Network Programing in Python

Lecture Notes: Compiled by: Mr. Daya Ram Budhathoki
Nepal Engineering College.

Table of Contents

Socket Introduction:

As an analogy, think of *TCP/IP* as the Postal Service. They pick up and leave mail in your mailbox, which is similar to a *port*. People who live in rural areas are always looking for friends, who might be traveling past the mailboxes, to shuttle their mail. A friend who carries a letter to your mailbox and delivers any mail waiting in your mailbox to you, is performing the same service that a *socket* does in relation to the application program and the port.

- A *socket* is a tool that carries data back and forth between a port and an application program, thus allowing the application to *read* from and *write* to the port, which in turns equates to sending and receiving data to and from the remote computer.
- A *socket* is an abstract resource of the operating system, which processes may request. We use the term *abstract* because it is not a piece of hardware. It is some data structures and instructions inside the operating system kernel.

Sockets

A logical endpoint for communication between two hosts on a TCP/IP network. A socket is also an application programming interface (API) for establishing, maintaining, and tearing down communication between TCP/IP hosts. Sockets were first developed for the Berkeley UNIX platform as a way of providing support for creating virtual connections between different processes

- API for applications to read and write data from TCP/IP or UDP/IP
- File abstraction (open, read, write, close)
- Abstract operating system resource
- First introduced with BSD UNIX
- De-facto standard API for TCP/IP

As an endpoint for network communication between hosts, a socket is uniquely identified by three attributes:

- The host's IP address
- The type of service needed—for example, a connectionless protocol such as User Datagram Protocol (UDP) or a connection-oriented protocol such as Transmission Control Protocol (TCP)
- The port number used by the application or service running on the host

Ports

The term port is overused in the computer world. In some cases it means something physical, such as a place into which you can plug your USB device. In our case here, though, ports are not physical. Instead, they are essentially tied to processes on a machine .

- The so called *well known ports* are those ports in the range of 0 to 1023 only the operating system or an Administrator of the system can access these. They are used for common services such as web servers (port 80) or e-mail servers (port 25).
- Registered Ports are in the range of 1024 to 49151.
- Dynamic and/or Private Ports are those from 49152 through 65535 and are open for use without restriction

Well-Known Ports" (0–1023) are for the most important and widely-used protocols. On many Unix-like operating systems, normal user programs cannot use these ports, which prevented troublesome undergraduates on multi-user machines from running programs to masquerade as important system services. Today the same protections apply when hosting companies hand out command- line Linux accounts.
"Registered Ports" (1024–49151) are not usually treated as special by operating systems—any user can write a program that grabs port 5432 and pretends to be a PostgreSQL database, for example—but they can be registered by the IANA for specific protocols, and the IANA recommends that you avoid using them for anything but their assigned protocol.

The remaining port numbers (49152–65535) are free for any use. They, as we shall see, are the pool on which modern operating systems draw in order to generate random port numbers when a client does not care what port it is assigned.

Types of Internet Socket:

Three types of Socket:

Stream Sockets (SOCK_STREAM) TCP

- Connection oriented
- Provides sequenced, and non duplicated flow of data
- Rely on TCP to provide reliable two-way connected communication .
- SOCK_STREAM socket type is implemented on the Transmission Control Protocol/Internet Protocol (TCP/IP) protocol.
- A stream socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to pipes.

Datagram Sockets (SOCK_DGRAM) UDP

- Rely on UDP
- Connection is unreliable
- This type of socket is generally used for short messages, such as a name server or time server, because the order and reliability of message delivery is not guaranteed.
- the SOCK_DGRAM socket type is implemented on the User Datagram Protocol/Internet Protocol (UDP/IP) protocol.
- A datagram socket supports the bidirectional flow of data, which is not sequenced, reliable, or

unduplicated. A process receiving messages on a datagram socket may find messages duplicated or in an order different than the order sent. Record boundaries in data, however, are preserved. Datagram sockets closely model the facilities found in many contemporary packet-switched networks

Raw sockets (or Raw IP sockets)

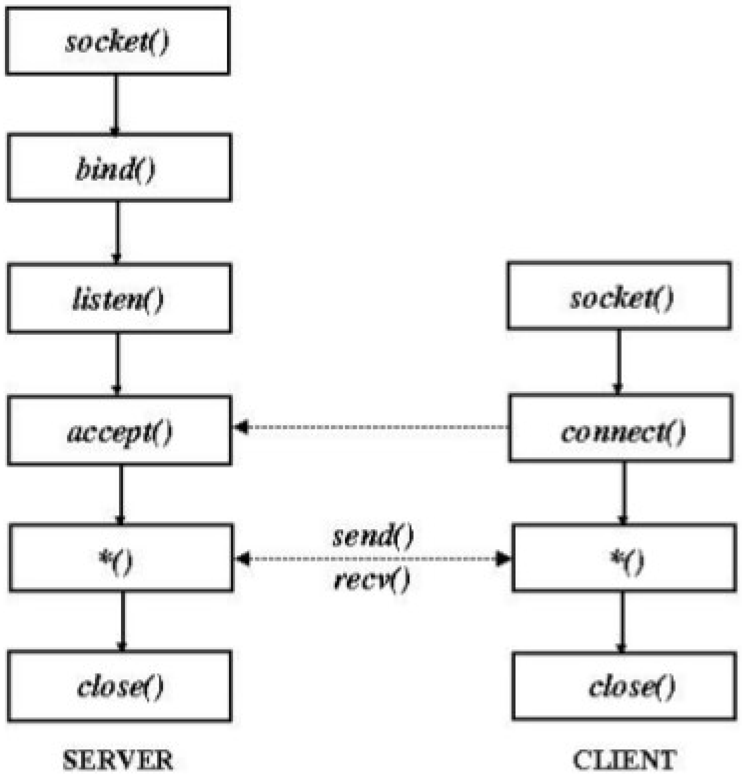
- Typically available in routers and other network equipment.
- Here the transport layer is bypassed, and the packet headers are not stripped off, but are accessible to the application.
- Application examples are Internet Control Message Protocol (ICMP, best known for the Ping suboperation), Internet Group Management Protocol (IGMP), and Open Shortest Path First (OSPF).

Why Python for Network Programing:

The Python socket module provides direct access to the standard BSD socket interface, which is available on most modern computer systems. The advantage of using Python for socket programming is that socket addressing is simpler and much of the buffer allocation is done for you. In addition, it is easy to create secure sockets and several higher-level socket abstractions are available.

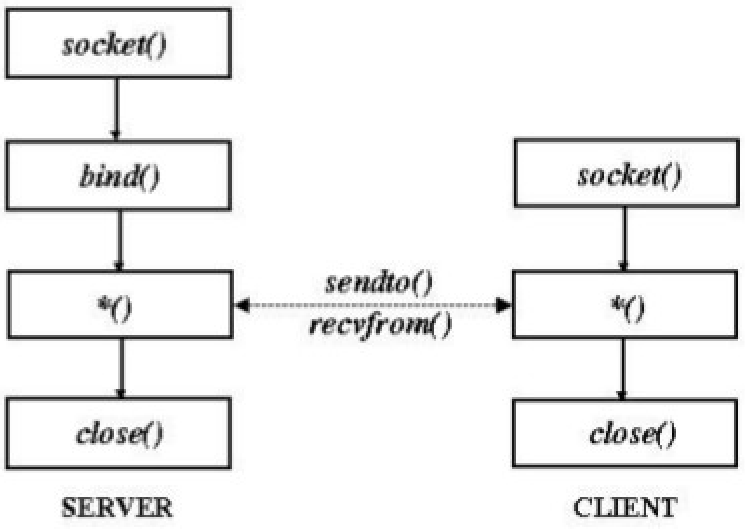
Socket Call

TCP Socket Call:



A stream connection is presented above. Observe how the processes interact: the server is started and must be in accept state before the client issue its request. The client's connect() method is trying to rendezvous with the server while this one is accepting connections. After the connection have been negotiated follows a data exchange and both sides call close() terminating the connection. Remember, this is a one connection diagram. In the real world, after creating a new connection (in a new process or in a new thread) the server return to accept state. *() functions are user-defined functions to handle a specific protocol. Data transfer is realized through send() and recv().

UDP Socket Call:



Python socket Module:

The `socket()` function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

To create a socket, you must use the socket.socket() function available in socket module, which has the general syntax:
s = socket.socket (socket_family, socket_type, protocol=0)
Here is the description of the parameters:
• socket_family: This is either AF_UNIX or AF_INET,
• socket_type: This is either SOCK_STREAM (Its create TCP socket) or SOCK_DGRAM.(udp)
• protocol: This is usually left out, defaulting to 0.
Once you have socket object, then you can use required functions to create your client or server program. Following is the list of functions required:

Server Socket Methods:

s.bind() This method binds address (hostname, port number pair) to socket.
s.listen() This method sets up and start TCP listener.
s.accept() This passively accept TCP client connection, waiting until connection arrives (blocking).

client socket methods:

s.connect() This method actively initiates TCP server connection.

General Socket methods:

Method	**Description**
s.recv()	This method receives a TCP message
s.send()	This method transmits TCP message.
s.recvfrom()	This method receives UDP message.
s.sendto()	This method transmits UDP message.
s.close()	This method close the socket.
socket.gethostname()	Returns the hostname.

Simple TCP Server and Client:

Here the Server will send a pre-defined message "Thank you for connecting" when a client connects to it.:

Server Program:

```
#!/usr/bin/python
import socket
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.bind(('localhost',8090))
s.listen(59)
print "The server is waiting for conneciton on port 8090"

while True:
  clientsocket,clientaddress=s.accept()
  print "Got the connection from", clientaddress
  clientsocket.send("Thank you for Connecting  \n")
  clientsocket.close()
```

When you telnet localhost once you run the server program above:

```
daya@daya-Aspire-4937:~$ telnet localhost 8090
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Thank you for Connecting
Connection closed by foreign host.
```

Writing separate client application instead of telnet client for above program:

```
#!/usr/bin/python
import socket
import sys
c=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
c.connect(('localhost', 8090))
data=c.recv(512)
print "Received data:", data
```

Socket API For Server Side:

First line in needed in Linux and is path to the python interpreter. Its not needed in the windows. second line , import socket. We import the socket class from Python's library; this contains all the communication methods we need.

Third line creates a socket.The two arguments state that we wish to the socket to be an Internet socket (socket.AF INET), and that it will use TCP (socket.SOCK STREAM), rather than UDP (socket.SOCK DGRAM). Note that the constants used in the arguments are attributes of the module socket, so they are preceded by 'socket.'; in C/C++, the analog is the #include file.

In the next line we specify the **bind functions,** it binds to a particular port in preparation to receive connections on this port. `bind()` takes one parameter, which is a two value tuple consisting of a host-name (string) and port number (integer).

the operating system will first check to see whether port 8090 is already in use by some other process. If so, an exception will be raised, but otherwise the OS will reserve port 8090 for the server. What that means is that from now on, whenever TCP data reaches this machine and specifies port 8090, that data will be copied to our server program. Note that bind() takes a single argument consisting of a two-element tuple, rather than two scalar arguments. so double brackets (('localhost', 8090))

The listen() method tells the OS that if any messages come in from the Internet specifying port 2000, then they should be considered to be requesting a connection to this socket.

The method's argument tells the OS how many connection requests from remote clients to allow to be pending at any give time for port 2000. The argument 1 here tells the OS to allow only 1 pending connection request at a time. We only care about one connection in this application, so we set the argument to 1. If we had set it to, say 5 (which is common), the OS would allow one active connection for this port, and four other pending connections for it. If a fifth pending request were to come it, it would be rejected, with a "connection refused" error. That is about all listen() really does.

We term this socket to be consider it a listening socket. That means its sole purpose is to accept connections with clients; it is usually not used for the actual transfer of data back and forth between clients and the server.
The accept() method tells the OS to wait for a connection request. It will block until a request comes in from a client at a remote machine. That will occur when the client executes a connect() call. In that call, the OS at the client machine sends a connection request to the server machine, informing the latter as to (a) the Internet address of the client machine and (b) the ephemeral port of the client

The accept method returns a tuple of a new allocated socket and the address of the client. The *address* is itself a tuple consisting of an IP address and port number.

Socket API for client side:

connect method:
socket.connect((hostname, portNumber)) Establish a connection to a server.
Example usage: sock.connect((hostname, portNumber))

- Connect to a server that is listening for connections
- The hostname may be either it's canonical name, or it's IP address.
- portNumber should match the port number that the sever was bound to.
- This is a blocking statement, until the connection is established.

The Socket API For Established Connections

socket.recv(N)
Receive up to N bytes from the socket. Block until a message is received on the socket, or the connection was closed.
Return type: string

Example usage:
message = sock.recv(1024)

socket.send(msg)

Send a string message using an established socket connection.
Parameters: msg (string) – The message to send

socket.close()
Close an established socket connection.

Example usage:
sock.close()
* Always a good idea to close sockets when finished with them.
* If the other end of the connection is blocked on socket.recv() when the socket is closed, it will return with a zero length message. see the note on detecting a closed connection and shutdown() below.

Building echo-server and echo-client programs"

TCP Echo Server and Client:

tcp-echoserver.py

```
#!/usr/bin/python
#A simple echo Server, send back the same message the client sent back to it.

import socket, sys

#Creat the socket
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.bind(('localhost',5090))
s.listen(5)
print "Server is running on port 5090:"

while True:
    clientsocket,clientaddress=s.accept()
    print "Received the connection from:", clientaddress
    while True:
        data=clientsocket.recv(512)
        if data=='q' or data=='Q':
            clientsocket.close()
            print "Client quits:"
            break
        else:
            print "Data received", data
            newdata="You Send:" +data
            clientsocket.send(newdata)
```

tcp-echoclient.py

```
#simple  tcp echo client
#!/usr/bin/python
import socket,sys
c=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
c.connect(('localhost', 5090))
while True:
	data=raw_input("Enter data to send to server: press q or Q to quit:\n")
	c.send(data)
	if data=='q' or data=='Q':
		c.close()
		break
	print c.recv(512)
```

UDP echo server and Client:

udp-echoserver.py

```
#UDP echo Server
#!/usr/bin/python
import socket
server_socket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
server_socket.bind(('localhost',5000))
print "UDP -Echo Server listening on port 5000:"
while True:
	data,address=server_socket.recvfrom(512)
	print address, ":said", data
	server_socket.sendto(data,address)
```

udp-echoclient.py

```
#!/usr/bin/python
import socket
client_socket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
while True:
	data=raw_input("Type Something(q or Q to exit:)")
	if data=='q' or data=='Q':
		client_socket.close()
		break;
	else:
		client_socket.sendto(data,("localhost", 5000))
		newdata=client_socket.recvfrom(512)
		print "Received:", newdata[0]
```

Catching Exceptions

In the previous examples, we have not checked for any exceptions that could be raised by methods in the socket module. One clear example is when the server tries to bind its socket to a particular port. Only one process is allowed to bind to each port; if this port is already being used then an exception is raised.

```
#!/usr/bin/env python
"""
A simple echo server that handles exceptions
"""
import socket,sys
host = ''
port = 50000
backlog = 5
size = 1024
s = None
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((host,port))
    s.listen(backlog)
    print "The server is listening on port: 50000"
except socket.error, (value,message):
    if s:
        s.close()
    print "Could not open socket: " + message
    sys.exit(1)
while True:
    client, address = s.accept()
    print "Received the connection from:", address
    while True:
        data=client.recv(size)
        if data=='q' or data=='Q':
            client.close()
            print "Client quits:"
            break
        else:
            print "Data received:", data
            newdata="You Send:" + data
            client.send(newdata)
```

```
#!/usr/bin/env python
"""
A simple echo client that handles some exceptions
"""
import socket
import sys
host = 'localhost'
port = 50000
size = 1024
s = None
try:
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((host,port))
except socket.error, (value,message):
        if s:
                s.close()
        print "Could not open socket: " + message
        sys.exit(1)

while True:
        data=raw_input("Type Something(q or Q to exit:)")
        s.send(data)
        if data=='q' or data=='Q':
                s.close()
                break
        else:
                newdata=s.recv(size)
                print newdata
```

(Simple command Peer to Peer Chat application):

```
# TCP server
import socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(("", 5000))
server_socket.listen(5)

print "TCPServer Waiting for client on port 5000"

while 1:
        client_socket, address = server_socket.accept()
        print "I got a connection from ", address
        while 1:
                data = raw_input ( "SEND( TYPE q or Q to Quit):" )
                if (data == 'Q' or data == 'q'):
                        client_socket.send (data)
                        client_socket.close()
                        break;
                else:
                        client_socket.send(data)

                data = client_socket.recv(512)
                if ( data == 'q' or data == 'Q'):
                        client_socket.close()
                        break;
                else:
                        print "RECIEVED:" , data
```

```
# TCP client
import socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(("localhost", 5000))
while 1:
    data = client_socket.recv(512)
    if ( data == 'q' or data == 'Q'):
        print "I received", data
        client_socket.close()
        break;
    else:
        print "RECIEVED:" , data
        data = raw_input ( "SEND( TYPE q or Q to Quit):" )
        if (data <> 'Q' and data <> 'q'):
            client_socket.send(data)
        else:
            client_socket.send(data)
            client_socket.close()
            break;
```

Multitasking: (Handling Multiple clients)

We need applications that can effectively handle multiple network connections simultaneously. As an example, consider a web server. If your server could only handle one connection at a time, you could only be transmitting a single page at a time. If you have a large file on your server and a user on a slow link is downloading it, that user could completely tie up your server for an hour or more. During that time, nobody else would be able to view any pages on that server. Virtually all servers want to be able to serve more than one client at once. To serve multiple clients simultaneously, you need to have some way to handle several network connections at once. Python provides three primary ways to meet that objective:

- forking
- threading
- asynchronous I/O (also known as nonblocking sockets).

Forking:

One of the way to handle multiple connections at once. It is easiest to underusing and use . However its not completely portable. However, Forking may be unavailable on platforms that aren't derived from UNIX.

The system call used to implement forking is called fork. Its a very unique call. After calling fork(), there are two copies of your program running at once. But the second copy doesn't restart from beginning; both copies continue directly after the call to fork(). The process's entire address space is copied. The hard part in understanding the fork is that it is called once but it returns twice. It returns once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child). It also returns once in the child with the return value of 0. Hence the return value tells the process whether it is the parent or the child.

Example:

A basic fork in action:

```
#!/usr/bin/python
import os,time
print "Before the fork my pid is:", os.getpid()
new_pid=os.fork()
if new_pid==0:
        print "Hello from the child my pid is:", os.getpid()
else:
        print "Hello from the parent my pid is:", os.getpid()

time.sleep(5)
print "Hello from both of us:"
```

Output of this Program:

```
Before the fork my pid is: 2925
Hello from the parent my pid is: 2925
Hello from the child my pid is: 2926
-----5 seconds later-----
Hello from both of us:
Hello from both of us:
```

The system call fork() creates a copy of the process which has called it. This copy runs as a child process of the calling process. The child process gets the data and the code of the parent process. The child process receives a process number(PID, process identifier) of its own from the Operating System. The child process runs as an independent instance, ie. Independent of the parent process. With the return value of fork we can decide in which process we are:

0 means that we are in the child process, while a positive value means that we are in the parent process, and a negative return value means that an error occurred while trying to fork().

Zombie Processes:

Process marked <defunct> are dead process (so called zombies) that remain because their parent has not destroyed them properly. Zombie processes are no longer executing , yet certain memory structures are still present and they continue to use the system resources to permit the parent to wait on it. For long running servers these zombie processes must be reaped. These bogus entries may be problematic.

```
#!/usr/bin/python
#Zombie problem Demonstration
import os,time

print "Before the fork my PID is:", os.getpid()
new_pid = os.fork()
if new_pid:
        print "Hello from the parent: child pid will be", new_pid
        print "sleeping 50 seconds..."
        time.sleep(50)
```

The child process will terminate immediately after the fork (fork() returns PID 0 for the child, so it will fail the if test, and there's nothing else for it to do). The parent doesn't clean it up, but rather waits around for a while. Run the program as follows:

```
daya@daya-Aspire-4937:~/Network-Programming/fork$ python zombie.py
Before the fork my PID is: 3456
```

```
Hello from the parent: child pid will be 3457
sleeping 50 seconds ....
```

Now, in another terminal session, inspect the results without stopping the program:

```
daya@daya-Aspire-4937:~$ ps aux |grep 3457
daya      3457  0.0  0.0      0     0 pts/0    Z+   06:40   0:00 [python] <defunct>
```

You can see that the child process is a zombie; the Z in the third column, as well as the <defunct> at the end of the output, indicate that. Once the parent terminates, you'll be able to confirm that neither process exists. The shell cleans up the parent process, and the child process gets re-parented to init, which will clean it up.

The Role of init

The init program is always the first process that runs on the system and always has PID 1. Its main roles are starting up and shutting down the system. In this case, there's another special role for in it. If a process dies, and there are still children of it out there on the system (zombie or not), the operating system will change that process's parent to be PID1 -init. The init program will watch for zombie children in the same way that normal processes will, so these processes will get cleaned up.

Managing Zombie processes:

Echo Server:

```
#################################################################################
# Server side: open a socket on a port, listen for a message from a client,
# and send an echo reply; forks a process to handle each client connection;
# child processes share parent's socket descriptors; fork is less portable
# than threads--not yet on Windows, unless Cygwin or similar installed;
#################################################################################
#!/usr/bin/python
import os, time, sys
from socket import *                    # get socket constructor and constants
myHost = ''                             # server machine, '' means local host
myPort = 50007                          # listen on a non-reserved port number
sockobj = socket(AF_INET, SOCK_STREAM)          # make a TCP socket object
sockobj.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)
sockobj.bind((myHost, myPort))                  # bind it to server port number
print "The server is listening on port 50007"
sockobj.listen(5)                       # allow 5 pending connects
```

```
activeChildren = []

def reapChildren():                              # reap any dead child processes
    while activeChildren:                        # else may fill up system table
        pid,stat = os.waitpid(0, os.WNOHANG)     # don't hang if no child exited
            if not pid:
               break
            else:
                    activeChildren.remove(pid)
                    print "Reaped Child Process:", pid



def handleClient(connection):                    # child process: reply, exit
    while True:                                  # read, write a client socket
        data = connection.recv(1024)             # till eof when socket closed
        if data=='q' or data=='Q':
               connection.close()
               break
            else:
           newdata="you send:" + data
               connection.send(newdata)
    os._exit(0)


def dispatcher():                                # listen until process killed
    while True:                                  # wait for next connection,
        connection, address = sockobj.accept()   # pass to process for service
        reapChildren()                           # clean up exited children now
        childPid = os.fork()                     # copy this process
        if childPid == 0:                        # if in child process: handle
            handleClient(connection)
        else:                                    # else: go accept next connect
            activeChildren.append(childPid)      # add to active child pid list


dispatcher()                                     # call the function dispatcher.
```

Echo Client:

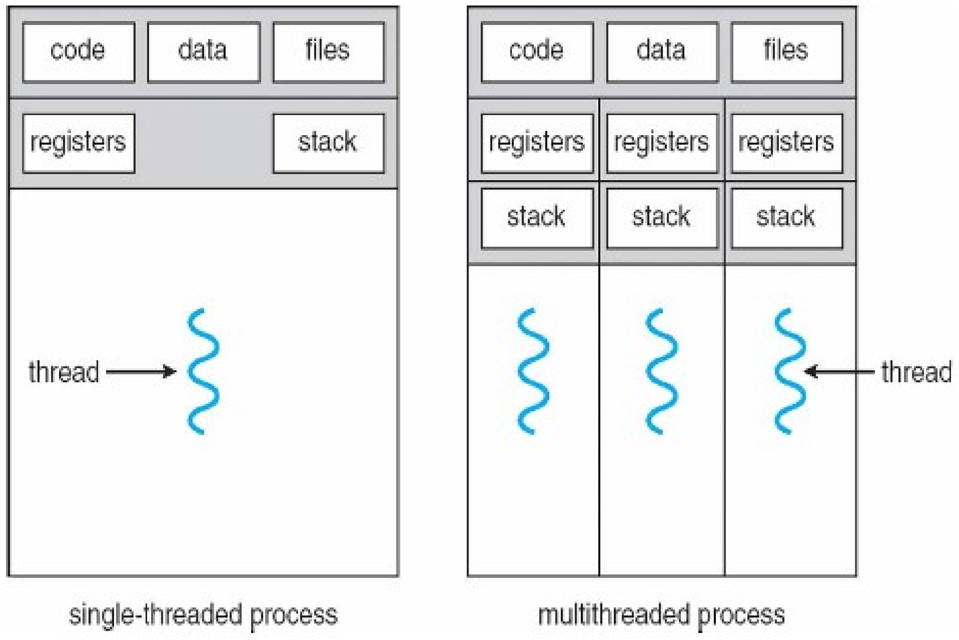
```
##############################################################################
# Client side: use sockets to send data to the server, and print server's
# reply to each message line; 'localhost' means that the server is running
# on the same machine as the client, which lets us test client and server
# on one machine
##############################################################################
```

```
#!/usr/bin/python
import socket
clientsocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
clientsocket.connect(('localhost',50007))
print "Enter q or Q to quits:"
while 1:
  data=raw_input('>')
  clientsocket.send(data)
  if data=='q' or data=='Q':
    break
  newdata=clientsocket.recv(1024)
  print newdata

clientsocket.close()
```

Threading:

A thread is a basic unit of CPU utilization, it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating system resources, such as open files and signals. A traditional (or heavy weight) process has a single thread of control. If a process has multiple thread of control, it can perform more than one task at a time. Fig below illustrate the difference between single threaded process and a multi-threaded process.



Thread simply enable us to split up a program into logically separate pieces and have the pieces run independently of one another until they need to communicate. In a sense threads are a further level of object orientation for multitasking system.

Multithreading:Many software package that run on modern desktop pcs are multi-threaded. An application is implemented as a separate process with several threads of control. A web browser might have one thread to display images or text while other thread retrieves data from the network. A word-processor may have a thread for displaying graphics, another thread for reading the character entered by user through the keyboard, and a third thread for performing spelling and grammar checking in the background.

Why Multithreading:
In certain situations, a single application may be required to perform several similar task such as a web server accepts client requests for web pages, images, sound, graphics etc. A busy web server may have several clients concurrently accessing it. So if the web server runs on traditional single threaded process, it would be able to service only one client at a time. The amount of time that the client might have to wait for its request to be serviced is enormous.

One solution of this problem can be thought by creation of new process. When the server receives a new request, it creates a separate process to service that request. But this method is heavy weight. In fact this process creation method was common before threads become popular. Process creation is time consuming and resource intensive. If the new process perform the same task as the existing process, why incur all that overhead? It is generally more efficient for one process that contains multiple threads to serve the same purpose. This approach would multithreaded the web server process. The server would cerate a separate thread that would listen for clients requests. When a request is made, rather than creating another process,
it will create a separate thread to service the request.

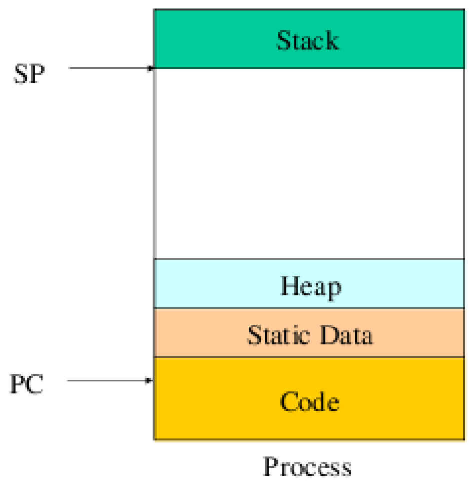
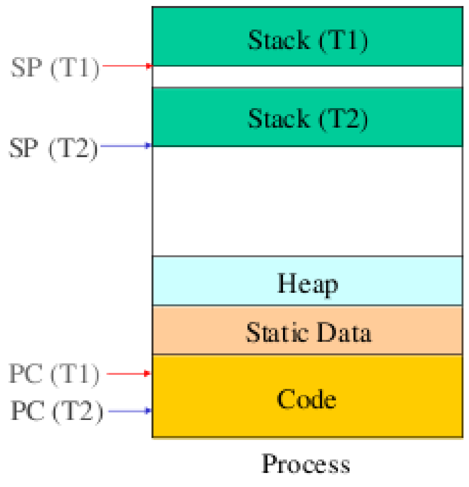
Benefits of Multi-threading:
Responsiveness: Mutlithreaded interactive application continues to run even if part of it is blocked or performing a lengthy operation, thereby increasing the responsiveness to the user.

Resource Sharing: By default, threads share the memory and the resources of the process to which they belong. It allows an application to have several different threads of activity withing the same address space.

Economy: Allocating memory and resources for process creation is costly. Since thread shares the resources of the process to which they belong, it is more economical to create and context switch threads. It is more time consuming to create and manage process than threads.

Utilization of multiprocessor architecture: The benefits of multi threading can be greatly increased in multiprocessor architecture, where threads may be running in parallel on different processors. Mutlithreading on a multi-CPU increases concurrency.

Process VS Thread:

Process	***Thread***
	 SP (T1) SP (T2) PC (T1) PC (T2) Stack (T1) Stack (T2) Heap Static Data Code Process
Heavy weight	Light weight
Unit of Allocation – Resources, privileges etc	Unit of Execution – PC, SP, registers PC—Program counter, SP—Stack pointer
Inter-process communication is expensive: need to context switch Secure: one process cannot corrupt another process	Inter-thread communication cheap: can use process memory and may not need to context switch Not secure: a thread can write the memory used by another thread
Process are Typically independent	Thread exist as subsets of a process
Process carry considerable state information.	Multiple thread within a process share state as well as memory and other resources.
Processes have separate address space	Thread share their address space
processes interact only through system-provided inter-process communication mechanisms.	Context switching between threads in the same process is typically faster than context switching between processes.

Creating threads in Python

Three ways to invoke the `threading.Thread` class from the `threading` module.

- Create Thread instance, passing in a function
- Create Thread instance, passing in a class
- Subclass Thread and create subclass instance

The first method is sufficient for most of our needs.

class `threading.Thread`(*target*, *args*)

Parameters:
- **target** (*callable function*) – identifies the code (function) for the new thread to execute
- **args** (*list*) – is the arguments to pass to the target function

`setDaemon`()

Old API for `daemon`.

`daemon`

A boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before `start()` is called, otherwise `RuntimeError` is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to `daemon` = `False`.

`start`()

Begin execution of the thread now. It must be called at most once per thread object. It arranges for the object's `run()` method to be invoked in a separate thread of control.This method will raise a `RuntimeError` if called more than once on the same thread object.

`join`([*timeout*])

Wait until the thread terminates. This blocks the calling thread until the thread whose `join()` method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs. When the *timeout* argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As `join()` always returns `None`, you must call `isAlive()` after `join()` to decide whether a timeout happened – if the thread is still alive, the `join()` call timed out.

Examples:

```
#!/usr/bin/python
import threading,time

def sleepandprint():
        time.sleep(1)
        print "Hello from both of us:"

def threadcode():
        print "Hello from new thread: My name is %s:" %threading.currentThread().getName()
        sleepandprint()


def main():
        print "Before starting a new thread My name is:", threading.currentThread().getName()
        #create a new thread
        t=threading.Thread(target=threadcode, name="Childthread")
        daemon=True
        t.start()
        print "Hello from main thread: My name is %s:" %threading.currentThread().getName()
        sleepandprint()
        t.join()


if __name__=='__main__':
        main()
```

Output from this program:

```
daya@daya-Aspire-4937:~/Network-Programming/blog$ python thread.py
Before starting a new thread My name is: MainThread
Hello from new thread: My name is Childthread:
 Hello from main thread: My name is MainThread:
Hello from both of us:
 Hello from both of us:
```

Three parts of a multi-threaded server

- **Parent thread**
    - Listen and accept socket connections
    - Create and start child threads
    - Infinite loop

- **Child thread**
    - Receive data from client
    - Send data to client
    - Call synchronized code as needed

- **Synchronized access to shared data**
    - Provides protected access to shared global data, which are often held in a global class, which contains the synchronization algorithms, as well as the global data.
    - Uses synchronization tools – Locks, semaphores, conditional waits (also called monitors) See *Synchronization tools (some of them)*, below.

Echo server using Thread:

#!/usr/bin/python

#Simple Echo server using thread to handle multiple connections at a time.

import threading, socket,sys

```
#The function handlechild handles each client.
def handlechild(clientsock):
        print "New child", threading.currentThread().getName()
        print "Got the connection from", clientsock.getpeername()
        while 1:
                data=clientsock.recv(2096)
                if data=='q' or data=='Q':
                        print "The client", clientsock.getpeername()," quits"
                        clientsock.close()
                        break
                print clientsock.getpeername() ,"send: ", data
                clientsock.send(data)



try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.bind(('localhost',5300))
        s.listen(5)
        print "Waiting for connection on port 5200:"
```

```
except socket.error,(value,message):
	if s:
		s.close()
	print "Couldn't open the socket",message
	sys.exit(1)

while 1:
	clientsocket,clientaddress=s.accept()
	t=threading.Thread(target=handlechild,args=[clientsocket])
	daemon=True
	t.start()
```

Echo Client

```
#!/usr/bin/python
import socket,sys

try:
	c=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
	c.connect(('localhost',5300))

except socket.error, (value,message):
	if c:
		c.close()
	print "Couldnot open the socket", message
	sys.exit(1)


while True:
	data= raw_input("Enter data to send:")
	if data=='q' or data=='Q':
		c.send(data)
		break
	c.send(data)
	print "Received data:", c.recv(1024)
```

Asynchronous Communication:

We introduced handling multiple connection at once through forking and threading. There's a different option available. Instead of running several processes (or threads) at once, one process could be used. This one process would watch over the various connections, switching between them and servicing each one as necessary. This is known as asynchronous communication. The traditional method, used everywhere else in this tutorial, is synchronous communication in which I/O is handled immediately and directly.

Operations of sockets:

Berkeley sockets can operate in one of two modes: blocking or non-blocking.

- Blocking Sockets:
- Non-blocking Sockets:

Blocking Socket: (Synchronous)

In blocking socket, the program is "blocked" until the request for data has been satisfied. The sockets shown in the first example above are called *blocking* sockets, because the Python program stops running until an event occurs. The accept() call blocks until a connection has been received from a client. The recv() call blocks until data has been received from the client (or until there is no more data to receive).

When a program uses *blocking* sockets it often uses one thread (or even a dedicated process) to carry out the communication on each of those sockets. The main program thread will contain the listening server socket which accepts incoming connections from clients. It will accept these connections one at a time, passing the newly created socket off to a separate thread which will then interact with the client. Because each of these threads only communicates with one client, it is ok if it is blocked from proceeding at certain points. This blockage does not prohibit any of the other threads from carrying out their respective tasks.

The use of blocking sockets with multiple threads results in straightforward code, but comes with a number of drawbacks. It can be difficult to ensure the threads cooperate appropriately when sharing resources. And this style of programming can be less efficient on computers with only one CPU.

Non-blocking socket(Asynchronous)

One is the use of *asynchronous* sockets. These sockets don't block until some event occurs. Instead, the program performs an action on an asynchronous socket and is immediately notified as to whether that action succeeded or failed. This information allows the program to decide how to proceed. Since asynchronous sockets are non-blocking, there is no need for multiple threads of execution. All work may be done in a single thread. This single-threaded approach comes with its own challenges, but can be a good choice for many programs. It can also be combined with the multi-threaded approach: asynchronous sockets using a single thread can be used for the networking component of a server, and threads can be used to access other blocking resources, e.g. databases.

Asynchronous communication using select (Handling multiple client at a time):

The *select()* One is the use of asynchronous sockets. These sockets don't block until some event occurs. Instead, the program performs an action on an asynchronous socket and is immediately notified as to whether that action succeeded or failed. This information allows the program to decide how to proceed. Since asynchronous sockets are non-blocking, there is no need for multiple threads of execution. All work may be done in a single thread. This single-threaded approach comes with its own challenges, but can be a good choice for many programs. It can also be combined with the multi-threaded approach: asynchronous sockets using a single thread can be used for the networking component of a server, and threads can be used to access other blocking resources, e.g. databases.method uses the following syntax:

select(*input*,*output*,*exception*[,*timeout*])
The first three arguments are lists of socket or file objects that are waiting for input, output, or exceptions. If you do not specify a timeout, then the select call will block until one of the listed sockets has an input, output, or exception event occur. If you specify a timeout period, given as a floating point number of seconds, then the socket call will return after this time if no sockets are ready. A timeout value of zero indicates that the sockets are checked but the call will not block if none are ready.

The *select()* method returns a tuple of three lists: the subset of the input sockets and files that have input, output, or exception events. If a timeout has occurred without any input, output or exception events, then all three lists will be empty.

Echo server using select:

```
#!/usr/bin/env python

"""
An echo server that uses select to handle multiple clients at a time.
Entering any line of input at the terminal will exit the server.
"""

import select
import socket
import sys

host = ''
port = 50000
backlog = 5
size = 1024
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((host,port))
server.listen(backlog)
input = [server,sys.stdin]
```

```
running = 1
while running:
    inputready,outputready,exceptready = select.select(input,[],[])
    for s in inputready:
        if s == server:
            # handle the server socket
            client, address = server.accept()
            print "Got the connection from:", address
            input.append(client)

        elif s == sys.stdin:
            # handle standard input
            junk = sys.stdin.readline()
            running = 0

        else:
            # handle all other sockets
            data = s.recv(size)
                print  data
            if data:
                s.send(data)
            else:
                s.close()
                input.remove(s)
server.close()
```

Echo client using select

```
#!/usr/bin/env python

"""
An echo client that allows the user to send multiple lines to the server.
Entering a blank line will exit the client.
"""

import socket
import sys

host = 'localhost'
port = 50000
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
sys.stdout.write('%')

while 1:
    # read from keyboard
    line = sys.stdin.readline()
    if line == ' ':
        break
    s.send(line)
    data = s.recv(size)
    sys.stdout.write(data)
    sys.stdout.write('%')
s.close()
```