System Programming

13. Network

Seung-Ho Lim

Dept. of Computer & Electronic Systems Eng.

Network Protocols for Communications

- Protocol
  - A pre-defined communication step for ***error-free*** communications on an erroneous data link.
  - Usually a protocol is implemented in a multiple-layered architecture. Why?
    - too big,
    - various level of abstraction,
    - various level of service,
    - variable media & communications types

Network Protocols for Communications

- OSI 7 layers (OSI reference model)
  - ***Physical layer*** : electrical signaling system, (wired, wireless)
  - ***Data link layer*** : error free communications between adjacent nodes;
    - MAC layer (Medium Access Control) : multiple shared accesses to a link/bus (ethernet, RF)
    - Point-to-point (a private link between nodes)
  - ***Network layer*** : Routing (which must be the next node to deliver the received packet to the final target?), ***IP in the Internet Protocol***
  - ***Transport layer*** : host APIs for end-to-end communications
    - ***TCP/UDP in the Internet Protocol***
  - ***Session layer*** : session management, error recovery
  - ***Presentation layer*** : Encryption/decryption, network standard data format, other libraries/utilities (address translation, etc.)
  - ***Application layer*** : ftp, email, rlogin, telnet, web server/browser (http), etc.

Network Protocols for Communications

- A message from a user process can be split into multiple segments in each protocol layer. (**Fragmentation**)
- Messages from a user can be merged into a segment in each protocol layer.
- Each layer attaches a layer's **packet header** to the segment. → a packet frame
  - *TCP segment : TCP Header + Data segment*
  - *IP : IP header + TCP segment + CRC checksum*
  - *…*

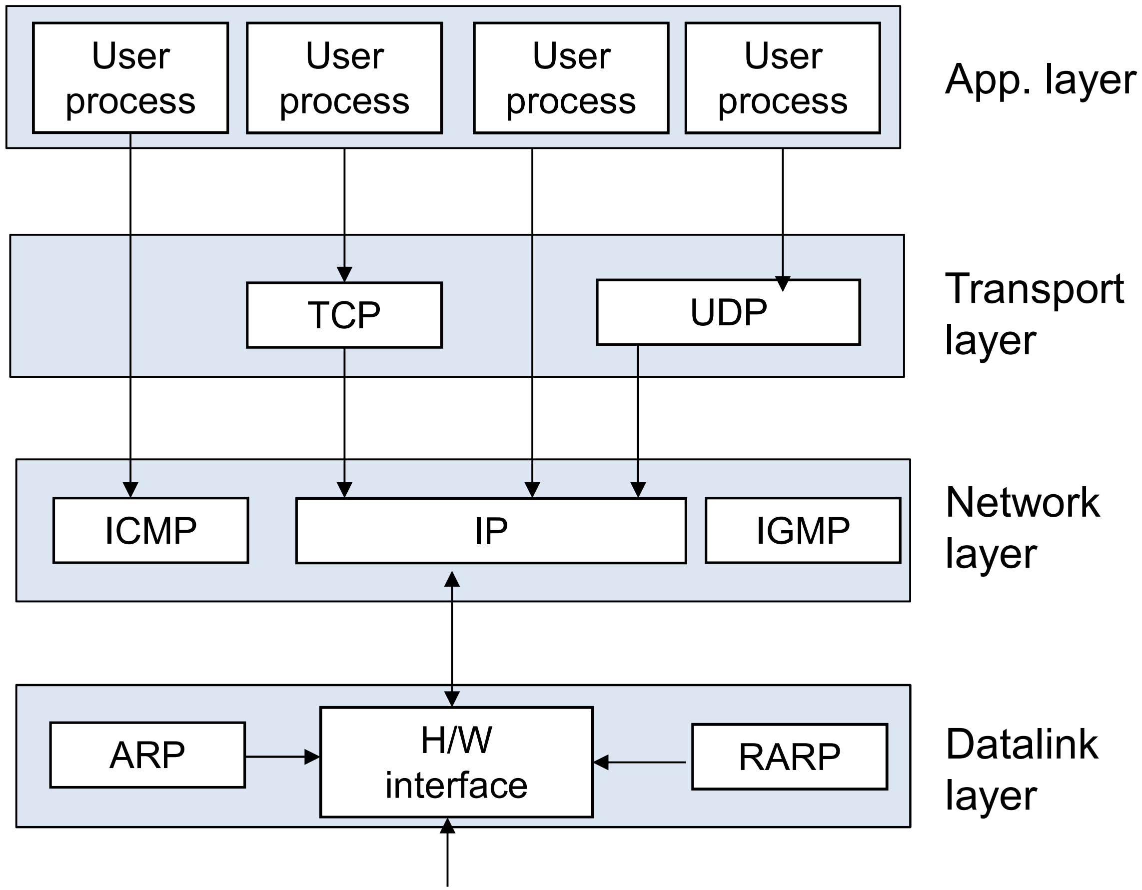
  - Each layer has its ***MTU (Maximum Transfer Unit)*** : max. sized data segment in each layer.

Protocol Examples

Application layer	*Telnet, ftp,* *SNMP (Simple Network Management Protocol), etc.*
Transport layer	*TCP (Transmission Control Protocol),* *UDP (User Datagram Protocol), etc.*
Network layer	*IP, ICMP, IGMP, etc.*
Data Link layer	*Network device drivers,* *Interface/controller cards, etc.*

- ICMP (Internet Control Message Protocol)
  - A protocol used by the "ping" service
- IGMP (Internet Group Message Protocol)
  - A router's protocol for multicasting

Protocol Usages

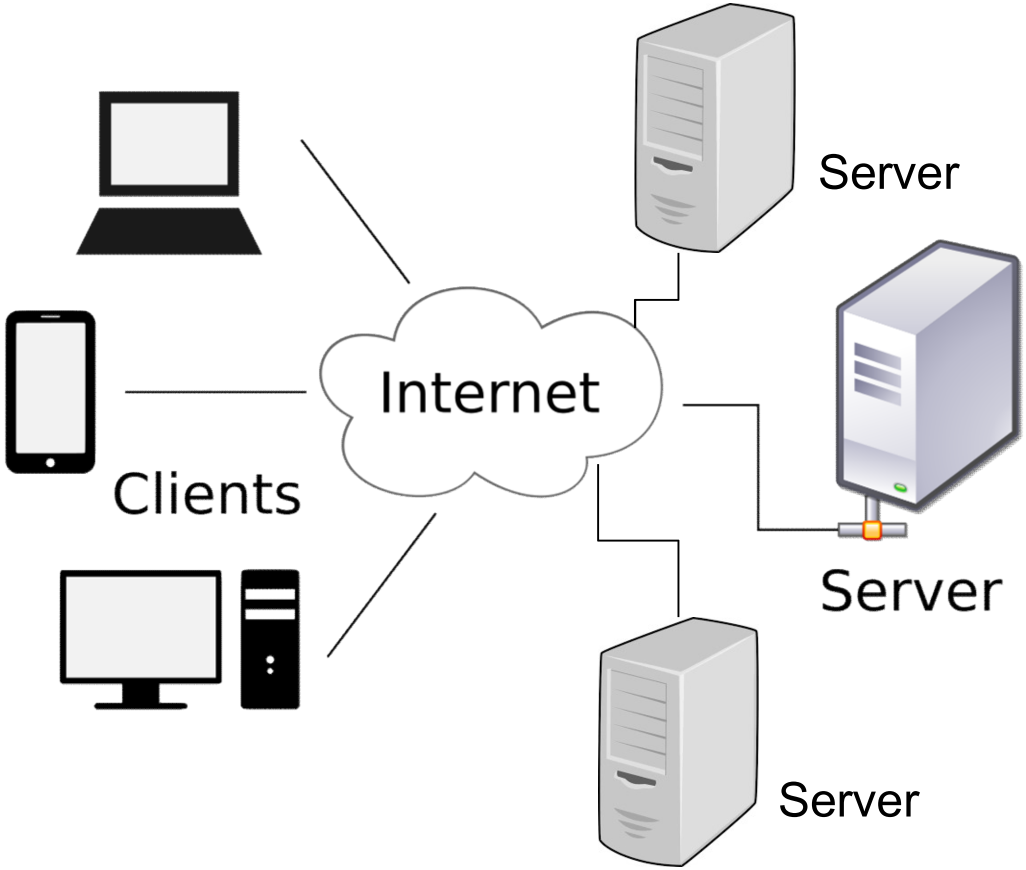


ARP (Address Resolution Protocol): IP to a physical MAC address
RARP (Reverse Address Resolution Protocol) : Physical MAC address to an IP

App. Layer Services on Transport Protocols

Transport Protocols	Application Services
TCP	FTP (File Transfer Protocol) TELNET SMTP (Simple Mail Transfer Protocol) HTTP (Hyper Text Transfer Protocol) of WWW
UDP	SNMP (Simple Network Management Protocol) TFTP (Trivial FTP)

Client/Server Model (1)

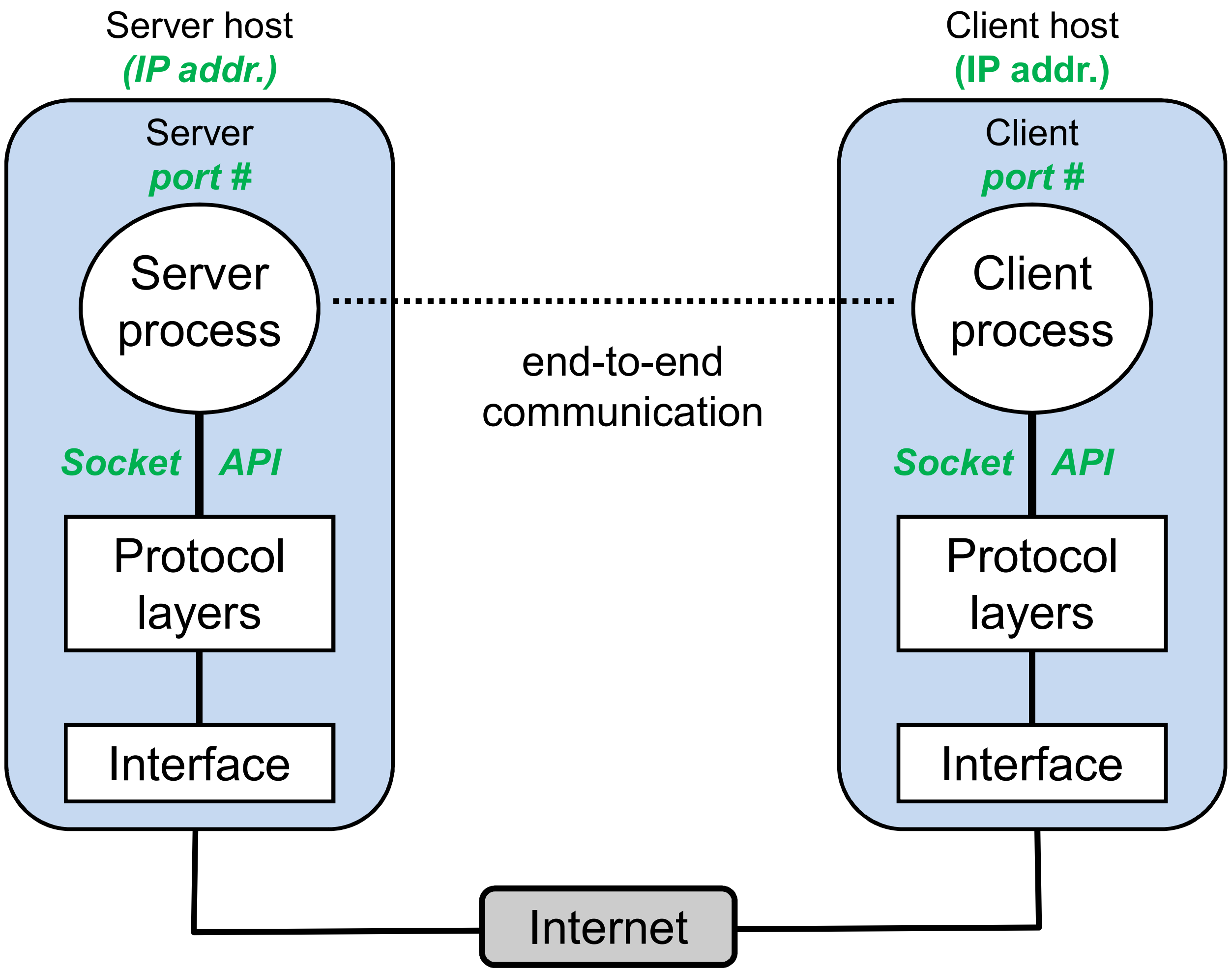


- *Types of network application services*
  - **Client/server** : a client program ↔ a server program
  - **Web-based** : web-browser(client) ↔ web-server(server)
  - **Peer-to-peer(P2P)** : a node can be a server or can be a client

Client/Server Model (2)

- Both of client & server processes (programs) are necessary.
  - Client & server processes use a set of protocol APIs.
  - A server demon is always waiting for connections of clients.
    - Upon a connection, the server prawns a child (or a thread) to serve the client.
  - A client tries to connect to a server.
    - Target (server/client) addresses are designated by the (***IP address, Port #***) couples.
    - Packets are delivered to a node with the IP address.
    - In the target node, a network protocol delivers the packet to the server/client process/thread using the port #.

Client/Server Model (3)



Connection-oriented vs. Connectionless Communications

Types	Description
Connection-oriented	Use the ***TCP*** (Transmission Control Protocol) protocol, Reliable data transfer is guaranteed, Must set up a link to each client, More clients, higher pressure on the server. Think telephone communications! (Virtual Circuit)
Connectionless	Use the ***UDP*** (User Datagram Protocol) protocol, Suitable for a single message transmission, No link for clients: low pressure to the server, Can be used for broadcast or multicast services. Think datagram of post offices!

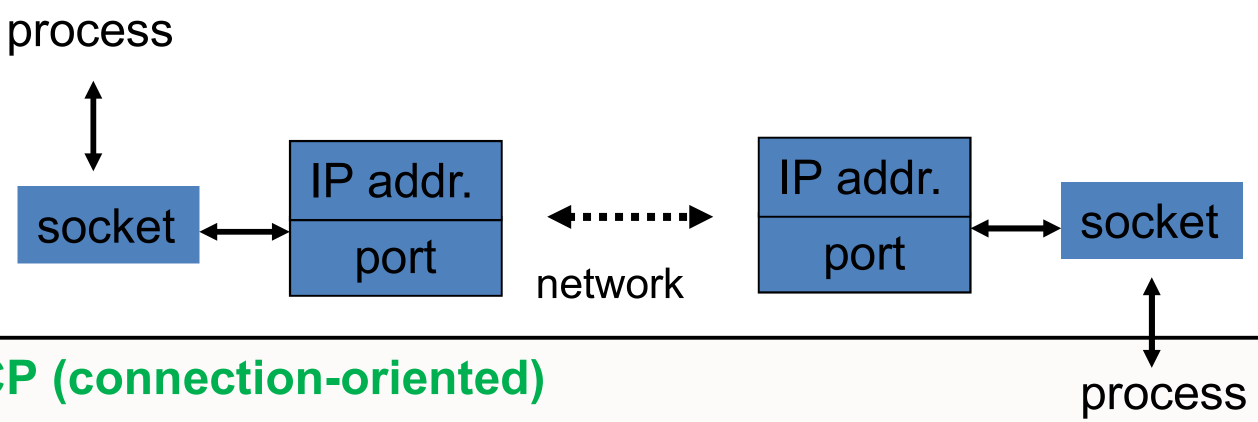
Server Types

Server types	*Description*
Repetitive server type ***(single-threaded server)***	A single server process handles the requests of multiple clients one by one. (e.g. FIFO) Slow response.
Concurrent server type (***multi-threaded server***)	A server process consists of several concurrent threads. For each client, a server thread/process is created at the connection time. (or pre-created at the time of server-launch) High performance. Concurrency problems (e.g. Mutex)

Programs/APIs of Each Layer

Layer	APIs, Programs	Description
App. layer	***http, ftp, email, rsh, RPC***	Application services for easy use.
Transport layer	***Socket (Berkley), Winsock, TLI***	For ***end-to-end*** communications (message, stream). Supports ***TCP/IP, UDP/IP***,
Device driver layer	Packet Driver, NDIS(Network Driver Interface Spec, window), ODI (Open DataLink Interface)	Handle MAC frame transmission on a LAN. Support various MAC protocols. Error control, flow control.

Socket Communications (end-to-end: TCP, UDP)

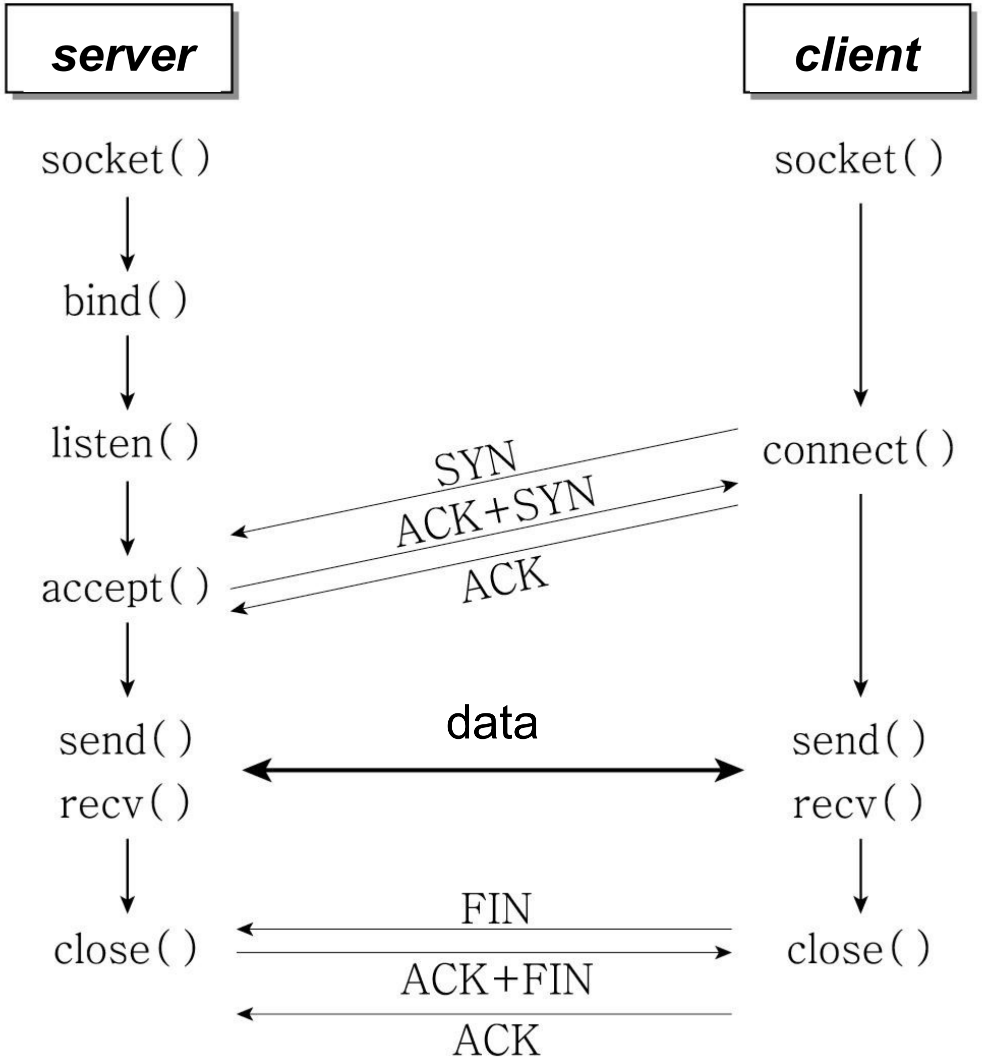


- **TCP (connection-oriented)**
  - Binding : assign (IP addresses, port #) to a socket
  - Every transmission of packets use the same link
  - Stream I/O, reliable, flow control, error control.
- **UDP (connectionless)**
  - For each transmission of a message, IP addr. & port # of the target are necessary.
  - Useful for one-time small message transmission.
  - Message-based, unreliable (a message can be lost), order of message delivery can be reversed.
  - Message size must be smaller than the UDP packet size
  - No flow control, restricted error control, low overhead.

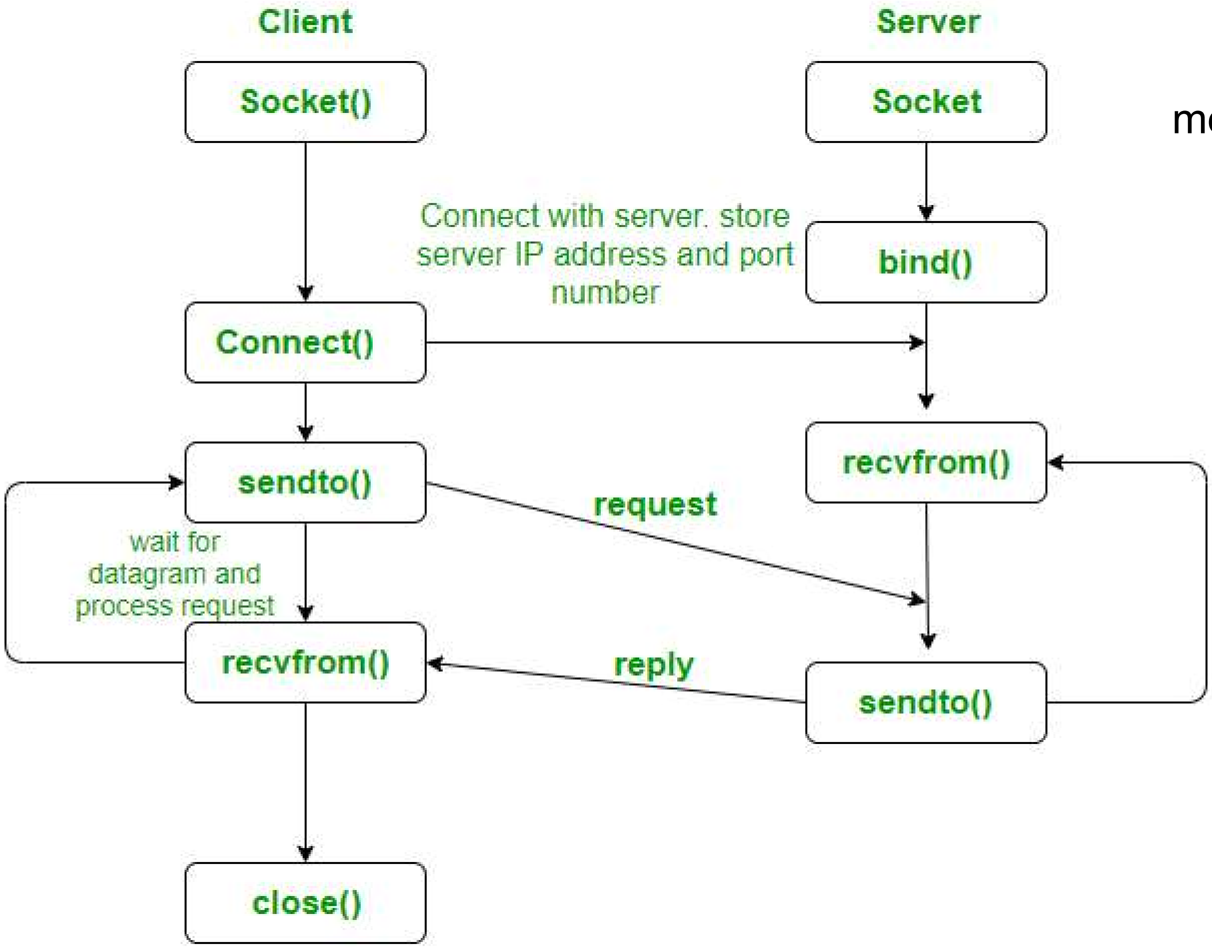
Ports

- A network application process uses a port in the local host.
  - So ***(IP addr. + port #)*** can designate the **peer** process of the remote host.
- Port number usage (0 ~ 65535), **IANA** allocation
  - ***Well-known ports*** (0 ~ 1023) : already assigned to exiting network services. (ex: 23: DNS)
    - *$cat /etc/services | grep tftp → tftp 69/tcp tftp 69/udp*
  - ***Registered ports*** (1024 ~ 49151), ***Dynamic ports*** (49152 ~ 65535)
    - All ports can be used by a user, but a registered port can be registered to IANA.
    - Dynamic ports: free use by users.

Connection-oriented Communication (TCP)



Connectionless Communication (UDP)



Wait for a message from any client

Socket Address Structure

- socket-addr structure (a representative structure for all types of networks)

```
#include <sys/socket.h>

struct sockaddr {
        u_char sa_len;      /* address structure length */
        u_char sa_family  /* address type */
        char sa_data[14]; /* 14 byte-address */
};
```

Socket Address Structure of the Internet

```
#include <netinet/in.h>
#include <sys/types.h>
// sockaddr internet (practical address structure used by internet TCP/UDP)
// when use this structure in relevant syscalls, address casting to sockaddr is needed

struct sockaddr_in {
        u_char sin_len;                 // address structure length
        u_char sin_family;              // address type
        u_short sin_port;  // 16 bit port #
        struct  in_addr  sin_addr // 32 bit  IP address
        char sin_zero[8];   // not used, for the further use: must set to be all zeros
};

struct in_addr
{
    u_long          s_addr;             // 32 bit IP address
}
```

Socket Address Structure

- **sin_family:**
  - ***AF_INET***: internet IP address
  - ***AF_UNIX***: UNIX or used for local communications
    - – The server & client are all in the local host,
    - – But use the whole protocol stack.
    - – Usually be used for testing
  - AF_NS: XEROX network address

socket()

```
#include <sys/socket.h>
#include <sys/types.h>
```

int socket (int domain, int type, int protocol);

input

- ***domain*** : address types
  - *PF_INET: internet protocol*
  - *PF_INET6: IPv6 protocol*
  - *PF_UNIX: UNIX, local communications*
  - *PF_NS: XEROX: Xerox network address*
  - *PF_IMPLINK: IMP link layer address*
- ***type*** : socket type
- ***protocol :*** protocol for use (packet header format)

return

- normal : socket id
- error : -1

type	*protocol*	*actual protocol*
SOCK_DGRAM	*IPPROTO_UDP*	*UDP*
SOCK_STREAM	*IPPROTO_TCP*	*TCP*
SOCK_RAW (user creates a packet header)	*IPPROTO_ICMP*	*ICMP*
SOCK_RAW (user creates a packet header)	*IPPROTO_RAW*	*RAW*

bind()

- Connect my host IP addr. & port # to my socket

```
#include <sys/socket.h>
#include <sys/types.h>
int bind (int sockfd, struct sockaddr *myaddr, int addrlen);
```

input

- sockfd : socket descriptor (file descriptor)
- myaddr : address of the socket address structure
- addrlen : size of the “myaddr” structure

return

- normal : 0
- error : -1

connect()

- A client waits for the connection to a server by sending a “connection-request message”. : telephoning
- Auto-binding to the socket will be done, no bind() is necessary!

```
#include <sys/socket.h>
#include <sys/types.h>
int connect (int sockfd, struct sockaddr *servaddr, int addrlen);
```

input:

- sockfd : socket descriptor
- myaddr : socket addr. structure that contains the IP & port of the server.
- addrlen : size of the “`servaddr`” structure

return:

- normal : 0, error : -1

listen()

- A server declares the max. queue-length of client-requests.

```
#include <sys/socket.h>
#include <sys/types.h>

int listen (int socket, int queuesize);
    input
        - socket : bound socket descriptor
        - queuesize : max. number of client connection requests
    return
        - normal : 0
        - error : -1
```

accept()

- A server waits for (being blocked) a connection request from a client.
- When a connection is established, returns a new socket descriptor for communication with the client.
- The old socket will be used for other client-requests for further connections.

#include <sys/socket.h>

#include <sys/types.h>

int accept (int sockfd, struct sockaddr *peer, int *addrlen);

input:

- sockfd : socket descriptor
- peer : the socket address structure that contains the IP & port of the connected client
- addrlen : size of the "peer" structure

return:

- normal : a new socket descriptor to communicate the client
- error : -1

send() for TCP

#include <sys/socket.h>

#include <sys/types.h>

int send (int sockfd, char * buf, int bytes, int flag);

input

- sockfd : socket descriptor
- buf : data buffer holding the data to be sent
- bytes : size of the buffer
- flag : options
  - » MSG_OOB: OOB(out of bound) data: used for urgent data sending
  - » MSG_PEEK: keep the data in the buffer
  - » MSG_DONTROUTE: ignore the usual routing

return

- normal : the size of the data that actually be sent
- error : -1

send() for TCP

- Completion of send() // successful return
  - This means that the data message has been stored in the sender's protocol buffer successfully.
  - Does not mean that the delivery has been completed.
- The data stored in the protocol buffer will be removed when an ACK message is arrived from the receiver's protocol.
- If **ACK timeout or NACK**, the sender's protocol resends the data packet. (ARP: Automatic Repeat Request)
- TCP send/recv : **stream I/O**
  - A message can be split or messages can be merged at the time of arrival.
  - So if a message splitting is required, the user must handle this by buffering, using of header/length.

recv() for TCP

```
#include <sys/socket.h>
#include <sys/types.h>

int recv (int sockfd, char * buf, int bytes, int flag);
    input
        - sockfd : socket descriptor
        - buf : user's receive-buffer
        - bytes : size of the buffer
        - flag : options (maybe NULL)
    return
        - normal : the size of data actually received
        - error : -1
```

sendto() for UDP

- Send a message to a server using the UDP protocol
- At every sendto(), the IP addr. & port # of the receiver must be given because this is a datagram.

```
#include <sys/socket.h>
#include <sys/types.h>
int sendto (int sockfd, char * buf, int bytes, int flag, struct sockaddr *to, int addrlen);
```

input: - sockfd : scoket descriptor
- buf : buffer address
- bytes : size of the buffer
- flag : options
- to : the address structure containing the receiver's IP address and port #
- addrlen : size of the "to" structure

return: - normal : the size of the data message actually transferred
- error : -1

recvfrom() for UDP

```
#include <sys/socket.h>
#include <sys/types.h>

int recvfrom (int sockfd, char * buf, int bytes, int flag,
        struct sockaddr *from, int *addrlen);
    input
        - sockfd : socket descriptor
        - buf : buffer for data reception
        - bytes : size of the buffer
        - flag : options
        - from : sender's address structure
        - addrlen : size of the "from" structure
    return
        - normal: the size of the message actually received
        - error : -1
```

close()

```
#include <unistd.h>

int close (int sockfd);
    input
        - sockfd : socket descriptor
    return
        - normal : 0
        - error : -1
```

Client of an echo program (TCP)

client.c

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 1024            // max KB input length
#define PORT 30000          // server port #
#define HOSTADDR            “xxx.xxx.xxx.xxx”    // server IP

int main (int argc, char *argv[])
{
    int sd, send_bytes, n, recv_bytes;
```

(con’t)

```
struct sockaddr_in servaddr;
char snddata[MAX], rcvdata[MAX];

bzero ((char*) &servaddr, sizeof(servaddr));  // prepare server address, port
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = inet_addr (HOSTADDR);
servaddr.sin_port = htons (PORT);

if (( sd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf( stderr, "can't open socket.\n");
        exit(1);
}
if (connect (sd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
        fprintf (stderr, "can’t connect to server.\n");
        exit(1);
}  // auto binding for this client and connect request
```

(con’t)

```
    while (fgets (snddata, MAX, stdin) != NULL) {   // get a string from KB
            send_bytes = strlen (snddata);
            if (send (sd, snddata, send_bytes, 0) != send_bytes) { // to server
                        fprintf( stderr, "can't send data.\n");
                        exit(1);
            }
            recv_bytes = 0;
            while (recv_bytes < send_bytes) {  // while loop for stream I/O !
               if ((n = recv (sd, rcvdata + recv_bytes, MAX, 0)) < 0) { // from server
                        fprintf (stderr, "can't receive data.\n");
                        exit(1);
               }
               recv_bytes += n;
            }
            rcvdata[recv_bytes] = 0;              // NULL char for string
            fputs (rcvdata, stdout); // display
    }
    close (sd);
    return 0;
}
```

Server of an Echo Program (TCP)

server.c

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 1024            // max client data length
#define PORT 30000          // server port #
#define BACKLOG 5           // queue length

int main (int argc, char *argv[])
{
    int sd, nsd, pid, bytes, cliaddrsize;
```

(con't)

```
struct sockaddr_in cliaddr, servaddr;
char data[MAX];

if (( sd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf( stderr, "can't open socket.\n");
        exit(1);
}
// to bind the server itself to the socket
bzero ((char*) &servaddr, sizeof( servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
servaddr.sin_port = htons (PORT);
```

- Why INADDR_ANY was used instead of the IP address?
  INADDR_ANY means the all of IP address of a multi-home host.

(con't)

```
if (bind (sd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
        fprintf (stderr, "can't bind to socket.\n");
        exit(1);
}  // bind itself to the socket

listen (sd, BACKLOG);                    // declare the client-queue length

while (1) {       // a typical server waiting loop
        cliaddrsize = sizeof (cliaddr);
        if (( nsd = accept (sd, (struct sockaddr *) &cliaddr, &cliaddrsize)) < 0) {
                fprintf (stderr, "can't accept connection.\n");
                exit(1);
        }  // upon return: client addr. is known and a new socket is created
        if ((pid = fork()) < 0) {   // fork error, a new thread may be used!
                fprintf (stderr, "can't fork process.\n"); exit(1);
        }
```

(con't)

child
server

```
if (pid == 0) {           // the new child server for the connected client
    close (sd);           // old socket is not necessary for me
    while(1) {
            bytes = recv (nsd, data, MAX, 0);  // from client
            if (bytes == 0)          //  client quit
                break;
            else if (bytes < 0) {  // error
                fprintf (stderr, "can't receive data.\n"); exit(1);
            }
            if (send (nsd, data, bytes, 0) != bytes) { // echo back
                fprintf (stderr, "can't send data.\n"); exit(1);
            }
    }   // end while, client quits
            return 0;                // child server exit.
}
else                    //: parent
    close (nsd);  // parent: close the new socket
  }                     // end while: parent goes to the client waiting-loop again
} /* main */
```

Run & Results

```
$ ./server&                    // run the server as a background process
[1] 25345

$ ./client                     // run the client
It's client-server test
It's client-server test
Linux programming
Linux programming
^c
$
```

TCP's stream I/O

- TCP does not guarantee to receive a whole message sent.
  - Messages can be split or merged.
- When the message size is fixed and known.
  - Method 1

```
while(1) {
    len = recv (sd, data, MAX, MSG_PEEK);  // do not remove from TCP buffer.
    if (len >= desired_length)
        break;
}
recv (sd, data, MAX, 0); // read a message & remove from the TCP buffer
```

  - Method 2

```
len=0;
size = sizeof (struct message);  // the fixed known message size
while(1) {
        p = (char*)message_buf +  len;
        len += recv (sd, (void*)p, size-len, 0);
        if ( len >= size)
            break;
}
```

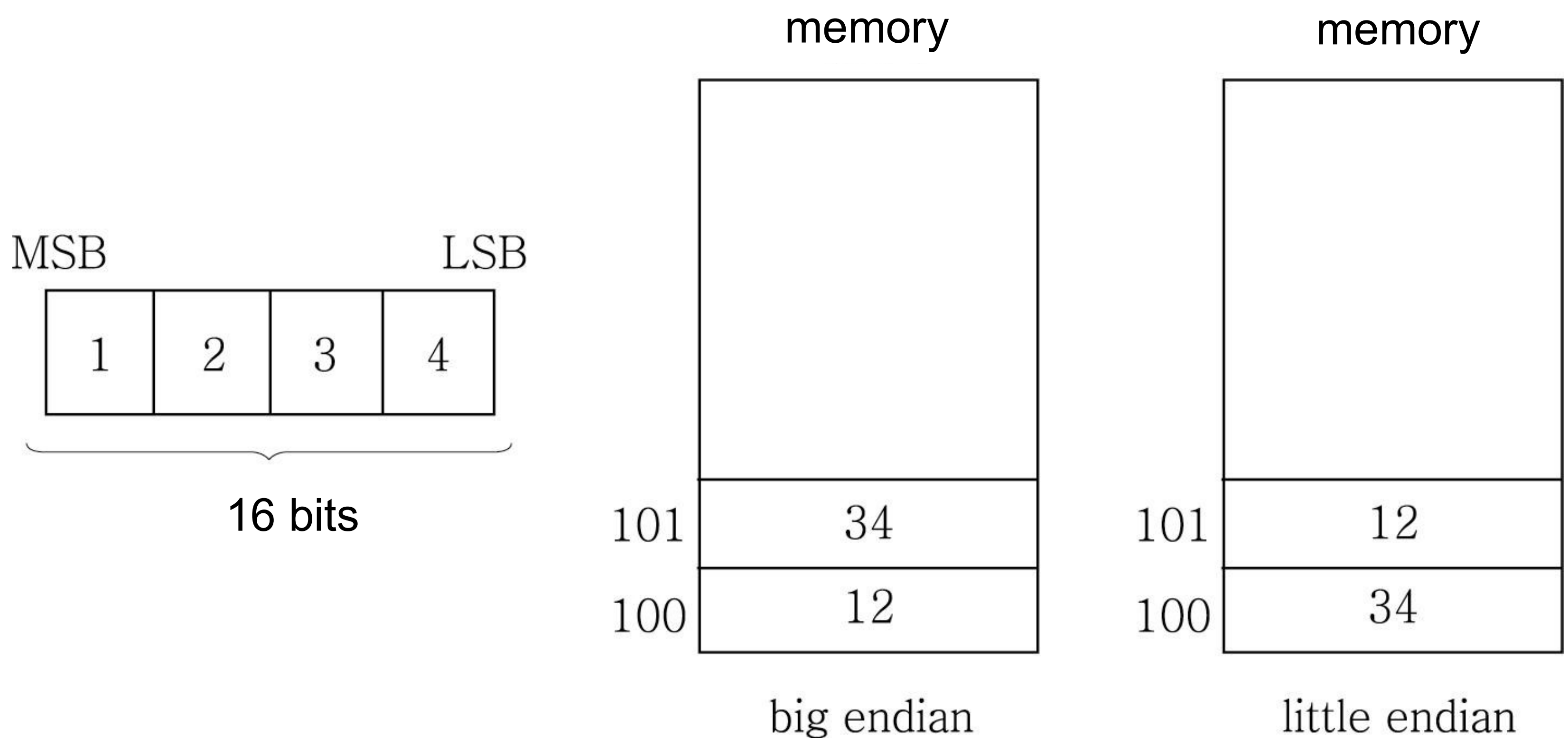
Data Format Conversion

Functions	Description
u_long **htonl** (u_long hostlong);	Convert host long format to the network standard
u_short **htons** (u_short hostshort);	Host to network short
u_long **ntohl** (u_long netlong);	Network to host long
u_short **ntohs** (u_short netshort);	Network to host short

- The above functions are necessary because data formats are machine dependent.
- ***Little endian vs. big endian***
- Different float/double format (exponent, mantissa)
  - Float/double must be handled by a user.

Functions	Description
void **bcopy** (char *src, char *dst, int bytes);	Bytes copy from src to dst
void **bzero** (char *dst, int bytes);	Clear the bytes in dst
int **bcmp** (char *ogn, char *tgt, int bytes);	Compare two byte strings, If same, return 0, else otherwise

Little Endian & Big Endian



Byte-orderings are different.
This is CPU dependent.

Byte ordering, Little endian, Big endian

- Variable's bytes-ordering is reversed in some other machines.
  - For integer, short, ***ntohl(), htonl()*** can be used.
    - Big endian, Little endian, network standard
  - But for float or double, the ordering must be cared by users.
  - Endian may differs from each other.