Application Server Herd with `asyncio`

Addison Lynch
904484099

Abstract

This paper explores a test application of an *application server herd* that asynchronously listens on five TCP ports, accepts data, relays the data, and responds to the client. To implement this, we have used the asyncio library of Python 3.6.

1 Python

1.1 Overview

There are both advantages and disadvantages to using Python for this project. Python, a high-level interpreted language, is strong, dynamic typed. It employs *duck typing* and does not check type constraints at compile time. It does, however, forbid certain undefined operations. There are two main implementations of Python, Cython and Jython, the former written in C and the latter in Java. The main difference between the two implementations (which are nearly syntactically-identical), is the way they manage memory. Jython uses the Java Memory Model (JMM), while Cython uses a specialized reference counting scheme. Given that this project's implementation is Cython, we explain Cython's memory management process.

1.2 Memory Management

Cython uses a combination of reference counting to detect garbage objects (which have no references) and a cycle detection algorithm to search for garbage cycles. Though the language contains built-in modules such as `gc` to handle garbage collection, most programs need not use it, as garbage collection is *automatic*. A private heap is used to hold each object and data structure of a program. This heap, which handles storage dynamically, is managed by the *Python Memory Manager*. The PMM has exclusive control over the heap, which cannot be controlled by the user, though certain high-level manipulation is allowed.

1.3 Asynchronous I/O

Here we arrive at the core of Python's usefulness in this project. Typical I/O operations (such as those in C/C++) are typically handled by servers on a thread-per-request basis. With a fixed thread pool, the application assigns each of these threads to a request as needed. This approach, however, is both limited and cumbersome in applications where many requests are received concurrently (such as our proxy for the Google Places API). With a limited number of threads in the thread pool, requests further down the call stack may be delayed if all currently-executing threads are blocked. Further, each thread requires a certain amount of often limited stack space (especially in C), placing further holds on the application when the memory is exhausted.

An improvement on this method for applications which conduct a high-level of concurrent I/O operations is coroutines. Coroutines are subroutines for non-preemptive multitasking. They allow multiple entry points for suspending and resuming execution of a function at certain locations. Essentially, coroutines are functions that can be *paused*. In the case of the Google Places Proxy, we implement a coroutine to await network transmission on an open TCP connection. While input is not received, the function pauses, or *yields* control to another coroutine and awaits transmission. The primitives for this method in Python were *Generators*, which were first introduced in Python 2.2 (PEP255). Generators introduced the `yield` command, which allowed a function to maintain its scope (a namespace of local variables) when returned. The primitive generators were purely one-directional, however, as it as not possible to pass values to a generator function.

Thus a feature was added in Python 2.5 (PEP342) which added the `send()` method, which allowed the calling scope to pass values to the generator. Then followed PEP380 of Python 3.3 which added the `yield from` command. `yield from` allows for refactoring generators, providing the ability to yield every value from an iterator and traverse the call stack.

Asynchronous I/O is a non-blocking form of I/O processing which allows processing to continue before a transmission is finished.

1.4 `asyncio`

`asyncio` is a module that ”provides infrastructure for writing single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resource, running network clients and servers, and other related primitives.” The library was introduced in Python 3.4 (PEP3156) and was born from the `yield from` command introduced in Python 3.3 (PEP380).

`asyncio` is the ideal Python module for implementation of the Wikimedia platform. To begin, it's simple and easily-implementable. As seen with the Application Server Herd abstract that has here been built, it's clear that the implementation is both stable and scalable.

2 Server Herd Implementation

2.1 Servers

The objective of the project is to have five concurrently-running servers named Alford, Ball, Hamilton, Holiday, and Welsh listening on five distinct TCP ports. To instantiate these servers, we first create an event loop with `asyncio.get_event_loop()`, which, when called with no arguments, returns an empty event loop. Next, `asyncio.AbstractEventLoop.create_server()` is a coroutine that will instantiate and return a `Server` object with attribute `sockets` which contains the any created sockets.

We pass `create_server()` a lambda-instantiated instance of our `ServerProtocol` class (see below) as well as a hostname (localhost in this case) and a port. To obtain the port for each server, we cache the desired port values in either a configuration file or in a dictionary atop the program (the latter in this case). Once the server is created, it will listen for socket connections on that port and handle them as we desire with the `ServerProtocol` callback methods.

2.2 Protocols

The server herd implements three network protocols (subclassed from `asyncio.Protocol`) to handle its transports.

1. `ServerProtocol` - base of the application and accepts/transmits data to and from the client.

2. `InterServerProtocol` - handles inter-server communications (propagation)

3. `GoogleProtocol` - handles http 1.1 requests between the server and the Google Places API

2.2.1 `ServerProtocol`

`ServerProtocol` is the primary protocol through which transmissions are sent and received between the server and the client. While subclassing `asyncio.BaseProtocol`, we provide implementations of each of its callback functions, notably `connection_made()` and `data_received()`. The server listens on its assigned port until certain events occur, such as the opening of a connection or the transmission of data. Connections are handled on a per-request basis, with callback functions returning control to the main event loop at the completion of their execution. The `ServerProtocol` reads transmissions from the client into a buffer, then sends this buffer to be tokenized, validated, and processed with `tokenize_request()`, `process_<COMMAND>()`, and `propagate_data()`.

While tokenizing the data, the protocol ensures that a proper command as well as properly-formatted ISO6709 location and POSIX time are provided. After determining the command that the client has passed, the protocol processes this request accordingly. In the case of a IAMAT command, the server updates the database accordingly with location, time, and skew information about the client. When doing so, it ensures that the request's timestamp is more recent than the current entry in the database, preventing untimely updates across the servers. It then propagates such data to the servers in its *friends* list as specified in a dictionary at the top of the program using `asyncio.AbstractEventLoop.create_connection()` and an instance of `InterServerProtocol`.

In the case of a WHATSAT command, the server queries the database for the latest location and time data from the client. If there exists no such data, the server returns an invalid response (led by `?`) as specified. Else, it uses the data to create a new TCP/SSL connection to the Google Places API, as specified with a new connection using the `GoogleProtocol`.

2.2.2 `InterServerProtocol`

2.2.3 `GoogleProtocol`

2.3 Propagation

2.4 HTTP Request from Google Places API

3 Operation & Testing

We see from the docs that `asyncio` currently implements transports for TCP, UDP, SSL, and subprocess pipes." Intuitively, we can see that two protocols need to be created. The first, a TCP protocol, will facilitate communications between each of the five servers. The other, an SSL protocol, will allow us to asynchronously obtain data from the Google Places API without the use of aiohttp or similar resources.

4 node.JS?

Node.js is a Javascript run-time environment that handles server-side operations with Javascript. Node is based on an event-driven framework which employs asynchronous I/O most commonly in a web environment. In serving dynamic web content and real-time applications, Node enables the program to conduct I/O operations outside of the main thread, preventing blocking and allowing the application to continue to serve its content. There has been much discussion since the advent of `asyncio` about the pros and cons of using Python versus Node.js, and it is clear that there are a number of distinctions between the two.

The clearest of these distinctions is performance, a category which Node.js dominates Python. Given that Cython is written *on top of* C, it must first compile to C before running, a process which deals a significant blow to performance. In a web environment, particuarly Further, the asynchronous architecture behind Node.js is built-in by default, whereas Python supports coroutines via the `asyncio` module. Error handling is frequently raised as an additional issue, with

5 Conclusion