gevent, threads & async frameworks

Denis Bilenko

What is gevent?

- green threads for Python
  - Cooperative, only switch at I/O
- greenlet for context switching
  - No syscalls
- libev for event loop
  - Pretty fast

status

- current stable: 1.0.1
  - Runs on Python 2.5 – 2.7
- master branch: 1.1
  - Support for PyPy: 85% tests pass
  - Support for Python3: 53% tests pass
    - Basics and sockets work
    - Subprocesses and fileobjects todo

```
import thread, socket


def send(host):
    sock = socket.create_connection((host, 80))
    sock.sendall("GET / HTTP/1.0\r\n\r\n”)

# do two requests in parallel
create_new_thread(send, ('python.org', ))
create_new_thread(send, ('gevent.org', ))
```

```
#import thread, socket
from gevent import thread, socket

def send(host):
    sock = socket.create_connection((host, 80))
    sock.sendall("GET / HTTP/1.0\r\n\r\n”)

# do two requests in parallel
create_new_thread(send, ('python.org', ))
create_new_thread(send, ('gevent.org', ))
```

drop-in modules

```
from gevent import
    socket,
    ssl,
    subprocess,
    thread,
    local,
    queue
```

stdlib compatibility

- less to learn
- API is more stable across gevent versions
- we can use stdlib's tests to verify semantics
- trivial to port libraries and apps to gevent

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

import requests
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return requests.get('http://python.org').content

app.run()
```

A TCP server

```
def echo(socket, address):
    for line in socket.makefile():
        socket.sendall(line)

gevent.server.StreamServer(':5000', handle).start()
```

gevent.wait([… objects …])

- wait for any gevent object
- extendable through rawlink(callback)

gevent.wait([… objects …], timeout=5)

- limit waiting to certain time

gevent.wait([... objects ...], count=N)

- only wait for N objects
- return value is a list of ready objects

gevent.wait()

- wait for everything
- “background” watchers ref=False
- graceful shutdown:
  - stop accepting new requests
  - gevent.wait()

geventserver

- pre-fork, <999LOC
- supports any gevent server, not just HTTP
- graceful shutdown
- can be embedded: import geventserver
- reports long loop iterations

github.com/surfly/geventserver

Why?

- Avoid complexities of event loops
- Avoid costs of real threads

Complexities of event loops

A simple

```
sock = socket.create_connection((host, 80))
sock.sendall("GET / HTTP/1.0\r\n\r\n”)
data = sock.recv(1024)
```

becomes

Complexities of event loops

```
def connectionMade(self):
    …
def dataReceived(self, data):
    …
def connectionError(self, error):
    …
```

async vs. sync

exceptions for I/O errors

```
try:
    connect()
except IOError:
    ...
```

```
def connectionMade(self):
    ...

def connectionError(self, error):
    ...
```

async vs. sync

context managers

```
with open('log', 'w') as log:
    io_operation()
    log("connected")
    io_operation()
    log("sent")

# log object is closed by "with"
```

explicit state machine

async vs. sync

synchronous programming model

```
handle_result(func(params))
```

```
d = Deferred()
func(d)
d.add_callback(handle_result)
d.add_errback(handle_error)
```

Giving up

- exception handling
- context managers
- synchronous programming style
- 3rdparty libraries

A subset of Python without batteries.

Generators / PEP-3156?

- Help *somewhat*
- But not enough:

```
with conn.cursor() as curs:
    curs.execute(SQL)
# cursor() needs to do I/O in __exit__
```

- No compatibility with threading / 3rdparty libraries

Coroutines

- generators: non-stackful coroutine
  - Only yield to parent
  - Need special syntax
  - Only saves the top frame
- greenlet: stackful coroutine
  - Switching is just a function call
  - Switching to any target
  - Saves the whole stack, like a thread

Why?

- Avoid complexities of event loops
- Avoid costs of real threads

Threads vs green threads

- creation
  - thread.start_new_thread: 28usec
  - gevent.spawn: 5usec
  - gevent.spawn_raw: 1usec

- does not matter if used via pool

Threads vs green threads

memory

- threads: 8MB of stack by default
- greenlet: only allocate what's actually used
  - 350 bytes per Python frame
  - 10-15KB

- does not matter since the memory is virtual
  - limits number of threads on 32bit arch

Gevent server

```
def handle(socket, addr):
    # read out the request
    for line in socket.makefile():
        if not line.strip():
            break
    # send the response
    socket.sendall(HTTP_RESPONSE)
    socket.close()

from gevent.server import StreamServer
StreamServer(':5000', handle).serve_forever()
```

Threaded server

```
queue = Queue()

def worker():
    while True:
        socket = queue.get()
        handle(socket)

for _ in xrange(1000):
    thread.start_new_thread(handle, ())

while True:
    socket, addr = listener.accept()
    queue.put(socket)
```

Benchmark

ab -n 1000 -c 100 http://localhost:5000/

- threaded: 7.1K RPS, latency 14ms
- gevent: 9.3k RPS, latency 11ms

- The threaded server can probably be improved

Mutex / context switch benchmark

```
def func(source, dest, finished):
    source_id = id(source)
    for _ in xrange(COUNT):
        source.acquire()
        dest.release()
    finished.release()

thread.start_new_thread(func, (sem1, sem2, a_finished))
thread.start_new_thread(func, (sem2, sem1, b_finished))
```

Threads vs green threads

- context switch
  - 2 threads switch to each other using 2 mutexes
- gevent threads: 15ns
- real threads: 60ns
  - 2 CPUs – GIL contention

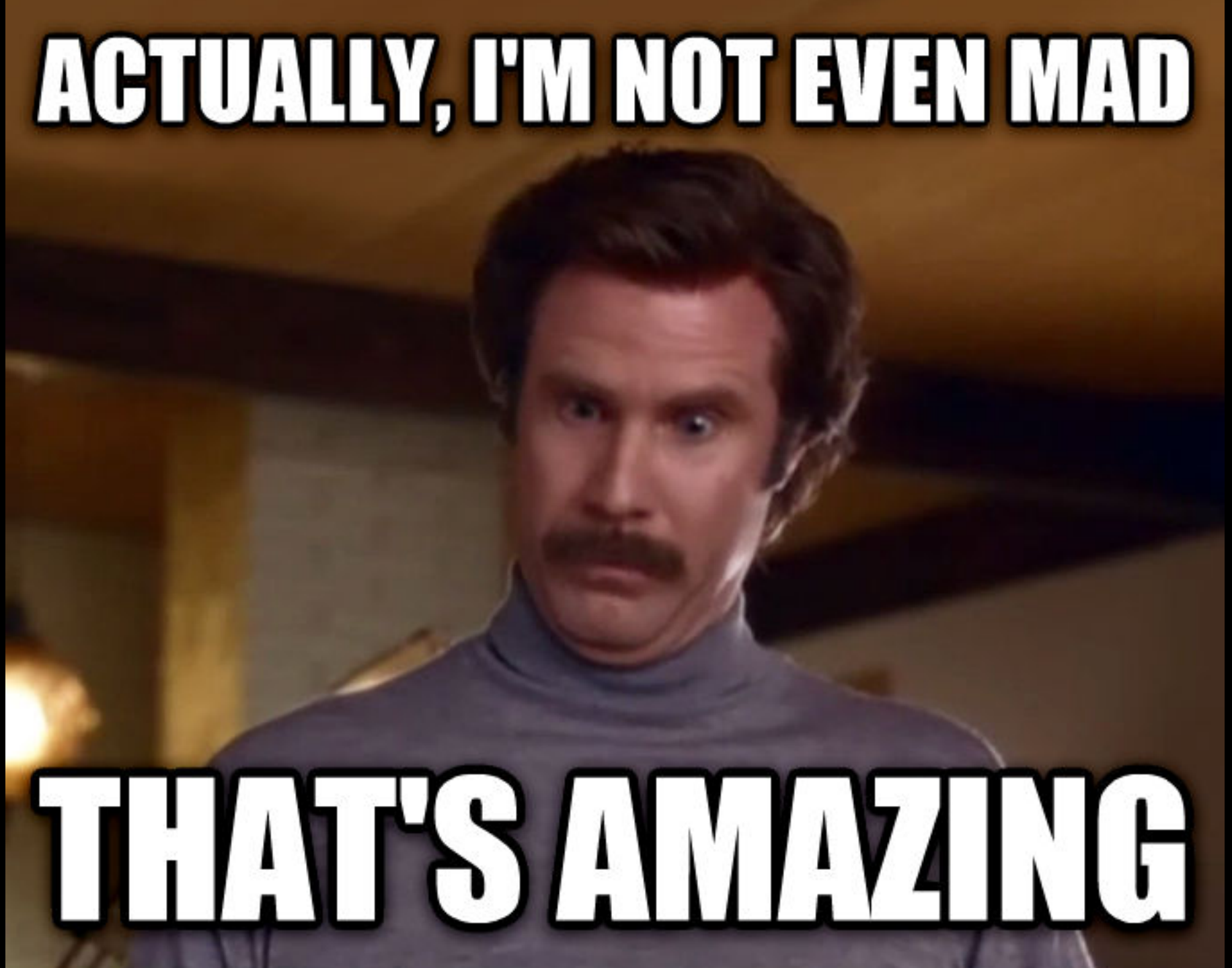
to avoid “taskset 1 python …”

Threads vs green threads

- gevent threads: 15ns
- real threads: 12ns

PyPy:

- gevent threads: 11ns
- real threads: 7ns
- requires warmup, with only 100 iterations: 115ns / 35ns


ACTUALLY, I'M NOT EVEN MAD
THAT'S AMAZING

JOIN THE REVOLUTION

Thousands of Threads and Blocking I/O

The old way to write Java Servers is New again

(and way better)

Paul Tyma

paul.tyma@gmail.com

paultyma.blogspot.com

http://www.mailinator.com/tymaPaulMultithreaded.pdf

SD WEST 2008

MARCH 3-7, 2008, SANTA CLARA, CA

Threads on Linux

- Threads used to have awful performance
- But since linux 2.6 / NPTL they've improved a lot
  - idle threads cost is almost zero
  - context switching is much faster
  - you can create lots of them
- (in Java) sync I/O is 30% faster than async I/O

cooperative OS threads?

- Already exclusive due to GIL
- sys.setswitchinteval(2 ** 31) # Python 3
- sys.setcheckinterval(2 ** 30) # Python 2

would not work if you have CPU-hungry threads, but neither will async frameworks

Can gevent be speed up?

- Switching and primitives are in Python
  - Let's try C
- Switching is done through Hub
  - Let's try directly

libgevent

- stacklet: a C module that PyPy uses to implement greenlet
- libuv: Node.JS event loop

libgevent

```
gevent_cothread t1;
gevent_cothread_init(hub, &t1, sleep0);
gevent_cothread_spawn(&t1);
gevent_wait(hub)
```

libgevent

- It's only a prototype
- spawn(), sleep(), wait()
- channels, semaphores
- getaddrinfo()
- naive Python wrapper

https://github.com/denik/libgevent

Threads vs green threads

- context switch
  - 2 threads switch to each other using 2 semaphores
- gevent threads: 15ns
- real threads: 12ns
- libgevent/gevent2: 1.8ns

Conclusions

- Thread pool is a deployment option with considering
- Gevent's performance can be pushed further
- Avoid framework lock-in
  - Migrating between gevent and threads is easy
  - Migrating between async and sync models is not
- The better threads are, the more irrelevant async frameworks are (gevent included)
  - And threads are already pretty fast

References

gevent:
http://gevent.org

faster gevent experiment:
https://github.com/denik/libgevent

pre-fork server for gevent:
https://github.com/surfly/geventserver

“Thousands of threads and blocking I/O” by Paul Tyma:
http://www.mailinator.com/tymaPaulMultithreaded.pdf