature and extensive.

net.connect() and net.createServer() in Node.js are similar in functionality to asyncio.open-connection() and asyncio.start-server() in Python's asyncio library. They establish network connections and create servers for handling incoming connections, respectively.

create-server() calls start-server(), which establishes a new socket and listens on it. create-connection() calls start-connection(), which connects to a (host, port) address and returns a socket object.

It's worth noting that Node.js also provides higher-level frameworks and libraries for building network applications, such as Express.js for web servers and Socket.IO for real-time

communication, which can simplify the process of creating servers and handling network connections.

JavaScript's OO design uses a prototype-based approach, which is even more dynamic and loose than Python's.

8 New Features in asyncio

Since Python 3.9, a number of features in asyncio have been added or updated. The introduction of Barriers in 3.11 as a synchronization primitive could help us manage updates to the hash table. In 3.12, the performance of writing to sockets in asyncio has been significantly improved, and current-task has been implemented in C for speedup as well (see citation). So our servers can get by with older versions of Python, but the newer versions are nice to have.

9 Problems Encountered

Checking that the messages aren't sent around in indefinite cycles proved to be tricky. In the end, I had each server "sign" their work with their server name. When considering who to send the message to, the server will skip over any friends that already signed the incoming request.

10 Recommendation for Using Python

Based on the project requirements and the analysis of Python's features, asyncio's approach to concurrency, and the similarities and differences with Node.js, using Python with the asyncio library is a recommended choice for implementing the herd servers.

Advantages of Using Python with asyncio:

- Simplified and readable asynchronous code with async/await syntax - Efficient handling of I/O-bound tasks like network communication and hash table updates - Task specific memory requirements (maintain one large hash table, connection objects that only have one reference) suitable for Python's memory management scheme.

Disadvantages and Considerations:

- Dynamic typing can lead to runtime errors if not properly handled - Potential memory management challenges large-scale, memory-intensive applications

While Python with asyncio offers significant advantages for this project, it is essential to consider performance requirements, scalability needs, and the development team's experience with the language and concurrency models. If true parallelism or deterministic memory management is a critical requirement, Java's thread-based concurrency model might be a better fit. However, for the given use case of implementing herd servers, Python with asyncio provides a compelling balance of ease of use, performance, and suitability for the task at hand.

11 Sources Cited

Python Documentation. What's New in Python 3.11 (2022).

Node.js. Node.js v20.11.1 Documentation (2024).

Wikipedia. ISO-6709 (2024).

ChatGPT 3.5. ChatGPT (2024).

Tall, Snarky Canadian. How the heck does async/await work in Python 3.5? (2016).