Sockets Programming with Python

Réseaux : Programmation & Configuration

Dino Lopez
http://www.i3s.unice.fr/~lopezpac/

Socket definition

- The sockets are the end points for the communication between 2 processes (Inter-Process Communications – IPC)
  - Local IPC
    - $ ls ~ | grep “^d” | wc -l
  - Remote IPC
    - BSD sockets
- Berkeley Sockets (BSD sockets) is a library to allow the programming of Internet Sockets
- BSD sockets evolved and make part now of the POSIX standard
- The Python `socket` module provides access to the BSD Socket interface

IPC in a nutshell

Receiver (Server)

- Something to read
- Open the recipient
- Define the communication method
- Wait for a call

Sender (Client)

- Something to write
- Identify the destination
- Identify the recipient
- Define the communication method
- Make the call

Client – Server Model

- Server
  - ➢ Daemon
- Client
  - ➢ Initialize a connection
  - ➢ Punctual communications

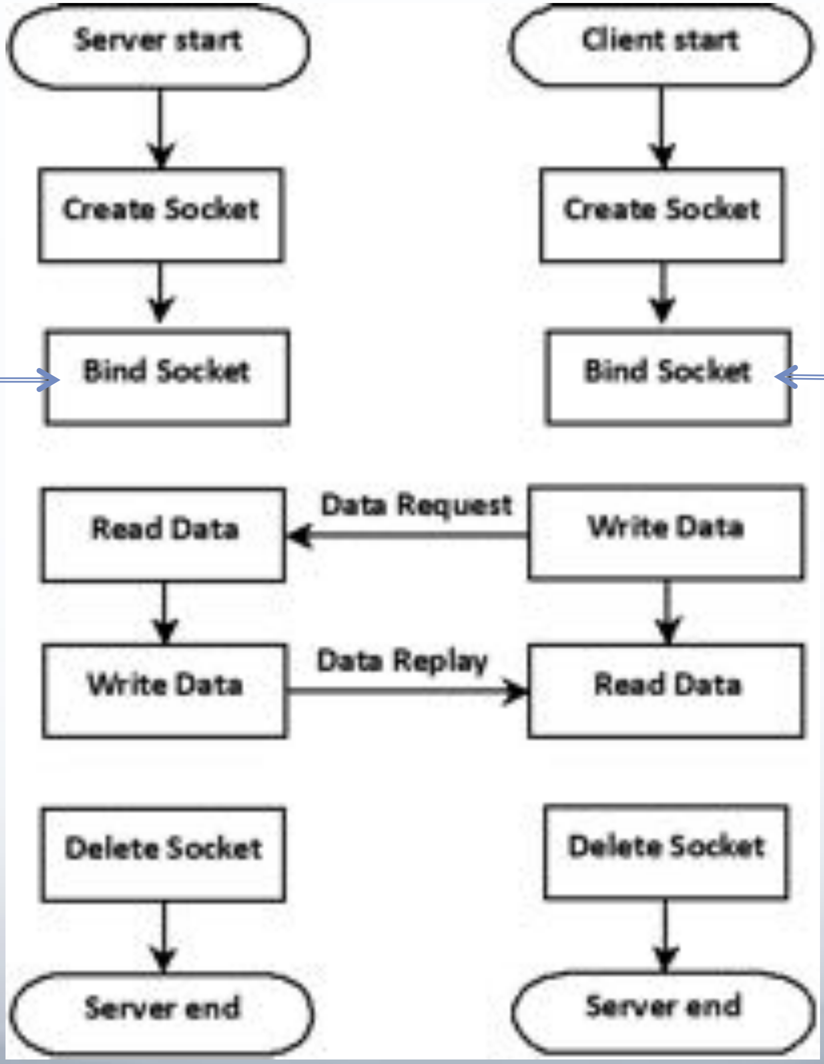
Communication modes

- Connectionless mode
  - Uses UDP
  - Unreliable
  - Datagram Sockets

- Connection oriented
  - Uses TCP
  - Reliable
  - Monopolize a socket descriptor

Connectionless mode

Mandatory



Optional

Writing your code

1. Create your socket – the socket() function

- Socket() returns a socket object which implements the BSD Socket system calls
- Definition socket.socket([family[, type[, proto]]])
  - Address family: by default, AF_INET
  - Socket type: by default SOCK_STREAM
  - Protocol number: 0 (zero) frequently

Socket domains and Types

- Several domains
  - *AF_INET: Socket in the IPv4 domain*
  - AF_INET6: Socket in the IPv6 domain
  - AF_BLUETOOTH: Socket in the Bluetooth domain (needs python-bluez)
  - …
- 3 socket types available for the AF_INET domain
  - *SOCK_STREAM: Connection-oriented communication - TCP*
  - *SOCK_DGRAM: connectionless communication – UDP*
  - SOCK_RAW: custom construction of headers
- All these constants are available in the socket class

Binding the socket with bind()

- To bind an IP address and define a listening port in a newly created socket, you will use the bind() method of the socket class

- According to the Python doc, bind() is declared as socket.bind(*address*)
  - Note that the format of address depend on the address family used to create your socket
  - For AF_INET, the address is a tuple (host,port), where
    - host is a string representing the IP address or the canonical name of a given interface: "mycomp.test.com", "192.168.0.12"
    - To leave the kernel to take any available interface, use None for host
    - Port is an integer

- Bind is mandatory at the server side, but optional at the client side

Sending/Receiving data – connectionless mode

- To send data in a non connected socket, you want to use socket.sendto(string, address)
  - String represents the message to be sent
  - Address is the tuple representing the remote host and recipient
  - It returns the number of bytes which were actually sent
- And to receive the data, you want to use socket.recvfrom(bufsize[, flags]) or socket.recvfrom_into(buffer[, nbytes[, flags]])
- Regarding socket.recvfrom(bufsize[, flags])
  - It returns a pair (string, address), where string represents the received data and address is the address of the remote peer

Closing a Socket

- To close the socket, you can either call the socket.close() or the socket.shutdown(*how*) method
- After close(), any operation on the socket object will fail
- shutdown() allows a finer control over the socket
  - If how is SHUT_RD, the reception of data is disabled
  - If how is SHUT_WR, data transmission is disabled
  - If how is SHUT_RDWR, the transmission and reception of data are disallowed
  - On some OSs, shutting down a half of the connection can close the opposite half
- *In connected mode, closing a socket triggers the transmission of EOF to the remote peer*

Endianess and Network Byte Order

Big Endian vs Little Endian

- Endianness refers to the order of the bytes, comprising a digital word, in computer memory. Definitions from Wikipedia
- Big Endian: the most significant byte of a word is stored in a particular memory address, and subsequent bytes are stored in the following higher memory addresses
- Little Endian: the least significant byte of a word is stored in a particular memory address, and subsequent bytes are stored in the following memory addresses
- Network byte order is Big-Endian

Example in an Intel x86_64 processor

```
from math import ceil
from struct import pack

def show_bytes(data):
    i = 0
    for b in data:
        print "Byte %d has %02x" % (i,ord(b))
        i=i+1

var = int("16909060",10)
numBytes = ceil(var.bit_length()/8.0)
print "Var has %08x" % (var)

i = 0
while numBytes > i:
    print "Byte %d has %02x" % (i,(var>>(i*8) & 0xff))
    i=i+1

print ""
show_bytes(pack(">I",var)); // Big-Endian

print ""
show_bytes(pack("!I",var)); // Network Byte Order
```

```
Var has 01020304
Byte 0 has 04
Byte 1 has 03
Byte 2 has 02
Byte 3 has 01

Byte 0 has 01
Byte 1 has 02
Byte 2 has 03
Byte 3 has 04

Byte 0 has 01
Byte 1 has 02
Byte 2 has 03
Byte 3 has 04
```

Communication without Network Byte Order

Short Integer = 255_{10} = 00 FF_{16}
Memory (X, X+1) = (FF,00)
Write 1st Byte = FF
Write 2nd Byte = 00

Receives 1st Byte = FF
Receives 2nd Byte = 00
Memory (X,X+1) = (FF,00)
Short Integer = 65280_{10} = FF 00_{16}

Little-Endian

Big-Endian

Network Byte Order formatting

- When sending 2 byte words or 4 byte words, the network byte order must be applied
- 2 bytes words are
  - written in network byte order with the htons(). If the device is BE, no actions are taken, otherwise, bytes are flipped
  - Translated to the device architecture with ntohs(). If the device is BE, no actions are taken, otherwise, bytes are flipped
- 4 bytes words are
  - written in network byte order with the htonl(). Same observations as above.
  - Translated to the device architecture with ntohl(). Same observations as above.
- Python is little bit special. Socket.htonl(), etc are available. It's preferable to use
  - struct.pack to pack binary data, convert to the network byte order and send it by the socket
  - Struct.unpack to receive bytes
- When the data has always the right order (e.g. ascii text, file content, etc.) the network byte order does not apply

Example of a connectionless communication

Connectionless mode – the integer is transmitted in Network Byte Order

Server

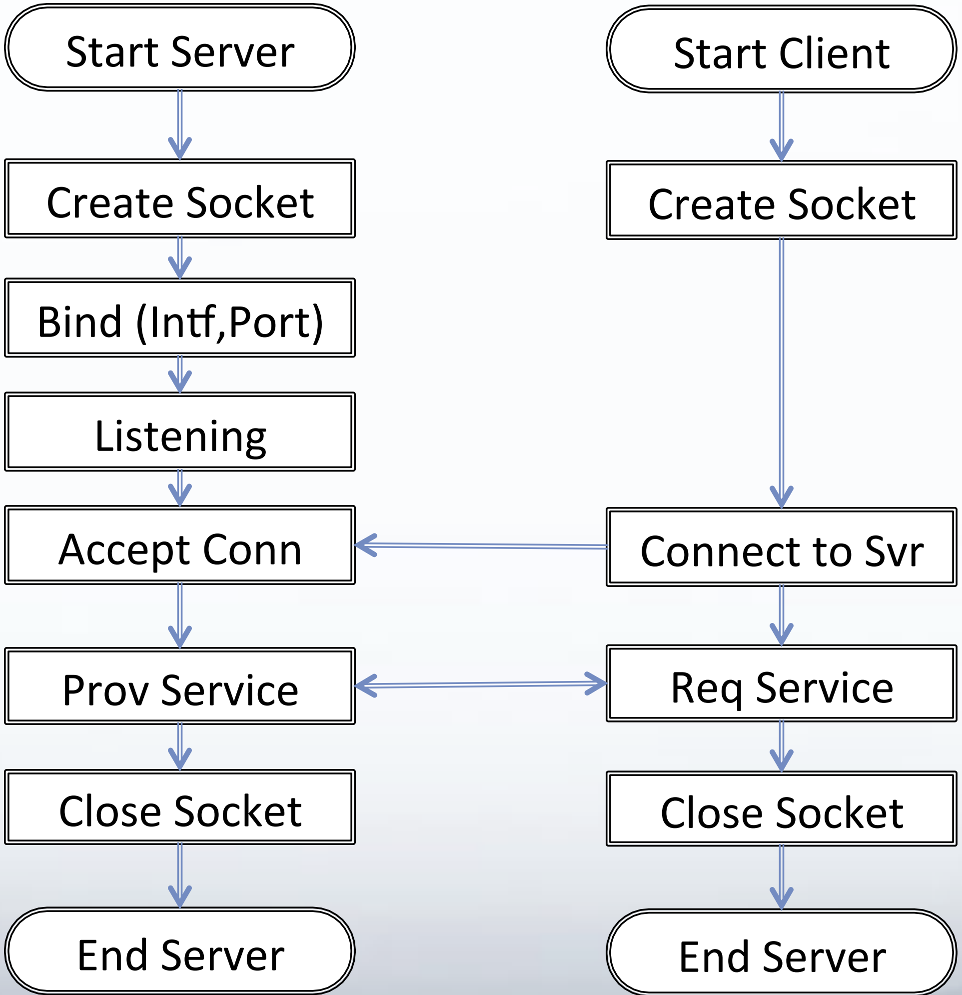
```
import socket
import struct

HOST = ''   # any available interf
PORT = 5000  # Arbitrary non-priv port
s = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
s.bind((HOST, PORT))
data, (HOST,PORT) = s.recvfrom(1024)

ndata = struct.unpack("!h",data)
data = ndata[0]+1

s.sendto(struct.pack("!h",data),
(HOST,PORT))
s.close()
```

Client

```
import socket
import struct

HOST = '127.0.0.1’  # The remote host
PORT = 5000         # The remote port
s = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)

val = struct.pack("!h",1)
s.sendto(val,(HOST,PORT))

data, (HOST,PORT) = s.recvfrom(1024)
print "Received %d" %
(struct.unpack("!h",data))

s.close()
```

Connection-based communication

The flow chart



Connection demand

- In a connection-based communication, after binding the socket to an interface and port, the server must listen for incoming connection.
  - socket.listen(*backlog*). *backlog* represents the number of incoming connection that can be queued at any time
- After listening for incoming connection, connection requests should be accepted
  - socket.accept(). accept() returns a pair *conn, address*, where
    - *conn* is a new socket object that the server will use to communicate with the remote host
    - *address* is the address of the remote host. The address format depends on the address family type
- The client connect to the server with the connect() method.
  - socket.connect(*address*). *address* is the address of the remote host. The address format depends on the address family type

Sending data

- To send data, you can use
  - socket.senda(*string*[, *flags*])
    - Returns the number of bytes sent. The application must verify that whole data has been sent, or retry the data transmission if needed
  - socket.sendall(*string*[, *flags*])
    - Send all data unless an error occur
    - It returns “None” on success. Otherwise, un exception is raised and there is no way to know how many data has been sent
- For both methods, the optional *flags* argument can be used to execute special sending methods (e.g. out-of-band data)

Receiving data

- To receive data through a connected socket you can use socket.recv(*bufsize*[, *flags*])
  - Bufsize represents the maximum amount o bytes to read
  - Flags can be used to perform “special” readings
  - It returns a string which is actually the data read

Connection-based communication

Server

```python
import socket

HOST = ''     # any available interf
PORT = 5000   # Arbitrary non-priv port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.sendall("Hi!")
conn.close()
s.close()
```

Client

```python
import socket

HOST = '127.0.0.1'    # The remote host
PORT = 5000           # The remote port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
val = "Hello!"
s.sendall(val)
data = s.recv(1024)
s.close()
print "Received: %s" %(data)
```

Server styles

- Iterating server
  - Only one socket is opened at a time
  - Clients are accepted one after the other
  - Slow service
- Forking server
  - With fork
    - After accept, the server creates a subprocess which will provide the service
    - The subprocess is a copy of the parent process
    - The used memory space is doubled
  - With POSIX Threads
    - The same memory space is shared between all the threads
    - Special attention must be taken to avoid race conditions
- Concurrent single server
  - Uses select to simultaneously wait over the whole set of opened socket IDs
  - The main process is waken up when new data arrives
  - There is no context switching, but it cannot benefit from multiprocessors

Some thoughts about Multiprocessing

Multiprocessing in Linux / Unix-like systems

- Quick overview of sub-process creation and termination
- One can use the multiprocess Python package to create and handle sub-processes
  - High-level API
  - Let’s play with the OS services to understand the concepts
- os.fork() spawns a child process
  - Returns 0 in the child process
  - Returns the PID of the child in the parent process
  - In case of error, an OSError exception is raised

Example 1 – No synch between processes

```python
import os
import time

def testdelay():
    for i in range(0,5):
        time.sleep(2)
        print "child is running... %d" %(i)
    os._exit(0)

pid = os.fork()
if pid == 0:
    testdelay()

print "parent is exiting..."
exit(0)
```

```
parent is exiting...
child is running... 0
child is running... 1
child is running... 2
child is running... 3
child is running... 4
```

Example 2- waiting for child termination

```
import os
import time

def testdelay():
    for i in range(0,5):
        time.sleep(2)
        print "child is running... %d" % (i)
    os._exit(0)

pid = os.fork()
if pid == 0:
    testdelay()
else:
    status = os.wait()
    print status

print "parent is exiting..."
exit(0)
```

```
child is running... 0
child is running... 1
child is running... 2
child is running... 3
child is running... 4
(59692, 0)
parent is exiting...
```

Example 3 – child exits faster than the parent process

```
import os
import time

def testdelay():
    for i in range(0,5):
        time.sleep(2)
        print "parent is running... %d" %(i)

pid = os.fork()
if pid == 0:
    print "child exits..."
    os._exit(0)
else:
    testdelay()
    status = os.wait()
    print status

print "parent is exiting..."
exit(0)
```

- Output

child exits...
parent is running... 0
parent is running... 1
parent is running... 2
parent is running... 3
parent is running... 4
(59692, 0)
parent is exiting...

- Process table status after child finishes but before parent finishes

59691 ttys002 0:00.03 python test-fork.py

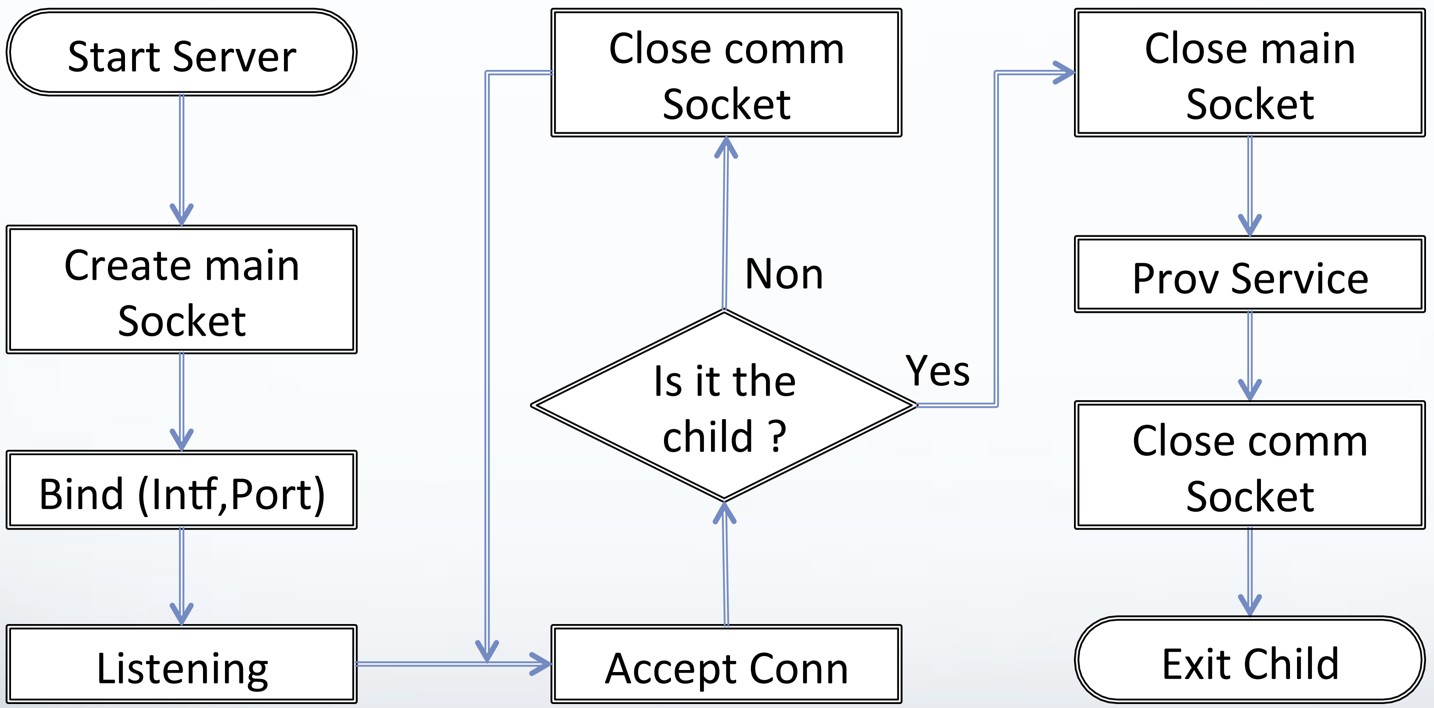
59692 ttys002 0:00.00 (Python)

Zombie process

SIGCHLD and SIG_IGN

- Upon exit, a child process reports its exit code to its parent
- While the process parent doesn't read the exit status of the child process, this last is keep in the process table
  - Leading to a so-called zombie process
  - Refer to wait(), waitpid()
- In Linux, Unix-like systems, whenever something interesting happens to a forked off child, the parent process receives a SIGCHLD signal
- By default, SIGCHLD is ignored
- To avoid zombie process, the parent should handle the SIGCHLD signal. Ex.
  - signal.signal(signal.SIGCHLD, signal.SIG_IGN)

Multi-process server



Multicast

Building a Multicast sender

- One-to-many communication
  - It is not broadcast
  - Multicast groups are identified by Class D IP addresses (224.0.0.1 → 239.255.255.255)
- Create your socket: SOCK_DGRAM or SOCK_STREAM ?
- Bind is optional. You should specify however on which interface multicast messages will be sent. Eg.
  - s.setsockopt(socket.IPPROTO_IP,socket.IP_MULTICAST_IF,socket.INADDR_ANY)
- Send the packet to the multicast address

Building a Multicast receiver

- Before binding your socket, it’s used to reuse the local address, so multiple applications can use at the same time that multicast port
  - s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)

- After binding, the receiver process must ask to the system to indeed receive any message addressed to the targeting multicast group and send it up to the application
  - `mreq = struct.pack("4sl",socket.inet_aton("226.1.1.1"),socket.INADDR_ANY)`
  - `s.setsockopt(socket.SOL_IP,socket.IP_ADD_MEMBERSHIP,mreq)`

- Before closing your socket, remove your membership (same operation as above, but with the **IP_DROP_MEMBERSHIP** option)