HOW TO DEVELOP SOCKET-BASED APPLICATIONS USING PYTHON

Introduction

Networking is the concept of two programs communicating across a network. Can be client-client,server-server or client-server. A client is an end device interfacing with a human(user). A server is a device providing a service for clients.

There are two models for networking:

- Client-Server module, is the most common because the server runs constantly & is available for clients to access/receive information at any time(Web browser-client connects to Google website-server).
- Peer-to-peer module, has a more complex setup and is useful for services that don't require to be constantly available,clients connect to other clients without central server at times, suitable for real-time programs like Skype & Game servers.

SOCKETS = These are programming concepts for connections among programs in the network. They allow communication to be bidirectional. Once connection is established, transmission of data is ready. Data is sent & received using sockets, TCP & UDP transport protocols are implemented.

Languages that are socket-based include Python, Ruby & Scheme. Python has helper classes and simplifies sockets programming.

BASIC PYTHON SOCKETS MODULES

Python offers two basic sockets modules: SOCKET which provides low-level interface to the basic sockets API, and SOCKETSERVER which provides server-centric class that simplifies the development of network servers. This is done in an Asynchronous way in which you can provide plug-in classes to do the application specific jobs of the server.

The Socket Module

This provides the basic networking services and has all that is needed to build socket servers & clients. In python, the socket method returns a socket object to which the socket methods can be applied.

Class methods for the Socket module:

Socket = low-level networking interface

socket.socket(family, type) = create &return a new socket object

socket.getfqdn(name) = convert a string IP address into a domain name

socket.gethostbyname(hostname) = resolve a hostname to string IP address

socket.fromfd(fd, family, type) = create a socket from existing file descriptor4

Instance methods for the Socket module:

sock.bind((adrs, port)) = binds the socket address to the port

sock.accept() = return a client socket with peer address information

sock.listen(backlog) = place the socket into listening state,able to pend backlog connection request

sock.connect = ((adrs, port)) = connect the socket to the defined host & port

sock.recv(buflen[, flags]) = receive data from the socket,up to the buflen bytes

sock.recvfrom(buflen[, flags]) = receive data from socket up to buflen bytes, returning also the remote host & port from which the data came

sock.send(data[, flags]) = send the data through the socket

sock.sendto(data[, flags], addr) = send data through the socket

sock.close() = close the socket

sock.getsockopt(lvl, optname) = get the value of the specified socket option

sock.setsockopt(lvl, optname, val) = set the value of the specified sock option.

The difference between a class method and an instance method is that, the instance method requires a socket occurrence to be performed(which is returned from the socket) whereas a class method does not.

The SocketServer Module

This is an interesting module that simplifies the development of socket servers. Using the example of implementing a simple "Hello World" server which emits the message once the client is connects:

-we first create a request handler that inherits the **SocketServer.SteamRequestHandler** class.

-define a method called handle, that deals with requests for the server

-everything the server does must be handled within the context of this function

-one can also implement HTTP request with this class.

Once the connection handler is ready, we create the socket server. The **SocketServer.TCPServer** class provides the address & port number to which the server will be bound and the request-handler method. The result is a **TCPServer** object. The serve_forever method is called and starts the server, making it available for connections.

Code for the simple "Hello Word" server:

```
import SocketServer

class hwRequestHandler( SocketServer.StreamRequestHandler ):
    def handle( self ):
```

```
self.wfile.write("Hello World! \n")

server = SocketServer.TCPserver( ("", 255), hwRequestHandler)
server.serve_forever()
```

Creating and Destroying Sockets

To create a new socket, the socket method from the socket class is used. It is a class method because there isn't a socket object from which methods are applied yet. Creating a stream(TCP) and datagram(UDP) socket respectively:

```
streamSock = socket.socket( socket.AF_INET, socket.SOCK_STEAM)
dgramSock = socket.socket( socket.AF_INET, socket.SOCK_DGRAM)
```

In each case, a socket object is returned. The AF_INET indicates that a n IPv4 socket is requested. The SOCK_STREAM is the transport protocol TCP socket & SOCK_DGRAM is the UDP socket. If using IPv6, specify as socket.AF_INET6 to create an IPv6 socket.

To close a connected socket:

```
streamSock.close()
```

Finally to delete a socket, this permanently removes the socket object:

```
del streamSock
```

Socket Address

An end point for a socket is a tuple consisting of an interface address & port number, for example ('192.168.1.1', 80) or have a domain name like ('www.google.com', 80).

Server Sockets

These sockets expose a service to the network. Server sockets are created differently from client sockets. After creating a server socket, use the bind method to bind the address to it, the listen method places the server in a listening state and finally the accept method to accept a new client connection.

```
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock.bind( ('',2525) )
sock.listen( 5 )
newsock, (remhost, remport) = sock.accept()
```

For this example, the address('',2525) is used which means that the wildcard is used for the interface address(''), this allows incoming connections from any interface on the host. The address is bound to port number 2525. The accept method returns not only the new socket object that represents the client connection(newsock) but also an address tuple(the remote address & port number of the peer end of the socket). Python's SocketServer module simplifies this process.

Datagram servers can be created but since they are connectionless, there's no associated accept method.

```
sock = socket( socket.AF_INET, socket.SOCK_DGRAM )
sock.bind( ('',2525 ))
```

Client Socket

The mechanisms are similar to setting up the server sockets. Upon creating a socket, an address is needed to identify where the socket should attach. There's no binding as seen in server sockets. Say we have a server on a host with the interface IP address of '192.168.1.1' &

port number 2525. Below is the new socket created that connects to the specified server:

```
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock.connect( ('192.168.1.1', 2525) )
```

For datagram sockets, by nature are connectionless, they are message-based able to communicate with multiple peers at the same time. Datagram sockets need require destination information + information to be sent. Example of a datagram client that uses the connect method but in essence there is no real connection between the client & server is below:

```
sock = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
sock.connect( ('192.168.1.1', 2525) )
```

One can call connect again to respecify the target of the datagram client's message.

Stream Sockets I/O

In Python it is simple to send and receive data through stream sockets. Methods used to move data through a stream socket include: send, recv, read & write. We'll demonstrate a client and server for stream sockets:

-the server echoes whatever it receives from the client
-the echo stream server is represented in code
-after creating a new stream socket, an address is bound to it
-the listen method is invoked to enable incoming connections
-the echo server then goes into a loop for client connections
-the accept method is called & blocks until a new client connects, at this point the new client socket is returned with address information

for the remote client

—with the new client socket, the recv is called to get a string from the peer and writes string back out of the socket

—close the socket

```
import socket

srvsock =socket.socket( socket.AF_INET, socket.SOCK_STREAM)
srvsock.bind( ('', 23000) )
srvsock.listen( 5 )

while 1:
    clisock, (remhost, remport) = srvsock.accept()
    str = clisock.recv(100)
    clisock.send( str )
    clisock.close()
```

Below is the echo client that corresponds with the echo server. After making a new stream socket, the connect method is used to attach its socket to the server. When connected, the connect method returns and the client emits a simple message with the send method. The client then awaits the echo with the recv method, the print statement is used to display what's read. The close method performs the close of the socket:

```
import socket

clisock = socket.socket( socket.AF_INET, socket.SOCK_STREAM)
clisock.connect( ('', 23000) )

clisock.send("Hello World!\n")
print clisock.recv(100)
```

```
clisock.close()
```

Datagram Sockets I/O

These sockets are disconnected by nature which require a destination address to be provided for communication. When a message is received through a socket, the source of the data must be returned. The recvfrom & sendto methods support the additional address information in the datagram echo server and client implementations. Starting with the datagram echo server, a socket is first created and using the bind method it is bound to an address. An infinite loop is entered for serving client clients requests. The recvfrom method receives a message from the datagram socket and returns the message + source of the message. The sendto method returns the message to the source.

```
import socket

dgramSock =socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
dgramSock.bind( ('', 23000))

while 1:
    msg, (addr, port) = dgramSock.recvfrom( 100 )
    dgramSock.sendto(msg, (addr, port) )
```

The datagram client is simpler, after creating the datagram socket the sendto method sends a message to a specific address. When sending is complete, an echo response is awaited by the recv method which then prints it.

```
import socket
```

```
dgramSock=socket.socket( socket.AF_INET,socket.SOCK_DGRAM)
dgramSock.sendto('Hello World \n', ('', 23000) )
print dgramSock.recv( 100 )
dgramSock.close()
```

Socket Options

Sockets by default are set to certain standard behaviors. It is possible to alter the behavior using options. Socket options are manipulated with the setsockopt method & are captured be the getsockopt. It is simple to use sockets in Python, for example to read the size of the socket send buffer or to get the value of the SO_REUSEADDR option that is used within the TIME_WAIT period and enabling it:

```
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

#get size of send buffer
bufsize =sock.getsockopt (socket.SOL_SOCKET, SOCKET.SO_SNDBUF)

#get state of the SO_REUSEADDR option
state =sock.getsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR )

#enable the SO_REUSEADDR option
sock.setsockopt( socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

The SO_REUSEADDR option is most often used in socket server development. One can increase the socket send & receive buffers if required.

ASYNCHRONOUS I/O

As part of the select module, Python offers asynchronous I/O. The select method allows one to multiplex events for several sockets & for different events. For example one can instruct select method to notify when socket has data available, to know when it's possible to write data through a socket and when an error occurs on a socket. These actions can be performed at the same time.
Below shows awaiting input from stdin:

```
rlist, wlist, elist = select.select( [sys.stdin], [], [] )

print sys.stdin.read()
```

The arguments passed to select are lists representing read events, write events & error events. The select method returns 3 lists containing objects whose events are satisfied. Upon return rlist should be [sys.stdin] indicating that data is available to read on stdin, the data is read in read method.

The select method also works on socket descriptors. An example to show two client sockets are created & connected to a remote peer. A select method is then used to identify which socket has data available for reading. The data is then read & emitted to stdout.

```
import socket
import select

sock1 = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock2 = socket.socket( socket.AF_INET, socket.SOCK_STREAM)

sock1.connect( ('192.168.1.1', 25) )
sock2.connect( ('192.168.1.1', 25) )
```

```
while 1:
    #await a read event
    rlist, wlist, elist =select.select( [sock1, sock2], [], [], 5)

    #wait for timeout
    if [rlist, wlist, elist ] == [ [], [], [] ]:
        print "Five seconds elapsed.\n"

    else:
        #loop through each socket,read & print the data
        for sock in rlist:
            print sock.recv( 100 )
```

BUILDING A CHAT SERVER

A simple chat server can incorporate the basic Python networking APIs. Using Telnet, clients can connect to the Python chat server & globally communicate with one another. Messages submitted to the chat server are viewed by others, as well as additional management of information such as clients joining/leaving the chat server.

The chat server uses the select method to support an arbitrary number of clients. It is also important to place the chat server to be scalable in order to support an arbitrary number of stream(TCP) clients. This is done by using the select method to asynchronously manage the client list. The read event of select determines when a client has data available for reading and to determine when a server has anew client trying to connect. This behavior is exploited to simplify the development of the server.

The ChatServer Class

The class is made up of 4 methods: RUN method that is invoked to start the server & permit client connections, BROADCAST_STRING method and ACCEPT_NEW_CONNECTION method are used internally in the class, the __INIT__ method is special because it's invoked when a new instance of class is created. All the methods take the self argument,which is a class instance itself, access instances of variables.

The __init__ method creates 3 instance variables: PORT which is the port number for the server passed in the constructor, SRVSOCK is the socket object for this instance, DESCRIPTORS is a list that contains each socket object used within the select method to identify read event list.

In the __init__ method, after creating a stream socket the SO_REUSEADDR socket option is enabled so that server can be quickly restarted, the wildcard address is bound to defined port number, the listen method is invoked to permit incoming connections. The server socket is added to the descriptors list, as all client sockets are added as they arrive then a salutation is provided to stdout indicating server has started.

The run method is the server loop for the chat server and when called it enters an infinite loop, providing communication between clients. The core of the server is the select method, the descriptor list with all the server's sockets is passed as the read event list to select. When the read event is detected it's returned as sread, the sread list contains the socket objects that will be serviced. The first check of the loop is if the socket object is the server, if it is a new client trying to connect then the accept_new_connection method is called.

A message is constructed & sent to all connected clients, then peer socket is closed and corresponding object from the descriptor list

is removed. If the recv return is not null, a message is available & stored in str. The message is distributed to other clients using broadcast_string.

There are 2 helper methods in the chat server class that provide methods for accepting new client connections & broadcasting messages to the connected client. The accept_new_connection method is called when a new client is detected on incoming connection queue and returns a new socket object & remote address information. The new socket is added to the descriptors list. A string identifies that the client has connected & information is broadcasted. A socket object is passed with the string to omit some sockets from getting certain messages. A status message shows the new client joining the group and is performed in broadcast_string with the omit_sock argument.

Instantiating a new ChatServer is the next step, start the new ChartServer object and pass the port number to be used. Call the run method to start the server & allow incoming connections. Once the server is running it can be connected by one/more clients. The two methods are chained together.

Below shows the code for ChatServer named **pchatsrv.py**:

```
import socket
import select.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 'ChartServer started on port %s' % port

    def run( self ):
        while 1:
            #await event on readable descriptor
            (sread,swrite,sexc)= select.select(self.descriptors,
                                               [], [] )

            #iterate through the tagged read descriptors
            if sock == self.srvsock:
                self.accept_new_connection()
            else:
                #received something on client socket
                str = sock.recv(100)

                #check to see if peer socket is 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 )
```

```
        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

        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 )

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

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.