Architecting an event-driven networking framework: Twisted Python

@jessicamckellar
I want to build

- a web server
- a mail server
- a BitTorrent client
- a DNS server
- an IRC bot
- clients and servers for a custom protocol

in Python
I want them to be

• event-driven
• cross-platform
• RFC-compliant
• testable
• deployable in a standardized fashion
• supporting code-reuse
What should I do?

The TCP/IP model (RFC 1122)

**Application Layer**
- BGP
- DHCP
- DNS
- FTP
- Gopher
- GTP
- HTTP
- IMAP
- IRC
- NNTP
- NTP
- POP
- RIP
- RPC
- RTCP
- RTP
- RTSP
- SDP
- SIP
- SMTP
- SNMP
- SOAP
- SSH
- STUN
- Telnet
- TIME
- TLS/SSL

**Transport Layer**
- TCP
- UDP
- DCCP
- SCTP
- RSVP

**Internet Layer**
- IP (IPv4, IPv6)
- ICMP
- ICMPv6
- IGMP

**Link Layer**
- ARP
- RARP
- NDP
- OSPF
- Tunnels (L2TP)
- Media Access
- Control (Ethernet, DSL, ISDN, FDDI)
- Device Drivers
Event-driven Python

- **asyncore**: standard library module for writing asynchronous socket service clients and servers. (Stackless)

- **gevent**: coroutine-based library using greenlets to provide a synchronous API on top of the libevent event loop. (ZeroMQ)

- **Tornado**: asynchronous web server. (FriendFeed)
asyncore echo server

class EchoHandler(asyncore.dispatcher_with_send):
    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):
    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accept(self):
        pair = self.accept()
        if pair is not None:
            sock, addr = pair
            handler = EchoHandler(sock)

server = EchoServer('localhost', 8000)
asyncore.loop()
Event-driven Python

- asyncore: standard library module for writing asynchronous socket service clients and servers. (Stackless)

- gevent: coroutine-based library using greenlets to provide a synchronous API on top of the libevent event loop. (ZeroMQ)

- Tornado: asynchronous web server. (FriendFeed)
Event-driven Python

- asyncore: standard library module for writing asynchronous socket service clients and servers. (Stackless)

- gevent: coroutine-based library using greenlets to provide a synchronous API on top of the libevent event loop. (ZeroMQ)

- **Tornado**: asynchronous web server. (FriendFeed)
Twisted: an event-driven networking framework

- event-driven
- cross-platform
- RFC-compliant
- “batteries-included”
- general and extensible
- clients and servers

TCP, UDP, SSL, HTTP, IMAP, IRC, SMTP, POP3, IMAP, DNS, FTP...
Architecture
while True:
    timeout = time_until_next_timed_event()
    events = wait_for_events(timeout)
    events += timed_events_until(now())
    for event in events:
        event.process()
Transports

Represent the connection between two endpoints communicating over a network

TCP, UDP, SSL, UNIX sockets
# Transports

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>Write some data to the physical connection, in sequence, in a non-blocking fashion.</td>
</tr>
<tr>
<td>writeSequence</td>
<td>Write a list of strings to the physical connection.</td>
</tr>
<tr>
<td>loseConnection</td>
<td>Write all pending data and then close the connection.</td>
</tr>
<tr>
<td>getPeer</td>
<td>Get the remote address of this connection.</td>
</tr>
<tr>
<td>getHost</td>
<td>Get the address of this side of the connection.</td>
</tr>
</tbody>
</table>
Protocols
Describe how to process network events asynchronously.

HTTP, IMAP, IRC, DNS
# Protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>makeConnection</code></td>
<td>Make a connection to a transport and a server.</td>
</tr>
<tr>
<td><code>connectionMade</code></td>
<td>Called when a connection is made.</td>
</tr>
<tr>
<td><code>dataReceived</code></td>
<td>Called whenever data is received.</td>
</tr>
<tr>
<td><code>connectionLost</code></td>
<td>Called when the connection is shut down.</td>
</tr>
</tbody>
</table>
Transports and protocols are decoupled
TCP echo server

class Echo(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()

reactor.listenTCP(8000, EchoFactory())
reactor.run()
TCP echo server

class Echo(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()

reactor.listenTCP(8000, EchoFactory())
reactor.run()
class Echo(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()

reactor.listenTCP(8000, EchoFactory())
reactor.run()
TCP echo server

class Echo(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()

reactor.listenTCP(8000, EchoFactory())
reactor.run()
class Echo(protocol.DatagramProtocol):
    def datagramReceived(self, data, addr):
        self.transport.write(data, addr)

reactor.listenUDP(8000, Echo())
reactor.run()
UDP echo server

class Echo(protocol.DatagramProtocol):
    def datagramReceived(self, data, addr):
        self.transport.write(data, addr)

reactor.listenUDP(8000, Echo())
reactor.run()
UDP echo server

class Echo(protocol.DatagramProtocol):
    def datagramReceived(self, data, addr):
        self.transport.write(data, addr)

reactor.listenUDP(8000, Echo())
reactor.run()
SSL echo server

class Echo(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()

context = DefaultOpenSSLContextFactory("server.key", "server.crt")
reactor.listenSSL(8000, EchoFactory(), context)
reactor.run()
SSL echo server

class Echo(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()

context = DefaultOpenSSLContextFactory("server.key", "server.crt")
reactor.listenSSL(8000, EchoFactory(), context)
reactor.run()
SSL echo server

class Echo(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()

context = DefaultOpenSSLContextFactory("server.key", "server.crt")
reactor.listenSSL(8000, EchoFactory(), context)
reactor.run()
Architecture

reactor

transports

protocols

TCP, UDP, SSL, UNIX sockets

HTTP, IMAP, IRC, DNS
Managing callbacks
import getPage

def processPage(page):
    print page

def logError(error):
    print error

def finishProcessing():
    print "Shutting down..."
    exit(0)

url = "http://google.com"
try:
    page = getPage(url)
    processPage(page)
except Error, e:
    logError(e)
finally:
    finishProcessing()
import reactor
import getPage

def processPage(page):
    print page
    finishProcessing()

def logError(error):
    print error
    finishProcessing()

def finishProcessing():
    print "Shutting down..."
    reactor.stop()

url = "http://google.com"
getPage(url, processPage, logError)
reactor.run()
Deferred

An abstraction of the idea of a result that doesn't exist yet; a **promise** that a function will have a result at some point.

A **Deferred** helps manage the callback chains for processing an asynchronous result.
import reactor
import getPage

def processPage(page):
    print page

def logError(error):
    print error

def finishProcessing(value):
    print "Shutting down..."
    reactor.stop()

url = "http://google.com"
deferred = getPage(url)
deferred.addCallbacks(processPage, logError)
deferred.addBoth(finishProcessing)

reactor.run()
# Deferred API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addCallback</code></td>
<td>Register a callback with the callback chain.</td>
</tr>
<tr>
<td><code>addErrback</code></td>
<td>Register an errback with the errback chain. The analogous synchronous logic is the except part of a <code>try/except</code> block.</td>
</tr>
<tr>
<td><code>addCallbacks</code></td>
<td>Add a callback and errback parallel to each other in the callback chains.</td>
</tr>
<tr>
<td><code>addBoth</code></td>
<td>Add the same callback to both the callback and errback chains. The analogous synchronous logic is the <code>finally</code> part of a <code>try/except/finally</code> block.</td>
</tr>
</tbody>
</table>
import reactor
import getPage

def processPage(page):
    print page

def logError(error):
    print error

def finishProcessing(value):
    print "Shutting down..."
    reactor.stop()

url = "http://google.com"
deferred = getPage(url)
deferred.addCallbacks(processPage, logError)
deferred.addBoth(finishProcessing)

reactor.run()
Deferreds

- Can only be fired once
- Can be paused/unpaused/cancelled

```
try
addCallback

except
addErrback

finally
addBoth
```
Architecture

reactor

Deferreds

transports

protocols

TCP, UDP, SSL, UNIX sockets

HTTP, IMAP, IRC, DNS
“Batteries-included”
HTTP server

```python
define twisted from twisted import import reactor
from twisted.web.server import Site
from twisted.web.static import File

resource = File("/var/www/mysite")
factory = Site(resource)
reactor.listenTCP(80, factory)
reactor.run()
```
from twisted.cred import checkers, portal
from twisted.internet import reactor
from twisted.words import service

wordsRealm = service.InMemoryWordsRealm("example.com")
wordsRealm.createGroupOnRequest = True

checker = checkers.FilePasswordDB("passwords.txt")
portal = portal.Portal(wordsRealm, [checker])

reactor.listenTCP(6667, service.IRCFactory(wordsRealm, portal))
reactor.run()
- scheduling
- authentication
- interacting with databases
- using threads and processes
- testing event-driven programs

POP3  HTTP
IMAP  DNS
SMTP  SSH
FTP  IRC
Twisted applications
class Echo(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()

reactor.listenTCP(80, EchoFactory())
reactor.run()
from echo import EchoFactory

application = service.Application("echo")

echoService = internet.TCPServer(80, EchoFactory())

echoService.setServiceParent(application)
$ twistd -y echo_server.tac
2011-11-19 22:23:07 [-] twistd 12.1.0 (/usr/bin/python...
2011-11-19 22:23:07 [-] Starting factory <EchoFactory>...
2011-11-19 22:23:20 [-] Received SIGTERM, shutting down.
2011-11-19 22:23:20 [-] (TCP Port 8000 Closed)
2011-11-19 22:23:20 [-] Stopping factory <EchoFactory>...
Twisted applications

- logging
- daemonization
- profiling
- authentication
twistd web --port 8000 --path .
twistd dns -v -p 5553 --hosts-file=hosts
sudo twistd conch -p tcp:2222
twistd mail -E localhost -d example.com=/tmp/example.com
twistd mail --relay=/tmp/mail_queue
Architecture

reactor

Deferreds

Applications

transports

protocols

TCP, UDP, SSL, UNIX sockets

HTTP, IMAP, IRC, DNS
Twisted: an event-driven networking framework

- event-driven
- cross-platform
- RFC-compliant
- “batteries-included”
- general and extensible
- clients and servers

TCP, UDP, SSL, HTTP, IMAP, IRC, SMTP, POP3, IMAP, DNS, FTP...
“This is a proposal for asynchronous I/O in Python 3, starting with Python 3.3... The proposal includes a pluggable event loop API, transport and protocol abstractions similar to those in Twisted, and a higher-level scheduler based on yield from (PEP 380).”
Food for thought

- Evolution of async primitives in Python and other languages
- Twisted as a monolithic framework
Thank you!
@inlineCallbacks

def download(url):
    try:
        page = yield getPage(url)
        processPage(page)
    except Error, e:
        logError(e)
    finally:
        finishProcessing(page)
@inlineCallbacks

• **generator functions**: “restartable functions” that use `yield` to produce iterators

• **coroutines**: generator functions that can accept arguments as well as `yield` them

• **decorators**: callables that takes a function as an argument and return a replacement function; they “wrap” functions.
@inlineCallbacks

- **generator functions**：“restartable functions” that use `yield` to produce iterators

```python
def countdown(n):
    while n > 0:
        yield n
    n -= 1

x = countdown(10)
for num in x:
    print(num)
```
inlineCallbacks

- coroutines: generator functions that can accept values as well as yield them

\[ x = \text{yield} \ n \]
@inlineCallbacks

• **decorators**: callables that takes a function as an argument and return a replacement function; they “wrap” functions.

```python
def decorate(func):
    print "decorating"
    return func

@decorate
def countdown(n):
    ...
```
@inlineCallbacks

- **generator functions**: “restartable functions” that use yield to produce iterators

- **coroutines**: generator functions that can accept arguments as well as yield them

- **decorators**: callables that takes a function as an argument and return a replacement function; they “wrap” functions.
def download(url):
    try:
        page = yield getPage(url)
        processPage(page)
    except Error, e:
        logError(e)
    finally:
        finishProcessing(page)