Eventlet

Asynchronous I/O with a Synchronous Interface

Network Servers

Processes, Threads, or Non-Blocking I/O?

The C10K Problem

- http://www.kegel.com/c10k.html
- “It's time for web servers to handle ten thousand clients simultaneously, don't you think?”

Processes, Threads, Non-Blocking I/O

- Processes
  - Too heavyweight
- Threads
  - Non-determinism sucks
- Non-Blocking I/O
  - Requires callback-style programming
    - Rules out many existing libraries

Solution: Coroutines

- **Callbacks**: Register a callback function and then **Return** to the main loop
- **Coroutines**: Register a callback coroutine and then **Call** the main loop
  - The call stack is preserved
  - Does not require cooperation from the caller

Enhanced Generator Coroutine Problems

- Python 2.5's Enhanced Generators can be used to implement coroutines
- The yield statement returns control to the caller, unlike a traditional coroutine
  - Requires caller participation
    - Java "Checked Exception" problem
- They have other caveats

Solution: Greenlet

- Greenlet Provides Hard Switching from Stackless in a Regular Python Module
- Stack Slicing is used to implement coroutine switching
  - Portions of the C Stack are copied to the Heap and vice versa

Eventlet

Green Threads on top of Greenlet

Green Threads: Lightweight Threads

- Regular POSIX threads are **Preemptive**
  - Non-Deterministic
- Green Threads are **Cooperative**
  - Deterministic
- Green Threads use much less memory

Spawning a Green Thread

- spawn(

  func,

  *args,

  **kwargs)

```
>>> def func(x, y):
...     return x + y
...
>>> from eventlet import api
>>> print api.spawn(func, 1, 2).wait()
3
```

Cooperating: Voluntarily Yielding

- sleep(0)
  - "Run something else, then switch back to me as soon as possible"
- sleep(1)
  - "Switch to me after 1 second"

```
from eventlet import api


def func1():
    api.sleep(2)
    print "func1"


def func2():
    api.sleep(1)
    print "func2"


greenlets = api.spawn(func1), api.spawn(func2)

for g in greenlets:
    g.wait()
```

Outputs:

```
func2
func1
```

Synchronization: Event

- One sender, multiple waiters
- One use
- Output:

```
func3 begin
func3 exit
func1
func2
```

```
from eventlet import api
from eventlet import coros

def func1(event):
    event.wait()
    print "func1"

def func2(event):
    event.wait()
    print "func2"

def func3(event):
    print "func3 begin"
    event.send()
    print "func3 exit"

evt = coros.event()

greenlets = [
    api.spawn(func1, evt),
    api.spawn(func2, evt),
    api.spawn(func3, evt)]

for x in greenlets:
    x.wait()
```

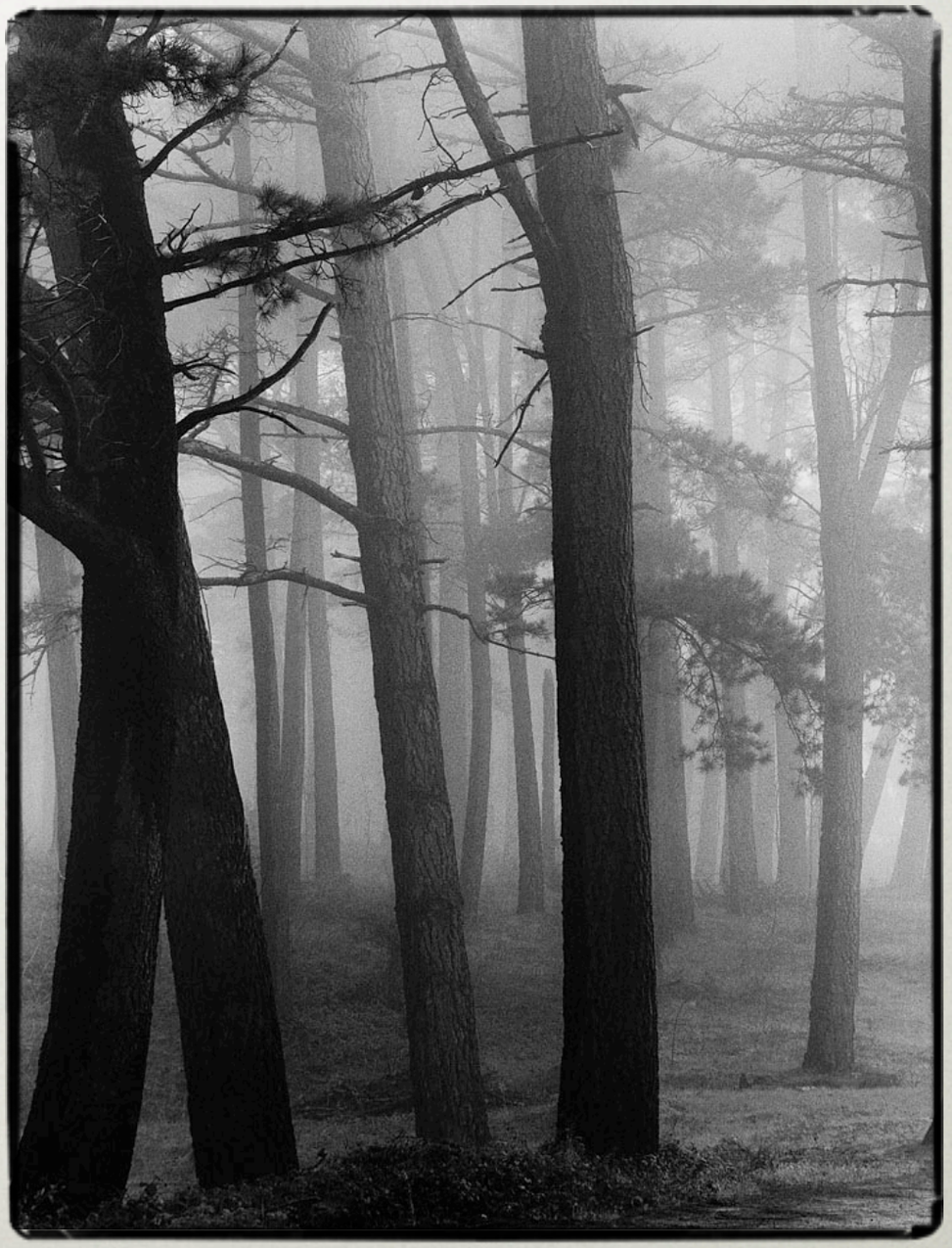
Synchronization: Queue

- Multiple senders, multiple waiters
- Multiple use
- Output:

```
func3 begin
func3 middle
func3 exit
func1 1
func2 2
```

```
from eventlet import api
from eventlet import coros

def func1(q):
    result = q.wait()
    print "func1", result

def func2(q):
    result = q.wait()
    print "func2", result

def func3(q):
    print "func3 begin"
    q.send(1)
    print "func3 middle"
    q.send(2)
    print "func3 exit"

q = coros.Queue()

greenlets = [
    api.spawn(func1, q),
    api.spawn(func2, q),
    api.spawn(func3, q)]

for x in greenlets:
    x.wait()
```

Concurrency Control: Pool



Integration with Twisted Deferred

- If you don't know what a Deferred is: you don't have to.
- eventlet.twistedutil
  - block_on(deferred)
    - Returns result when Deferred fires
  - deferToGreenThread(func, *args, **kw)
    - Returns Deferred

EVENTLET.GREEN

Cooperative Sockets

eventlet.green: Cooperative Sockets

- Same interface as socket.socket
- Instead of blocking, the cooperative socket switches to the main loop
- Main loop runs select (or poll, etc) and switches back to “blocked” coroutine when I/O is ready

Socket Example

```
from eventlet import api
from eventlet.green import socket

def handle_socket(reader, writer):
    print "client connected"
    while True:
        # pass through every non-eof line
        x = reader.readline()
        if not x: break
        writer.write(x)
        print "echoed", x
    print "client disconnected"

print "server socket listening on port 6000"
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('', 6000))
server.listen(100)

while True:
    try:
        new_sock, address = server.accept()
    except KeyboardInterrupt:
        break
    # handle every new connection with a new coroutine
    api.spawn(handle_socket, new_sock.makefile('r'), new_sock.makefile('w'))
```

Sockets Have Implicit Cooperation Points

- Any API which would normally block cooperates instead
  - connect
  - read
  - write
  - etc.

Emulated Modules

- BaseHTTPServer
- httplib
- os
- select
- socket
- SocketServer
- thread
- threading
- time
- urllib
- urllib2
- Easy to add more

Patching Other Libraries to Cooperate

- Import one module patched with cooperative sockets
- Create a new module which is a patched version of the original
- Monkeypatch sys.modules globally for the entire process

Spawning

WSGI Server Written Using Eventlet

Spawning: Highly Configurable

- Can be configured to use:
  - Multiple OS Processes
  - Multiple POSIX Threads
  - Green Threads
- And various combinations of the three

Spawning: Designed for COMET

- "Real Time" web applications are finally becoming popular
- Servers must keep open one connection per active user
- When Spawning is configured to use eventlet's green threads it is perfect for COMET

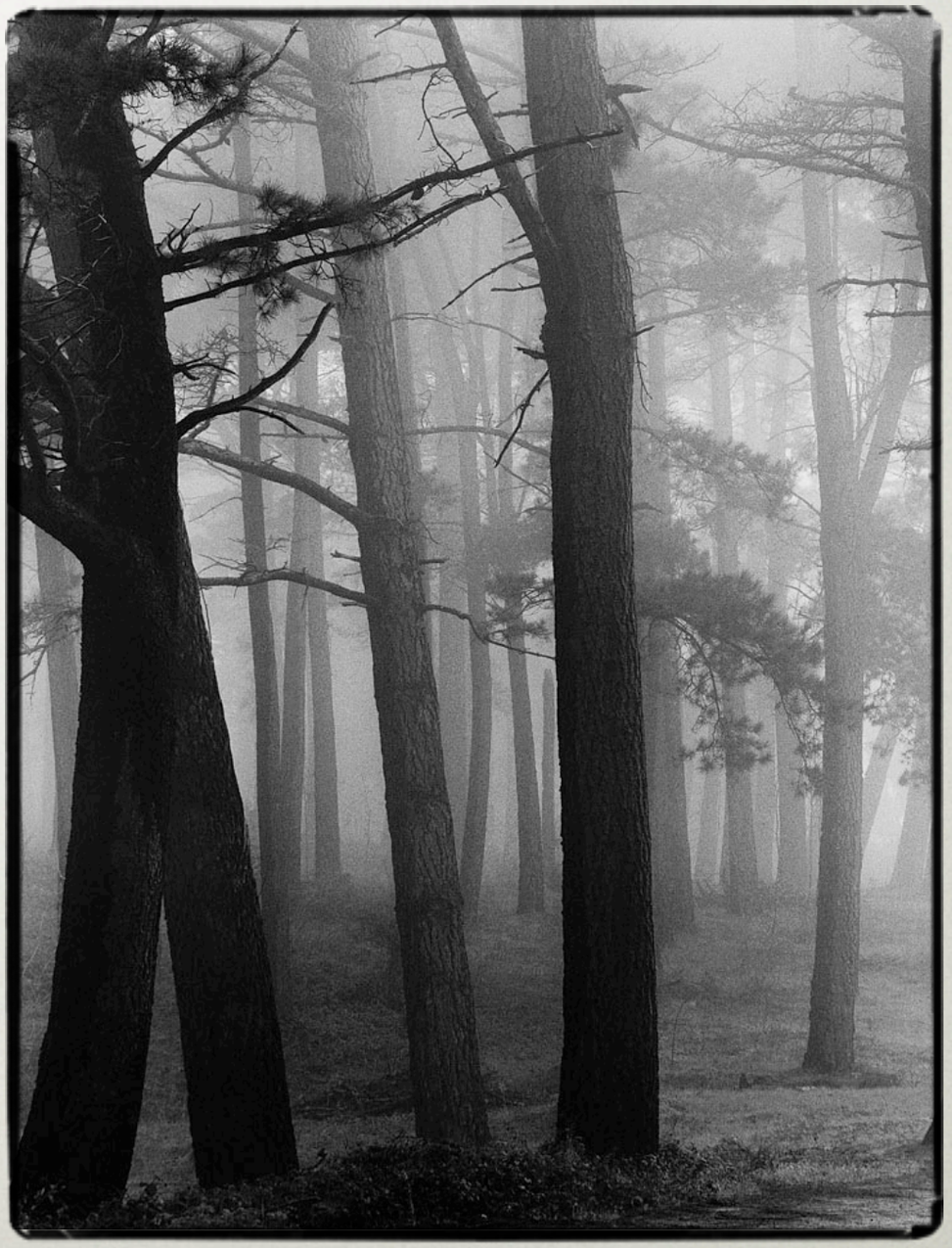
Summary

Eventlet

- High Scalability Non-Blocking I/O
- True Coroutines using Greenlet
- Green Threads with Scheduler
- Cooperative socket Implementation
- Easy to Integrate with Existing Libraries

Eventlet in Production

- In production at Linden Lab (Second Life) since 2006
- Handles a huge amount of traffic



Q&A