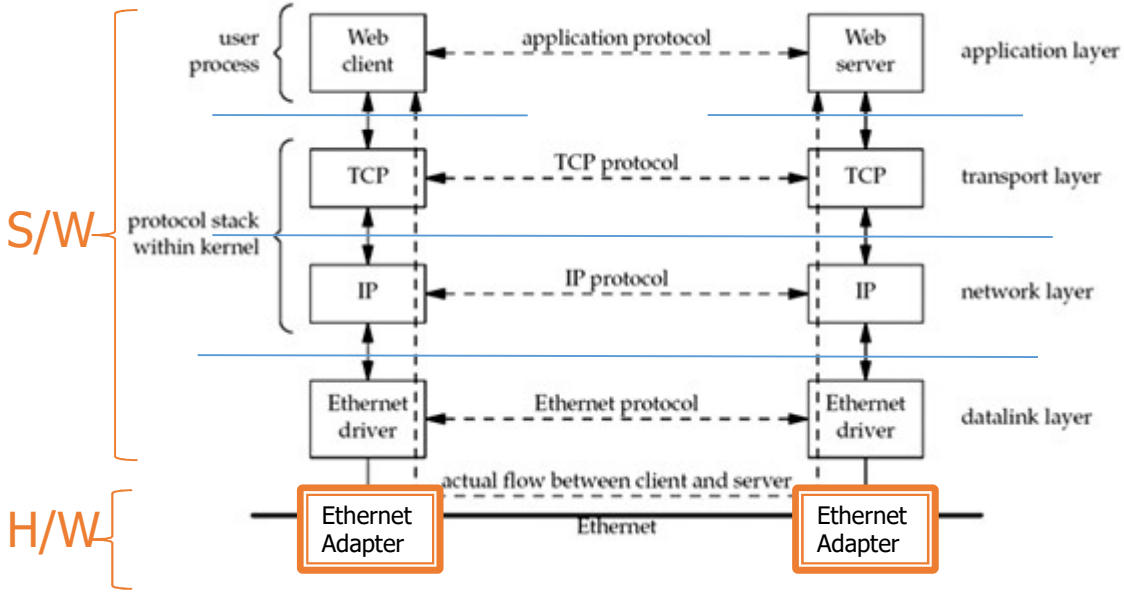
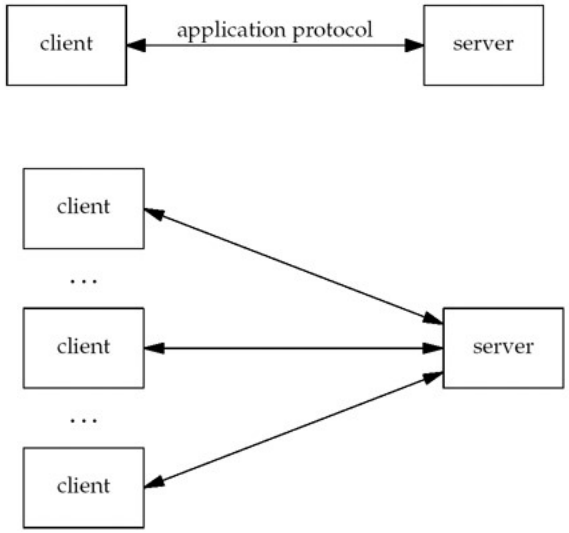
Network Programming in Python

Introduction

Protocols and their Implementation

- Protocol Implementation
  - 보통 transport layer 이하 protocol들은 kernel (OS) 내에 상주한다. Why?
  - Link layer는 H/W와 이를 컴퓨터에 장착하기 위한 S/W (예: Ethernet driver)로 구현된다.
  - Application protocol은 user process (즉, application) 으로 구현된다.
- Client-server model
  - Internet에서 많은 application protocol은 이 model을 가정하고 있다.
  - Server는 여러 client의 요청을 동시에 처리할 수 있어야 한다.
- Peer-to-peer model
  - Peer내에 client와 server 모두 가지고 있는 model로 생각할 수 있다.





Identify a Processes on Internet

- Host(machine) by 32-bit IP address
  - dotted decimal notation: '203.253.70.32'
  - domain name: 'mclab.hufs.ac.kr'
  - 'localhost' → '127.0.0.1'
- Process in the host by 16-bit port number
  - well-known port: 0 ~ 1023
    - reserved by standard protocols
    - identifies the standard service(server process)
  - registered port: 1024 ~ 49151
  - dynamic port: 49152 ~ 65535
    - dynamically assigned for clients
- A process on Internet can be identified by
  - socket address: (host, port)
    - `('mclab.hufs.ac.kr', 80)`

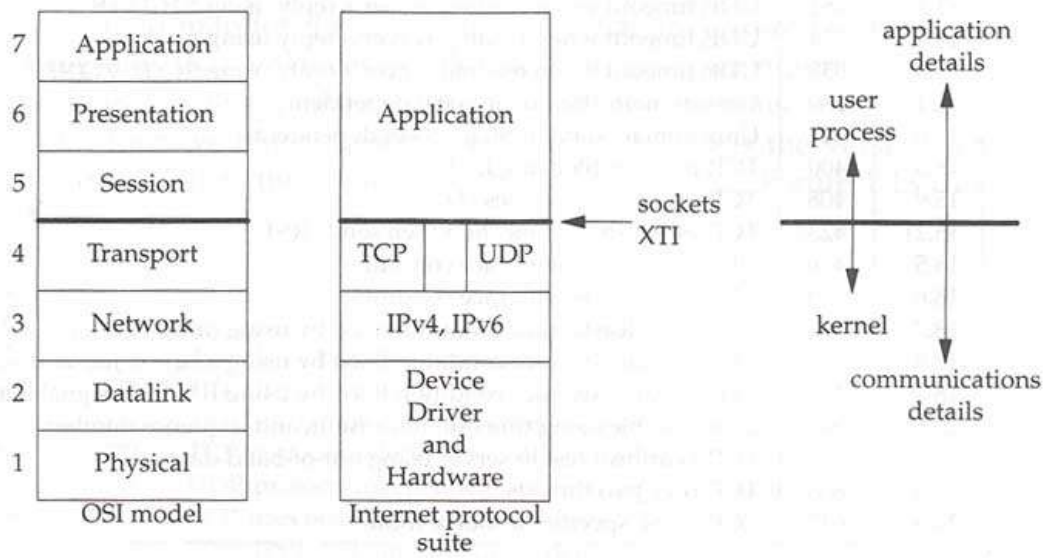
Protocol	Common function	Port number	Python module
HTTP	Web pages	80	`http.client, http.server`
NNTP	Usenet news	119	`nntplib`
FTP data default	File transfers	20	`ftplib`
FTP control	File transfers	21	`ftplib`
SMTP	Sending email	25	`smtplib`
POP3	Fetching email	110	`poplib`
IMAP4	Fetching email	143	`imaplib`
Finger	Informational	79	n/a
SSH	Command lines	22	n/a: third party
Telnet	Command lines	23	`telnetlib`

Common Internet-related Python modules

Python modules	Utility
socket, ssl	Network and IPC communications support (TCP/IP, UDP, etc.), plus SSL secure sockets wrapper
cgi	Server-side CGI script support: parse input stream, escape HTML text, and so on
urllib.request	Fetch web pages from their addresses (URLs)
urllib.parse	Parse URL string into components, escape URL text
http.client, ftplib, nntplib	HTTP (web), FTP (file transfer), and NNTP (news) client protocol modules
http.cookies, http.cookiejar	HTTP cookies support (data stored on clients by website request, server- and client-side support)
poplib, imaplib, smtplib	POP, IMAP (mail fetch), and SMTP (mail send) protocol modules
telnetlib	Telnet protocol module
html.parser, xml.*	Parse web page contents (HTML and XML documents)
xdrlib, socket	Encode binary data portably for transmission
struct, pickle	Encode Python objects as packed binary data or serialized byte strings for transmission
email.*	Parse and compose email messages with headers, attachments, and encodings
mailbox	Process on disk mailboxes and their messages
mimetypes	Guess file content types from names and extensions from types
uu, binhex, base64, binascii, quopri, email.*	Encode and decode binary (or other) data transmitted as text (automatic in email package)
socketserver	Framework for general Net servers
http.server	Basic HTTP server implementation, with request handlers for simple and CGI-aware servers

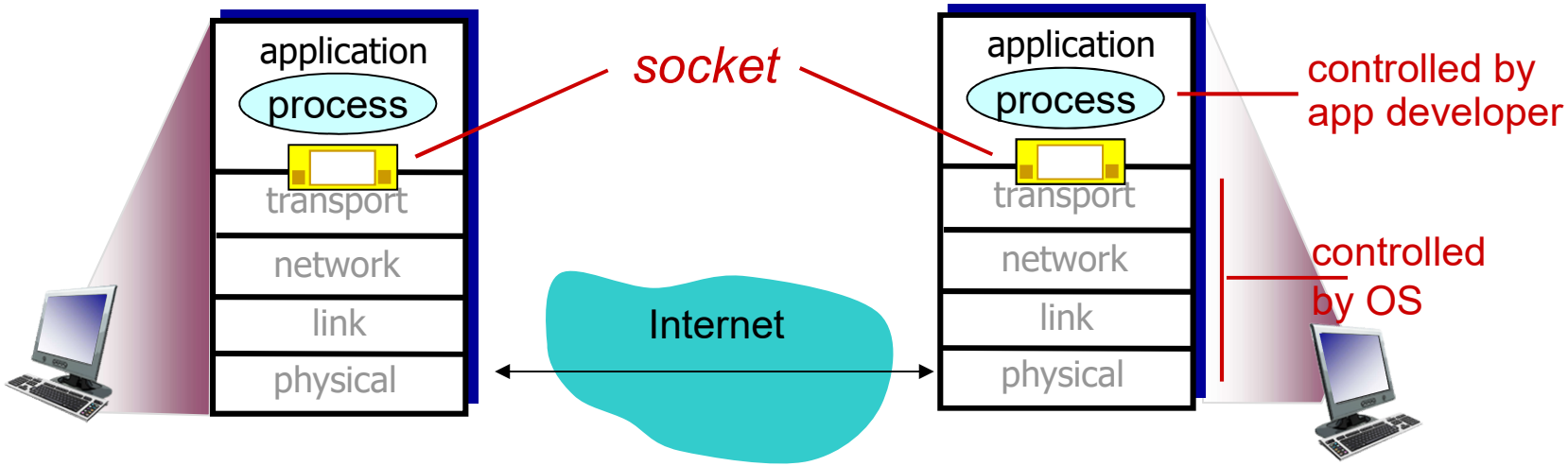
Socket API

- API: Application Program Interface
  - User process가 kernel사이의 interface를 API라고 한며, 흔히 system call이라고 부른다.
  - Function call vs. system call
- Socket API is an API for communications
  - 가장 널리 쓰이는 Communication API
  - 여러 protocol suite에 대해 generic API 제공
  - Transport, network, link layer의 서비스를 받을 수 있는 API도 제공



Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



The First Socket Program

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # create a TCP socket
>>> s.connect(('np.hufs.ac.kr', 7)) # connect to echo server
>>> s.send('Hello, np\n')
10
>>> s.recv(1024)    # recv. buf size = 1014 bytes
'Hello, np\n'
>>> s.close()
```

- socket() returns socket object `_socketobject`
  - Most of socket API are methods on socket objects or functions
- Socket address is a tuple: (host, port)

Lab. Get home page of mclab.hufs.ac.kr

```
>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.connect(('mclab.hufs.ac.kr', 80))
>>> sock.send('GET /\r\n')   # '/' refers to home page (web root dir.)
>>> print sock.recv(1024)
>>> sock.close()
```

Socket programming with TCP

- Client must contact server
  - server process must first be running
  - server must have created socket (door) that welcomes client's contact
- Client contacts server by:
  - creating client-local TCP socket
  - specifying IP address, port number of server process
  - When client creates socket: client TCP establishes connection to server TCP

- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Echo Client

```
import sys, socket

def echo_client(server_addr):
    """Echo client"""
    # make TCP/IP socket obj
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(server_addr)        # connect to server process
    while True:
        message = sys.stdin.readline()
        if message == '\n': break
        sock.send(message)           # send message to server over socket
        data = sock.recv(1024)       # receive response from server: up to 1KB
        print data,
    sock.close()                     # close socket to send eof to server

if __name__ == '__main__':
    echo_client(('localhost', 50007))
```

Echo Server - iterative

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_server(my_port):
    """Echo server (iterative)"""
    sock = socket(AF_INET, SOCK_STREAM) # make listening socket
    sock.bind(('', my_port))          # bind it to server port number
                                      # '' = all available interfaces on host
    sock.listen(5)                    # listen, allow 5 pending connects
    print 'Server started'
    while True:                       # do forever (until process killed)
        conn, cli_addr = sock.accept()  # wait for next client connect
                                      # conn: new socket, addr: client addr
        print 'Connected by', cli_addr
        while True:
            data = conn.recv(1024)    # recv next message on connected socket
            if not data: break        # eof when the socket closed
            print 'Server received:', data
            conn.send(data)           # send a reply to the client
        print 'Client closed', cli_addr
        conn.close()                  # close the connected socket

if __name__ == '__main__':
    echo_server(50007)
```

Problems

1. handles only one client at a time
2. may down – no exception handling
   When client abnormally terminated, socket.error exception raised
3. bind error when server relaunched

Echo Server – a solution for problem 2, 3

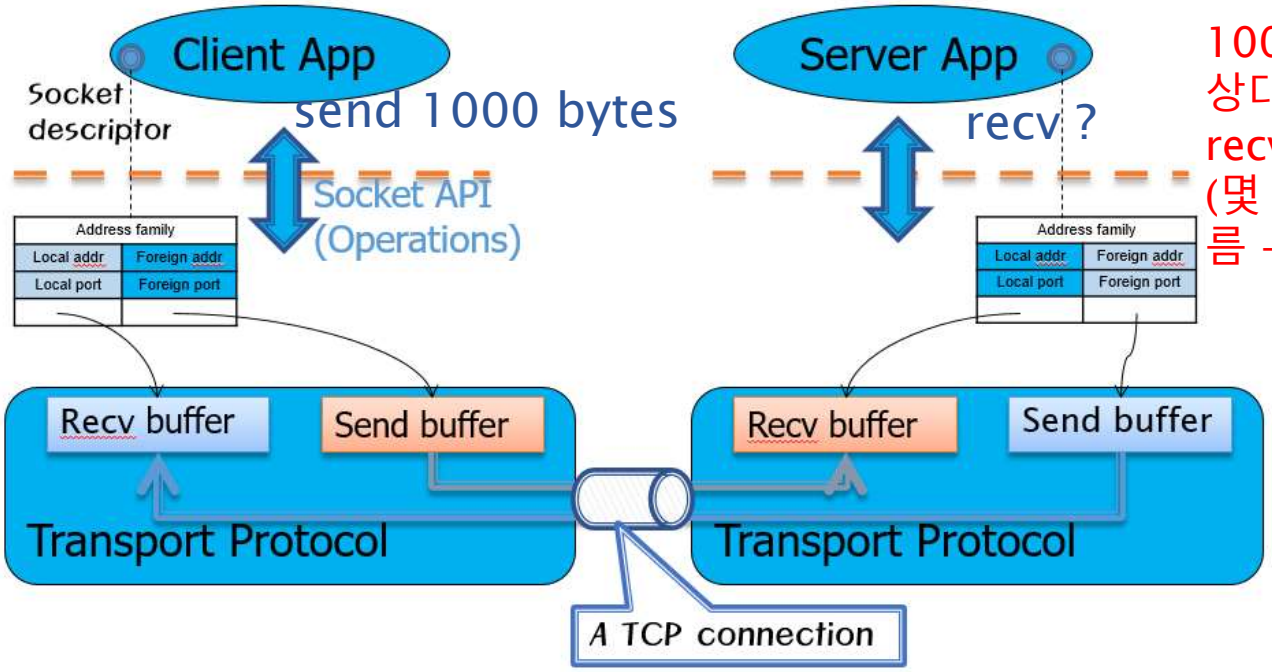
```
from socket import *

def echo_server(my_port):
    """Echo server (iterative)"""
    sock = socket(AF_INET, SOCK_STREAM) # make listening socket
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1) # Reuse port number if used
    sock.bind(('', my_port))        # bind it to server port number
    sock.listen(5)                  # listen, allow 5 pending connects
    print 'Server started'
    while True:                     # do forever (until process killed)
        conn, cli_addr = sock.accept()  # wait for next client connect
                                        # conn: new socket, addr: client addr
        print 'Connected by', cli_addr
        try:
            while True:
                data = conn.recv(1024)  # recv next message on connected socket
                if not data: break      # eof when the socket closed
                print 'Server received:', data
                conn.send(data)         # send a reply to the client
        except error as e:              # socket.error exception
            print 'socket error:', e
        except Exception as e:
            print e
        else:
            print 'Client closed', cli_addr
        finally:
            conn.close()                # close the connected socket

if __name__ == '__main__':
    echo_server(50007)
```

Sending/Receiving via TCP Socket

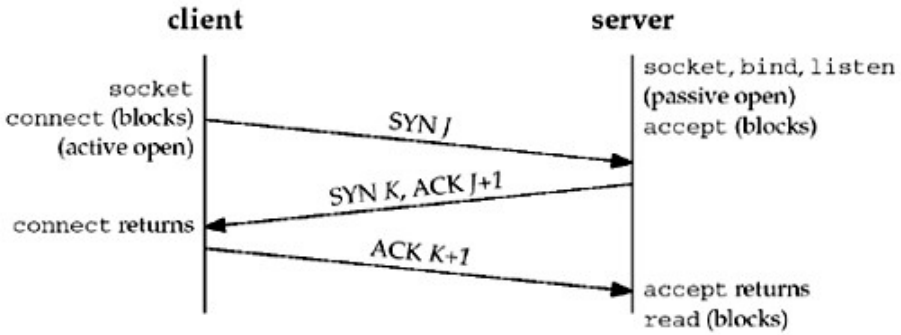
- Full duplex, byte stream transmission (record boundary가 없다.)
- send(msg): put the msg into TCP(socket) send buf.
  - may blocked until TCP send buffer has enough space
- recv(bufsize): get some bytes(<=bufsize) from the TCP recv buf.
  - blocked if TCP recv buffer is empty
- sendall(msg): (non-blocking socket에서도) msg 전부를 send buffer에 저장하기 위해 계속 send()

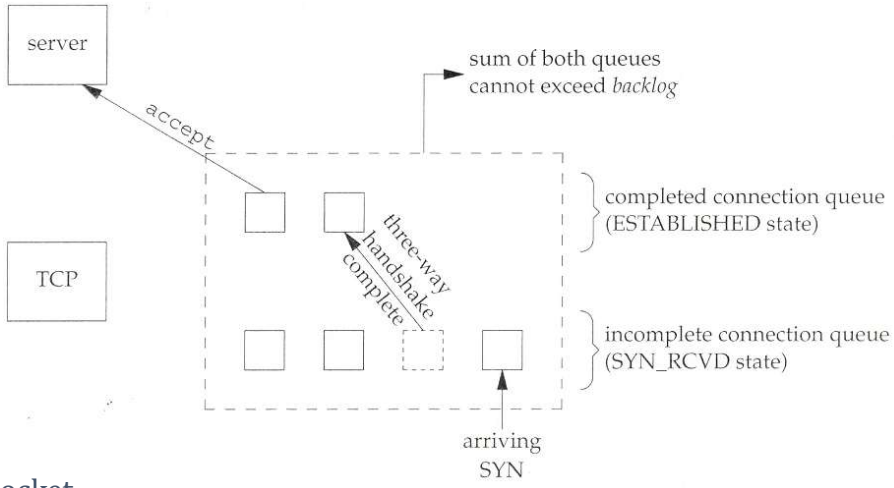


1000 bytes를 send했다고 해서 상대편에서 같은 1000 bytes를 recv 하는 것이 아니다.
(몇 byte를 받을 수 있을지는 모름 – TCP 마음)

Establishing TCP Connection

- 3-way handshake
- Client's perspectives
  - Client connect()가 return되기까지 RTT가 소요된다.
  - Connect()가 fail되는 경우
    - No such server process
    - host/network unreachable
    - 3번 retry하기 때문에 fail임을 아는데 시간 많이 걸린다. (75초)
    - fail되면 socket을 close해야
- Server's perspectives
  - bind(address): set my address (host, port)
    - '' host means any interfaces
  - listen(number): convert to the listening socket
    - number: connection queue size (not exactly)
    - TCP state: CLOSED → LISTEN (see netstat)
  - conn, addr = accept(listening_socket)
    - accept a connection request and create the connected socket

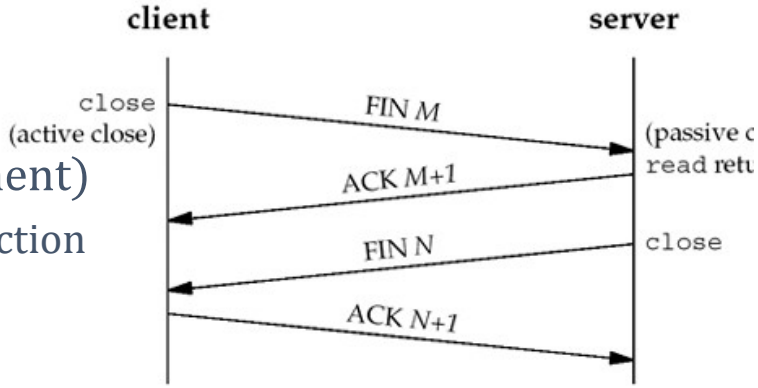




참고: process가 종료될 때 open된 file이나 socket은 자동적으로 close된다

Closing TCP Connection

- close(): close TCP connection and the socket
  - TCP send buffer에 보낼 것이 있으면 모두 send
  - close TCP connection (TCP shall sends FIN segment)
    - 이 socket이 다른 process와 share하고 있다면, no action
  - close the socket: no more use



- shutdown(how): terminate one direction (half closing)
  - initiate TCP's normal termination regardless of sharing with other processes
  - how:
    - SHUT_RD: further receives are disallowed
    - SHUT_WR: further sends are disallowed (TCP sends FIN)
  - socket이 close된 것이 아니다.
    - 예를 들어, shutdown(SHUT_WR) 후에 serve의 response를 recv할 수 있다.