Pyro - Python Remote Objects

Eine Bibliothek zum Erstellen verteilter Anwendungen unter Python

Oliver Burger

DHBW Stuttgart - Campus Horb

Zusammenfassung. In der hier vorliegenden Arbeit werden die Grundzüge der Programmierung mit der Pyro-Bibliothek zum Erstellen verteilter Anwendungen unter Python dargestellt. Hierbei soll das grundlegende Vorgehen der Programmierung dargestellt werden sowie die Architektur dieser Anwendungen betrachtet werden.

1 Einleitung

In der Vorlesung *Verteilte Systeme* wurde das *Remote Message Invocation* unter Java behandelt, einer Java-Bibliothek, die die Kommunikation zwischen verschiedenen Objekten über das Netwerk erlaubt. Diese Aufgabe erfüllt unter Python die *Python Remote Objects*-Bibliothek.

Das Projekt Pyro wurde 1998 begonnen, als Technologien wir *Java RMI* und *CORBA* in aller Munde waren und nichts vergleichbares in Python existierte. Bis zum Jahr 2010 wurde Pyro stetig weiterentwickelt und um neue Features ergänzt.
Dann wurde entschieden, eine komplette Re-Implementierung durchzuführen, da der alte Pyro-Code - der zu diesem Zeitpunkt in Version 3.10 vorlag - seine Wartbarkeit immer mehr einbüßte und nur schwer weiterentwickelt werden konnte.
Aus diesem Grund begann 2010 die Entwicklung an Pyro4, das mittlerweile eine genügend hohe Stabilität erreicht hat, um es als den neuen Hauptzweig von Pyro betrachten zu können. Der Funktionsumfang von Pyro4 umfasst zwar alle Funktionen, die Pyro-3.x auch geboten hat, es ist jedoch nicht API-kompatibel, so dass die Umstellung existierender Pyro-Anwendungen Code-Änderungen erfordert.

2 Grundlagen

Pyro nutzt die Python-Bibliothek *pickle* zur Serialisierung von Python-Objekten. Dies ist ein im Python-Umfeld bekanntes Sicherheitsproblem, da die Rückrichtung, also die Deserialisierung, das „unpickling“ beliebiger Daten es erlaubt, beliebigen Code in eine Anwendung einzuschleußen und auszuführen, was zu einer Kompromitierung des Systems führen kann.
Um diese Probleme mit *pickle* zu umgehen, setzt Pyro eigene Sicherheitsmechanismen ein. Diese werden später kurz beschrieben.
Auch ist zu beachten, dass Pyro den gesamten Objektbaum serialisiert und überträgt, auch wenn nur ein kleiner Teil davon tatsächlich vom Gegenüber benötigt wird. Hierauf muss bei der Implementierung von Pyro-basierten Anwendungen geachtet werden. So ergibt es evtl. Sinn eigens Datenobjekte zu schaffen, die dann mittels Pyro übertragen werden.

3 Ein Beispiel

Die Funktionsweise von Pyro wird nun anhand eines kleinen Beispieles vorgestellt:
Die Anwendung besteht aus zwei Teilen, einem Server und einem Client, der Server läuft hierbei im Hintergrund und wartet auf eine Client-Anfrage.

Listing 1.1. simpleGreeting-Server

```
1  import Pyro4
2
3  class GreetingMaker(object):
4      def get_fortune(self, name):
5          return "Hello, {0}. Here is your fortune message:\n" \
6             "Behold the warranty -- the bold print giveth and the " \
7             "fine print taketh away.".format(name)
8
9  greeting_maker=GreetingMaker()
10
11 # create a Pyro daemon
12 daemon=Pyro4.Daemon()
13 # register the greeting object as a Pyro object
14 uri=daemon.register(greeting_maker)
15
16 # print the uri so we can use it in the client later
17 print "Ready. Object uri =", uri
18 # start the event loop of the server to wait for calls
19 daemon.requestLoop()
```

Der dazugehörige Client ist genauso einfach aufgebaut:

Listing 1.2. simpleGreeting-Client

```
1  import Pyro4
2
3  # ask for the uri of the remote object
4  uri=raw_input("What is the Pyro uri of the greeting object? ") \
```

```
  .strip()
name=raw_input("What is your name? ").strip()

# get a Pyro proxy to the greeting object
greeting_maker=Pyro4.Proxy(uri)

# call method normally
print '\n'+greeting_maker.get_fortune(name)+'\n'
```

Auf dem Server wird zuerst ein Daemon gestartet, der die notwendigen Dienste bereit stellt. Danach wird das zu übertragende Objekt bei diesem Daemon registriert und die „Request Loop“ gestartet, der Daemon wartet nun also auf die Anfrage eines Clients.
Der Client holt sich zuerst einen Proxy zu dem übertragenen Objekt, danach kann die Methode dieses Objektes genau so aufgerufen werden, wie dies bei einem lokalen Objekt der Fall wäre.

4 Benutzung eines Nameservers

In einer praktischen Umsetzung, besonders, wenn die Kommunikation zwischen dem Client und dem Server ohne menschliches Zutun geschehen soll, ist es natürlich nicht machbar, die URI des Objektes abzufragen. Woher sollte der Client diese auch kennen.
Für diesen Zweck gibt es einen Nameserver innerhalb von Pyro, der das manuelle Eintragen der URI im Client unnötig macht. dieser Nameserver wird auf dem Server mit dem Befehlsaufruf „python -m Pyro4.naming“gestartet und steht dann beliebig vielen Pyro-Anwendungen zur Verfügung. Wenn man mehrere Pyro-Anwendungenauf seinem System betreibt, ist es am einfachsten, den Start des Nameservers vom Init-Prozess des Betriebssystems als System-Daemon ausführen zu lassen. Folgende Änderungen an Server und Client sind nun notwendig:

Listing 1.3. Server-Änderungen (Diff)

```
--- simpleGreeting.py   2012-05-13 12:57:00.130317847 +0200
+++ greeting.py 2012-05-13 13:04:01.545071905 +0200
@@ -10,10 +10,14 @@

 # create a Pyro daemon
 daemon=Pyro4.Daemon()
+# find the name server
+ns=Pyro4.locateNS()
 # register the greeting object as a Pyro object
 uri=daemon.register(greeting_maker)
+# register the object with a name in the name server
+ns.register("example.greeting", uri)

 # print the uri so we can use it in the client later
-print "Ready. Object uri =", uri
+print "Ready."
 # start the event loop of the server to wait for calls
```

```
18  daemon.requestLoop()
```

Listing 1.4. Client-Änderungen (Diff)

```
1  --- simpleClient.py 2012-05-13 12:57:53.717668952 +0200
2  +++ client.py   2012-05-13 13:05:30.995654686 +0200
3  @@ -1,13 +1,9 @@
4   import Pyro4
5
6  -# ask for the uri of the remote object
7  -uri=raw_input("What is the Pyro uri of the greeting object? ") \
8  -    .strip()
9  -
10  name=raw_input("What is your name? ").strip()
11
12 -# get a Pyro proxy to the greeting object
13 -greeting_maker=Pyro4.Proxy(uri)
14 +# use name server object lookup uri shortcut
15 +greeting_maker=Pyro4.Proxy("PYRONAME:example.greeting")
16
17  # call method normally
18  print '\n'+greeting_maker.get_fortune(name)+'\n'
```

Der vergebene Name muss natürlich eindeutig gewählt werden, um keine Kollisionen zwischen verschiedenen Pyro-Anwendungen zu erhalten.

5 Sicherheitsaspekte

Wie bereits im Grundlagen-Kapitel erwähnt, ist die Nutzung von pickle für die Serialisierung und Deserialisierung der Python-Objekte ein potentielles Sicherheitsrisiko.
Um dem entgegenzuwirken setzt Pyro verschiedene eigene Sicherheitsmechanismen ein:

- Standardmäßig werden Pyro-Server nur an localhost gebunden, damit soll verhindert werden, dass Angriffe von außerhalb möglich sind. Für den Fall, dass man eine tatsächliche Netzwerkkommunikation nutzen möchte, kann man dieses Verhalten mit einer Umgebungsvariable unterbinden, sollte dann aber andere Sicherheitsmechanismen einsetzen.
- Pyro nutzt keine eigenen Verschlüsselungsmechanismen. Man sollte daher darauf achten, die zu übertragenden Daten vorher zu verschlüsseln, oder die Pyro-Kommunikation mit anderen Maßnahmen wie einem VPN oder einem ssh-Tunnel abzusichern.
- Es wird empfohlen, HMAC-Signaturen im Netzwerk zu nutzen. Hiermit wird sichergestellt, dass sich nur rechtmäßige Clients mit einem Server verbinden können. Hierzu wird ein Private-Key-/Public-Key-Verfahren genutzt. Es wird jedoch explizit darauf hingewiesen, dass man selbst dafür sorgen muss, dass der Private-Key wirklich geheim bleibt.

– Generell sollte darauf geachtet werden, mit welchen Rechten und unter welchem Benutzer die Server-Anwendung läuft. Es ist ratsam, hier einen eigenen Server-Benutzer zu erstellen, der außerhalb seines eigenen Verzeichnisses keine Berechtigungen hat, ähnlich wie das üblicherweise unter Linux bei anderen Server-Anwendungen wie zum Beispiel einem Webserver durchgeführt wird.

Literaturverzeichnis

1. http://packages.python.org/Pyro4/, 13.05.2012