gevent: Asynchronous I/O made easy

Daniel Pope — 2014-07-31 19:00 — 2 Comments

This blogpost is a write-up of a talk I gave at Europython 2014. This is my initial draft of the talk that was turned into slides for presentation.

The video is already on Youtube, so if you prefer you can watch:

> gevent: Asynchronous I/O made easy (https://www.youtube.com/watch?v=0wpYQr-_kqg) (44 minutes)

You can see the slides used for the presentation here (../../../presentations/gevent-talk/).

Introduction

gevent (http://www.gevent.org/) is a framework for scalable asynchronous I/O with a fully synchronous programming model.

Let's look at some examples of where we are going. Here's an echo server:

```
from gevent.server import StreamServer


def connection_handler(socket, address):
     for l in socket.makefile('r'):
          socket.sendall(l)


if __name__ == '__main__':
     server = StreamServer(('0.0.0.0', 8000), connection_handler)
     server.serve_forever()
```

In this example we make 100 web requests in parallel:

```
from gevent import monkey
monkey.patch_all()

import urllib2
from gevent.pool import Pool


def download(url):
     return urllib2.urlopen(url).read()


if __name__ == '__main__':
     urls = ['http://httpbin.org/get'] * 100
     pool = Pool(20)
     print pool.map(download, urls)
```

What is the strange monkey.patch_all() call? Not to worry, this isn't like your every day monkey patching. This is just a distribution of Python that happens to be shipped as a set of monkey patches.

This final example is a chat server:

```
import gevent
from gevent.queue import Queue
from gevent.server import StreamServer

users = {}  # mapping of username -> Queue


def broadcast(msg):
    msg += '\n'
    for v in users.values():
        v.put(msg)


def reader(username, f):
    for l in f:
        msg = '%s> %s' % (username, l.strip())
        broadcast(msg)


def writer(q, sock):
    while True:
        msg = q.get()
        sock.sendall(msg)


def read_name(f, sock):
    while True:
        sock.sendall('Please enter your name: ')
        name = f.readline().strip()
        if name:
            if name in users:
                sock.sendall('That username is already taken.\n')
            else:
                return name


def handle(sock, client_addr):
    f = sock.makefile()

    name = read_name(f, sock)

    broadcast('## %s joined from %s.' % (name, client_addr[0]))

    q = Queue()
    users[name] = q

    try:
        r = gevent.spawn(reader, name, f)
        w = gevent.spawn(writer, q, sock)
        gevent.joinall([r, w])
    finally:
        del(users[name])
        broadcast('## %s left the chat.' % name)


if __name__ == '__main__':
    import sys
    try:
        myip = sys.argv[1]
```

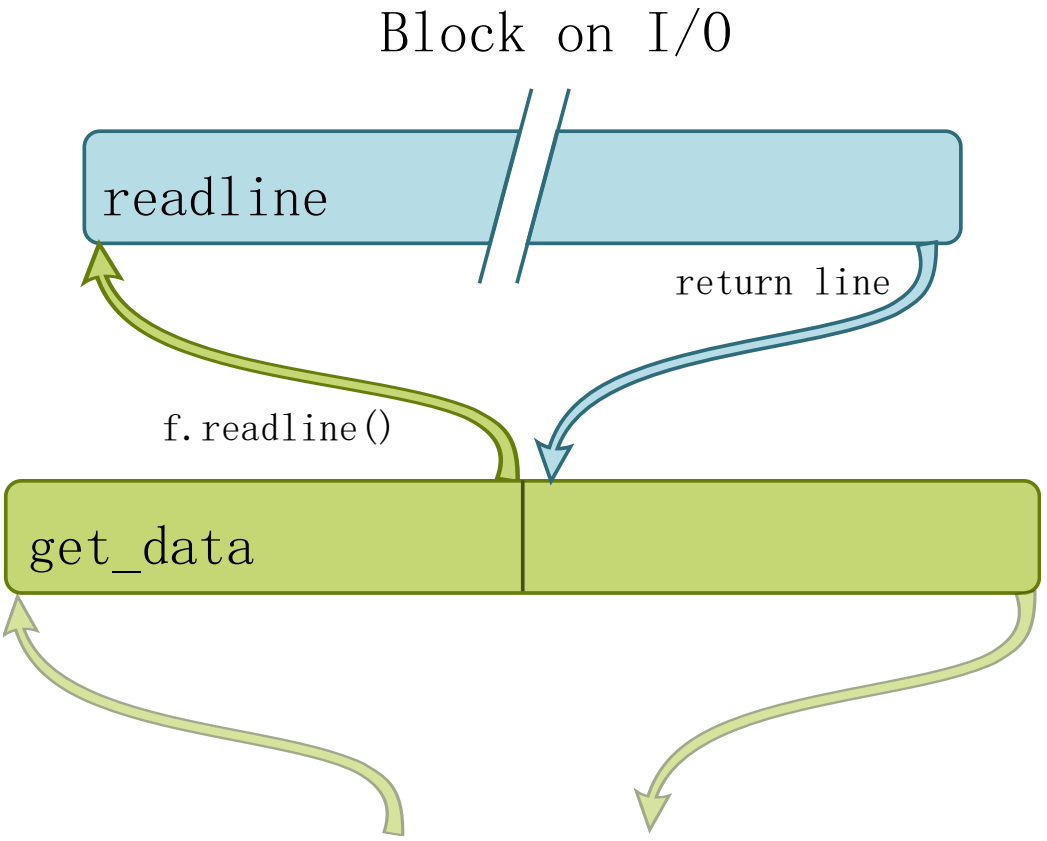
```
    except IndexError:
        myip = '0.0.0.0'

    print 'To join, telnet %s 8001' % myip
    s = StreamServer((myip, 8001), handle)
    s.serve_forever()
```

Simple enough? Let's look at why this gevent stuff is written like this.

Synchronous I/O

Synchronous I/O means that each I/O operation is allowed to block until it is complete.

To scale this to more than one user at a time, we need threads and processes. Each thread or process is allowed to individually block waiting on I/O. Because we have full concurrency, this blocking doesn't affect other operations; from the point of view of each thread/process the world stops until the operation is complete. The OS will resume the thread/process when the results are ready.



Drawbacks of threads: terrible performance. See Dave Beazley's notes on the GIL. Also high memory use. Threads in Linux allocate stack memory (see ulimit -s). This is not useful for Python - it will just cause you to run out of memory with relatively few threads.

Drawbacks of processes: No shared memory space. High memory use, both because of the stack allocation and copy-on-write. Threads in Linux are like a special kind of process; the kernel structures are more or less the same.

Asyncronous I/O

All asynchronous I/O falls back to the same pattern. It's not really about how the code executes but where the waiting is done. Multiple I/O activities need to unify their efforts to wait so that the waiting happens in a single place in the code. When an event occurs, the asynchronous system needs to resume the section of the code that was waiting for that event.

The problem then, is not how to do the waiting in a single place, but how to resume the one piece of code expecting to receive that event.

There are several different approaches to how to organise a single-threaded program so that all of the waiting can be done in a single place in code, ie. there are several different ideas about what resume() and waiter might be in the following event loop code:

```
read_waiters = {}
write_waiters = {}
timeout_waiters = []

def wait_for_read(fd, waiter):
    read_waiters[fd] = waiter

wait_for_write = write_waiters.__setitem__

def event_loop():
    while True:
        readfds = read_waiters.keys()
        writefds = write_waiters.keys()

        read, write, error = select.select(
            readfds,  # waiting for read
            writefds,  # waiting for write
            readfds + writefds,  # waiting for errors
        )

        for fd in read:
            resume(read_waiters.pop(fd))

        for fd in write:
            resume(write_waiters.pop(fd))

        # something about errors
```

We may want to add timeouts to the above code, in which case we might write something also include something like the following:

```
timeout_waiters = []

def wait_for_timeout(delay, waiter):
    when = time.time() + delay
    heapq.heappush(timeout_waiters, (when, waiter))

def event_loop():
    while True:
        now = time.time()
        read, write, error = select.select(
            rfds, wfds, efds,
            timeout_waiters[0][0] - time.time()
        )
        while timeout_waiters:
            if timeout_waiters[0][0] <= now:
                _, waiter = heapq.heappop(timeout_waiters)
                resume(waiter)
```

All asynchronous I/O frameworks are built on the same kind of model; they just take different approaches to how to structure your code so that it can be suspended when an I/O operation is requested and resumed when that operation is completed.

Callbacks

Examples:

- Javascript/Node
- Tornado IOStream
- Twisted Deferred
- asyncio, under the hood

One approach is to just call a callable whenever data is available to read. Typically we want to act on a higher level than chunks of data, so have a callback read and parse individual chunks of binary data and call an application callback when the parsed content is complete (such as a HTTP request or response).

What the user code looks like:

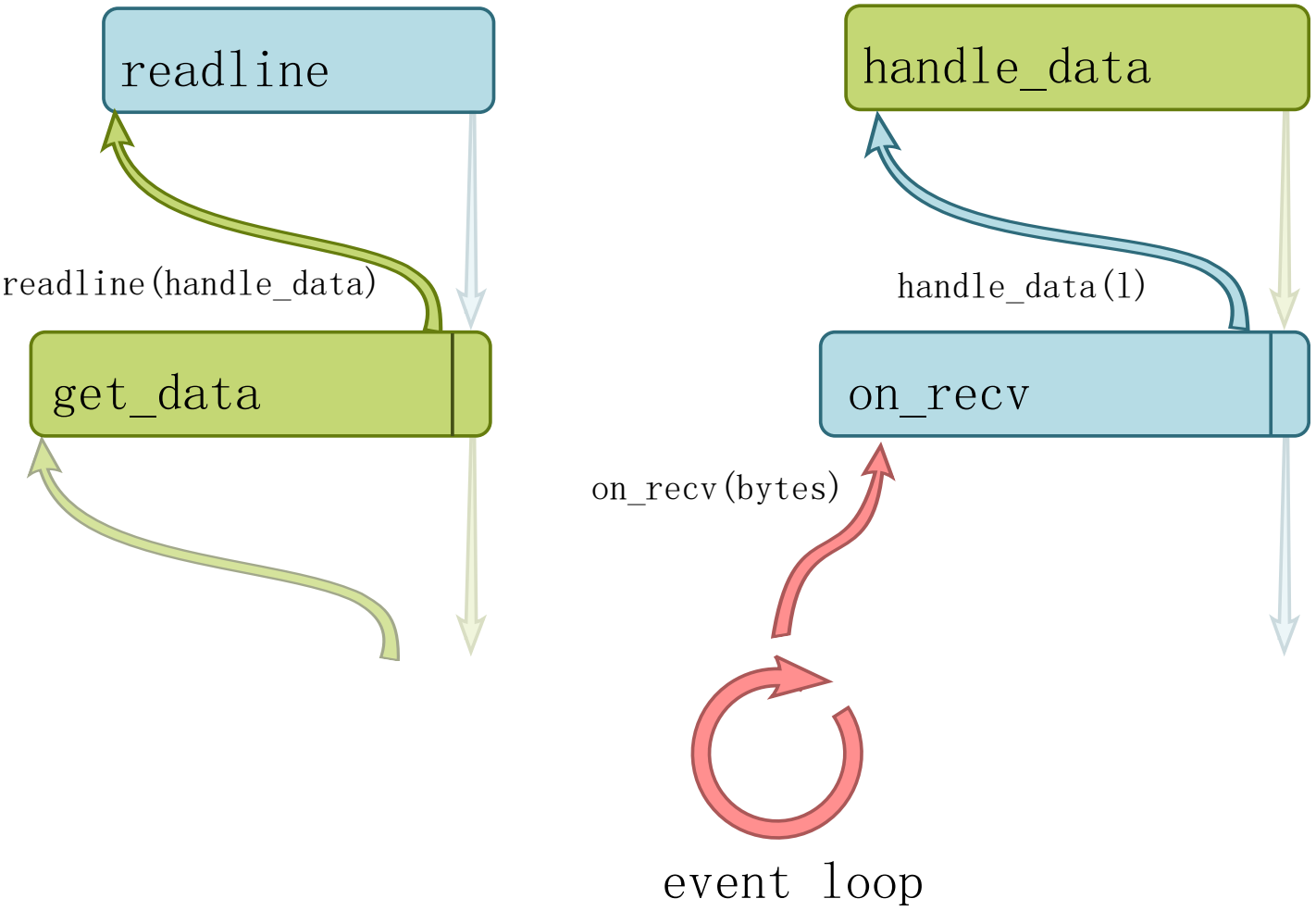
```
def start_beer_request():
    http.get('/api/beer', handle_response)

def handle_response(resp):
    beer = load_beer(resp.json)
    do_something(beer)
```

How can we tie the response back to a specific request? One answer is closures:

```
def get_fruit(beer_id, callback):
    def handle_response(resp):
        beer = load_beer(resp.json)
        callback(beer)

    http.get('/api/beer/%d' % beer_id, handle_response)
```

Either way is ugly, especially if we wanted to chain more I/O calls (anyone fancy this nesting any deeper?). There's no escape from the nesting and programming in pieces. As it has been said, "Callbacks are the new Goto".

The call stack looks like this:



Note how the return values never go anywhere useful; indeed the only way to pass results around is to write additional callbacks.

Method-based callbacks

Examples:

- Twisted Protocols
- Tornado RequestHandler
- asyncio Transports/Protocols

Callbacks are a mess! So we might organise them into interfaces where the methods are automatically registered as callbacks, so that users can subclass the interfaces to implement them. For example, this is asyncio example code:

```
import asyncio

class EchoClient(asyncio.Protocol):
    message = 'This is the message. It will be echoed.'

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('data sent: {}'.format(self.message))

    def data_received(self, data):
        print('data received: {}'.format(data.decode()))

    def connection_lost(self, exc):
        print('server closed the connection')
        asyncio.get_event_loop().stop()
```

This solution doesn't completely solve the problem of eliminating callbacks, it just gives you a neater framework that avoids callbacks being dotted all over the code. It places constraints on what callbacks can be called and where they can be defined. Suppose you want to link two protocols together - for example, you need to make an HTTP request to a backend REST service in the middle of handling a request from a user. You still get the problem of logic being fragmented over multiple callbacks, and you still can't use the return values from any asynchronous function.

Error handling in callbacks

When using a callback-based programming model, you have to register additional error handler callbacks.

Sadly not all frameworks enforce this. One reason is that it makes every single program twice as hard to read if you do enforce it. So it's optional, and consequently programmers don't always (even often) do it.

This violates PEP20 (http://legacy.python.org/dev/peps/pep-0020/):

> Errors should never pass silently.
> Unless explicitly silenced.

One of the bigger risks of not properly handling errors is that your internal state may get out of sync, deadlock waiting for an event that will never arrive, or holding resources indefinitely for a connection that has already disconnected.

Generator-based Coroutine

Examples:

- tornado.gen
- asyncio/Tulip

Generators have been used to implement coroutine-like functionality. This allows us to defer to some event loop system which will resume the after-the-callback section of our code after the I/O operation has completed:

```
import asyncio

@asyncio.coroutine
def compute(x, y):
    print("Compute %s + %s ..." % (x, y))
    yield from asyncio.sleep(1.0)
    return x + y

@asyncio.coroutine
def print_sum(x, y):
    result = yield from compute(x, y)
    print("%s + %s = %s" % (x, y, result))

loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()
```

(In this example we should note that we will want to spawn other async activities to get any scalability benefit out of this approach).

When generators were introduced with PEP255 they were described as providing coroutine-like functionality. PEP342 and PEP380 extended this with the ability to send exceptions to a generator and to yield from a subgenerator respectively.

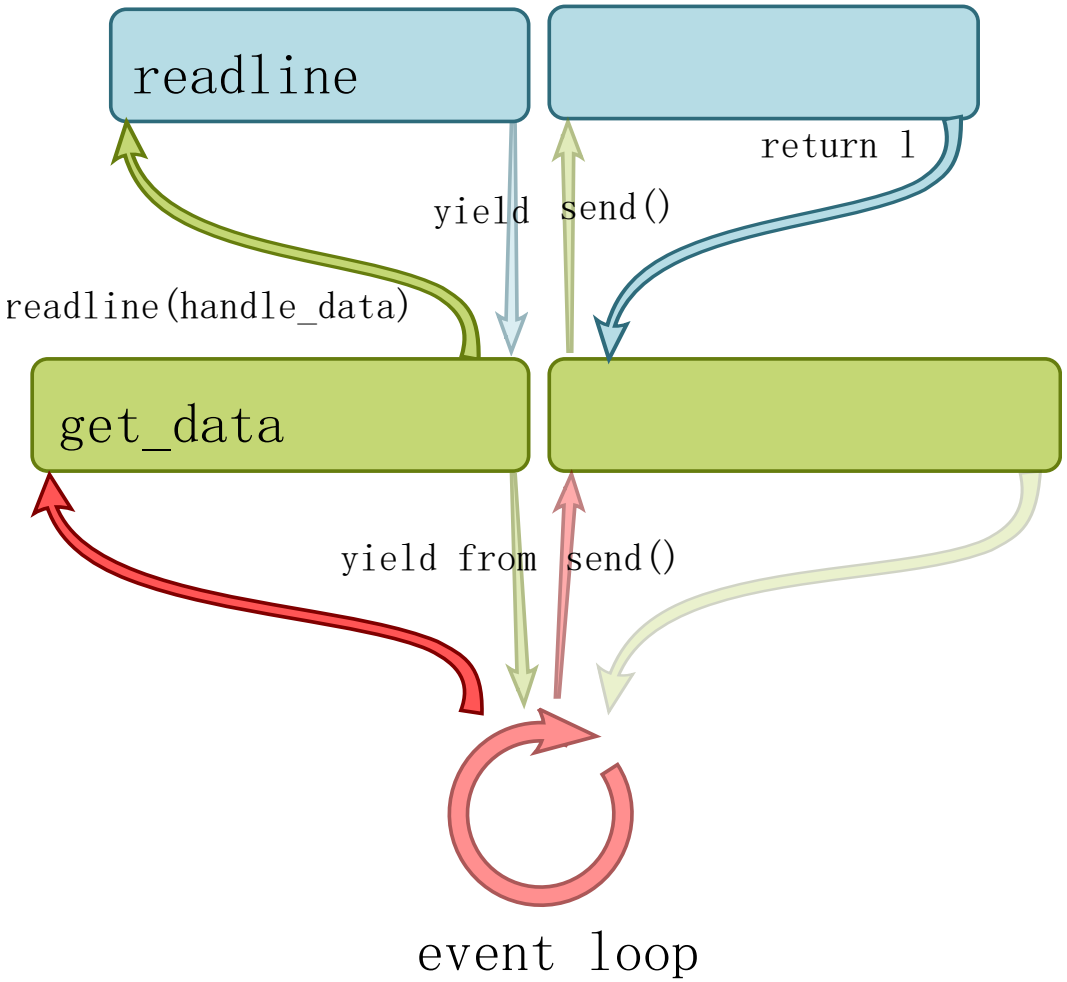
The term "coroutine" denotes a system where you have more than one routine - effectively, call stack - active at a time. Because each call stack is preserved, a routine can suspend its state and yield to a different, named coroutine. In some languages, there is a yield keyword of some form that invokes this behaviour (so different to the yield keyword in Python).

Why would you want to do this? This provides a primitive form of multitasking -- cooperative multitasking. Unlike threads, they are not preemptive, which means that they aren't interrupted (preempted) until they explicitly yield.

Generators are a subset of this behaviour, right down to the 'yield' terminology. Wikipedia calls them semicoroutines (http://en.wikipedia.org/wiki/Coroutine#cite_ref-Ralston2000_4-0). However there are two significant differences between generators and coroutines:

1. Generators can only yield to the calling frame.
2. Every frame in the stack needs to collaborate in yielding to the calling frame - so the top frame might yield, and all other calls in the stack need to be made with yield from.

The call stack will look as below:



Note that this use of yield means that you can't also use yield to write asynchronous generators. You have to return lists instead.

Greenlets/green threads

A greenlet is a full coroutine.

Examples:

- gevent
- greenlet
- Stackless Python

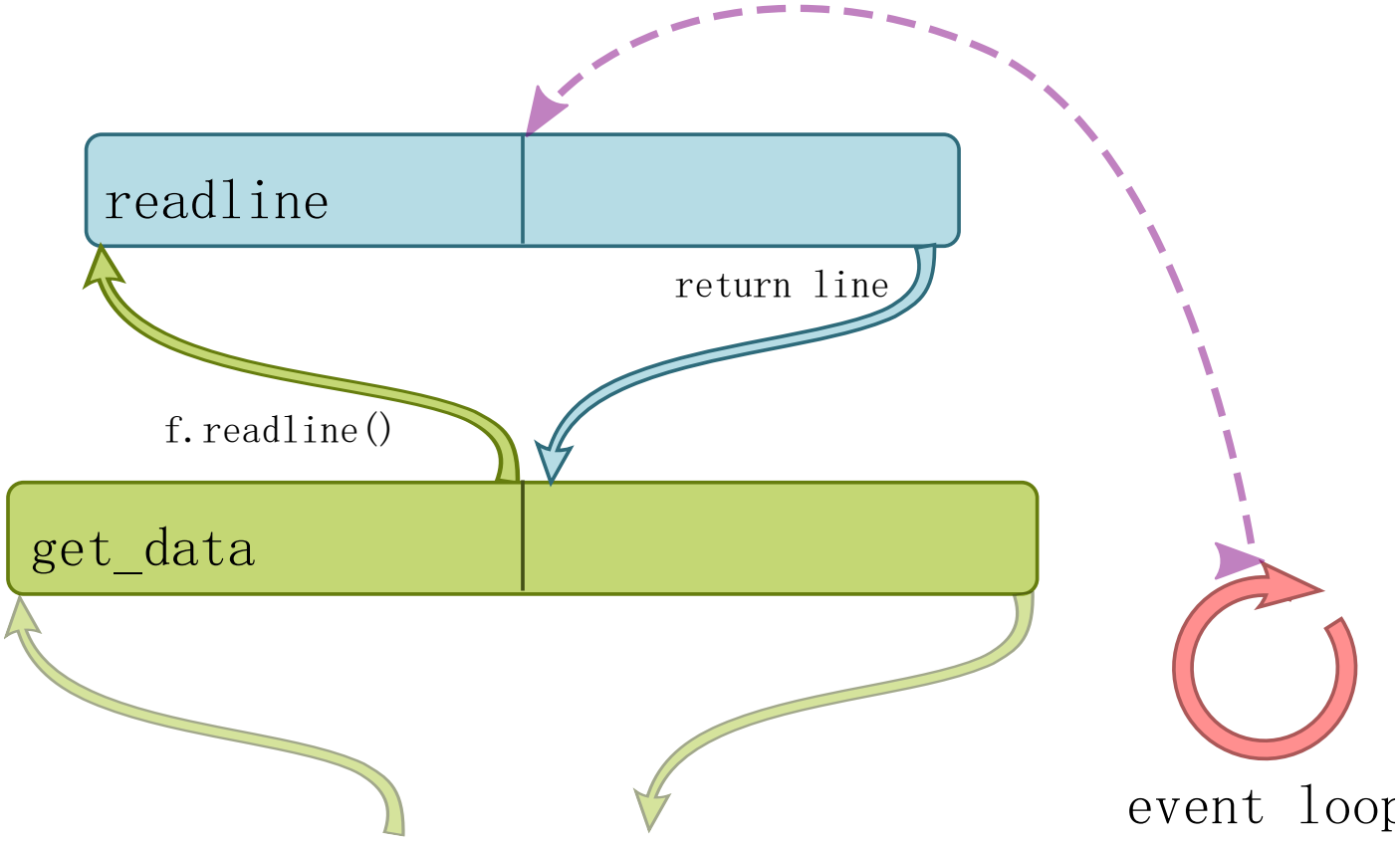
Let's rewrite that asyncio example with gevent:

```
import gevent

def compute(x, y):
     print "Compute %s + %s ..." % (x, y)
     gevent.sleep(1.0)
     return x + y

def print_sum(x, y):
     result = compute(x, y)
     print "%s + %s = %s" % (x, y, result)

print_sum(1, 2)
```

(Again, we'll want to start other greenlets to get any scalability benefit out of this approach).

That's much simpler! We can simply omit all the generator cruft because gevent.sleep() can yield to the event loop (called the hub in gevent) without the involvement of the calling frames. Also, because we have first class co-routines, the hub can be instantiated on demand; it doesn't need to be created as the explicit parent of the stack.

Coroutines give us a magic jump right from our current stack to the event loop - and back when ready:



The single piece of C-level magic allows us to write code that appears synchronous, but actually has all the benefits of asynchronous I/O.

Gevent: greenlets plus monkey-patching

Suppose though the sleep wasn't in our code? We can use gevent's monkey-patching to ensure that no code changes are necessary:

```
# These two lines have to be the first thing in your program, before
# anything else is imported
from gevent.monkey import patch_all
patch_all()

import time

def compute(x, y):
     print "Compute %s + %s ..." % (x, y)
     time.sleep(1.0)
     return x + y

def print_sum(x, y):
     result = compute(x, y)
     print "%s + %s = %s" % (x, y, result)

print_sum(1, 2)
```

The bulk of blocking calls in the standard library are patched so that they yield to the hub rather than blocking. Likewise the threading system is patched so that greenlets are spawned instead of threads.

Isn't monkey-patching bad?

In this case it's better to think of gevent as a distribution of Python that happens to use green threads, and includes different implementations of standard library code.

This is part of the reason it has to come right at the top of the module that contains the entry point of the program: it's a statement that before anything else happens, we want to use the gevent's distribution of the stdlib.

While the monkey patching is elegant in that it allows pure Python applications and libraries to become asynchronous with no modifications, it is an optional component of gevent, and you can write asynchronous programs without it.

- Works with any existing synchronous, pure Python code.
- Generally works with async code too - only select() is emulated, but most async frameworks have a select()-based implementation. No reason epoll() etc couldn't be implemented.

Examples of gevent threading primitives

Because gevent is based on lightweight "threads" the gevent library contains a wealth of concurrency tools that help you spawn greenlets, implement critical sections (locks/mutexes), and pass messages between greenlets.

Because it's a networking library it also has some higher-level network server modules, such as a TCP server and WSGI server.

Spawning and killing greenlets

- gevent.spawn(function, *args, **kwargs) - spawn a greenlet

- gevent.kill(greenlet, exception=GreenletExit) - 'kill' a greenlet by raising an exception in that greenlet.

There are also higher-level primitives like gevent.pool, a greenlet-based equivalent to multiprocessing.pool.

Synchronisation primitives

- gevent.lock.Sempahore
- gevent.lock.RLock
- gevent.event.Event

Message Passing

- gevent.queue.Queue
- gevent.event.AsyncResult - block waiting for a single result. Also allows an exception to be raised instead.

Timeouts

There's a useful wrapper to kill a greenlet if it doesn't succeed within a certain period of time. This can be used to wrap timeouts around pretty much any sequence of operations:

```
from gevent import Timeout

with Timeout(5):
    data = sock.recv()
```

Higher level server kit

- gevent.server.StreamServer - TCP server. Also supports SSL.
- gevent.server.DatagramServer - UDP server.
- gevent.pywsgi.WSGIServer - WSGI server that supports streaming, keepalives, SSL etc.

Gevent I/O patterns

The recommended way of using gevent (and avoiding select()) is to spawn one greenlet for each direction of communication - read or write. The code for each greenlet is a simple loop that "blocks" whenever necessary. See the chat server (gevent-asynchronous-io-made-easy.html#chat-server) code earlier for an example.

Why do we want a synchronous programming model?

First of all, it makes the code easier to read and understand. It's the same kind of programming you do when single-threaded. The code isn't dotted with yield from statements.

More significantly, of the methods described above, only gevent requires no changes to the calling conventions of other code. The importance of this should not be understated: it means that your business logic can happily call into blocking code.

As a very simple example, say we want to do a streaming API. Our pre-existing business logic (process_orders()) expects an iterable. With gevent, we can stream this iterable asynchronously from a remote server with no code changes!

```
from gevent.socket import create_connection

def process_orders(lines):
    """Do something important with an iterable of lines."""
    for l in lines:
        ...


# Create an asynchronous file-like object with gevent
socket = create_connection((host, port))
f = socket.makefile(mode='r')
process_orders(f)
```

Another advantage is that exceptions can always be raised, in a sensible place.

Unlike real threads, greenlets are never suspended at arbitrary times, so you can get away with many fewer locks and mutexes - you only need a mutex if there's a risk you might "block" between atomic operations.

Also unlike real threads, a greenlet can "kill" another greenlet - causing an exception to be raised in that greenlet when it next resumes.

Disdvantages

Bad news: the Python 3 branch of gevent isn't finished. I have not investigated whether it is at all usable.

One solution is to allow different implementation languages in different parts of your stack. gevent is great at dealing with scalable I/O, Python 3 is good at Unicode and user-facing applications. It may be possible to arrange to use the right tool for each job.

Pitfalls of asynchronous I/O

Gevent shares several pitfalls with many asynchronous I/O frameworks:

- Blocking (real blocking, at the kernel level) somewhere in your program halts everything. This is most likely in C code where monkey patches don't take effect. You need to take careful steps to make your C library "green".
- Keeping the CPU busy. greenlets are not preempted, so this will cause other greenlets never to be scheduled.
- There exists the possibility of deadlock between greenlets.

In summary gevent has very few pitfalls that are not present in other async I/O frameworks (sure, you probably can't deadlock callbacks but only because the event loop probably doesn't provide the synchronisation primitives to do so, so you also can't implement critical sections).

One pitfall gevent sidesteps is that you are less likely to hit an async-unaware Python library that will block your app, because pure Python libraries will use the monkey patched stdlib.

n to m concurrency

An approach for even better scalability is to run n greenlets on m physical threads. In Python we need to do this with processes. This gives very good performance on multiprocessor systems, as well as adding resilience.

This is the model that is used by Rust and Java among others.

Experiences with gevent

I evaluated gevent as well as the other systems mentioned in 2011. Gevent was the clear winner. There was little to choose from in terms of performance, but gevent's significantly simpler programming model was a major selling point. Not all developers are at the level where they are comfortable with generators, closures and callbacks, and gevent doesn't require them: you can use whatever techniques make your code most legible.

The ability to use gevent with existing code, or business logic that is agnostic about IO, was very valuable too: you probably want to re-use certain business logic libraries in both high-performance network apps and offline batch processes.

Over the following 18 months we wrote a variety of network applications. One byproduct was a web service framework called nucleon (http://nucleon.readthedocs.org/en/latest/), which aimed to connect RESTful JSON, PostgreSQL and AMQP, all with 'green' driver code for high scalability.

Though we had to make code changes to use PostgreSQL without blocking, gevent's monkey patching meant that the pure-Python drivers for other data stores just worked - so Redis, ElasticSearch and CouchDB could all be used transparently.

The AMQP library was originally a fork of Puka (not Pika), an AMQP library that didn't try to force its own brand of async on you (like Pika). I eventually completely rewrote this and split it into a separate project called nucleon.amqp (http://pythonhosted.org/nucleon.amqp/). nucleon.amqp allows interaction with an AMQP server with a fully synchronous programming model - remote queues on the AMQP broker can be exposed locally with the Queue API.

It wasn't without head-scratching moments, as with any concurrent programming project. However as a team we adapted to gevent and found ourselves developing a language of diagrams to explain to each other and understand how the flow of control moves between greenlets, how they block and how they signal each other.

The project was successful - we were able to keep our code simple and maintainable while ensuring our services were fast and scalable (a claim backed up by load testing).

Network programming Python
