a Twisted introduction

Use and advantages

Presented par

Jean Daniel Browne

Plan

1. Twisted: use and key concepts
2. the mechanism at the core of Twisted
3. example: a notification client

Twisted use and key concepts

applications
distributed on multiple servers,
using multiple protocols

Non-blocking interfaces

Common code for subsequent examples

```
from lxml.html import parse
dig = lambda html,pattern: parse(html).xpath(pattern)[0]


planets = ["http://planet.debian.net",
           "http://planetzope.org",
           "http://planet.gnome.org",
           "http://gstreamer.freedesktop.org/planet/"]:
```

Blocking code

```
from urllib2 import urlopen

def first_title(url):

    article = dig(urlopen(url ), '//h3' ).text
    print "first article on " url, title


for planet in planets:
    first_title(planet)
```

Twisted's non-blocking equivalent

```
from twisted.internet import reactor
from twisted.web.client import getPage


def print_first_title(html):
    article = dig(html, '//h3').text
    print "first article on ", url, article, title

for p in planets:
    getPage(p).addCallback(print_first_title)

reactor.run()
```

No threads, no locks

No recursive locks,
read/write locks, deadlocks
interlocking ...

Blocking parallel ... and buggy code

```
from urllib2 import urlopen
from threading import Thread

def first_title(url):

    title = dig(urlopen(url), '/html/head/title').text
    print "first article on:", url
    print "the title is:", title

threads = (Thread(first_title, (p,)) for p in planets)

for t in threads:
    t.start()

for t in threads:
    t.join()
```

Functions must return quickly

Can't mix blocking and non-blocking code

```
from smtplib import SMTP

msg = """From: notifyer@m.com
To: admin@m.com

"""

def print_first_title(html):
    title = dig( html, '//h3').text)
    s = SMTP('localhost')
    s.sendmail(notifyer@m.com, admin@m.com, msg + title)
    s.quit()

for p in planets:
    getPage(p).addCallback(print_first_title)
```

The mechanism at the core of Twisted

A system call
for supervising a list of sockets

like *select()*

For each socket:

an instance of the **Protocol class** holds **the callbacks** processing the data received from the socket

1. When data is available in a socket,

2. the reactor triggers *dataReceived(data)* of of the Protocol associated to the socket,

3. the data is parsed, formatted then routed toward the user callbacks

Example:
a notification client

A simple client/server protocol

A simple client/server protocol

Client		**Server**
`classified`	→	
	←	`nice flat in the 11e`
`random?`	→	
	←	`46774`

A simple client/server protocol

Client		**Server**
“classified?\n”	→	
	←	“nice flat in the 11e\n”
	...	
“random?\n”	→	
	←	“46774\n”

Public interface: 2 commands and an event

```
from twisted.protocols import basic

class Client(basic.LineReceiver):


    def classified(self):
        “Sends the request for a classified ad”

    def random(self):
        “Sends the request for a random number”



    def connectionMade(self):     # Event from the mother class
        “Code called by the reactor when the TCP connection is ready”
```

From the user point of view

```
from twisted.internet import reactor, protocol


class MyClient(Client):


    def connectionMade(self):
        self.random().addCallback(self.print_and_get_classified)

    def print_and_get_classified(self, result):
        print result
        self.classified().addCallback(self.print_and_stop)

    def print_and_stop(self, result):
        print result
        reactor.stop()

factory = protocol.ClientFactory()
factory.protocol = Client
reactor.connectTCP("localhost", 6789, factory)
reactor.run()
```

From the user point of view

```
def connectionMade(self):
    self.random().addCallback(self.print_and_get_classified)

def print_and_get_classified(self, result):
    print result
    self.classified().addCallback(self.print_and_stop)

def print_and_stop(self, result):
    print result
    reactor.stop()
```

From the user point of view

```
from twisted.internet.defer import inlineCallbacks as _o
```

```
    def connectionMade(self):
        self.random().addCallback(self.print_and_get_classified)

    def print_and_get_classified(self, result):
        print result
        self.classified().addCallback(self.print_and_stop)

    def print_and_stop(self, result):
        print result
        reactor.stop()
```

From the user point of view

```
from twisted.internet.defer import inlineCallbacks as _o


    @_o
    def connectionMade(self):
        print yield self.random()
        print yield self.classified()
        reactor.stop()
```

From the user point of view

```
from twisted.internet.defer import inlineCallbacks as _o
from twisted.internet import reactor, protocol

class MyClient(Client):

    @_o
    def connectionMade(self):
        print yield self.random()
        print yield self.classified()
        reactor.stop()

factory = protocol.ClientFactory()
factory.protocol = Client
reactor.connectTCP("localhost", 6789, factory)
reactor.run()
```

Implementing the public interface

```
from twisted.protocols import basic

class Client(basic.LineReceiver):

    def classified(self):
        return self.command("classified?")

    def random(self):
        def gotRandom(number):
            return int(number)
        return self.command("random?").addCallback(gotRandom)
```

Private methods

```
from twisted.protocols import basic

class Client(basic.LineReceiver):

    def command(self, cmd):
        self.sendLine(cmd)
        self.d = defer.Deferred()
        return self.d

    def lineReceived(self, data):
        self.d.callback(data)
```

Supporting notifications in the protocol

Extending the protocol with **notifications**

```
Client                  Server

notif       →

                    ← notif: random
stop        →
random?     →
                    ← 46774
notif       →
```

Notification interface

```
def notify(self):
    “Request the server to switch to notification mode”
    self.sendLine(“notify”)

def stopNotify(self):
    “Request to switch back to normal client/server mode”
    self.sendLine(“stop”)

def waitNotif(self):
    “Returns a placeholder for the notification callback”
    self.d = Deferred()
    return self.d
```

From the user point of view

```
class Client(basic.LineReceiver):

    @_o
    def connectionMade(self):
        self.notify()

        while True:
            notif = yield self.waitNotif()

            if notif=='notif: random':
                self.stopNotify()
                print yield self.random()
                self.notify()

            else:
                print "not interested, will wait for the next notification"
```

Pb 1. The user must test the data received and interpret the events to branch to the correct handling code

Can't the protocol author handle the parsing/dispatch?

Pb 2. How to model, with a programming language, a request which result in an **unknown number** of responses of **differing nature**?

Notifications API: 2 callbacks

```
def randomAvailable(self):
    “Callback triggered when a random number is available”

def classifiedAvailable(self):
    “Callback triggered when a classified number is available”
```

Simplification of the user code

```
class MyClient(Client):

    @_o
    def connectionMade(self):
        yield self.notify()

    @_o
    def randomAvailable(self):
        yield self.stopNotify()
        print (yield self.random())
        yield self.notify()
```

1. The user only focus on processing the events

2. The protocol details can be updated without modification of the user code

Parsing/dispatch by the protocol author

```
class Client(basic.LineReceiver):

    def lineReceived(self, data):
        if data.startswith('notif:'):

            prefix, command = data.split()

            if command == 'random' and hasattr(self, 'randomAvailable'):
                self.randomAvailable()

            elif command == 'classified' and hasattr(
                                            self, 'classifiedAvailable'):
                self.classifiedAvailable()
        else:
            d.callback(data)
```

Summary

- Splitting blocking functions between *emission* and *callback* makes it possible to use a reactor for handling concurrent requests without threads,
- The reactor: a socket supervision syscall,
  each socket has its callbacks associated thanks to a Protocole instance,
- **A**synchronous API : good for handling events (ex: servers)
  Blocking API : good for sequential clients

Questions?

Contact:
jeandaniel.browne@gmail.com