

business

collaboration

people

Twisted: il framework per la **tua** Internet

Alex Martelli



Questo talk e il suo pubblico

- siete esperti di reti TCP/IP e dei vari protocolli che le riguardano
- conoscete bene il linguaggio Python (o siete pronti a impararlo se val la pena)
- vorreste sviluppare e usare programmi di rete (server, clienti, proxy, ecc ecc) usando Python
- vi chiedete quale sia il modo migliore

Python e la rete: lib. standard

- Le librerie standard Python comprendono amplissime funzionalità di rete:
 - modulo socket (oggi con timeout, SSL, ...)
 - moduli server sincroni (opz. multithreading, ...)
 - moduli asincroni (asyncore/asynchat)
 - moduli cliente (urllib, urllib2, ftplib, nntplib, ...)
 - moduli per cgi, cookie, ...
 - moduli ausiliari per trattare dati che spesso girano in rete (xml, html, package email, ...)
- che altro potrà mai servirmi?!

Python e la rete: 3e parti

- basta googlare un minimo per trovare una miniera di altre cose (gratis!)...:
- Zope/Plone &c
- mod_pyton x collaborare con Apache
- Webware, Quixote, CherryPy, ...
- gestione RSS, forms (lato client), SSL...
- che **altro** potrà mai servirmi **ancora**?!

Twisted: un **killer-framework**

- architettura **asincrona** (come asyncore / asynchat) → scalabilità, prestazioni
- supporto di alto livello per l'asincronia
- le Design Pattern giuste per lavorare in rete (reactor, acceptor, ...) incarnate
- sfrutta appieno le capacità di Python (dinamismo, introspezione, ...)
- tonnellate di funzionalità già pronta

Twisted: come iniziare?

- Twisted é un framework ricco → vasto
- digerire "tutto" Twisted → lavoro!
- una strategia preferibile...:
 - capire "a volo d'uccello" il panorama Twisted
 - afferrare gli elementi fondamentali
 - concentrarsi su quel che serve per 1 applicazione
 - imparare "un'applicazione alla volta"
- comunque indispensabile: capire Python, e la rete!-)



quantifichiamo i vantaggi...

- un tipico web-server Twisted ha fatto registrare numeri del tipo:
- oltre 3000 utenze simultanee
- oltre 2 milioni di pagine al giorno
- uptime oltre 99.99%
- su di un Pentium 150 MHz, 256MB RAM (con dischi SCSI, OS FreeBSD)

Twisted: bassissimi livelli

- il "perno": twisted.internet
 - supporta TCP, UDP, SSL, I/O ...
- la "piattaforma": interfacce, adattatori, componentizzazione
- le interfacce IReactor...
- le implementazioni di xxReactor:
 - default (con select)
 - specializzate x sistema (java, poll, win32, kq, cf, ...)
 - specializzate x gui toolkit (gtk, qt, wx -- pyui, tk, ...)

Twisted: la Deferred

- coordina "chiamate future" (una o +, anche "in cascata") che avverranno...:
 - quando i risultati saranno pronti
 - quando ci sarà un errore
 - dopo un certo timeout
- può coordinare una lista di callback
- fa da base per l'integrazione trasparente del supporto per processi e thread

Twisted: i Transport

- astraggono "una socket" (TCP, UDP, SSL)
- ma anche (e.g.):
 - "un processo con cui sto parlando direttamente"
 - "un wrapper attorno a un file"
 - "un loopback diretto fra cliente e servente"
 - ...
- altre interfacce ausiliarie imparentate:
 - (read/write)descriptor,consumer,(push/pull)producer

Twisted: i Protocolli

- interfacce IProtocol e IProtocolFactory
- marea di protocolli standard già pronti...:
 - dns, finger, ftp, gps (nmea), http, ident, imap, irc, jabber, msn, nntp, oscar (aim/icq), pop3, smtp, socksV4, telnet, RFC 862...868, ...
- supporto x scrivere altri protocolli:
 - ethernet, ip, icmp, tcp e udp 'crudi', seriali, ...
 - line-based, netstring, length+data, xml, ...
 - forwarding, throttling, load balancing, ...



Twisted: livelli superiori

- **marshaling, serializzazione, persistenza**
 - "Jelly", "Banana", Twisted.Persisted
- **programmazione distribuita**
 - Perspective Broker
- **autenticazione**
 - Twisted.Cred
- **interfacce a database**
 - Twisted.Enterprise



Twisted: livelli applicativi

- gestione DNS
 - Twisted.Names
- email, chat, SSH
 - Twisted.Mail, Twisted.Words, Twisted.Conch
- servizi web
 - Twisted.Web
 - Nevow, Freeform, LiveEvil

Twisted: oggetti "Servizio"

- **interfacce** `IService`, `IServiceCollection`
 - x implementare: classi `Service`, `MultiService`
 - già pronti ("precotti"): Client e Server per TCP, UNIX, SSL, UDP, `UNIXDatagram`, Multicast
 - interfacciano ai `listenTCP/connectTCP`, `listenUNIX/connectUNIX` eccetera dei vari Reactor
 - inoltre: `TimerService` (x chiamate periodiche)
- **microlinguaggio di factory** x facilitare:
 - "tcp:80:interface=192.168.3.1"
 - "ssl:443:privateKey=pvk.pem"

Twisted: oggetti "Applicazione"

- implementano 4 interfacce-chiave:
 - `IService`, `IServiceCollection`,
`IProcess`, `IPersistable`
- creati dalla factory `service.Application`
- persistibili e ricaricabili
 - anche in forma crittata
 - file TAC e TAP
 - utility `twistd`, `mkctap`, `tapconvert`, ...



Esempio: "echo" via .TAC

```
from twisted.[[pkg]] import [[modules]]

application = service.Application('echo')
factory = protocol.ServerFactory()
factory.protocol = wire.Echo
serv = internet.TCPServer(7, factory)
serv.setServiceParent(application)
```

Come si usa questo echo.tac

```
$ twistd --python echo.tac
```

- demonizza il processo (PID in `twistd.pid`)
 - mantiene un log su `twistd.log`

```
$ kill `cat twistd.pid`
```

- termina il processo (e logga il fatto)
 - salva lo stato in `echo-shutdown.tap`

```
$ twistd -r echo-shutdown.tap
```

- riparte dal punto d'interruzione

Il contenuto del file .tap

- reso leggibile da tapconvert (in .tas):

```
app=Ref(2, Instance('twisted.python.components.Componentized',{
'twisted.python.components.Componentized.persistenceVersion':1,
  '_adapterCache':{'twisted.application.service.IServiceCollection':
Ref(1, Instance('twisted.application.service.MultiService',
  name='echo', namedServices={}, parent=None,
services=[Instance('twisted.application.internet.TCPServer',
  args=(7,Instance('twisted.internet.protocol.ServerFactory',numPorts=0,
protocol=Class('twisted.protocols.wire.Echo')),),),
  kwargs={}, parent=Deref(1),),],)),
'twisted.persisted.sob.IPersistable':
Instance('twisted.persisted.sob.Persistent',
  name='echo', original=Deref(2), style='source',),
'twisted.application.service.IProcess':
Instance('twisted.application.service.Process',gid=501, uid=501,),
'twisted.application.service.IService':Deref(1), }, }))
```

- → codice Python, ricostruisce lo stato

I formati di persistenza....:

- TAC: source Python x startup
- TAS: source Python "ricostruito"
- TAX: formato XML
- TAP: pickle (+ conciso e rapido)
- costruibili/arricchibili con: `mktao`
- convertibili fra loro con: `taoconvert`
- eseguibili con: `twistd`
 - inoltre, solo x Linux: `tao2deb`, `tao2rpm`

"eco" customizzato mioeco.py

```
from twisted.protocols import basic

class EcoPerRiga(basic.LineReceiver):
    def lineReceived(self, riga):
        print self.transport.getPeer(), riga
        self.sendLine(riga)

class Fabbrica(protocol.ServerFactory):
    protocol = EcoPerRiga
```

Nota: DP "2-phase init"

- non fare lavoro "serio" in `__init__`
- lavorare invece nel "metodo di partenza" tipico di una classe, tipo:
 - `connectionMade` per un `Protocol`
 - `startFactory` per una `Factory`
 - ...
- vantaggi tipici come in altri FW
 - controllo + articolato di sequenza, persistenza...
 - si evitano circoli viziosi...



Un TAC x l'eco customizzato

```
import mioeco
```

```
application=service.Application('mioeco')  
factory=mioeco.Fabbrica()  
serv=internet.TCPServer(7777, factory)  
serv.setServiceParent(application)
```

Il log dell'eco customizzato:

```
2004/04/24 17:08 CEST [-] mioeco.Fabbrica starting on  
7777
```

```
...
```

```
2004/04/24 17:08 CEST [EcoPerRiga,0,127.0.0.1]  
IPv4Address(TCP, '127.0.0.1', 49514) una riga
```

- **NB:** l'uso di moduli è **necessario** perché la persistenza dello stato sia corretta



Twisted e i thread

```
from twisted.python import threadable
threadable.init()

def lineReceived(self, riga):
    reactor.callInThread(self.lavora, riga)
def lavora(self, riga):
    time.sleep(5.0 + 5.0*random.random())
    reactor.callFromThread(
        self.sendLine, riga)
```

Twisted, threads e deferred

```
from twisted.internet import threads

def lineReceived(self, riga):
    d = threads.deferToThread(self.wrk, riga)
    d.addCallback(self.sendLine)

def wrk(self, riga):
    time.sleep(5.0 + 5.0*random.random())
    return riga
```

Twisted.Web: concetti base

- una `Resource` é un singolo "segmento" della `path-URL` (interfaccia `IResource`)
 - navigazione per "child"
 - arrivati alla "leaf" → "rendering" (produce HTML, o scrive sulla `Request`)
- un `site` fabbrica un `HTTPChannel` e contiene una `Resource` "radice"
- una `session` identifica una sessione di browser (ci si giunge dalla `Request`)

Twisted.Web: oggetti Site

- `twisted.web.server.Site` → protocol factory per HTTP con una `Resource` detta "root" (radice) del sito
- creabili in automatico con `mktap` (su di una `Resource` radice su disco)
- creabili manualmente istanziando `Site` con una qualsiasi `Resource` radice

Twisted.Web: "risorse"

- interfaccia `IResource`
- metodo `getChild` per "navigare"
 - `putChild` per "popolare" staticamente
- giunti alla foglia (e.g. `isLeaf`), metodo `render(request)` → torna HTML (o scrive sulla request stessa)
- un file `.rpy` sul percorso viene eseguito dinamicamente → definisce `resource`

metodo render di una Resource

- può tornare una stringa di HTML
- può schedulare `request.write` e `request.finish` (via `deferred`) e tornare la costante `server.NOT_DONE_YET`
- (può anche effettuare `write` e `finish` direttamente invece che via `deferred`... sempre tornando `NOT_DONE_YET`)

Twisted.Web: "sessione"

- offre continuità a una sequenza di visite-a-pagina dallo stesso browser
- internamente costruita con cookie
- per accedere alla sessione:
 - `s = request.getSession()`
- poi dinamicamente con normale Python:
 - `s.foo = 'bar'`
 - `foo = getattr(s, 'foo', None)`

Esempio di .rpy

```
class Ciao(resource.Resource):
    def render_GET(self, request):
        request.setHeader('cache-control',
            'no-cache, must-revalidate')
        return '<p>ciao da %s: %s</p>' % (
            request.prepath(), time.asctime())
resource=Ciao()
resource.putChild('', resource)
resource.putChild('foo', resource)
```

Come usare questo ci.rpy

```
$ mktap web --resource-script ci.rpy
```

```
$ twisted -f web.tap
```

```
$ browser http://localhost:8080
```

```
$ browser http://localhost:8080/foo/
```

- per usare altri percorsi: `putChild`
 - e/o: `isLeaf`, override di `getChild`, ...

Accesso alla sessione

- nel body di `render_GET`, ad es., ... :

```
sess = request.getSession()  
sess.nv = 1+getattr(sess, 'nv', 0)  
return '<p>visita n. %s</p>' % (  
    sess.nv)
```

Il templating: Nevow

- x la generazione di pagine via template HTML si usava `twisted.web.woven`
- oggi deprecato: usare invece `nevow`
 - progetto separato da Twisted
 - si appoggia su Twisted e ne dipende
- non ancora integrato in `mktao`: per ora, usare l'approccio diretto con `.tac`

business

collaboration

people

Risorse principali per Twisted e Nevow:

`www.twistedmatrix.com`

`www.nevow.com`

e x info su Python in genere:

`www.python.org`

Python in a Nutshell

`www.strakt.com`