Python Network Programming

- Python offers two basic sockets modules. The first, **Socket**, provides the standard low-level BSD Sockets API. The second, **SocketServer**, is a high-level server-centric class that simplifies the development of network servers.

- Python's socket module supports socket programming on any system that supports BSD-style sockets.

Server socket calls

- The following are some of the socket calls used in the implementation of a typical server (these will require the socket module: "***from socket import ****")

- The first step is to create a socket:

  sockobj = socket(*addr_family, type*)

  - ***addr_family***

    AF_INET Internet protocol (IPv4)
    AF_INET6 Internet protocol (IPv6)

  - ***type***

    SOCK_STREAM Connection based stream (TCP)
    SOCK_DGRAM Datagrams (UDP)

  - The following will create an IPv4 TCP socket:

    sockobj = socket(AF_INET,SOCK_STREAM)

- The newly created socket must be bound to a IP and port:

  sockobj.bind((myHost, myPort))

  - **myHost**

    In server programs this argument is set to an empty string (""), which means that the application will run on the default localhost.

  - **myPort**

    This can be set to any high port (> 1024) that the server will listen on for inbound connections.

- Once the socket is bound to an IP/port pair, the server will execute the listen call to start listening for connections:

  sockobj.listen(backlog)

  - ***backlog***

    This parameter sets a limit of the maximum number of simultaneous connection requests in the processing queue.

- At this point the server is ready to accept connections from remote clients. This is done as follows (note that the ***accept()*** call is a blocking call):

  conn = sockobj.accept()

  - ***conn***

  The accept call will return a new socket (***conn***) which is used for further communications between the server and client.

- Once the connection has been established, the client and server can start exchanging data using a variety of calls:

  - For TCP connections we receive data using:

    data = conn.recv(buflen)

    This will read up to ***buflen*** bytes. Returns the **bytes** received.

  - For UDP connections we receive data using:

    data, addr = sockobj.recvfrom(buflen[, flags])

    This will read up to ***buflen*** bytes. Returns the **bytes** received, together with the remote host IP and port from which the data was received.

  - For TCP connections we send data using:

    conn.send(data)

    This will send up to ***data*** bytes on the connected/accepted socket.

- For UDP sockets we send data using:

  sockobj.sendto(data, (host, port))

  This will send a datagram containing up to ***data*** bytes to the host/port address.

Python also provides another method for sending which will block until all data is transmitted:

conn.sendall(data)

- Once the application have completed their communications the sockect on both sides must be closed:

  conn.close() # TCP connected socket
  sockobj.close() # UDP socket

- In Python 3, all strings are **Unicode**. Thus, if all text strings is that are to be sent across the network, must be encoded using the **encode('ascii')** method on the data it transmits.

- Likewise, when a client receives network data, that data is first received as raw unencoded bytes, which must be decoded before printing (**decode('ascii')).**

<u>Client socket calls</u>

- The following are some of the socket calls used in the implementation of a typical server (these will require the socket module: "***from socket import ****")

- The first step is to create a socket; this has already been covered in the previous section.

- The client has to initiate the connection (3-way handshake) first:

  sock.connect((serverHost, serverPort))

  - ***serverHost # remote server IP or name***
  - ***serverPort # remote server port***

- After the connection has been established the same calls as before are used to communicate back and forth between the two systems.

- For UPD there is no such thing as a connection so all the client has to do is execute a ***recvfrom*** call as described in the previous section.

Examples

- The first example (**echo-server.py**) is a very basic server application that listens for connections on port 8000 and simply echoes the strings send by a client application.

- The client application is provided (**echo-client.py**). The client application will be default connect to “localhost” and port 8000. Optionally the user can provide a remote IP and port number for the server. The client simply sends a default string to the server and then reads the echo back from the server.

- The second server example (**echo-server2.py**) is an example of a multi-threaded server which can accept connections from multiple clients. Each client request is serviced by a new thread.

- The UDP implementations of the echo client/server are much simpler as can be seen in the examples (**udps.py** and **udpc.py**).