SOCKET

Valerio Di Valerio
The Problem

- Communication between computers connected to a network
Network applications

- A set of processes distributed over a network that communicate via messages
  - Ex: Browser Web, BitTorrent, ecc...

- Processes communicate via services offered by the operating system
  - What kind of services?! TCP, UDP and IP protocols...

- Most famous network application architecture: client/server
Network application has two components: client and server

- **Client:**
  - Initiates communication
  - Requests a service
    - Es: Chrome sends a request for a Web site:

- **Server:**
  - Waits a request
  - Provides the service
    - Es: Web server providing an html page
Peer-to-Peer model

See previous lesson!
Two main problems

- **Network addressing**: how to unambiguously identify the process running on a remote host
- **Data transport**: how to transfer bits to the destination
Addressing and data transport in TCP/IP

- **Addressing** based on two components
  - IP address: identifies the remote host (actually the network interface)
  - Port number: identifies the running process

- **Data transport** based on two protocols
  - TCP: connection oriented, stream oriented, reliable data transfer
  - UDP: message oriented, no connection, no reliable data transfer
How to interact with TCP/UDP

- Protocols run “inside” the operating system
  - OSs usually implement the protocol stack TCP/IP
- Our applications run “outside” the operating system
- Result: our applications need to interact with the OS to send data to TCP/IP
- Interaction is possible using a set of interfaces called Application Programming Interface (API)
Application Programming Interfaces (API)

- They standardize interaction with the OS specifying:
  - Function prototypes
  - Input/output parameters

- **Socket**: Internet API
  - Originated with the BSD Unix operating system in 1983 and developed in C
  - Now available on many OSs
  - The Python interface is a straightforward transliteration of the Unix interface for sockets implemented in C
Socket

- It is a “door” between application and transport protocols (TCP or UDP)
  - Allows to send/receive data from the network
- It represents the communication endpoint
  - A socket is owned by the application
- It provides to developers a high level interface to transport protocols
Socket

Controlled by application developer

Controlled by operating system

Network
Socket in Python

- **Socket creation:**

  ```python
  import socket
  s = socket.socket(addr_family, type, protocol)
  ```

- It is the first function executed both by the client and the server
  - The OS initializes all the resources needed to manage data transfer

- It returns the socket...
  - or raises an exception if something goes wrong
Socket in Python

- **addr_family**: the protocol family
  - `socket.AF_INET`: IPv4 protocol
  - `socket.AF_INET6`: IPv6 protocol
  - `socket.AF_UNIX`: to manage communication between processes on the same host

- **type**: the communication type
  - `socket.SOCK_STREAM`: stream (connection) oriented
  - `socket.SOCK_DGRAM`: message oriented
  - `socket.SOCK_RAW`: provide access to the network layer

- **protocol**: a specific protocol
  - If set to 0 (or omitted) the default protocol defined by the couple `addr_family + type` will be used
  - Es: `socket.AF_INET + socket.SOCK_STREAM = TCP`
Connection oriented communication

Server

socket()  
bind()  
listen()  
accept()  
recv()  
send()  
recv()  
close()

Server blocks on accept() waiting for a new connection

Create a connection

Send request data

Send response data

End the communication

Client

socket()  
connect()  
send()  
recv()  
close()
Bind a socket to an address

```
socket.bind(address)
```

- Thanks to the `bind()` function the OS will forward the received packets to the correct process!
- `address` is a tuple (host, port) for the AF_INET address family
  - `host` is a string representing either a hostname in Internet domain notation like “www.repubblica.it” or an IPv4 address like “213.92.16.191”
  - `port` is an integer
Socket addresses in Python

- `host = ""` (i.e., an empty string) specifies all local network interfaces
- `host = "localhost"` specifies the *loopback* interface
  - A virtual network interface used to manage communication between processes running on the same machine
  - Bypasses local network interface hardware and lower layers of the protocol stack
  - Useful for testing and development
  - “localhost” corresponds to the reserved IP address 127.0.0.1
- **Example:**

```python
import socket
sock = socket.socket(AF_INET, SOCK_STREAM)
sock.bind(('',9000))
sock.bind(('localhost',9000))
sock.bind(('192.168.2.1',9000))
```
A note on port numbers

- **Managed by Internet Assigned Numbers Authority (IANA)**
  - maintains the official assignments of port numbers for specific uses

- **Well-known ports (range 0-1023)**
  - Used by system processes that provide widely used network services
    - 21 -> FTP, 23 -> Telnet, 25 -> SMTP (Mail), 80 -> HTTP (Web)
  - On Unix OS a process needs root privileges to be able to bind on these ports

- **Registered ports (range 1024-49151)**
  - The IANA registers uses of these ports as a convenience to the Internet community
    - 1863 -> MSNP, 3074 -> Xbox LIVE,
  - Registered ports can be used by ordinary users

- **Dynamic ports (range 49152–65535)**
  - They cannot be registered with IANA
  - Used for custom or temporary purposes
listen() function

- Tells the OS to start listening for connections on the socket
- `backlog` argument specifies the maximum number of queued connections
  - the maximum value is system-dependent
- On Linux it refers to the established connections (3-way handshake completed)
  - Security reason: SYN flood attack
- If backlog is full, new connection requests can be ignored or refused by the OS
- 3-way handshake completely managed by the OS
Example: a simple server (to be cont’d)

```python
import socket

HOST = ""
PORT = 1060

sock = socket.socket(AF_INET, SOCK_STREAM)
sock.bind(((HOST,PORT)))
sock.listen(5)
```

- **Create socket**
- **Bind to the specified address**
- **Start listening for connections on the socket**
The `connect()` function

- Connects to a remote socket at `address`.
- If a TCP socket is used, `connect()` tells the OS to start the 3-way handshake.
- `address` is a tuple `(host, port)` (for the AF_INET address family).

Example:

```python
import socket
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('www.python.org', 80))
```
accept() function

sock, address = socket.accept()

- It allows the server to take the first established connection from the backlog
- If backlog is empty, accept() blocks until a connection is received
- Return values:
  - address is the address of the client that connected
  - sock is a new socket, the one actually used to transfer data with the connected client
Passive and active sockets

- Server uses two different sockets for each client connection

  - The **passive** socket, created by `socket()`
    - Holds the “socket name” (i.e., the address and port number) at which the server is ready to receive connections
    - No data can ever be received or sent by this kind of port
    - It does not represent any actual network conversation
    - Used to listen to incoming connections (using `listen()` function)

  - The **active** socket, returned by `accept()`
    - It has the same “socket name” of the passive socket
    - It is bound to one particular remote conversation partner
    - It can be used only for talking back and forth with that partner
Passive and active sockets

- **Problem**: there can be many active sockets that all share the same IP address and port number
  - Ex: a busy web server, to which a thousand clients have made HTTP connections, will have a thousand active sockets all bound to its public IP address at port 80

- What makes an active socket unique is a **four-tuple**: 
  
  (local_ip, local_port, remote_ip, remote_port)

- It is this four-tuple through which the operating system names and manages each active TCP connection
Example: a simple server (cont’d)

import socket

HOST = ""
PORT = 1060

sock = socket.socket(AF_INET, SOCK_STREAM)
sock.bind(((HOST,PORT)))
sock.listen(5)
while 1:
    sock_cli, addr = sock.accept()
...

# SERVE THE REQUEST
Send data

`numBytesSent = socket.send(string[, flags])`

- `string` represents the data to be sent
- `numBytesSent` represents the number of bytes sent
- NB: applications are responsible for checking that all data have been sent
  - if only some of the data were transmitted, the application needs to attempt delivery of the remaining data.
- TCP considers your outgoing and incoming data as streams, with no beginning or end
  - It feels free to split them up into packets however it wants!
Send data

- After a TCP send(), networking stack will face one of three situations
  - The data can be immediately accepted by the system
    - send() returns immediately, and it will return the length of your data string
  - The network card is busy and outgoing internal data buffer for this socket is full
    - send() blocks, pausing your program until the data can be accepted
  - The outgoing buffer is almost full
    - send() completes immediately and returns the number of bytes accepted from the beginning of your data string, but leaves the rest of the data unprocessed
Send data

- `send()` is usually called inside a loop like this...
  ```python
  bytes_sent = 0
  while bytes_sent < len(message):
      message_remaining = message[bytes_sent:]
      bytes_sent += sock.send(message_remaining)
  ```

- ...or it is replaced by:
  ```python
  socket.sendall(string[, flags])
  ```

- It continues to send data from `string` until either all data have been sent or an error occurs.
- It is more efficient than the above example, because it is implemented in C.
- Example: `sock.sendall(message)`
Receive data

\[ data = \text{socket.recv}(\text{bufsize}, \text{flags}) \]

- `bufsize` is an integer that specifies the maximum amount of data to be received at once.
- `data` is a string representing the data received.
- NB: similarly to `send()`, applications are responsible for checking that all data have been received!
- Unfortunately, we do not have a function similar to `sendall()`.
Receive data

- The operating system’s implementation of recv() is similar to that of send():
  - If no data are available, then recv() **blocks** and your program pauses until the data arrive.
  - If plenty of data are available in the incoming buffer, then recv() returns `#bufsize` bytes.
  - If the buffer contains a bit of data, but less than `#bufsize`, then you are immediately returned the available data, even if they are not as much as the requested data.

- recv() returns empty string if there are no more data.
  - This means that the other end of the connection has been closed (see next slides).
Receive data

- **Problem**: how can we understand if we have received **all** the data?
def recv_all(sock, length):
    data = ''
    while 1:
        read_data = sock.recv(length)
        if read_data == '':
            break
        data += read_data
    return data

We read data until the other end of the connection has been closed.
def recv_all(sock, length):
    data = ''
    while len(data) < length:
        read_data = sock.recv(length - len(data))
        if read_data == '':
            raise EOFError('socket closed')
        data += read_data
    return data

We keep reading until we receive #length bytes

If the connection is closed unexpectedly we raise an error
Example: a simple server

```python
import socket

HOST = ""
PORT = 1060

sock = socket.socket(AF_INET, SOCK_STREAM)
sock.bind((HOST,PORT))
sock.listen(5)

while 1:
    sock_cli, addr = sock.accept()
    message = recv_all(sock_cli, 16)
    print 'The incoming sixteen-octet message says', repr(message)
    sock_cli.sendall('Hello World!')
    sock_cli.close()
    print 'Reply sent, socket closed'
```
Close a connection

- Close the socket
- All future operations on the socket object will fail
- Releases the resource associated with a connection but does not necessarily close the connection immediately
  - Operating system first sends data that are still in the buffer

socket.close()
Close a connection

socket.shutdown(how)

- Shut down one or both halves of the connection
  - Shut down communication in one direction but without destroying the socket itself
- how can be set to:
  - SHUT_RD, further receives are disallowed
  - SHUT_WR, further sends are disallowed
  - SHUT_RDWR: further sends and receives are disallowed
  - NB: It is different from close()
Socket options

socket.setsockopt(level, optname, value)

- There are many options that can be set to sockets
  - level specify the protocol level
    - SOL_SOCKET: generic socket options
    - SOL_TCP: TCP socket options
  - optname is the name of the option
    - SO_KEEPALIVE: enables the periodic transmission of messages on a connected socket
    - SO_REUSEADDR: enables local address reuse
    - SO_SNDBTIMEO: set timeout value for output
    - SO_RCVTIMEO: set timeout value for input
  - value is the option value (it is option dependant)
Example: TCP ECHO server!
Connectionless communication

Server

- socket()
- bind()
- recvfrom()
- sendto()

Client

- socket()
- connect()
- sendto()
- recvfrom()

bind the socket to a well-known port

Send request

Receive response
Send data

$numBytesSent = socket.sendto(string[, flags], address)$

- $string$ represents the data to be sent
- $address$ represents the address of remote host
  - Communication is connectionless!!
- $numBytesSent$ represents the number of bytes sent
- NB: communication is not reliable!
- There are no guarantees that the packet is successfully delivered to remote host
Receive data

```
string, address = socket.recvfrom(bufsize[, flags])
```

- `bufsize` is the maximum amount of data to be received
- `string` represents the received data
- `address` represents the address of remote host
  - Communication is connectionless!!
- NB: receives packets from any remote host
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

MAX = 65535
PORT = 1060

sock.bind(('127.0.0.1', PORT))

while True:
    msg, address = sock.recvfrom(MAX)
    print 'The client at', address, 'says', repr(msg)
    response = 'The msg was %d bytes long' % len(msg)
    sock.sendto(response, address)
Connecting UDP sockets

- We can use the `connect()` function with UDP sockets!
- We can avoid to specify every time the server address when we call `sendto()`!
- Client is not susceptible to receiving packets from other senders
- NB: using `connect()` on an UDP socket does not send any data over the network!!
Problem: What if the response sent by the server is lost?

We do not want to block the client forever...

...but it is not easy to understand why the packet has not arrived:
- The reply is only taking a long time to come back
- The reply (or the request!) is lost
- Server is down

Solution: use a timeout!

if #value seconds elapse since the process is blocked, the OS raises a socket.timeout exception
Example: setTimeout()

sock.connect((HOST, PORT))
delay = 0.1
while True:
    sock.send('Send this message!')
    sock.settimeout(delay)
    try:
        data = sock.recv(MAX)
    except socket.timeout:
        delay *= 2  # Exponential backoff
        if delay > 2.0:
            raise RuntimeError('Maybe the server is down')
    else:
        break  # we are done
Example: UDP server!
Want to know more?

- **Book:**
  - Foundations of Python Network Programming, by Brandon Rhodes and John Goerzen

- **Python official documentation:**
  - [https://docs.python.org/2/library/socket.html](https://docs.python.org/2/library/socket.html)
  - [https://docs.python.org/2/howto/sockets.html](https://docs.python.org/2/howto/sockets.html)