09-Sockets

November 21, 2019

1 Socket Programming

1.1 Sockets

- Inter process and network communication among programs.
- Sockets establish communication channels among programs.
- Subject to network protocols.
- A socket is constructed, defined by two basic information:
  - **Address Family or Domain:** Protocol family of the socket. `AF_UNIX`, `AF_INET`, `AF_INET6`, `AF_IPX`, `AF_BLUETOOTH`
  - **Socket Type**: The type of socket, how data is packed, sequenced, reliability etc. `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_SEQPACK`, `SOCK_RAW`
- Each socket domain and type is internally handled by operating system as networking protocols.
- Details are abstracted from user. User just use socket like a file handle, data send over the socket can be received by another node in the network.

1.2 `AF_UNIX` Family

- A family without networking. Only processed within the same host/node can communicate.
- A path on file system is used as an address. Path should be writable and when bound, a special file is created

1.3 `AF_INET` Family

- Internet protocol version 4 family.
- Socket address is a tuple of IP address and a port. An IP address is a 4 byte address. Python uses a dot separated sequence of decimal numbers as a string. i.e. '144.122.171.123'. Port is a positive integer in [1:65535]
- `('144.122.145.146',80)` is address of departments web server.

1.4 `AF_INET6` Family

- Internet protocol version 6 family.
- Socket address is a tuple of IPv6 address and a port. An IPv6 address is a 16 bytes address. A column separated sequence of 2 bytes hexadecimal values are used (as `'2001:a98:30:cc::4:f101'`). Port is a positive integer in [1:65535]

1.5 SOCK_STREAM vs SOCK_DGRAM

- SOCK_STREAM is a stream sequence of bytes. The underlying protocol provides that the packets are:
  - Ordered
  - Reliable
  - No duplication
- SOCK_DGRAM is a datagram based socket type. Has a packet boundary. Not a stream. Each datagram is a standalone structure. In datagram packets can:
  - Arrive in arbitrary order to receiver
  - be lost
  - be duplicated, same datagram can be received multiple times.
- In AF_INET **tcp** and **udp** are examples of SOCK_STREAM and SOCK_DGRAM respectively.
- SOCK_DGRAM is faster by definition. Trade off between reliability and speed.

```
In [35]: # a simple unix domain datagram communication in a thread
         from socket import *
         from threading import Thread
         import os,stat

         def cleansocketfile(path):
             '''Test if the path is a socket file and clean it'''
             try:
                 st = os.stat(path)
                 if st and stat.S_ISSOCK(st.st_mode):
                     print("removing ",path)
                     os.unlink(path)
                     return True
                 else:
                     return False
             except:
                 return False
             return False

         def readandprintone():
             # create a socket, which is not bound yet
             s = socket(AF_UNIX, SOCK_DGRAM)

             # bind an adress to it.
             cleansocketfile("/tmp/mysocket")
             s.bind("/tmp/mysocket")

             res = s.recv(1000)
             print(res)
             s.close()
```

```
        t=Thread(target=readandprintone, args=())
        t.start()

removing  /tmp/mysocket


In [22]: c = socket(AF_UNIX, SOCK_DGRAM)
         c.sendto(b'hello',"/tmp/mysocket")

Out[22]: 5

b'hello'


In [38]: # same example in INET domain
         def readandprintone():
             # create a socket, which is not bound yet
             s = socket(AF_INET, SOCK_DGRAM)

             # bind an adress to it.

             s.bind(('0.0.0.0', 10447))

             res = s.recvfrom(1000)
             print(res)
             s.close()

         t=Thread(target=readandprintone, args=())
         t.start()

         c = socket(AF_INET, SOCK_DGRAM)
         c.sendto(b'hello how are you', ('127.0.0.1', 10447))
         c.close()

(b'hello how are you', ('127.0.0.1', 51750))
```

2 Datagram Communication

	Server	Client
Request	create a socket `s`,bind it to address `p`,`recvfrom()` on `s`,`recvfrom` returns `(reqbody, peeraddr)`	create a socket `c`,`sendto()` request to `p`
Response	get `peeraddr` `sendto(response, peeraddr)`	`recv()` on `c`

- No stream channel as established connection
- All communication is through single shot messages in datagrams.

```
In [51]: # A typical datagram service loop and clients
         import time, random

         def echoservice(n,port):
             ''' n times read a request and echo uppercase back'''
             s = socket(AF_INET, SOCK_DGRAM)
             s.bind(('', port))

             for i in range(n):
                 req, peer = s.recvfrom(10000)
                 print("request",req, " from ", peer)
                 s.sendto(req.decode().upper().encode(), peer)
             s.close()

         def client(port):
             c = socket(AF_INET, SOCK_DGRAM)
             time.sleep(random.random()*2)
             c.sendto(b'hello', ('127.0.0.1', port))
             result = c.recv(1000)
             print("Result:" , result)

         # create a server
         serv = Thread(target=echoservice, args=(5,20445))
         # create 5 clients
         clients = [Thread(target = client, args=(20445,)) for i in range(5)]
         serv.start()
         for cl in clients: cl.start()


Exception in thread Thread-112:
Traceback (most recent call last):
  File "/usr/lib/python3.5/threading.py", line 914, in _bootstrap_inner
    self.run()
  File "/usr/lib/python3.5/threading.py", line 862, in run
    self._target(*self._args, **self._kwargs)
  File "<ipython-input-51-7a8932e4468a>", line 7, in echoservice
    s.bind(('', port))
OSError: [Errno 98] Address already in use
```

RuntimeError Traceback (most recent call last)

```
    <ipython-input-51-7a8932e4468a> in <module>()
     25 clients = [Thread(target = client, args=(20445,)) for i in range(5)]
     26 serv.start()
---> 27 for cl in clients: cl.start()
     28


    /usr/lib/python3.5/threading.py in start(self)
    842             _limbo[self] = self
    843         try:
--> 844             _start_new_thread(self._bootstrap, ())
    845         except Exception:
    846             with _active_limbo_lock:


    RuntimeError: can't start new thread
```

3 Stream type and establishing a connection

- DGRAM is connectionless
- Streams are used for longer term reliable connections
- A stream is a bidirectional channel among the peers

Server: * create socket `s=socket(..., SOCK_STREAM)` * bind it `s.bind(...)` * listen to it `s.listen(queuesize)` * per connection request a loop `python while True: ns, peer = s.accept() # now we have a different socket object for each new channel # serve ns on a concurrent thread/process # ready to accept new connection`

Client: * create a socket * connect to server address

```
In [12]: # A thread per connection stream service and clients
         import time
         import random

         def echoservice(sock):
             ''' echo uppercase string back in a loop'''
             req = sock.recv(1000)
             while req and req != '':
                 # remove trailing newline and blanks
                 req = req.rstrip()
                 sock.send(req.decode().upper().encode())
                 req = sock.recv(1000)
             print(sock.getpeername(), ' closing')

         def client(n, port):
             # send n random request
             # the connection is kept alive until client closes it.
```

```
            mess = ['hello', 'bye', 'why', 'yes', 'no', 'maybe', 'are you sure', 'why not?']
            c = socket(AF_INET, SOCK_STREAM)
            c.connect(('127.0.0.1', port))
            for i in range(n):
                time.sleep(random.random()*3)
                c.send(random.choice(mess).encode())
                reply = c.recv(1024)
                print(c.getsockname(), reply)
            c.close()

        def server(port):
            s = socket(AF_INET, SOCK_STREAM)
            s.bind(('',port))
            s.listen(1)    # 1 is queue size for "not yet accept()'ed connections"
            try:
                #while True:
                for i in range(5):    # just limit # of accepts for Thread to exit
                    ns, peer = s.accept()
                    print(peer, "connected")
                    # create a thread with new socket
                    t = Thread(target = echoservice, args=(ns,))
                    t.start()
                    # now main thread ready to accept next connection
            finally:
                s.close()



        server = Thread(target=server, args=(20445,))
        server.start()
        # create 5 clients
        clients = [Thread(target = client, args=(5, 20445)) for i in range(5)]
        # start clients
        for cl in clients: cl.start()
```

```
('127.0.0.1', 41620) connected
('127.0.0.1', 41622) connected
('127.0.0.1', 41624) connected
('127.0.0.1', 41626) connected
('127.0.0.1', 41628) connected
('127.0.0.1', 41620) b'NO'
('127.0.0.1', 41620) b'WHY'
('127.0.0.1', 41622) b'MAYBE'
('127.0.0.1', 41628) b'HELLO'
('127.0.0.1', 41624) b'WHY'
('127.0.0.1', 41624) b'NO'
('127.0.0.1', 41626) b'BYE'
```

```
('127.0.0.1', 41620) b'WHY NOT?'
('127.0.0.1', 41628) b'MAYBE'
('127.0.0.1', 41622) b'MAYBE'
('127.0.0.1', 41624) b'NO'
('127.0.0.1', 41622) b'WHY'
('127.0.0.1', 41628) b'HELLO'
('127.0.0.1', 41620) b'BYE'
('127.0.0.1', 41628) b'ARE YOU SURE'
('127.0.0.1', 41626) b'YES'
('127.0.0.1', 41620) b'MAYBE'
('127.0.0.1', 41620)  closing
('127.0.0.1', 41622) b'BYE'
('127.0.0.1', 41628) b'HELLO'('127.0.0.1', 41628)
  closing
('127.0.0.1', 41624) b'NO'
('127.0.0.1', 41622) b'BYE'
('127.0.0.1', 41622)  closing
('127.0.0.1', 41626) b'WHY'
('127.0.0.1', 41624) b'BYE'
('127.0.0.1', 41624)  closing
('127.0.0.1', 41626) b'HELLO'
('127.0.0.1', 41626)('127.0.0.1', 41626)  b'WHY' closing
```