Lucrarea nr. 3
Aplicaţii de tip server

Mihai IVANOVICI

20 martie 2006

Scopul acestei lucrări este de a vă familiariza cu arhitectura şi modul de implementare al unei aplicaţii de tip server.

1 Pregătirea pentru conexiune

Pentru un server, procesul stabilirii unei conexiuni presupune patru paşi:

1. Crearea obiectului socket,
2. Setarea opţiunilor pentru socket (etapă opţională),
3. Legarea la un anumit port (şi opţional, la o interfaţă de reţea,
4. Ascultarea / aşteptarea de conexiuni de la clienţi.

Exemplu:

```
host = ''      #Bind to all interfaces
port = 55999

#Pasul 1
s = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

#Pasul 2
s.setsockopt( socket.SOL_SOCKNET, socket.SO_REUSEADDR, 1 )

#Pasul 3
s.bind((host, port))

#Pasul 4
s.listen(3)
```

Pentru servere de uz general, cea mai utilizată opţiune este `SO_REUSEADDR`. De obicei, după ce un server îşi încetează existenţa, sistemul de operare nu permite reutilizarea portului pentru câteva minute, pentru a preveni folosirea aceluiaşi port de către alte procese sau chiar alte instanţe ale serverului. Dacă opţiunea este setată TRUE, atunci sistemul de operare poate realoca portul imediat ce socketul serverului este închis sau procesul server se termină.

Primul parametru al funcţiei `setsockopt` defineşte setul de opţiuni care este folosit, care de regulă este `SOL_SOCKET`, indicând faptul că setul de opţiuni se referă la socket.

Opţiune	Comportament	Valoare
SO_BINDTODEVICE	Socketul va fi valid pentru o anumită interfaţă	Şir de caractere
SO_BROADCAST	Permite transmiterea şi recepţionarea de pachete la o adresă de broadcast. Valabil doar pentru UDP	Boolean
SO_DONTROUTE	Interzice trecerea pachetelor printr-un router sau gateway. Folosită pentru comunicaţii UDP într-o reţea locală, ca o măsură de securitate	Boolean
SO_KEEPALIVE	Permiterea transmiterea de packete keep-alive pentru conexiuni TCP. Aceste pachete permit verificarea faptului că o conexiune rămâne deschisă chiar şi atunci când nu se transmit date	Boolean
SO_REUSEADDR	Portul devine imediat reutilizabil după închiderea socket-ului	Boolean

Table 1: Opţiuni din setul SOL_SOCKET.

Alte opţiuni pot fi disponibile, în funcţie de sistemul de operare.

Următorul program tipăreşte pe ecran opţiunile disponibile suportate de varianta de Python instalată:

```
#Get lisy of available socket options - sockopts.py

import socket

solist = [x for x in dir(socket) if x.startswith( 'SO_' )]
solist.sort()
```

```
for i in solist:
   print i
```

Pentru a lega un socket la o anumită adresă IP, de exemplu 192.168.1.1, şi la portul HTTP standard, apelul funcţiei `bind` va fi următorul:

```
s.bind(('192.168.1.1', 80))
```

2 Acceptarea conexiunilor

Cele mai multe servere sunt proiectate să ruleze într-o buclă infinită şi să deservească mai multe conexiuni. Clienţii de obicei realizează câteva conexiuni

De regulă, aplicaţiile de tip server rulează în mod continuu într-o buclă infinită, ca în exemplul de mai jos:

```
#Basic server - basicserver.py

import socket

host = ''#bind to all interfaces
port = 55999

s = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
s.setsockopt( socket.SOL_SOCKET, socket.SO_REUSEADDR, 1 )
s.bind((host, port))
print "Waiting for connections..."
s.listen(1)

while 1:
   clientsock, clientaddr = s.accept()
   print "Got connection from", clientsock.getpeername()
   clientsock.close()
```

Buclele infinite consumă timp de procesor, ocupând resursele sistemului. În exemplul de mai sus, deşi bucla `while` rulează la infinit, când este apelată funcţia `accept()` se va reveni din funcţie doar după ce un client s-a conectat la server. Între timp, programul este suspendat ("stalled") şi nu consumă timp de procesor. Un program care este suspendat aşteptând o cerere se numeşte că este "blocat".

Pentru a testa exemplul de mai sus, puteţi utiliza comanda `telnet` a sistemului de operare. Mai întâi porniţi serverul într-un terminal. În alt terminat tastaţi "`telnet localhost 55999`". Serverul ar trebui să tipărească

pe ecran informaţii despre conexiunea pe care a creat-o cu clientul: adresa IP a acestuia (în cazul nostru adresa IP de loopback) şi portul folosit. În funcţie de varianta de telnet, acesta va afişa "Connection reset by peer" sau "Connection closed" sau programul se va termina fără a afişa vreun mesaj, ceea ce este normal deoarce serverul simplu prezentat nu face nimic după stabilirea conexiunii.

3 Tratarea erorilor

Orice excepţie netratată va duce la terminarea programului Python. Pentru un client este în general acceptabil: este normal ca acesta să îşi înceteze execuţia în cazul apariţiei unei erori. Pentru un server, însă, acest comportament este inadmisibil: imaginaţi-vă ce s-ar întâmpla dacă un server de web s-ar opri de fiecare dată când apare o eroare generată de un client (de exemplu când un utilizator apasă butonul Stop al browserului).

Prin urmare, orice eroare trebuie tratată. Următorul exemplu încearcă să trateze cât mai multe din erorile ce pot apărea, în aşa fel încât serverul să îşi continue execuţia. Erorile ce apar la apelul funţiei `accept()` sunt tipările doar şi ignorate: instrucţiunea `continue` va face ca execuţia programului să revina la următorul apel al funcţiei `accept()`. Cu combinaţia Ctrl-C sau Ctrl-Break se poate întrerupe execuţia serverului.

```
#Server With Error Handling - errorserver.py

import socket, traceback

host = ''
port = 55999

s = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
s.setsockopt( socket.SOL_SOCKET, socket.SO_REUSEADDR, 1 )
s.bind((host, port))
s.listen(1)

while 1:
   print "Server running"
   try:
      clientsock, clientaddr = s.accept()
   except KeyboardInterrupt:
      raise
   except:
      traceback.print_exc()
      continue
```

```
#Process the connection
try:
   print "Got connection from", clientsock.getpeername()
   #Process the request here
except (KeyboardInterrupt, SystemExit):
   raise
except:
   traceback.print_exc()

#Close the connection
try:
   clientsock.close()
except KeyboardInterrupt:
   raise
except:
   traceback.print_exc()
```

Observaţie: În programele Python de dimensiuni mai mari decât exemplul de mai sus, se recomandă folosirea de blocuri `try... finaly` pentru a ne asigura de faptul că socketul a fost închis. Înainte de ultimul apel `close()` se poate folosi o instrucţiune `finaly` pentru a închide socketul. Astfel, orice excepţie netratată care poate apărea între `try` şi `finally` va duce la închiderea socketului. Totuşi această excepţie va duce la terminare programului.

Pentru clienţi este important apelul funcţiei `shutdown()` (vezi Lucrarea nr. 2), dar pentru servere nu este necesar. Dacă apare vreo eroare, clientul este deconectat, iar serverul îşi va continua execuţia, ignorând eroarea.

4 Folosirea protocolului UDP

Din perspectiva clientului, folosirea protocolului UDP este mai dificilă decât cea a protocolului TCP, deoarece acesta (clientul) trebuie să se ocupe de recuperarea pachetelor pierdute. Din punctul de vedere al serverului, folosirea protocolului UDP este simplă.

De regulă, cele mai multe comunicaţii UDP presupun o cerere din partea clientului şi un răspuns din partea serverului: serverul nu poate face nimic pentru a detecta sau trata un eventual pachet pierdut. Responsabilitatea îrevine în întregime clientului.

Pentru a folosi UDP pentru un server, se crează socketul, se setează opţiunile acestuia şi se apelează funcţia `bind()`, la fel ca atunci când se foloseşte protocolul TCP. Nu mai este nevoie de apelurile funţiilor `listen()` sau `accept()`, ci doar de funţia `recvfrom()`. Aceasta returnează datele primite şi adresa şi portul clientului care a trimis datele. Deoarece UDP nu

se bazează pe conexiune, adresa şi portul sunt suficiente pentru a trimite înapoi clientului un raspuns.

În continuare este prezentat codul unui server "echo" simplu, ce poate fi folosit pentru testarea clienţilor UDP:

```
#UDP Echo Server - udpechoserver.py

import socket, traceback

host = ''#Bind to all interfaces
port = 55999

s = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
s.setsockopt( socket.SOL_SOCKET, socket.SO_REUSEADDR, 1 )
s.bind((host, port))

while 1:
   try:
      message, address = s.recvfrom( 8192 )
      print "Got data from", address
      #Echo it back
      s.sendto( message, address )
      print message
   except (KeyboardInterrupt, SystemExit ):
      raise
   except:
      traceback.print_exc()
```

Codul clientului este următorul:

```
#UDP Client Example - udp.py

import socket, sys

host = sys.argv[1]
textport = sys.argv[2]

s = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
try:
   port = int( textport )
except ValueError:
   #That didn't work. Look it up instead.
   port = socket.getservbyname( textport, 'udp' )
```

```
s.connect((host, port))
print "Enter a message for the server:"
data = sys.stdin.readline().strip()
s.sendall( data )

print "Waiting for reply. Press Ctr-C to stop."
while 1:
   buf = s.recv( 8192 )
   if not len( buf ):
      print "No reply"
      break
   print "The server replied: %s" % buf
```

Rulaţi serverul într-un terminal, tastând `python2.4 udpechoserver.py`, iar clientul într-un alt terminal, tastând `python2.4 udp.py localhost 55999`. Observaţi comportamentul celor două programe.

Modificaţi clientul sau serverul astfel încât unul dintre ele să doarmă pentru 10 secunde şi încercaţi să generaţi o eroare prin întreruperea cu Ctrl-C a unuia dintre ele, simulând pierderea sau neprimirea de către client a răspunsului dat de server.