Asynchronous I/O: A Case Study in Python

SALEIL BHAT

A library for performing "await"-style asynchronous socket I/O was written in Python. It provides an event loop, as well as a set of asynchronous functions to connect to a remote socket and read data from a socket. With this library, a user can write custom routines which utilize these asynchronous functions while maintaining a "synchronous style." Two versions of a test application, which reads and processes data from a large number of parallel connections, were written: one using this library, the other using threads. An attempt was made to compare the performance of the two versions as the number of connections increased. In the few-connections regime, both the threaded and asynchronous solutions performed similarly. However, both solutions failed in the many-connections regime. The application failed for more than 500 connections in the threaded case, and it failed for more than 300 connections in the asynchronous case. Further work is needed to investigate how to scale both solutions to a level in which the cost of threading will actually be non-negligible.

1 BACKGROUND

Web programming necessarily entails interacting with remote users. Given the time scales involved in such interactions (e.g. sending data across a transcontinental link or waiting for a user to type a command in their browser), the I/O routines of a web application will spend most of their time just waiting. However, a good web application should avoid blocking on these waiting periods when there is other work to be done. That is, while the I/O is not yet complete, the application should continue making progress on its local computations. The traditional method for doing so is to use threads; each I/O routine runs in its own thread, while yet another thread performs the backend computations. There are several drawbacks to this scheme, however. First, multi-threaded programming is widely regarded as complex and leaves a lot of room for programmer error. Common issues include race-conditions, dead-locks and resource starvation. Second, using threads delegates responsibility for switching between routines to the operating system. The CPU can switch between the threads at any (non-deterministic) point in time and the cost of such a context-switches is non-negligible [1].

Asynchronous programming is a programming paradigm which can be used to perform non-blocking I/O while avoiding some of the costs associated with threading. In asynchronous programming, instructions from the various application tasks (the threads from the previous model) are interleaved with one another in a single thread of execution. In an asynchronous program, the code itself explicitly manages when the execution thread switches from one task to another [2].

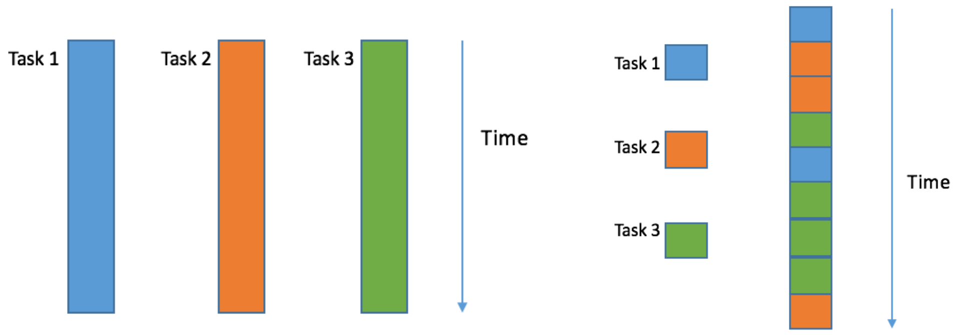

Fig. 1. The threaded programming model (left) vs. the asynchronous programming model (right)

One way to do of asynchronous programming is the event-callback technique. In callback-style programming, the programmer associates various events with a callback function such that when the event occurs, the thread of execution switches to executing that callback function. Callback programming is driven by an event loop. The event loop polls the various events which have been registered to it and when one such event is triggered, it executes the appropriate callback function.

Note that callback programming already solves many of the problems with using threads to do I/O. First, the execution "context" only switches on events, so the programmer knows that the switch will result in meaningful work. Second, since there is a single thread of execution, the developer does not have to worry about locking and race conditions; it is easy to reason about what the state of a given global variable will be just by walking through the control flow of the code. Unfortunately, callback programming has its own drawback: the colorfully-labeled "callback hell." Most applications require that the more than one thing occur on a given event (e.g. an attempted user login, the application needs to first authenticate the user and if that goes through then fetch their data). To implement such behavior, the programmer must use nested callbacks. The control flow of a program with deeply nested callbacks can be difficult to follow and such code is also difficult to modify. This concern is the primary motivation behind Promise/await style asynchronous programming.

A Promise (or a Future, in Python terminology) is an object which represents the result of an asynchronous call which may or may not have completed. A Promise stores a state (e.g. whether the call has resolved or not) and a value (the result of the call). It can also store a set of callback functions to execute when its state changes.
Asynchronous functions in this model return Promise objects rather than values. The benefit of this is that the user has more flexibility in expressing their control flow. In particular, a set actions which must be performed in sequence (which, in the callback style, would be expressed as a series of nested callbacks) can now be written in one block of "sequential" code [3].

Await-style asynchronous programming provides some syntax to make the common pattern "wait until this Promise resolves" into a one-liner: 'await Promise'.

In this project, I investigated how an asynchronous programming framework might be implemented. To this end, attempted to build a rudimentary asynchronous I/O library of my own in Python 3. Using Python was particularly illustrative because the introduction of native async/await to the language was quite incremental and clearly documented. Thus, to build my own library, I could simply "follow along" with the various Python Enhancement Proposals which laid the groundwork for each phase.

2 APPROACH

2.1 Project Scope

In investigating how thorough my implementation should be, I realized that building a full, general-purpose asynchronous programming framework would be beyond the scope of this project. Implementing a flexible event loop capable of taking and scheduling arbitrary functions is a very complex task; furthermore, doing so requires digging into platform-specific non-Python code as event loops are powered by system poll facilities. As such, I decided to limit my scope to an asynchronous socket I/O library. That is, the only asynchronous code my library can schedule and execute is that related to reading from and writing to sockets. This decision allowed me to use the Python 'selector' module as my event loop engine. The selector module uses a system poll utility to monitor a set of registered file descriptors and execute code when a read and/or write event occurs at one of these files. This decision also meant that it made little sense to compare the performance of my own library with that of the native Python version; my library is not nearly as fully-featured so a comparison in the one narrow case where my library is useful is not exactly fair. As such, I decided to try experimentally verifying the original premise of the asynchronous paradigm; namely, that it would perform better than a threaded program in the I/O bound domain.

2.2 System Design

NOTE: most of the design insights contained herein were obtained from [4]

Asynchronous programming in Python is based on the concept of coroutines. When an asynchronous function reaches the part where it would normally block, it yields back to its caller. The job of the asynchronous framework is to ensure that when the event which allows that function continue occurs, the program steps back into the same function right where it left off. In Lua terminology, the async calls coroutine.yield() to yield its caller and an event triggers a coroutine.resume() back into that function.

Python technically does not have a coroutine construct; it uses generators in the place of coroutines. The pure form of a generator should not be able to function as a coroutine because it is one-directional; you can yield *from* a pure generator, but you cannot step back *to* a generator. Fortunately, Python generators are NOT pure generators; they support a 'send' function that lets you push a value back down into a generator and resume execution at that point. This means it is possible to use them as coroutines.

My library consists of the following components:

1) A selector-based event loop
2) An implementation of a Promise class
3) An asynchronous function for connecting to a remote host and port; it returns a Promise which resolves into a socket object which is connected to the remote address
4) An asynchronous function for reading data from a given socket; it returns a Promise which resolves into the data which was read
5) A Task class used to wrap user-defined functions (which must "yield from" the asynchronous I/O calls provided by the library) and register them with the event loop

2.2.1 Event Loop. The selector object is global to the library. The event loop repeats the following: "wait for an event registered to selector to fire, when it does, call its associated callback." In isolation, the event loop does nothing; other code must first register an event with the library's selector for the loop to be meaningful.

2.2.2 Promise Class. The Promise class is quite simple; it only has two states, ‘resolved’ or ‘not resolved.’ It also maintains a list of callbacks and has one method: resolve. The ‘resolve’ method sets the Promise’s result to a given value, then calls every function in the Promise’s list of callbacks.

```python
class Promise:
    def __init__(self):
        self.result = None
        self._callbacks = []

    def add_callback(self, fn):
        self._callbacks.append(fn)

    def resolve(self, result):
        self.result = result
        for fn in self._callbacks:
            fn(self)
```

Fig. 2. Source code for the Promise class

2.2.3 Aysnc I/O Functions. Both the async connect and the async read functions follow the same basic template. I have put annotated source code which explains the format that these functions follow. The basic idea is to create an empty Promise object, then define the callback function inside the async function. By doing so, the Promise object is always in the scope of the callback. Then, register the relevant socket to the event loop and yield to the caller. When the event occurs, the callback will execute. The key idea is that the callback should resolve the Promise object with the desired result; because of the closure, the Promise object which is modified is the same object which the outer async function had yielded. At this point, the async function is ready to resume execution at the point where it last yielded. It is the role of the Task class to ensure that this occurs.

```python
def async_function:
    """The generic template for an internal async function"""

    # 1. do some pre-processing here
    #    <insert code>

    # 2. make a Promise object
    p = Promise()

    # 3. define an 'on event' callback
    # note: thanks to closures, p is always in scope when the callback is run.
    #  so the callback DIRECTLY modifies the function's Promise object
    def my_callback():
        # 4. get the desired result from the socket
        #   <insert code here>

        # 5. RESOLVE the promise with the result obtained from step 4
        p.resolve(some_value)

    # 6. register the event trigger with the global SELECTOR object
    SELECTOR.register(socket, some_event, my_callback)

    # 7. yield the Promise to the caller
    data = yield p

    # 8. we will reach here when the caller resumes execution of this function;
    #  the caller will 'send' p.result into the function, so data will equal the
    #  desired result

    # unregister the event trigger (so we don't execute it multiple times)
    SELECTOR.unregister(sock.fileno())

    # 9. return data, which should be p.result
    return data
```

Fig. 3. An annotated template of what an internal async library function should look like. My implementations of read and connect both follow this template

2.2.4 Task Class. The Task class is the "driver" of an async function. By wrapping an async function in a Task object, we ensure that the program steps back into the function when it is ready to resume execution. The Task class has a method called 'step', which steps into the async function where it last yielded, passing in the result of a Promise as a parameter. 'step' will receive the Promise the async function yields. Then, it takes that Promise object and adds *itself* to that Promise's list of callbacks. Now, when the Promise resolves, 'step' will execute again; but all 'step' does is step back into the async function where it last yielded, with the variable now containing the *result* of the promise. This is exactly the desired behavior.

```
class Task:
    def __init__(self, coro):
        self.coro = coro
        p = Promise()
        p.resolve(None)
        self.step(p)

    def step(self, promise):
        try:
            next_promise = self.coro.send(promise.result)
        except StopIteration:
            return

        next_promise.add_callback(self.step)
```

Fig. 4. Source code for the task class

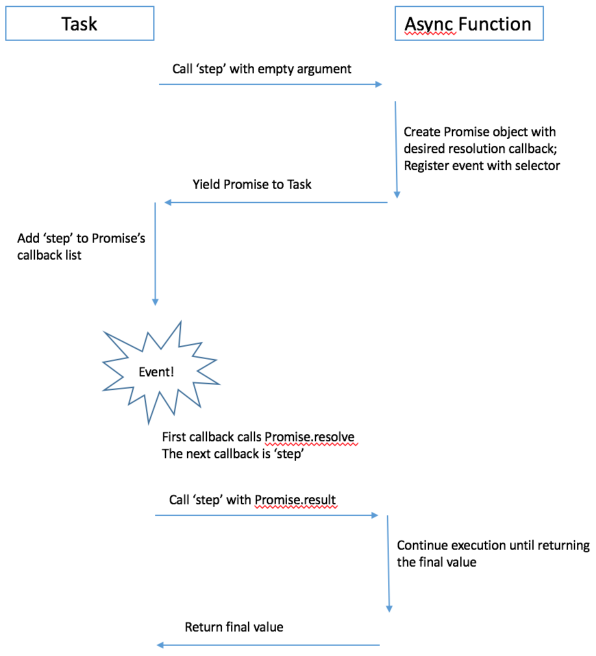
Task
Async Function
Call 'step' with empty argument
Create Promise object with desired resolution callback; Register event with selector
Yield Promise to Task
Add 'step' to Promise's callback list
Event!
First callback calls Promise.resolve
The next callback is 'step'
Call 'step' with Promise.result
Continue execution until returning the final value
Return final value

Fig. 5. An illustration of the control flow of a Task object driving an async function; note time increases as you move down the image

2.2.5. Nesting Coroutines/Await. The final component necessary to make this library useful is the ability to nest coroutines. The user should be able to define their own asynchronous function, which in turn makes calls to the internal library's async functions, and use their user-defined function as the driver of the event loop. The "yield from" feature of Python generators makes this possible. "yield from" allows a generator to yield a value from another nested generator to its caller; one can conceptualize it as making the intermediate generator "transparent." So, for example, one can define a new function which "yields from" the async socket read function described above and wrap this new function in a Task. The Task will drive the chain of coroutines exactly as it had done before for a single coroutine. A template for doing something like this is shown below. Note that if we replace substitute "await" for the phrase "yield from" in the code below, the code looks exactly like asynchronous code written in the async/await style.

```
def user_function():
    """A generic template for using this async IO library"""

    sock = yield from IOlibrary.connect(IP, PORT)
    data = yield from IOlibrary.read(sock)
    while data:
        # do some backend data processing here
        #   <insert code>
        data = yield from IOlibrary.read(sock)

    # do final backend procedures here
    #    <insert code>

# register as many instances of the user_function coroutine as you want to the event
  loop
for _ in range(N_CONNECTIONS):
    IOlibrary.Task(user_function())

# start the loop
IOlibrary.start_loop()
```

Fig. 6. An annotated template of what a program which uses this library might look like

3 RESULTS

To compare the performance of my asynchronous I/O library with a traditional threaded system, I implemented a simple test application. My application's "function" is to display a sorted list of the top 10 most frequently seen words thus far. The application opens N connections to a remote server(s) and waits for these servers to start sending strings. Each time it receives a new string, the application updates a hash table of words and their corresponding frequencies, updates its "top-10" list, then prints the current "top-10" list. I implemented two versions of this application, one with threads and the other with my asynchronous I/O library.

For the sake of minimizing variation, I ran a single threaded server on the same host as the application. Each serving thread read from a text file consisting of a series of (word, wait_time) pairs. For every (word, wait_time) pair, the thread sent the word over the socket to the application, then waited for wait_time seconds before moving on to the next pair. The contents of this text file remained constant across runs. Thus, the amount of time spent "idle" (waiting for an input to arrive) was the same for every run. This made the relative total runtime of the two applications a good measure of the cost of switching between threads/tasks. My hypothesis was that as the number of connections got larger, the runtime of the two versions would diverge (with the asynchronous version being relative faster than the threaded version).

Unfortunately, I was unable to observe that effect. For small numbers of connections, the performance of the two versions were virtually identical and both versions failed to scale to large numbers of connections. The threaded version failed at 500 connections while the asynchronous version failed at 300 connections. The threaded version fails with a "Connection Refused" error while the async version fails with an OSError. These (non)-results are shown in the table below:

Number of Connections	Threaded Runtime (s)	Async Runtime (s)	Relative Performance (ratio of run-times)
10	44.22	42.89	0.97699
100	45.01	44.82	0.9958
500	45.56	CRASHED	N/A

600	CRASHED	CRASHED	N/A

Fig. 7. Results from the time-trial of threaded and asynchronous test applications

Further work is needed to investigate why neither solution was able to scale. My hypothesis is that it has to do with number open file descriptors/active threads on the machine. Since both the server and the client were running on the same machine, it is more likely that the system ran out of these resources. For the threaded case, it is highly plausible that the system cannot handle more than a thousand Python threads. However, it is less clear what is going wrong in the async case. I attempted to test the thread theory by putting the server and application on two different machines in the myth cluster; however, it seems that TCP connections across myth machines are not allowed.

Though my quantitative results proved to be inconclusive, I do have some qualitative results. It took me several iterations to get the threaded version working properly; the threads needed acquire and release a lock since they were each reading and writing to the shared hash table. My first version of the code didn't account for all edge cases and I was getting unexpected results. For the asynchronous side, the code was very easy to write. Having already written the "business logic" of the app, all I needed to do was follow the template from Section 2 and insert the "business logic" functions where appropriate. Thus, my user experience does lend credence to the "ease of coding" claim.

REFERENCES

[1] Nick Humrich. 2016. Asynchronous Python: Await the Future. https://hackernoon.com/asynchronous-python-45df84b82434

[2] Rodrigo Fonseca. 2012. Introduction to Asynchronous Programming. http://cs.brown.edu/courses/cs168/s12/handouts/async.pdf

[3] Nicolas Bevacqua. 2016. Understanding JavaScript's async await. https://ponyfoo.com/articles/understanding-javascript-async-await

[4] A. Jesse Jiryu Davis and Guido van Rossum. 2016. 500 Lines or Less: A Web Crawler with asyncio Coroutines. http://www.aosabook.org/en/500L/a-web-crawler-with-asyncio-coroutines.html

[5] Jake Archibald. 2017. Javscript Promises: An Introduction. https://developers.google.com/web/fundamentals/primers/promises