Introduction à Twisted

Usages et avantages

Presenté par

Jean Daniel Browne

Plan

1. Twisted: usages et points clés
2. le mécanisme au cœur de Twisted
3. exemple: un client de notification

Twisted
usages et points clés

Des applications
distribuées sur plusieurs serveurs,
utilisant plusieurs protocoles

interfaces non bloquantes

Code commun

```
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/"]:
```

Code bloquant

```
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)
```

Equivalent Twisted non bloquant

```
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()
```

Ni thread, ni verrou

Code bloquant, parallèle … et buggé

```
from urllib2 import urlopen

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()
```

Code bloquant, parallèle … et buggé

```
from urllib2 import urlopen

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()
```

La nécessité de retourner rapidement

Ne pas mélanger bloquant et non bloquant

```
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)
```

le mécanisme au cœur de Twisted

Un appel système type *select* de supervision d'une liste de sockets

Pour chaque socket:

une instance de la class Protocol
contient les callbacks

1. A l'arrivée des données dans une socket,

2. le reactor déclenche *dataReceived*() du protocole associé à la socket avec les données,

3. les données sont formattées et routées vers les callbacks de l'utilisateur

Exemple: un client de notification

Un simple protocol client/serveur

```
Client                  Serveur

classified?      →
                 ←      nice flat in the 11e

                ...

random?          →
                 ←      46774
```

Interfaces publiques

```
from twisted.protocols import basic

class Client(basic.lineReceived):


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

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



    def connectionMade(self):
        “Code called by the reactor when the TCP connection is ready”
```

Du point de vue de l'utilisateur

```
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()
```

Du point de vue de l'utilisateur

```
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()
```

Du point de vue de l'utilisateur

```
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()
```

Du point de vue de l'utilisateur

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

[ … ]

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

Implémentation des interfaces publiques

```
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)
```

Méthodes privées

```
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)
```

Méthodes privées

```
from twisted.protocols import basic
from twisted.internet import defer

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)
```

Extension du protocole: les notifications

```
Client            Serveur

notif     →


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

Interfaces de notifications

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

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

def waitNotif(self):
    “Returns a placeholder for the code to trigger on a notification”
```

Interfaces de notification

```
def notify(self):
    self.sendLine("notif")

def stopNotify(self):
    self.sendLine("stop_notif")

def waitNotif(self):
    self.d = defer.Deferred()
    return self.d
```

Du point de vue de l'utilisateur

```
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"
```

L'utilisateur doit interpréter les évenements à partir des données reçues

(l'auteur du protocole ne peut-il pas s'occuper du parsing/dispatch?)

Comment écrire un appel à une fonction qui génère **plusieurs** réponses?

API de notifications: 2 callbacks

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

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



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

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

L'utilisateur implémente les callbacks

```
class MyClient(Client):

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

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

L'utilisateur se concentre sur le traitement des évèments,

Le protocole peut évoluer sans mise à jour du code utilisateur

Parsing/dispatch par l'auteur du protocol

```
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)
```

Résumé

- Scinder les fonctions bloquantes entre *emission* et *callback* permet d'utiliser un reactor pour effectuer des requètes concurrentes,

- Le reactor: un syscall de supervision d'une liste de socket, chaque socket a ses callbacks associés grâce à une instance de Protocole,

- API **A**synchrone: pour les évènements (ex: les serveurs)
  API synchrone: pour les clients séquentiels

Questions?

Contact:

jeandaniel.browne@gmail.com

jdb.github.com/imap_idle.html

Credits: template de Emily Dirsh (Fedora Design Team) –