LẬP TRÌNH SOCKET

CHATTING

Giảng viên: Mai Xuân Phú

Mục tiêu:

- *Hiểu và giải thích được các hàm cơ bản với lập trình socket trên nền tảng sử dụng ngôn ngữ python.*
- *Lập trình 1 ứng dụng đơn giản minh họa mô hình kết nối client-server.*
- *Lập trình 1 ứng dụng mô phỏng chat đa luồng: multi-client*

Nội dung chính:

- *Xây dựng ứng dụng chat bằng ngôn ngữ python, cho phép nhiều client kết nối đến server*

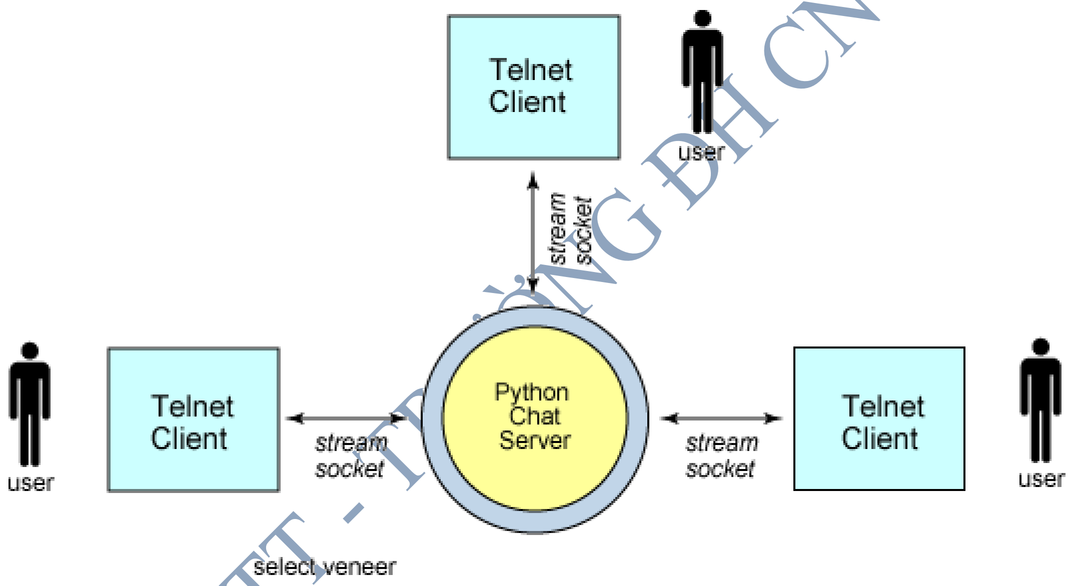
Building a Python chat server

(Source: https://www.ibm.com/developerworks/linux/tutorials/l-pysocks/)

A simple chat server

You've explored the basic networking APIs for Python; now you can put this knowledge to use in a simple application. In this section, you'll build a simple chat server. Using Telnet, clients can connect to your Python chat server and globally communicate with one another. Messages submitted to the chat server are viewed by others (in addition to management information, such as clients joining or leaving the chat server). This model is shown graphically in Figure 1.

Figure 1. The chat server uses the select method to support an arbitrary number of clients



An important requirement to place on your chat server is that it must be scalable. The server must be able to support an arbitrary number of stream (TCP) clients.

To support an arbitrary number of clients, you use the select method to asynchronously manage your client list. But you also use a feature of select for your server socket. The read event of select determines when a client has data available for reading, but it also can be used to determine when a server socket has a new client trying to connect. You exploit this behavior to simplify the development of the server.

Next, I'll explore the source of the Python chat server and identify the ways in which Python helps simplify its implementation.

The ChatServer class

Let's start by looking at the Python chat server class and the __init__ method -- the constructor that's invoked when a new instance is created.

The class is made up of four methods. The run method is invoked to start the server and permit client connections. The broadcast_string and accept_new_connection methods are used internally in the class and will be discussed shortly.

The __init__ method is a special method that's invoked when a new instance of the class is created. Note that all methods take the self argument, a reference to the class instance itself (much like the this parameter in C++). You'll see the self parameter, part of all instance methods, used here to access instance variables.

The __init__ method creates three instance variables. The port is the port number for the server (passed in the constructor). The srvsock is the socket object for this instance, and descriptors is a list that contains each socket object for the class. You use this list within the select method to identify the read event list.

Finally, Listing 16 shows the code for the __init__ method. After creating a stream socket, the SO_REUSEADDR socket option is enabled so that the server can be quickly restarted, if necessary. The wildcard address is bound with the defined port number. Then the listen method is invoked to permit incoming connections. The server socket is added to the descriptors list (the only element at present), but all client sockets will be added as they arrive (see accept_new_connection). A salutation is provided to stdout indicating that the server has started.

Listing 16. The ChatServer class with the `init` method

```
import socket
import select

class ChatServer:

  def __init__( self, port ):
    self.port = port;

    self.srvsock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
    self.srvsock.setsockopt( socket.SOL_SOCKET, socket.SO_REUSEADDR, 1 )
    self.srvsock.bind( ("", port) )
    self.srvsock.listen( 5 )

    self.descriptors = [self.srvsock]
    print 'ChatServer started on port %s' % port

  def run( self ):
    ...

  def broadcast_string( self, str, omit_sock ):
    ...

  def accept_new_connection( self ):
    ...
```

The run method

The run method is the server loop for your chat server (see Listing 17). When called, it enters an infinite loop, providing communication between connected clients.

The core of the server is the select method. I pass the descriptor list (which contains all the server's sockets) as the read event list to select (and null lists for write and exception). When a read event is detected, it's returned as sread. (I ignore the swrite and sexc lists.) The sread list contains the socket objects that will be serviced. I iterate through the sread list, checking each socket object found.

The first check in the iterator loop is if the socket object is the server. If it is, a new client is trying to connect and the accept_new_connection method is called. Otherwise, the client socket is read. If a null is returned from recv, the peer socket closed.

In this case, I construct a message and send it to all connected clients, close the peer socket, and remove the corresponding object from the descriptor list. If the recv return is not null, a message is available and stored in str. This message is distributed to all other clients using broadcast_string.

Listing 17. The chat server `run` method is the core of the chat server

```
def run( self ):

  while 1:

    # Await an event on a readable socket descriptor
    (sread, swrite, sexc) = select.select( self.descriptors, [], [] )

    # Iterate through the tagged read descriptors
    for sock in sread:

      # Received a connect to the server (listening) socket
      if sock == self.srvsock:
        self.accept_new_connection()
      else:

        # Received something on a client socket
        str = sock.recv(100)

        # Check to see if the peer socket closed
        if str == '':
          host,port = sock.getpeername()
          str = 'Client left %s:%s\r\n' % (host, port)
          self.broadcast_string( str, sock )
          sock.close
          self.descriptors.remove(sock)
        else:
          host,port = sock.getpeername()
          newstr = '[%s:%s] %s' % (host, port, str)
          self.broadcast_string( newstr, sock )
```

Helper methods

The two helper methods in the chat server class provide methods for accepting new client connections and broadcasting messages to the connected clients.

The accept_new_connection method (see Listing 18) is called when a new client is detected on the incoming connection queue. The accept method is used to accept the connection, which returns the new socket object and remote address information. I immediately add the new socket to the descriptors list, then send a salutation to the new client welcoming the client to the chat. I create a string identifying that the client has connected and broadcast this information to the group using broadcast_string (see Listing 19).

Note that in addition to the string being broadcast, a socket object is also passed. The reason is that I want to selectively omit some sockets from getting certain messages. For example, when a client sends a message to the group, the message goes to the group but not back to itself. When I generate the status message identifying a new client joining the group, it goes to the group but not the new client. This task is performed in broadcast_string with the omit_sock argument. This method walks through the descriptors list and sends the string to all sockets that are not the server socket and not omit_sock.

Listing 18. Accepting a new client connection on the chat server

```
def accept_new_connection( self ):

  newsock, (remhost, remport) = self.srvsock.accept()
  self.descriptors.append( newsock )

  newsock.send("You're connected to the Python chatserver\r\n")
  str = 'Client joined %s:%s\r\n' % (remhost, remport)
  self.broadcast_string( str, newsock )
```

Listing 19. Broadcasting a message to the chat group

```
def broadcast_string( self, str, omit_sock ):

  for sock in self.descriptors:
    if sock != self.srvsock and sock != omit_sock:
      sock.send(str)

  print str,
```

Instantiating a new `ChatServer`

Now that you've seen the Python chat server (under 50 lines of code), let's see how to instantiate a new chat server object in Python.

Start the server by creating a new ChatServer object (passing the port number to be used), then calling the run method to start the server and allow incoming connections:

Listing 20. Instantiating a new chat server

```
myServer = ChatServer( 2626 )
myServer.run()
```

At this point, the server is running and you can connect to it from one or more clients. You can also chain methods together to simplify this process (as if it needs to be simpler):

Listing 21. Chaining methods

```
myServer = ChatServer( 2626 ).run()
```

which achieves the same result. I'll show the ChatServer class in operation.

Demonstrating the ChatServer

Here's the ChatServer in action. I show the output of the ChatServer (see Listing 22 and the dialog between two clients (see Listing 23 and Listing 24). The user-entered text appears in bold.

Listing 22. Output from the `ChatServer`

```
[plato]$ python pchatsrvr.py
ChatServer started on port 2626
Client joined 127.0.0.1:37993
Client joined 127.0.0.1:37994
[127.0.0.1:37994] Hello, is anyone there?
[127.0.0.1:37993] Yes, I'm here.
[127.0.0.1:37993]  Client left 127.0.0.1:37993
```

Listing 23. Output from Chat Client #1

```
[plato]$ telnet localhost 2626
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
You're connected to the Python chatserver
Client joined 127.0.0.1:37994
[127.0.0.1:37994] Hello, is anyone there?
Yes, I'm here.^]
telnet> close
Connection closed.
[plato]$
```

Listing 24. Output from Chat Client #2

```
[plato]$ telnet localhost 2626
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
You're connected to the Python chatserver
Hello, is anyone there?
[127.0.0.1:37993] Yes, I'm here.
[127.0.0.1:37993] Client left 127.0.0.1:37993
```

As you see in Listing 22, all dialog between all clients is emitted to stdout, including client connect and disconnect messages.

Reference:

1. Charles Severance. *Python for Informatics - Exploring Information*. 2013
2. https://www.ibm.com/developerworks/linux/tutorials/l-pysocks/