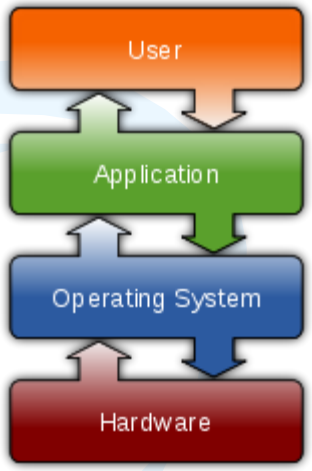
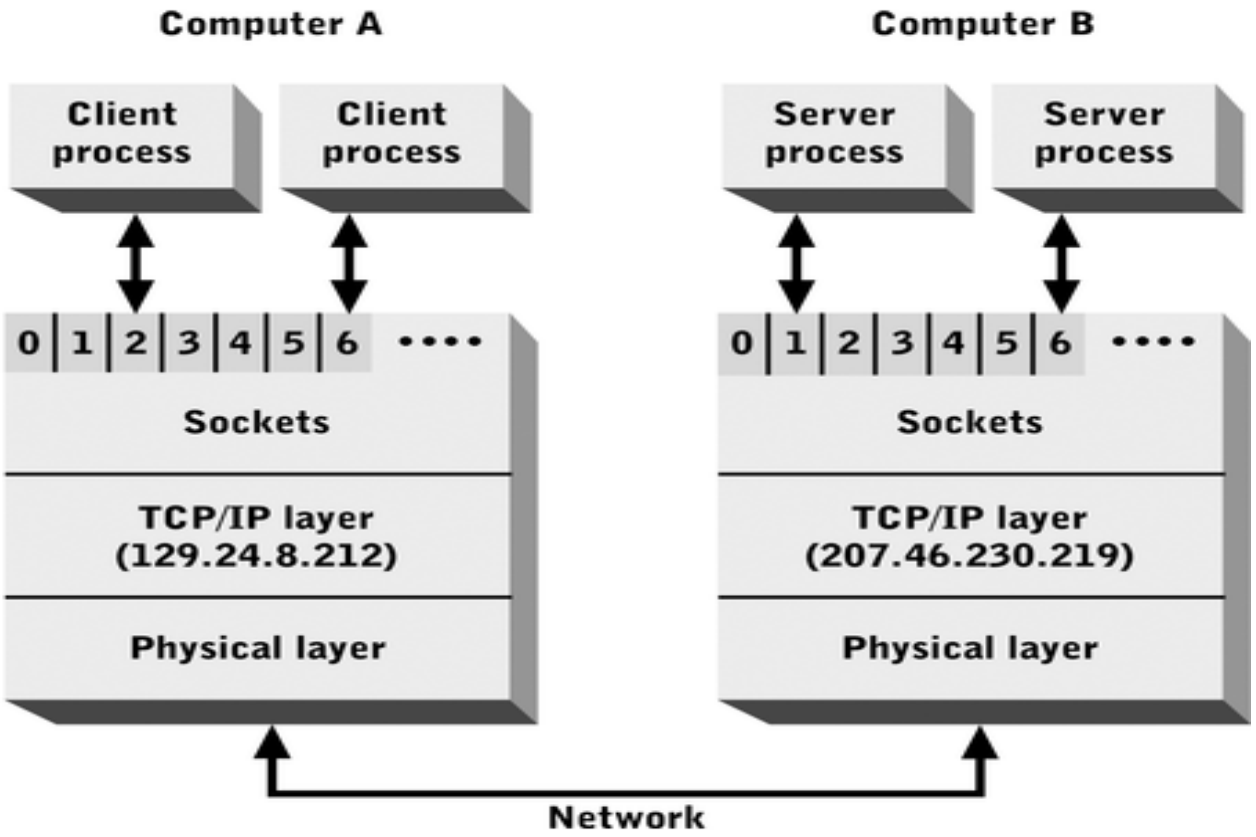
ENCE360
Operating Systems



Interprocess Communication

Sockets

Interprocess Communication Using Sockets



(TCP=Transport Control Protocol, IP=Internet Protocol)

Sockets

Socket API

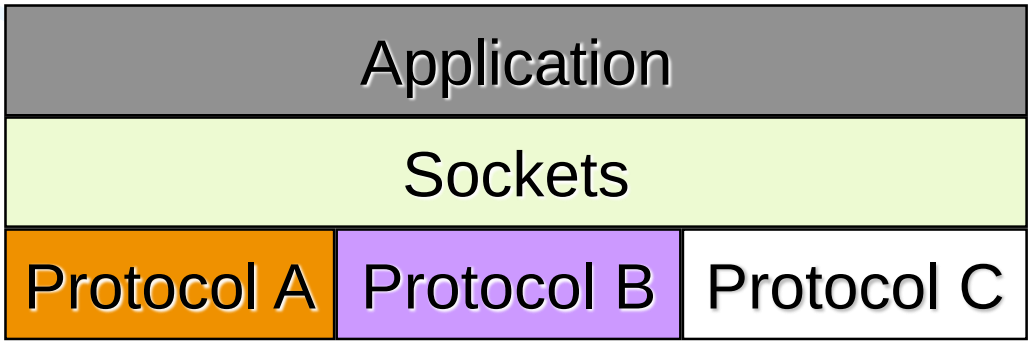
- introduced in BSD4.1 UNIX, 1981
  Two types of sockets
  - connection-oriented (stream)
  - Connectionless (datagram)

socket

an interface into which one application process can both send and receive messages to/from another (remote or local) application process

What is a socket?

- API: network programming interface
- Socket: abstraction through which an application may send/receive data.
- Allows an application to “plug-in” a network and communicates with other applications in the same network.
- Sockets: support different underlying protocol families.



Sockets

- Sockets provide an interface for programming networks at the transport layer.
- Network communication using Sockets is very much similar to performing file I/O
  - In fact, socket handle is treated like file handle.
  - The streams used in file I/O operation are also applicable to socket-based I/O
- Socket-based communication is programming language independent.
  - That means, a socket program written in Python can also communicate with a socket program written in C or Java.

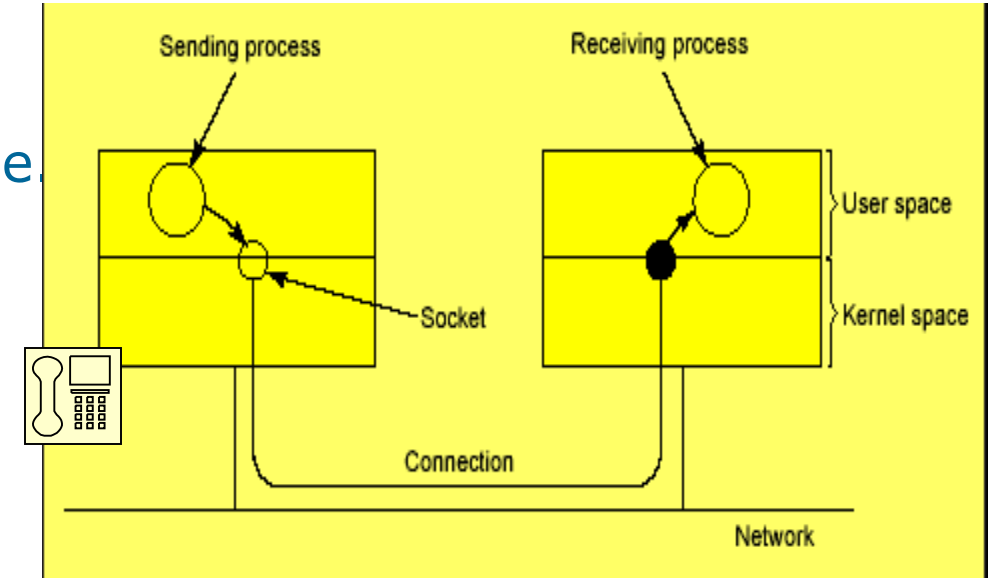
Socket

A socket is one endpoint of a two-way communication link between two programs running on the network.

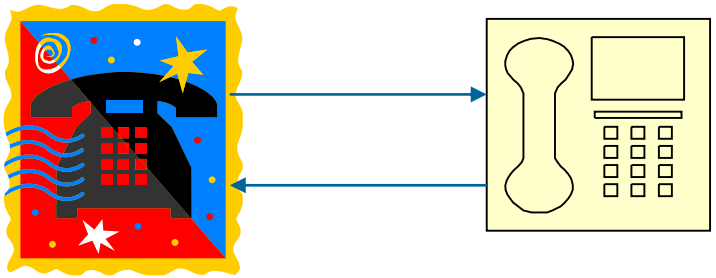
A socket is bound to a port number so that the network can identify the application that data is destined to be sent.

Socket basics

- Sockets connect unrelated processes that may be on different computers.
  - Once established, the socket connection may be read and written like a pipe.
- There are different domains of communication:
  - UNIX/LINUX domain
    - Sockets have actual file names. They can only be used with processes that reside on the same host.
      - Type is **AF_UNIX**
  - Internet domain
    - Allows unrelated processes different hosts to communicate via the Internet
      - Type is **AF_INET**



Analogy:



Socket Names

- Applications refer to sockets by name.
- But within the communication domain sockets are referred by addresses.
- Name to address translation is usually done outside the operating system.

A Socket-eye view of the Internet

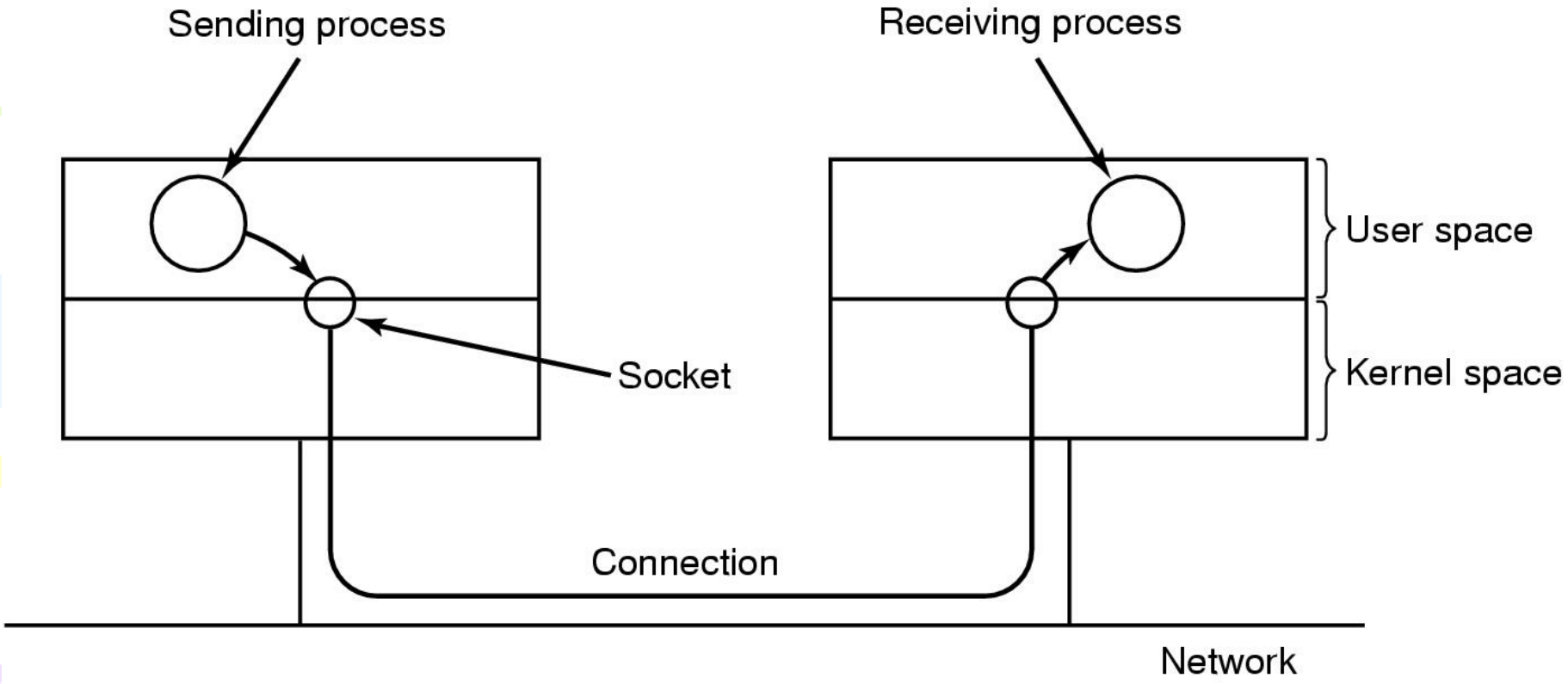
atlas.cs.uga.edu
(128.192.251.4)

www.google.com
(216.239.35.100)

church.cse.ogi.edu
(129.95.50.2, 129.95.40.2)

- Each host machine has an IP address

Networking



Use of sockets for networking

Socket

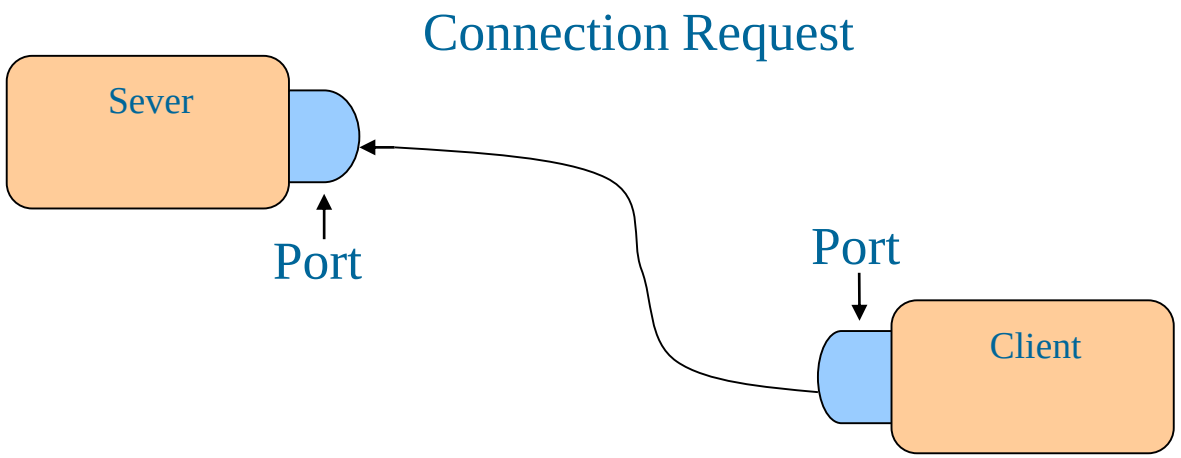
How it works?

The client must know the hostname of the machine on which the server is running and the port number to which the server is connected.

To make a connection, the client sends a request to the server on the server's machine and port.

Socket

How it works?



Socket

How it works?

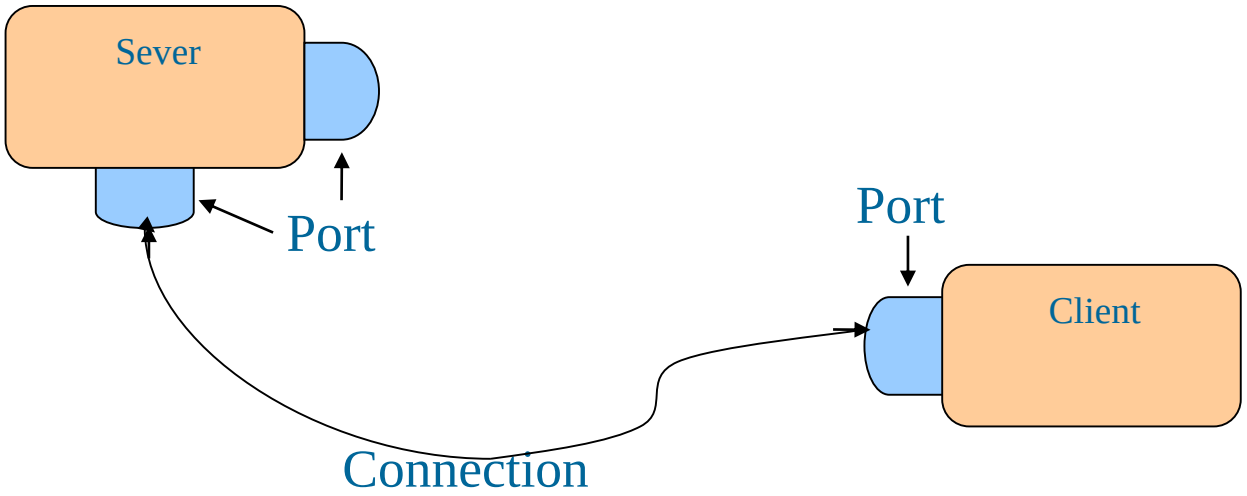
If everything goes well, the server accepts the connection.

Upon acceptance, the server creates a new socket bound to a different port.

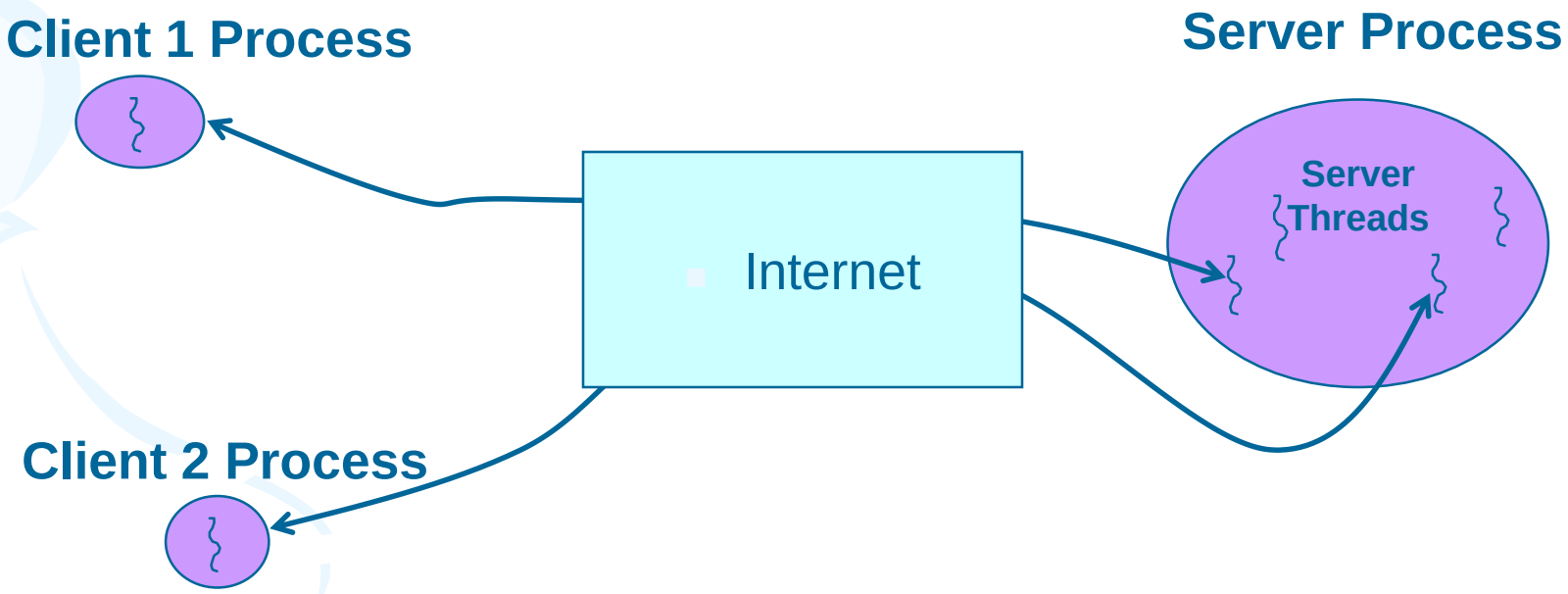
The original server port will be ready to listen for other connection request to the server.

Socket

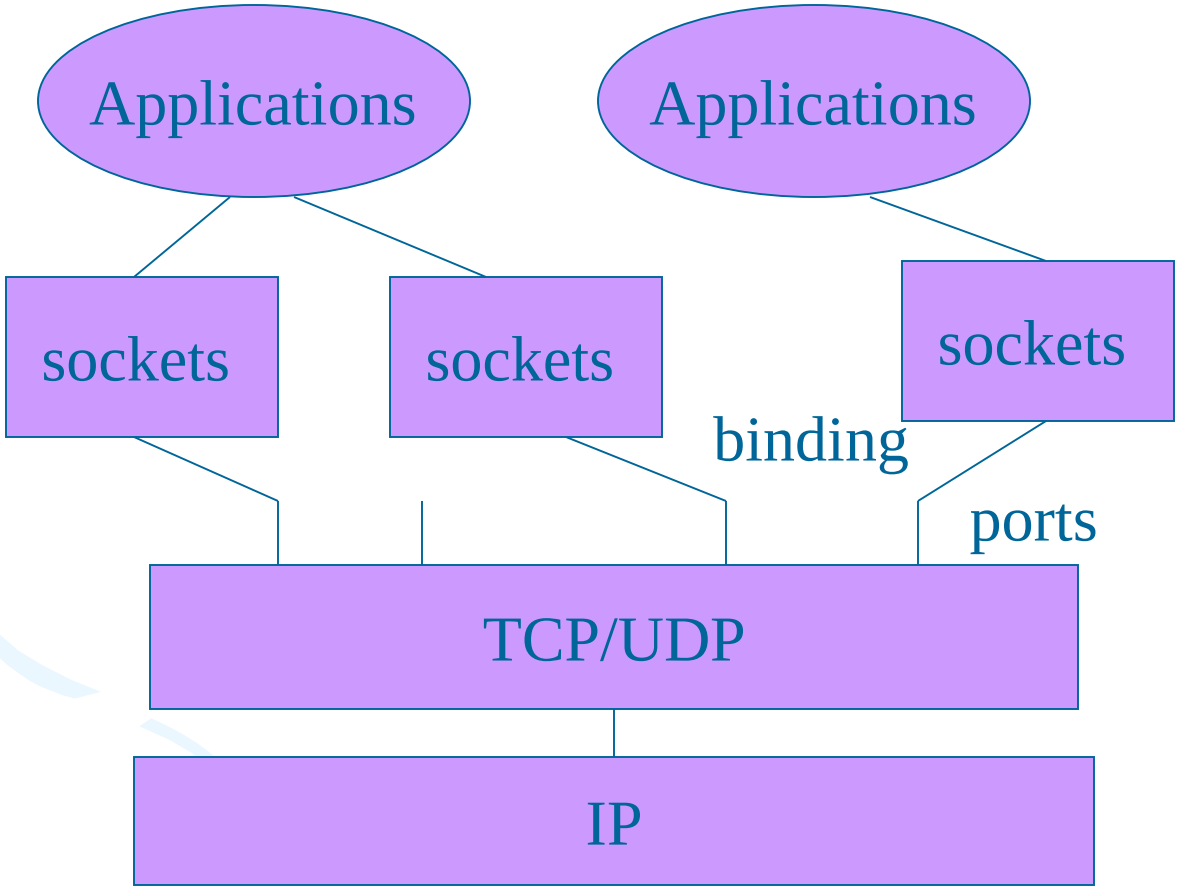
How it works?



Multithreaded Server: For Serving Multiple Clients Concurrently



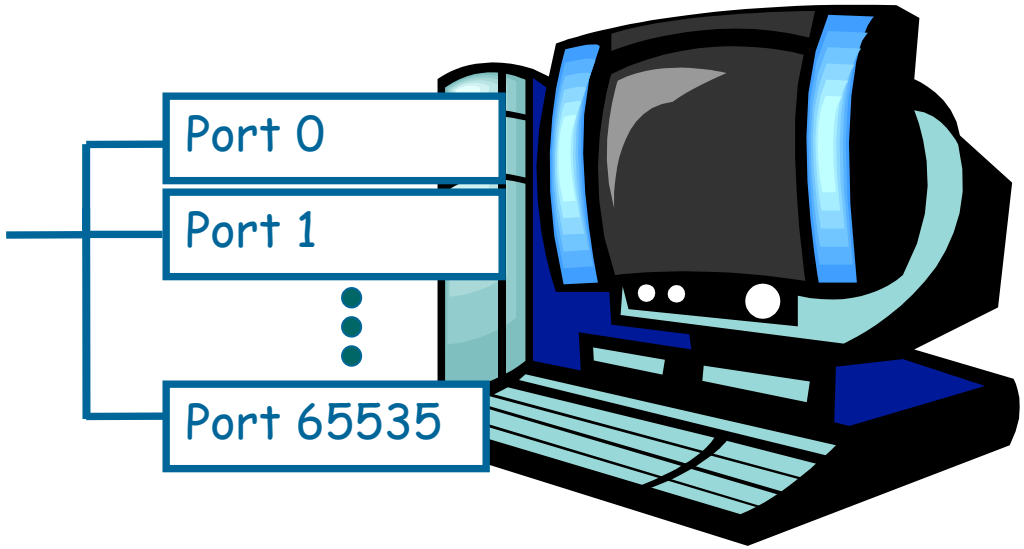
Sockets, protocols, ports



(**TCP**=Transport Control Protocol, **UDP**=User Datagram Protocol **IP**=Internet Protocol)

Ports

- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
  - 20,21: FTP
  - 23: Telnet
  - 80: HTTP



- A socket provides an interface to send data to/from the network through a port

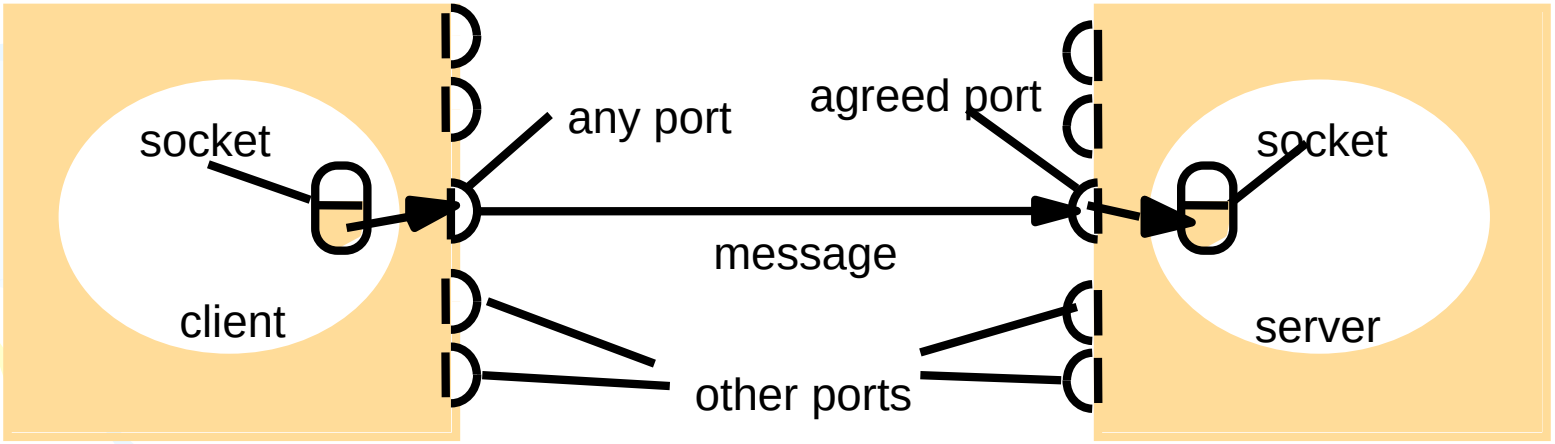
about 1023 ports are reserved

User level process/services generally use port number value >= 1024

Addresses, Ports and Sockets

- Like apartments and mailboxes
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network

Sockets and ports

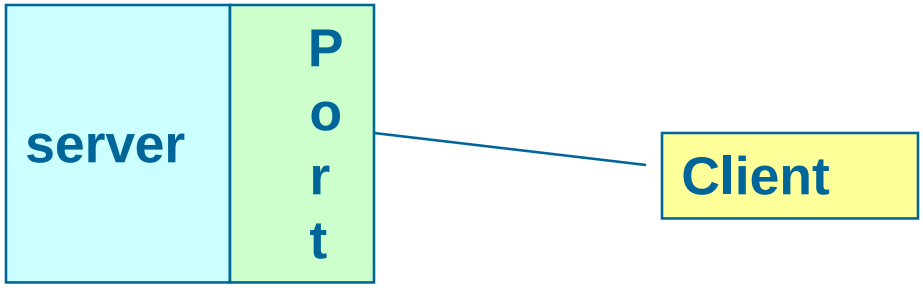


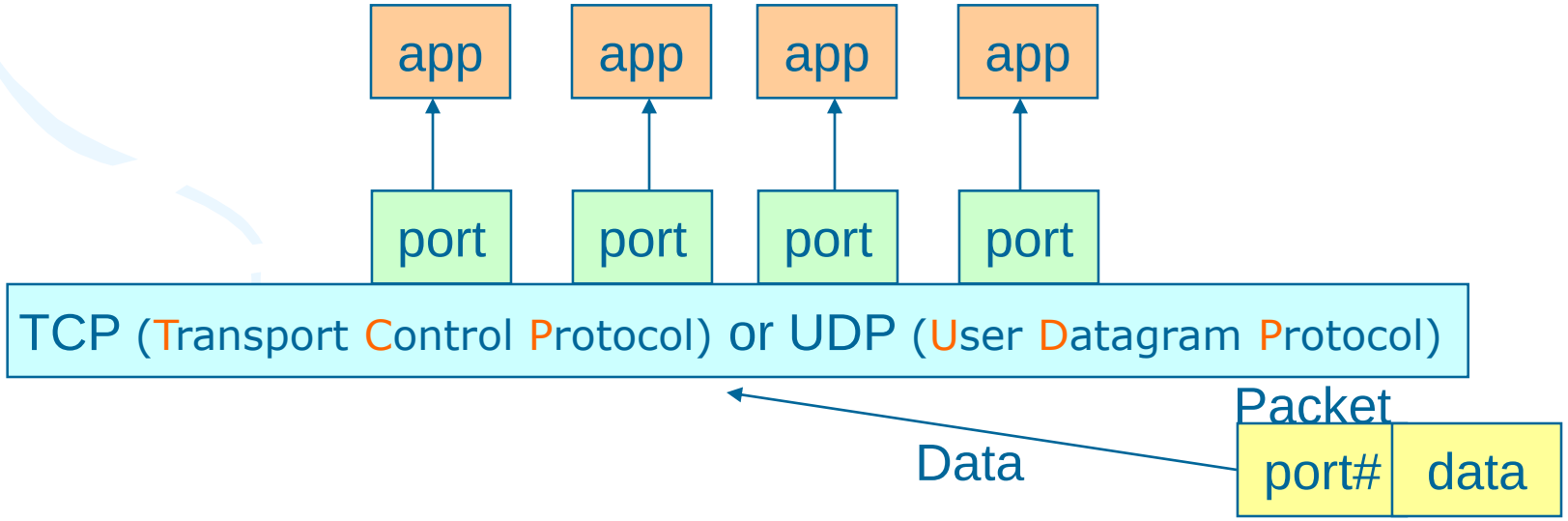
Internet address = 138.37.94.248

Internet address = 138.37.88.249

Understanding Ports

- Sockets use *ports* to map incoming data to a particular *process* running on a computer.





Understanding Ports

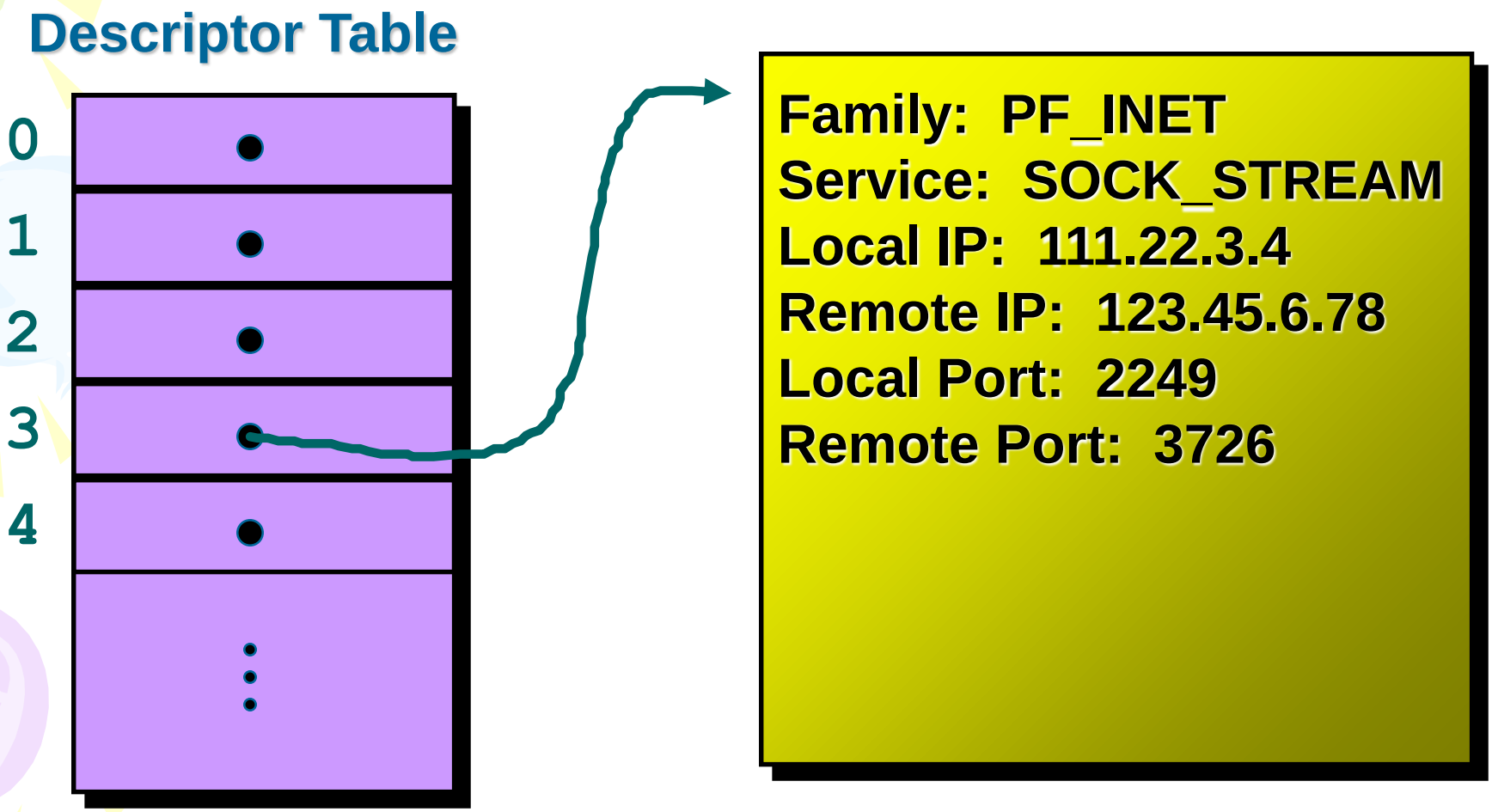
- Port is represented by a positive (16-bit) integer value
- Some ports have been reserved to support common/well known services:
  - ftp 21/tcp
  - telnet 23/tcp
  - smtp 25/tcp
  - login 513/tcp
- User level process/services generally use port number value >= 1024

Standard Web Protocols & Services

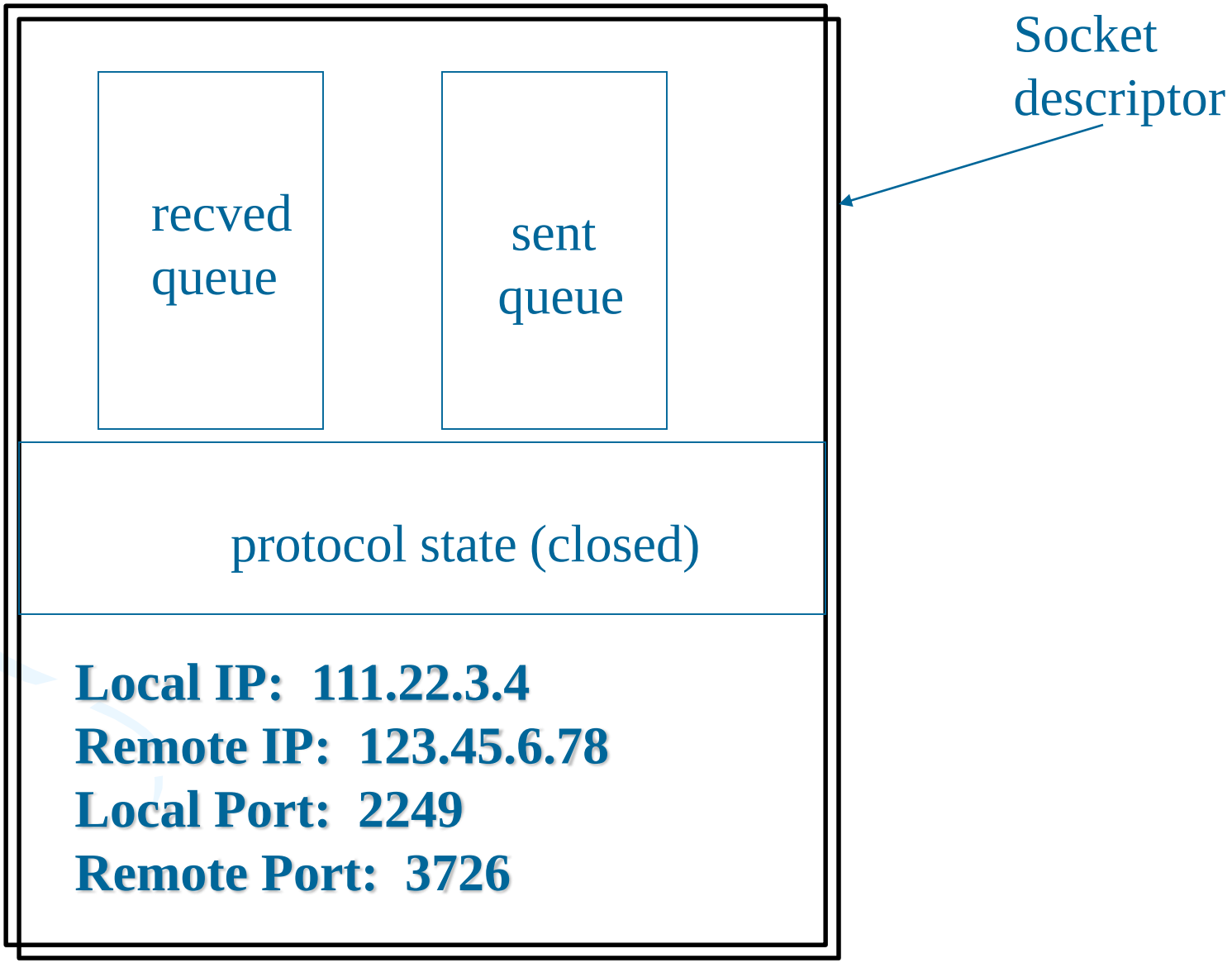
All resources are identified by a unique **Uniform Resource Locator** (URL). The URL has four parts: protocol, host, port and resource.

- **Protocol** – an optional header specifying the resource access protocol.
- **Host** – the IP number or registered name of an Internet host computer or device.
- **Port** – an optional port number that specifies the socket.
- **Resource** – the complete path name of a resource on the host.

Socket Descriptor Data Structure

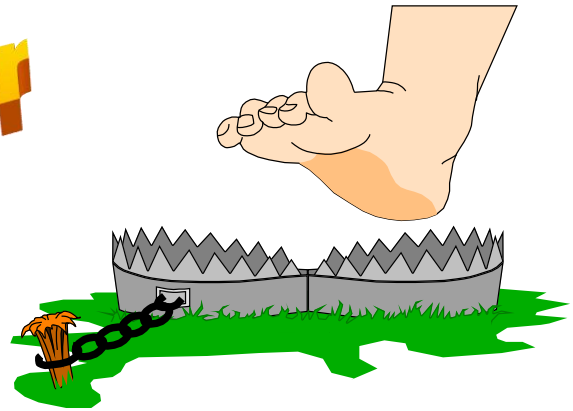


Data structure of a socket



Network Byte Order

- All values stored in a **`sockaddr_in`** must be in network byte order.
  - **`sin_port`** a port number.
  - **`sin_addr`** an IP address.



Byte Ordering

```
union {
    u_int32_t addr;  /* 4 bytes address */
    char c[4];
} un;

un.addr = 0x8002c25f;  /* 128.2.194.95 */
/* c[0] = ? */
```

- Big Endian

c[0]	c[1]	c[2]	c[3]
128	2	194	95

  - Sun Solaris, PowerPC, ...

- Little Endian

95	194	2	128

  - Intel x86, DEC Alpha, ...

- Network byte order = Big Endian

Network Byte Order Functions

'h' : host byte order

'n' : network byte order

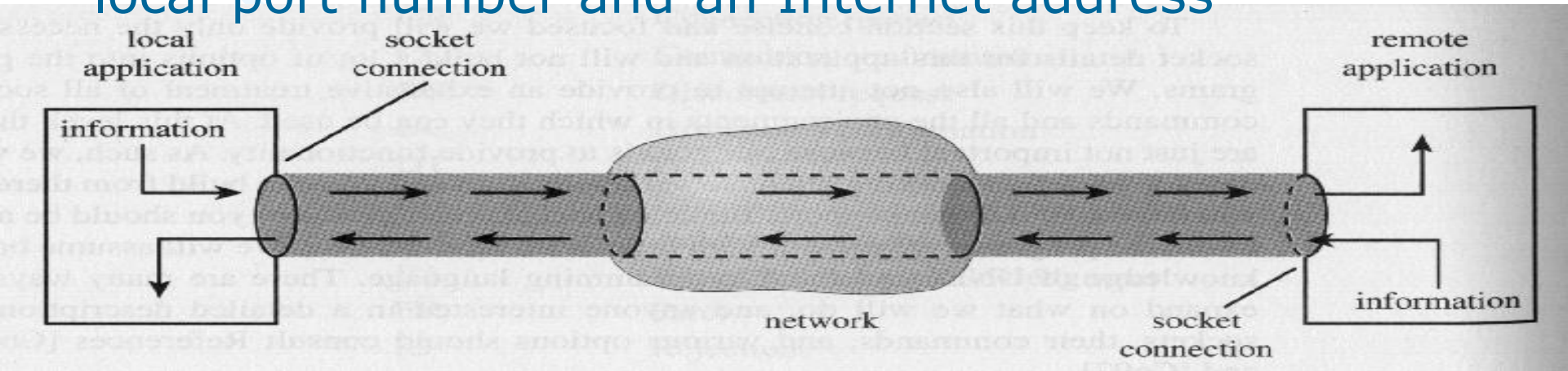
's' : short (16bit)

'l' : long (32bit)

```
uint16_t htons(uint16_t);
uint16_t ntohs(uint_16_t);

uint32_t htonl(uint32_t);
uint32_t ntohl(uint32_t);
```

Socket

- Socket is a UNIX construct and is the basis for UNIX network I/O
  - It is available under MS Windows as well, i.e. WinSock 2
- Developed by UC Berkeley: **BSD Socket** (**B**erkeley **S**oftware **D**istribution)
- Originally used in UCB Unix as an API for TCP (Transport Control Protocol) and UDP (User Datagram Protocol)
- Socket and its related primitives are transport interface to higher layer applications
- Socket is a communication identifier consisting of a local port number and an Internet address



BSD Sockets

- Message passing
- Most general IPC (Inter Process Communication) mechanism
- Basic building block of Internet and WWW
- Sockets have addresses either in
  - the **UNIX domain** (same machine)
  - the **Internet domain**

BSD Sockets

- Three important types of sockets
  - **stream sockets** (TCP Transport Control Protocol) provide reliable delivery of data
    - No data will be lost, replicated or arrive out of sequence
  - **datagram sockets** (UDP User Datagram Protocol) make no such warranties (e.g. ping)
  - **raw sockets:** very low level

Sockets basics for Linux

Client server applications are often using sockets

- The server provides services to client's programs that connect to the server via socket interface
- Associated with services is a number known as
  - Port
    - 0 -1023 are reserved
    - User code must use numbers< 1023
- Ports combined with the IP address of the computer executing the server program form a unique point of contact (address) for this server.
- For example, the address **114.58.1.6:6072** comprises the computers IP number **114.58.1.6** and the port number **6072** separated by a colon.

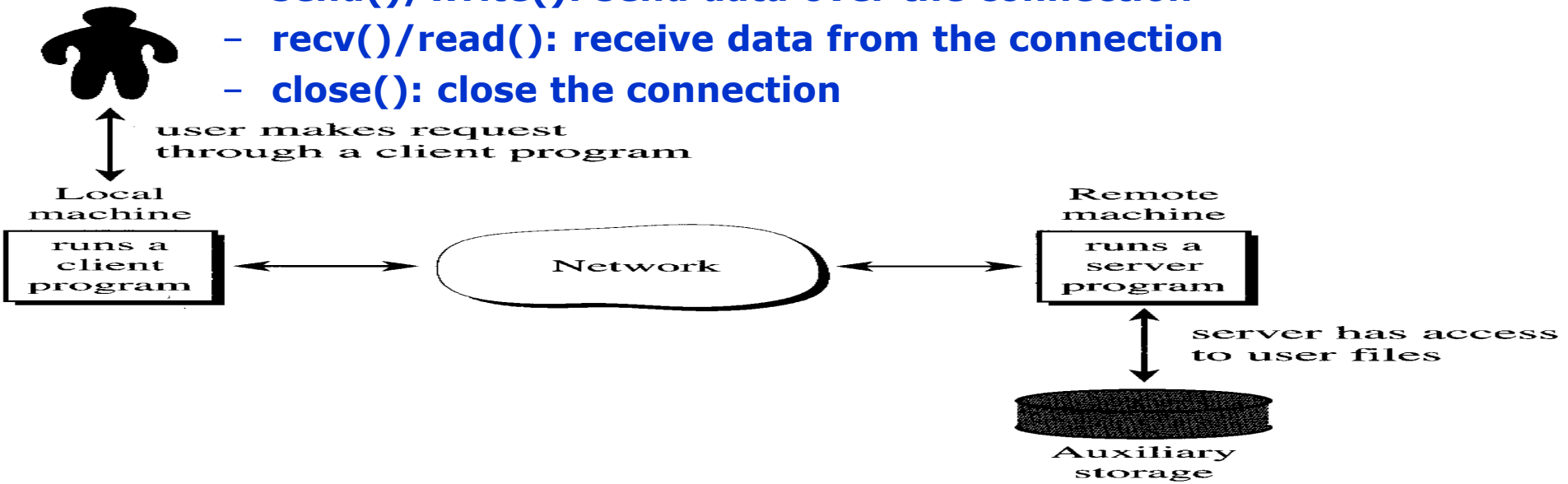
```
# /etc/services is a file on most computers
    used to specify socket addresses

ftp  21/tcp
Ssh 22/tcp SSH Remote Login Protocol
Ssh 22/udp SSH Remote Login Protocol
telnet   23/tcp
smtp     25/tcp     mail
smtp     25/udp     mail
time     37/tcp     time server
time     37/udp     time server
whois++ 63/tcp     nick name
bootps  67/tcp     BOOTP server
bootpc  68/tcp     BOOTP client
gopher  70/tcp     Internet Gopher
gopher  70/udp
finger   79/tcp
http     80/tcp     WorldWideWeb HTTP
```

Socket functions needed

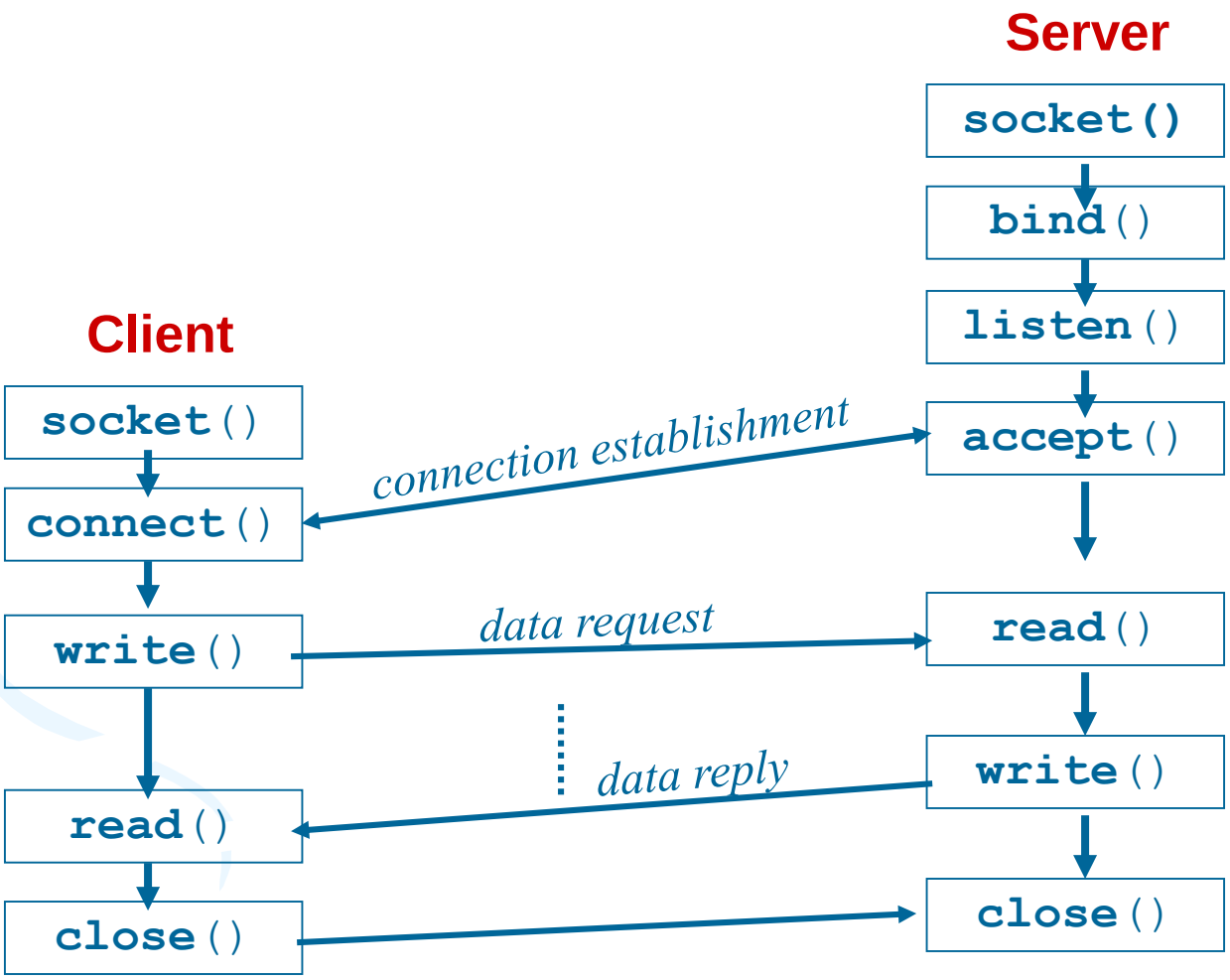
- Specify local and remote communication endpoints
- Initiate a connection
- Wait for incoming connection
- Send and receive data
- Terminate a connection gracefully
- Error handling

Socket

- **Client/server model**
  - **Involves two distinct programs running on different machines at different locations**
  - **They have network connections**
  - **The server provides services and responds to requests coming in from clients**
- **Primitives for socket**
  - **socket(): create a communication end point**
  - **bind(): attach a local address to a socket**
  - **listen(): announce number of connections**
  - **accept(): waiting for a connection call**
  - **connect(): attempt to establish a connection**
  - **send()/write(): send data over the connection**
  - **recv()/read(): receive data from the connection**
  - **close(): close the connection**



Socket Client-Server Interaction



Simple Socket Server

```c
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>

int main() {

    int server_socket = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in server_address, client_address;
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = INADDR_ANY;
    server_address.sin_port = htons(1234);
    bind(server_socket,(struct sockaddr *)&server_address,sizeof(struct sockaddr_in));

    listen(server_socket, 5);

    socklen_t l = sizeof(struct sockaddr_in); /* client address length */
    int client_socket = accept(server_socket, (struct sockaddr *) &client_address, &l);


    char data[512] = { '\0' };
    int message_length = read(client_socket, data, 512);
    printf("from client: %s\n", data);
    message_length = sprintf(data, "back to ya client");
    message_length = write(client_socket, data, message_length);

    close(client_socket);

    return 0;
}
```

Simple Socket Client

```c
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>

int main() {

    int client_socket = socket(AF_INET, SOCK_STREAM, 0);

    struct addrinfo *server_address = NULL;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    getaddrinfo( "localhost", "1234", &hints, &server_address);
    connect(client_socket, server_address->ai_addr, server_address->ai_addrlen);

    char data[512] = {'\0'};
    int message_length = sprintf(data, "hello server");
    message_length = write(client_socket, data, message_length);
    message_length = read(client_socket, data, 512);
    printf("from server: %s\n", data);

    close(client_socket);

    return 0;
}
```

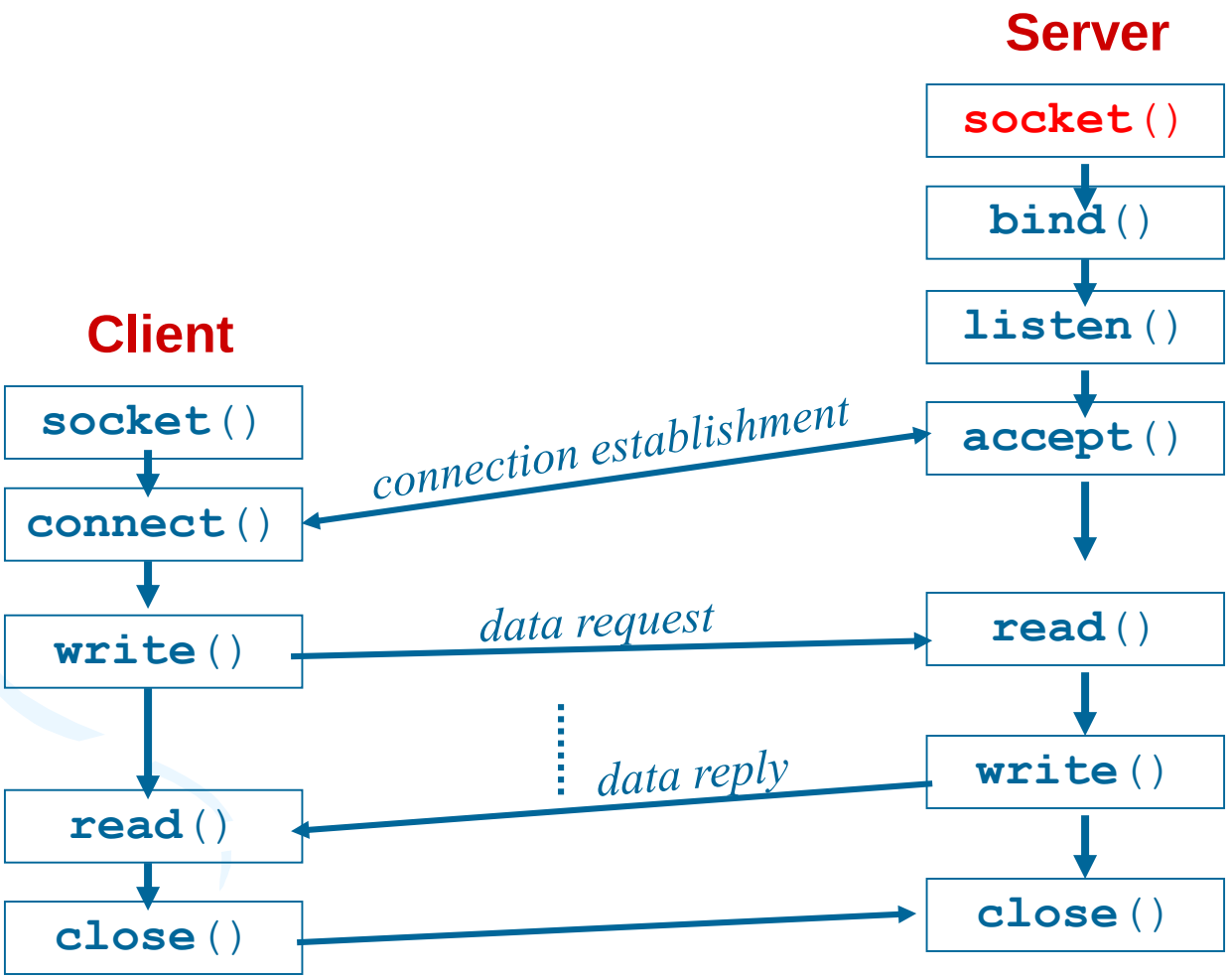
Simple Python Socket Server

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_socket.bind(("", 1234))

server_socket.listen(5)

client_socket, address = server_socket.accept()

data = client_socket.recv(512)
print "from client: " , data
data = "back to ya client"
client_socket.send(data)

client_socket.close()
```

Simple Python Socket Client

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

client_socket.connect(( "localhost", 1234))

data = "hello server"
client_socket.send(data)
data = client_socket.recv(512)
print "from server: " , data

client_socket.close()
```

Socket Client-Server Interaction



The socket() function

- Creates a socket:
  - Not associated with a client or server process yet
- Socket type:
  - SOCK_STREAM vs. SOCK_DGRAM

Prototype:
int **socket**(int *family*, int *type*, int *protocol*)

Example – Internet socket:

```
if (( sock = socket(AF_INET,SOCK_STREAM,0))
== -1)
    { printf(": error opening socket");
      exit(-1);
    }
```

Server code (get socket and initialize struct)

```
main (int argc, char *argv[])

{
   int rc,                    /* system call return code */
       msgsock, sock,         /* server/listen socket descriptors */
       adrlen,                /* sockaddr length */
       cnt;                   /* number of bytes I/O */

   struct sockaddr_in         sockname;  /* Internet socket name */
   struct sockaddr_in         *nptr;    /* ptr to get port number after bind*/
   struct sockaddr            addr;     /* generic socket name */

   char buf[80];              /* I/O buffer for messages - could be larger*/
   /* You need to lookup in /etc/hosts file.   */

   struct hostent *hp, *gethostbyaddr();

/*get socket  API*/

if (( sock = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
     perror("Get server socket");
      exit(1);
      }
   /* Initialize the fields in the Internet socket name structure*/
   sockname.sin_family = AF_INET;        /* Internet address */
   sockname.sin_port = 0;                /* System will assign port #  */
   sockname.sin_addr.s_addr = INADDR_ANY;  /* "Wildcard" */
```

```
/*
struct    hostent {
   char   *h_name;       //official name
   char   **h_aliases;   // alias list
   int    h_addrtype;    //address type
   int    h_length;      // address length
   char   **h_addr_list; // address list
  #define h_addr h_addr_list[0]
  /*backward compatibility*/
};
```

The structures *sockaddr , sockaddr_in*

>> **sockaddr**

Contain address information associated with a socket (generic structure)

>> **sockaddr_in** applies to Internet protocols (*contains IP address and port*)

They have the same length in bytes

Unix domain sockets

```
struct sockaddr_un {
    short sun_family; //AF_UNIX
    char sun_path (108);//path
};
```

```
struct sockaddr {
        u_short sa_family;
        u_short sa_data[14];
}
```

```
struct sockaddr_in {
        short sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];  // unused
}
```

Using sockaddr_in structure

Example I:

```
sock_addr.sin_addr.s_addr = htonl(INADDR_ANY);
sock_addr.sin_port = htons(0);
sock_addr.sin_family = AF_INET;
```

Example II:

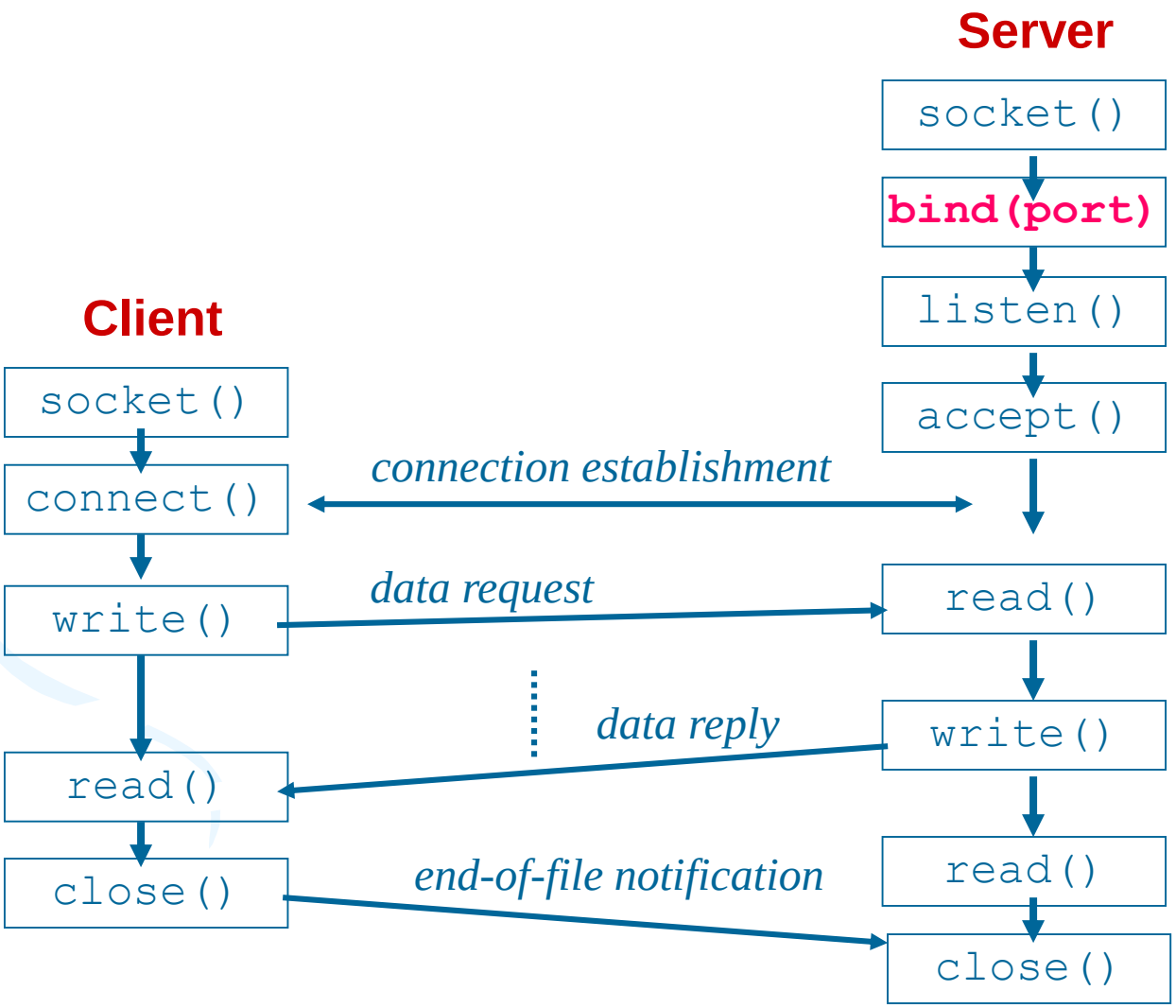
```
sock_addr.sin_addr.s_addr = inet_addr("127.59.69.7");
sock_addr.sin_port = htons(5115);
sock_addr.sin_family = AF_INET
```

Example 3: we need more:

```
struct sockaddr_in sa;
    struct sockaddr_in caller;
    struct sockaddr_in *nptr; /*ptr to get port number*/
    struct sockaddr    addr; /*generic socket address*/
```

- **htonl(), htons()** handle byte ordering differences between computer architectures and network protocols
  - *redundant in Sun's Sparc*
  - *needed in x86 architectures*
  - **INADDR_ANY** is used for accepting connections on any local interface if the system is multi-homed
  - We assign a zero port number when filling the **sockaddr_in** data structure, if we want a free port to be allocated to a socket during the bind() procedure

Socket Client-Server Interaction



from UNIX Network Programming Volume 1, figure 4.1

The bind() function

int bind(int *sockfd*, struct sockaddr **myaddr*, int *addrlen*)

Example:

```
if ( bind(sock, (struct sockaddr *) &sock_addr, sizeof(sock_addr)) < 0)
     { printf(": error binding socket to local address");
       exit(-1);
     }
```

This call binds a socket to an IP address and port.
It is not associated with a client or server process yet

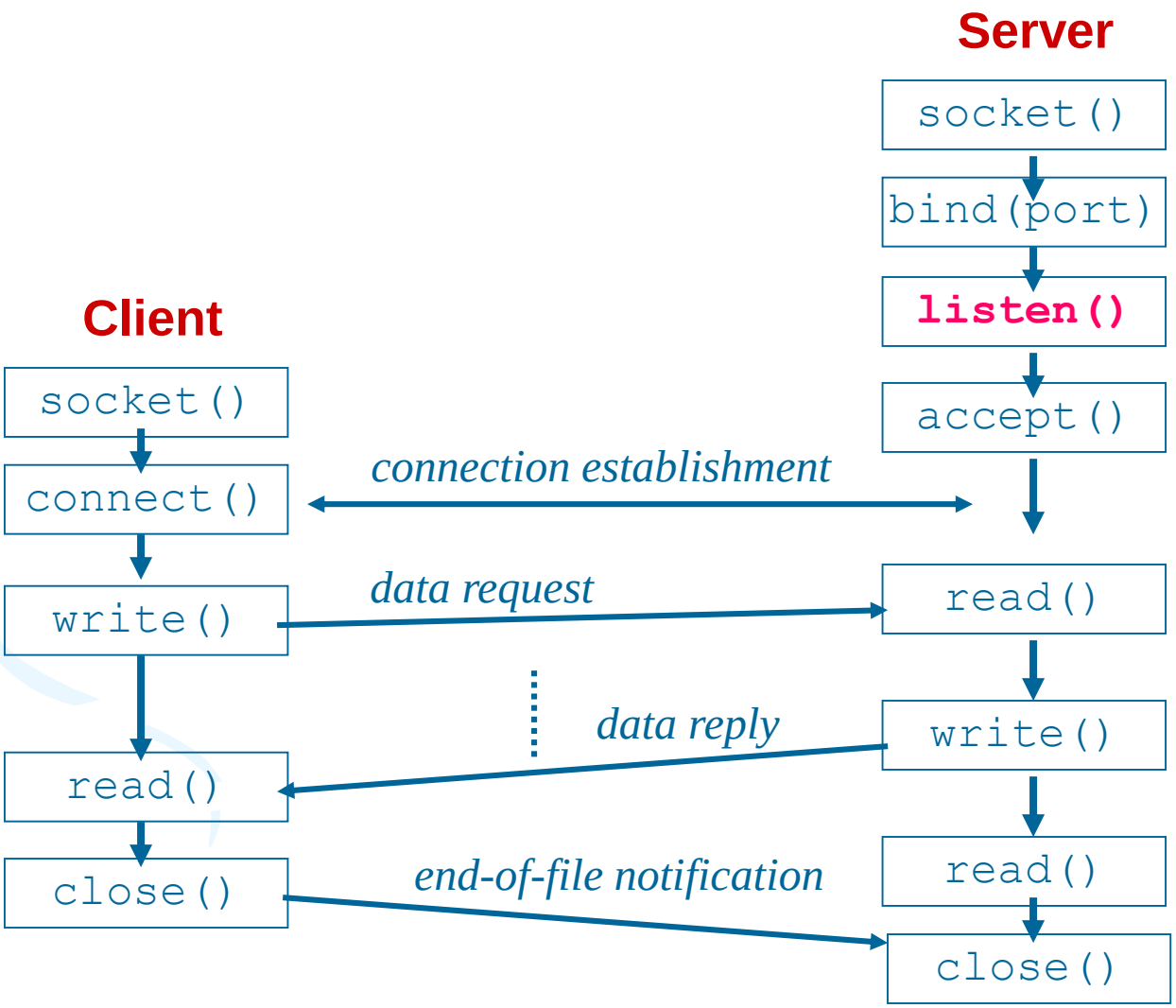
1. **sock** is a **file descriptor** for the socket
2. **name** is **a pointer to a structure** of type sockaddr

Server code – bind() and get port number

```
if (bind(sock, &sockname, sizeof(sockname) ) < 0 ) {
    close(sock);                          /* always close socket  */
    perror("network server bind");
    exit(2);
}

/*Get the port number assigned to the Internet socket and
  print it for use as arg to client*/

adrlen = sizeof(addr);
if ( ( rc = getsockname( sock, &addr, &adrlen ) ) < 0 )
    {
    perror("network server getsockname");
    close (sock);
    exit(3);
}

/*tell the client the socket number.
The pointer nptr points to the generic address structure. */

nptr = (struct sockaddr_in *) &addr;    /* port number  */
                /* now you can print port number is for client to use */
```

socket port is in nptr->sin_port >>>>see next slide

Socket Client-Server Interaction



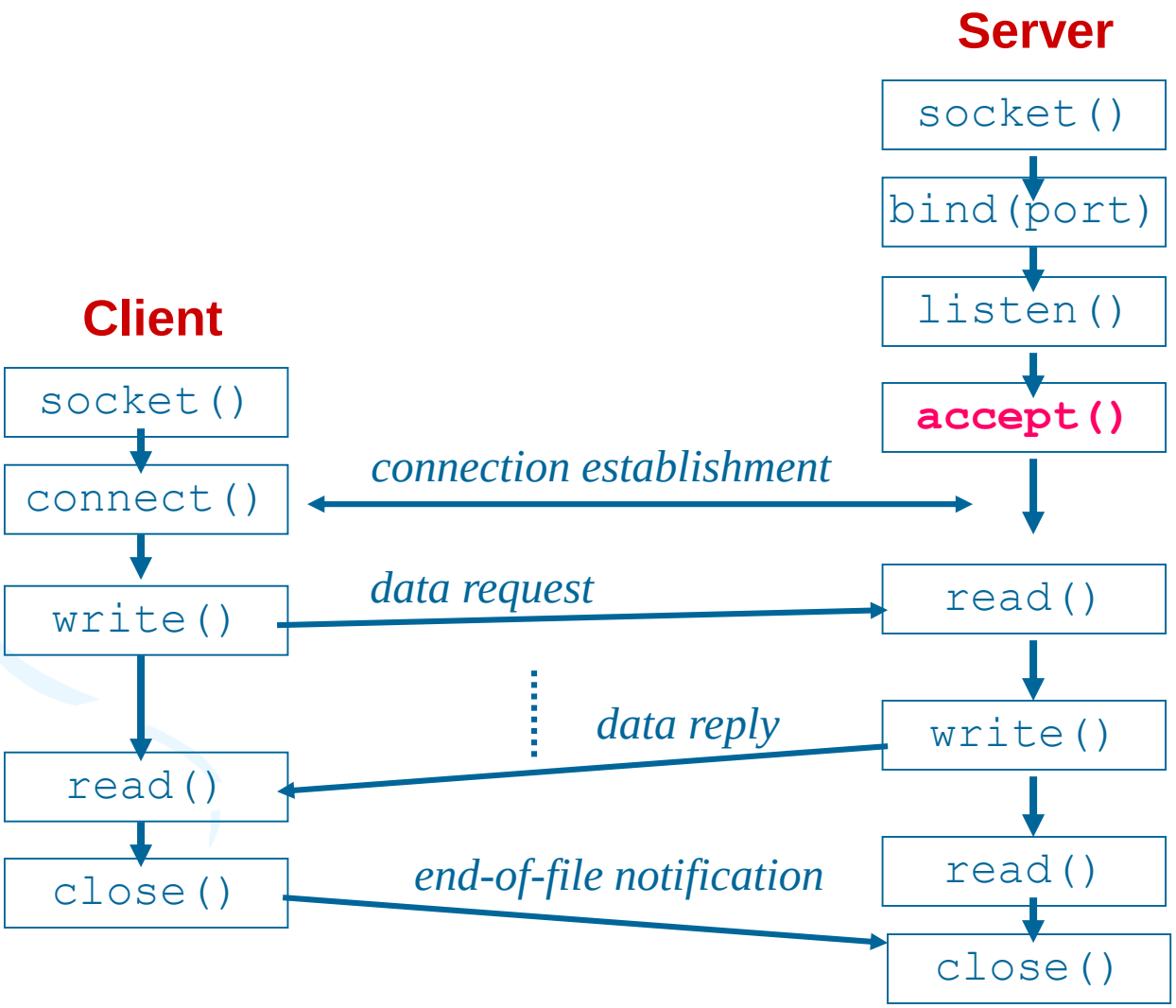
from UNIX Network Programming Volume 1, figure 4.1

Function listen()

```
/*get the port assigned to the Internet socket and print it */

length = sizeof(sa);      /*length of entire structure!!*/
        /*check if function worked */
       if ((rc=getsockname(s, (struct sockaddr_in *)&sa, &length) )< 0){
              close(s);
               ERROR("getsockname");
                }
        nptr = (struct sockaddr_in *) &sa;    /*get port number */
        printf("\n\Server has port number: %d\n", ntohs(nptr->sin_port));

        /* listen for connections on a socket */

        listen (s, 5);
```

/*listen establishes the socket as a passive endpoint of a connection.
It does not suspend the process execution; No messages can be sent through this socket, however, incoming messages can be received
sock is a file descriptor associated with the socket*/

Socket Client-Server Interaction



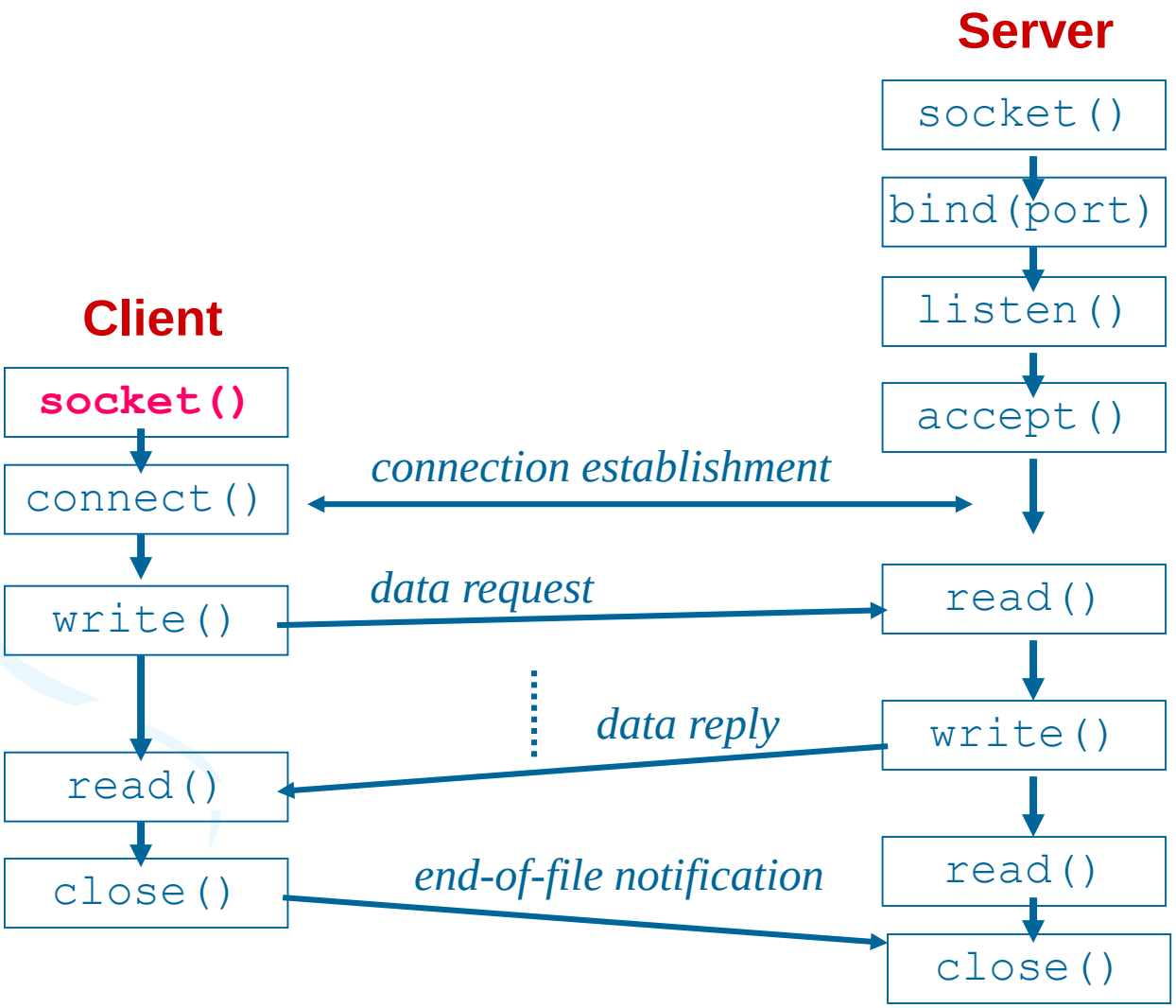
from UNIX Network Programming Volume 1, figure 4.1

Function accept()

```
/* Use accept to wait for calls to the socket.  Accept returns
*a new socket which is connected to the caller.  msgsock will be a
 *temporary (non-reusable) socket different from s
*msgsock is returned if the call is successful          */
```

```
length = sizeof(caller);
if ((msgsock = accept(s,(struct sockaddr *)&caller,&length)) < 0) {
        printf("\nNetwork server accept failure %d\n", errno);
        perror("network server");
        close(s);
        exit(5);
…..}
```

Socket Client-Server Interaction



from UNIX Network Programming Volume 1, figure 4.1

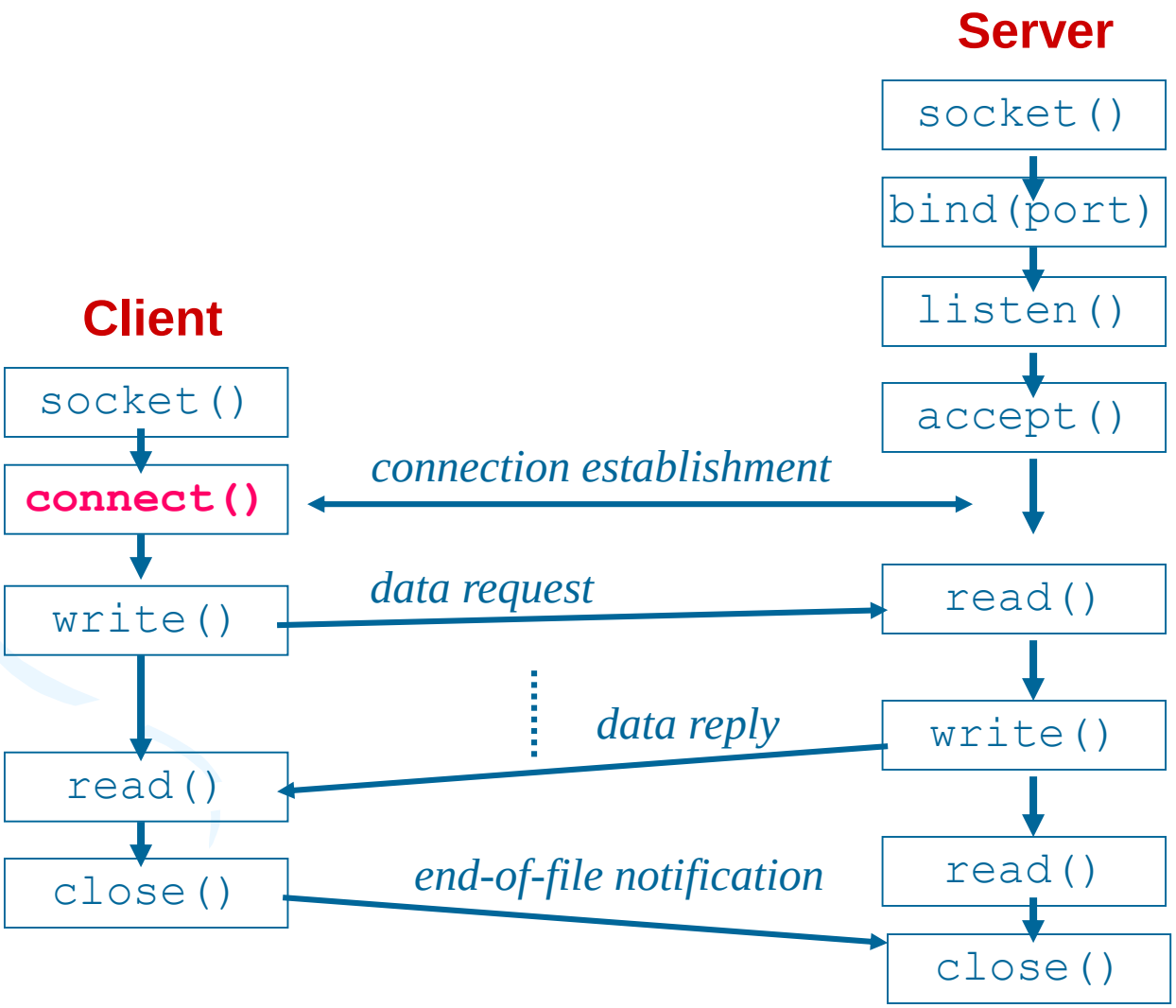
Client programming

```
/* Expected parameters on command line:

   argv[0] -- name of executable (client)
   argv[1] -- the host name to which connection is desired (use localhost)
   argv[2] -- the port number to be used by the client: the
              value is the port number assigned to the
              server by the server's host system.        */

{
   int sock,           /* socket descriptor */
       val,            /* scratch variable */
       cnt;            /* number of bytes I/O */

   struct sockaddr_in client;  /* Internet socket name (addr) */
   struct sockaddr_in *nptr;   /* pointer to get port number */

   char buf[80];   /* I/O buffer, for the message  - could be larger */

/* For lookup in /etc/hosts file.   */
   struct hostent *hp, *gethostbyaddr();
```

Client code to establish the socket

```
{
   int sock,                    /* socket descriptor */
       val,                     /* scratch variable */
       cnt;                     /* number of bytes I/O messages*/

   struct sockaddr_in client;  /* Internet socket name (addr) */
   struct sockaddr_in *nptr;   /* pointer to get port number */

   char buf[80];   /* I/O buffer, kind of small  , could be larger*/

   /* For lookup in /etc/hosts file.   */
   struct hostent *hp, *gethostbyaddr();

if (( sock = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
      perror("network client");
      exit(2);
   }
   /* Convert user-supplied port number to integer.  */

   client.sin_port = htons( atoi(argv[2]) );              /* Server port number */

   client.sin_family = AF_INET;                           /* Internet domain */
```

Socket Client-Server Interaction



from UNIX Network Programming Volume 1, figure 4.1

The function connect() – connect to the server socket

int **connect** (int *sockfd*, struct sockaddr **myaddr*, int *addrlen*)

Example:

```
if (connect(sock, ( struct sockaddr *) &sock_addr, sizeof sock_addr) == -1)
    { printf(": socket connection error");
     exit(-1);
}
```

- used in connection oriented communications

- Connects a client to a server

Client can get the host name

```
hp = gethostbyname (argv[1]);

   /* It is good to know to whom we are connected  to    */
   if ( hp == NULL ) {
      perror("network client");
      close (sock);
      exit(3);
   }
   else {
       printf("\tThe official host name is %s\n", hp -> h_name);
       printf("\tThe first host address is %lx\n",
                  ntohl ( * (int * ) hp -> h_addr_list[0] ) );
       printf("\tAlias names for the host are:\n");

      while ( *hp -> h_aliases )       //get all aliases
         printf( "\t\t%s\n", *hp -> h_aliases++ );
   }

   bcopy ( hp -> h_addr_list[0], &client.sin_addr.s_addr,
           hp -> h_length );   /* save the first name */
```

Connection between server and client

```
/* Client: Establish socket connection with (remote) server.  */

   if ( ( connect ( sock, &client, sizeof(client) ) ) < 0 ) {
         printf("network client %s connect failed %d\n",argv[0], errno);
         perror("network client on connect");
         close (sock);
         exit(4);
      }
```

listen()

```
/*SERVER:  Set up "infinite loop" to listen for clients.  Since the
      structure "sockname" is bound to the listen socket, the
      socket structure name and socket length parameter
      values are omitted from the accept call.  The bound values
      are used. */

   while (1) {
      if ( ( msgsock = accept (sock, 0, 0 ) ) < 0 ) {
         printf("Network server accept failed %d\n", errno);
         perror("network server accept");
         close (sock);
         exit(5);
    else { /*processing of the call comes here) /*}
```

The read() and write() functions

int **read**(int *sockfd*, char **buf*, unsign int *nbytes*)
int **write**(int *sockfd*, char **buf*, unsign int *nbytes*)

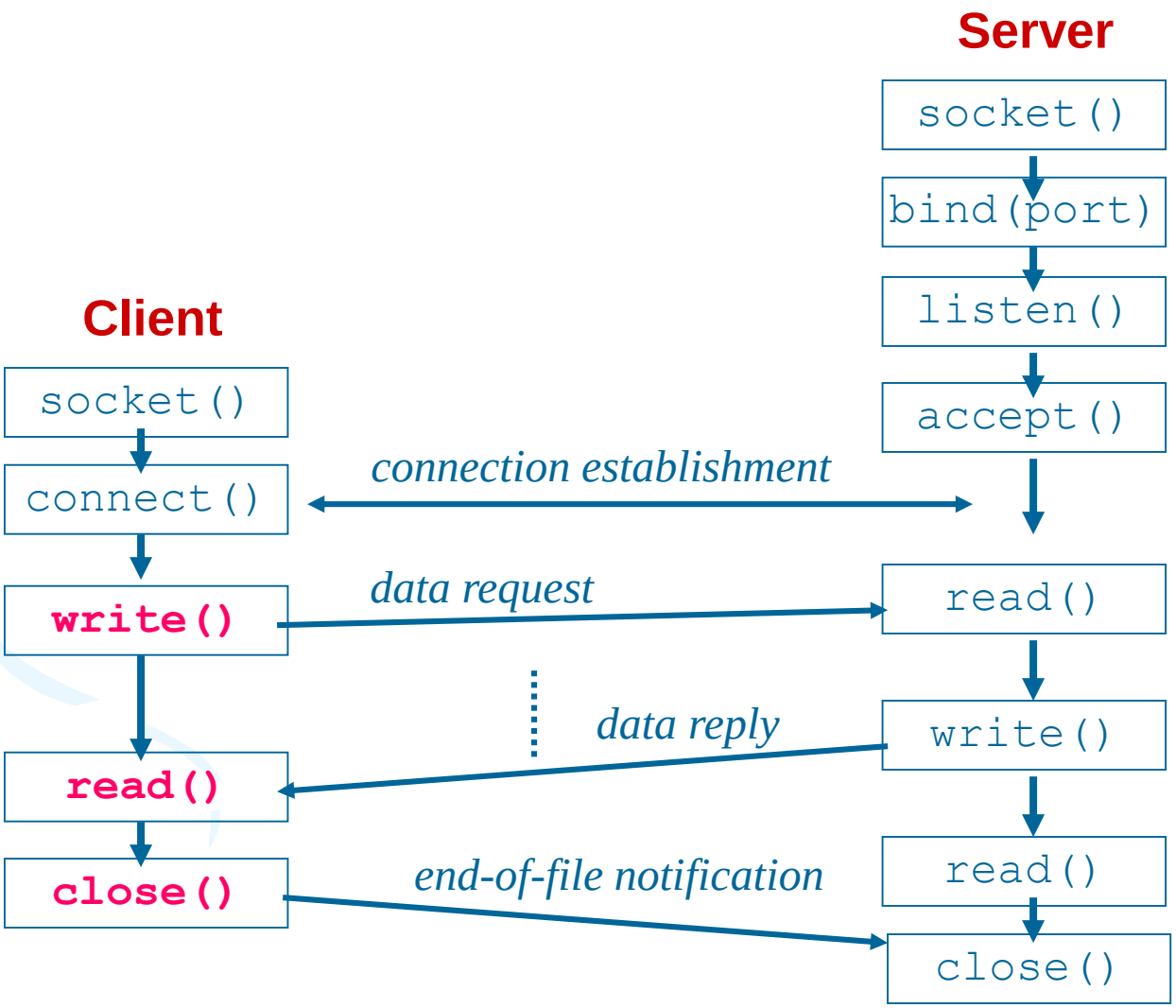
Examples:

```
if (write(sock, message, size) <0 )
 { printf(": error writing to remote host");
    exit(-1);
 }

bytes_received = read(sock, reply, sizeof(reply));
```

- used in connection oriented communications

- used for requesting and transmitting data

Socket Client-Server Interaction

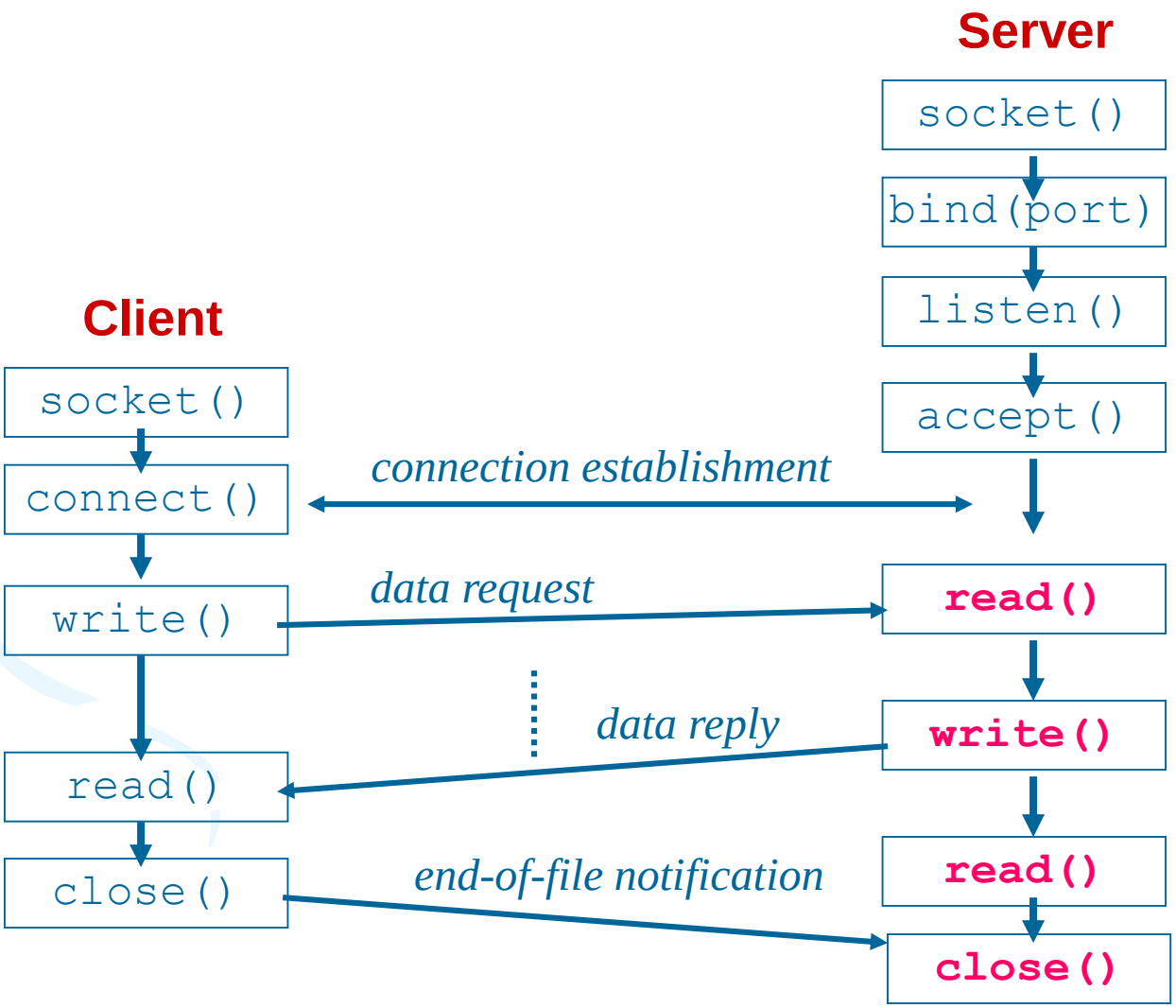


from UNIX Network Programming Volume 1, figure 4.1

Client established the communication

```
if ( ( connect ( sock, &client, sizeof (client) ) ) < 0 ) {
        perror ("network client on connect");
        close (sock);
        exit(4);
    }
    /*  Exchange data with server.  Clear buffer bytes first. */

    bzero  ( buf,   sizeof( buf) );  /* zero buffer, BSD. */
    strcpy ( buf, "Hello server, please respond to client!" );
    write  ( sock, buf, sizeof(buf) );

    /* Now read message sent back by server.    */

    if ( ( cnt = read (sock, buf, sizeof(buf) ) ) < 0 ) {
       perror("network client on read");
       close(sock);
       exit(5);
    }
    else
        printf("I am client and I have received this message %s\n", buf);
        /* Send a message with 0 bytes to end the conversation. */
       bzero  ( buf, sizeof( buf) );  /* zero buffer, BSD. */
       write  ( sock, buf, 0 );
       close (sock);
       exit(0);
}
```

Socket Client-Server Interaction



from UNIX Network Programming Volume 1, figure 4.1

Server communication

```
/* Server communication includes:
   Set up "infinite loop" to listen for clients (see slide 21).
   The bound values are used. */

   while (1) {
      if ( ( msgsock = accept (sock, 0, 0 ) ) < 0 ) {
         perror("network server accept");
         close (sock);
         exit(5);
      }

      /* Fork child process to handle client service request */

   if ( ( fork() ) == 0 ) {    /* Child process to handle communication*/
         int pid;
         pid = getpid();      /* PID of child process */
         close (sock);                  /* Do not need listen socket in child. */
         /* Find out who the client is. */
                  */
    if ((rc = getpeername( msgsock, &addr, &adrlen )) < 0) {
            perror("network server getpeername");
            close(msgsock);
            exit(6);
         }
   printf("\n\tnetwork server %d:", pid);
   printf(" client socket from host %s\t has port number %d\n",inet_ntoa ( nptr -> sin_addr ),
                              nptr -> sin_port );
```

```
do {
      bzero(buf, sizeof(buf));    /* zero buf, BSD call. */
          if ((cnt = read (msgsock, buf, sizeof(buf))) < 0 ) {
             perror("network server read");
             close(msgsock);
             exit(7);
          }
          else
          if (cnt == 0) {   //close on 0 bytes received
             printf("\nServer closing client connection\n");
             close (msgsock);
             continue; /* break out of loop */
          }
          else {  /*Print the message received from client.
                        Send a message back response*/
             printf("Server %d received your message\n",pid);
             printf("The message from client is %s\n", buf);
             bzero (buf, sizeof(buf));                  /* zero buf, BSD. */
             strcpy(buf, "\nI am server and this is my message\n");
             write (msgsock, buf, sizeof(buf));
          }  /* end of message-print else */

       }  /* end of do loop statement */

       while (cnt != 0);  /* end of do loop*/
       exit(0);  /* Exit child process */

    } /* End of child */

    else      /*  Parent process */
       close(msgsock); /* Parent doesn't need socket. */

  }  /* end of while */
```

Exchange data between client and server – server's child processing

Assumption:
close the connection if you receive 0 bytes from client

The close() function

- When finished using a socket, the socket should be closed:
- status = close(s);
  - status: 0 if successful, -1 if error
  - s: the file descriptor (socket being closed)
- Closing a socket
  - closes a connection (for SOCK_STREAM)
  - frees up the port used by the socket

Socket Server: Python code with equivalent C code

```python
import socket
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
int main() {
```

```python
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```c
int server_socket = socket(AF_INET, SOCK_STREAM, 0);
```

```python
server_socket.bind(("", 1234))
```

```c
struct sockaddr_in server_address, client_address;
server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = INADDR_ANY;
server_address.sin_port = htons(1234);
bind(server_socket,(struct sockaddr *)&server_address,sizeof(struct sockaddr_in));
```

```python
server_socket.listen(5)
```

```c
listen(server_socket, 5);
```

```python
client_socket, address = server_socket.accept()
```

```c
socklen_t l = sizeof(struct sockaddr_in); /* client address length */
int client_socket = accept(server_socket, (struct sockaddr *) &client_address, &l);
```

```python
data = client_socket.recv(512)
print "from client: " , data
data = "back to ya client"
client_socket.send(data)
```

```c
char data[512] = { '\0' };
int message_length = read(client_socket, data, 512);
printf("from client: %s\n", data);
message_length = sprintf(data, "back to ya client");
message_length = write(client_socket, data, message_length);
```

```python
client_socket.close()
```

```c
close(client_socket);
return 0;
}
```

Socket Client: Python code with equivalent C code

```
import socket
```

```
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
int main() {
```

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
int client_socket = socket(AF_INET, SOCK_STREAM, 0);
```

```
client_socket.connect(( "localhost", 1234))
```

```
struct addrinfo *server_address = NULL;
struct addrinfo hints;
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
getaddrinfo( "localhost", "1234", &hints, &server_address);
connect(client_socket, server_address->ai_addr, server_address->ai_addrlen);
```

```
data = "hello server"
client_socket.send(data)
data = client_socket.recv(512)
print "from server: " , data
```

```
char data[512] = {'\0'};
int message_length = sprintf(data, "hello server");
message_length = write(client_socket, data, message_length);
message_length = read(client_socket, data, 512);
printf("from server: %s\n", data);
```

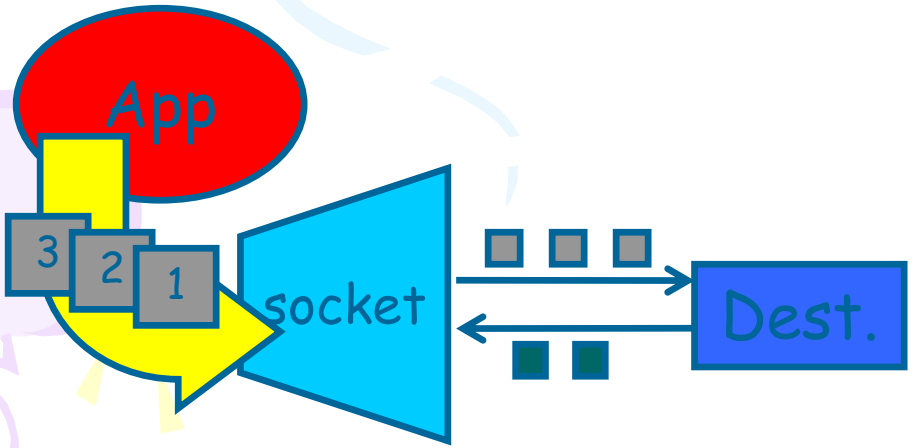
```
client_socket.close()
```

```
close(client_socket);
return 0;
}
```

Two essential types of sockets

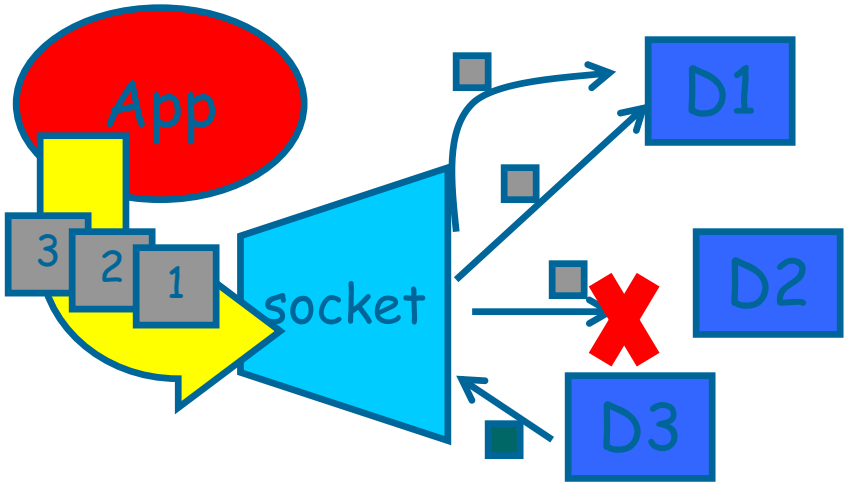
stream socket

- a.k.a. TCP
  (Transport Control Protocol)
- reliable delivery
- in-order guaranteed
- connection-oriented
- bidirectional

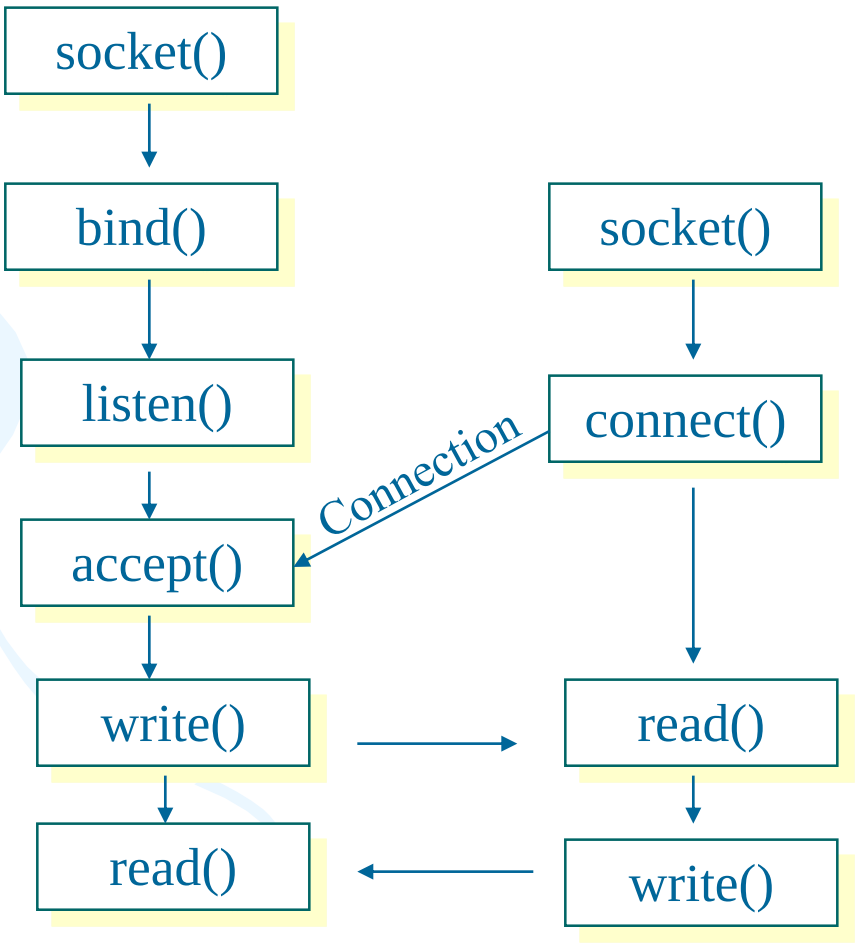


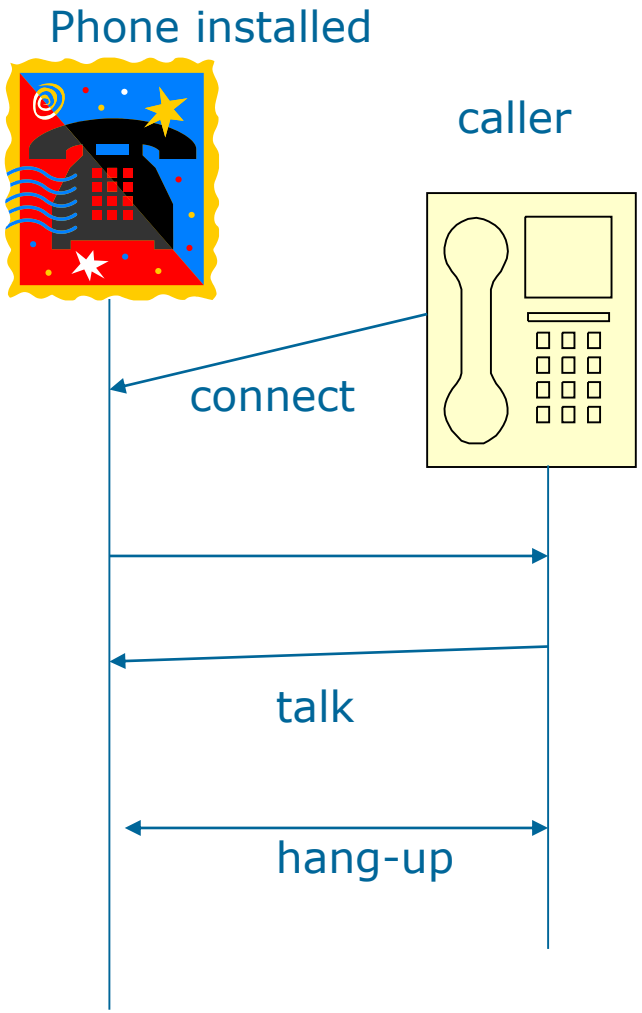
datagram socket

- a.k.a. UDP (e.g. ping)
  (User Datagram Protocol)
- unreliable delivery
- no order guarantees
- no notion of "connection" where app indicates a destination for each packet
- can send or receive

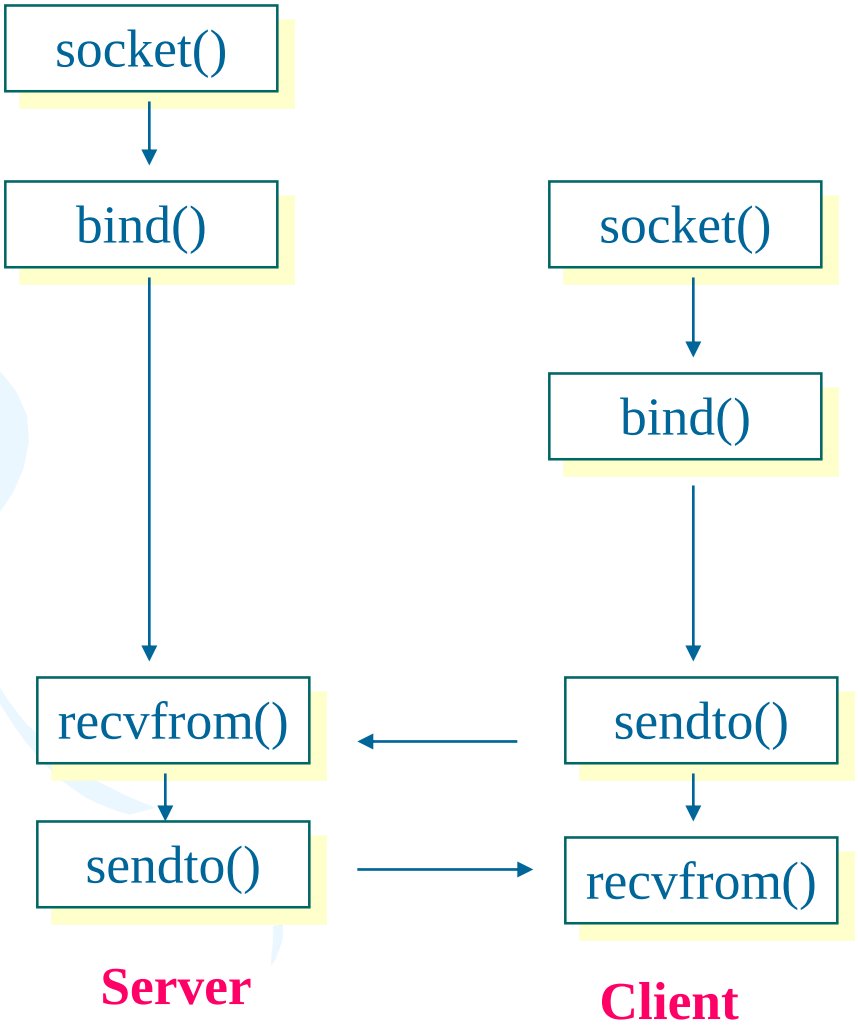


Connection oriented protocol = stream socket





Connectionless Protocol = datagram socket



Both sockets must use bind() call

Sockets

- **Comments:**

Streams vs Datagrams: **If a sender makes 5 separate calls to write, each for 512 bytes and the receiver asks for 2560 bytes then in the case of a**

- **stream:** **the receiver gets all the requested bytes in the correct order**
- **datagram:** **only 512 bytes is returned - four other calls will be needed to retrieve all 2560 bytes**

Datagrams **are less frequently used than streams. Datagrams are used primarily for services that send small amounts of data on short time connections, or send only a little amount of data at a time.**

Summary
Sockets are:

- Communication abstractions
- Support the TCP/IP protocol stack
- Provide access to both
  - reliable (TCP Transport Control Protocol) and
  - unreliable (UDP User Datagram Protocol)

  transport services
- Programming tools to implement client-server applications

Sample Exam Question

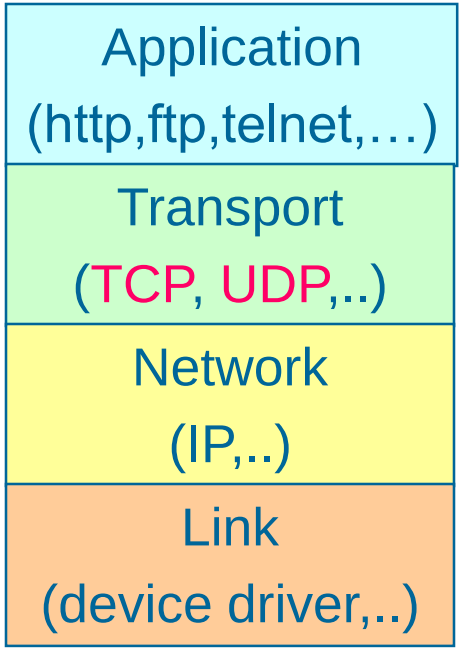
List eight differences between stream and datagram sockets.

Stream Socket	**vs**	**Datagram Socket**
uses TCP		uses UDP
bidirectional		one way only – send or receive
(synchronous)		(asynchronous)
stream data (large)		single packet (small)
connection orientated		not connection orientated
(persistent connection)		(single-use connection)
(need to close connection)		(no connection to close)
in order guaranteed		no order of data guaranteed
(so message boundaries not ..)		(message boundaries preserved)
reliable delivery		unreliable delivery
(acknowledge receipt)		(do not acknowledge receipt)
more complex		simpler – less overhead
application: HTTP, FTP		application: ping

Networking Basics

- Applications Layer
  - Standard apps
    - HTTP
    - FTP
    - Telnet
  - User apps
- Transport Layer
  - TCP (Transport Control Protocol)
  - UDP (User Datagram Protocol)
  - Programming Interface:
    - Sockets
- Network Layer
  - IP
- Link Layer
  - Device drivers

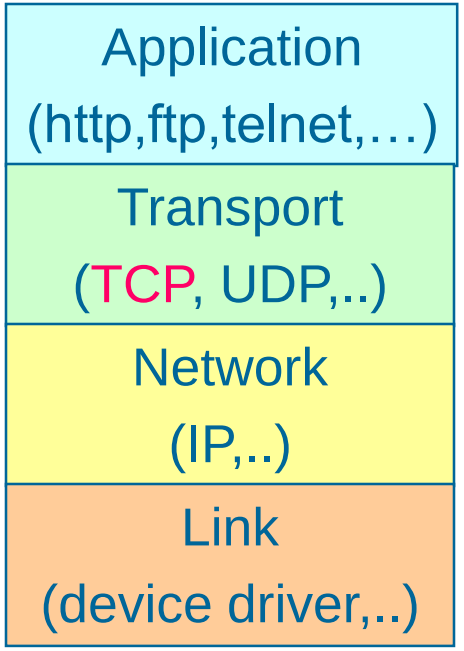
TCP/IP Stack:



Networking Basics

- TCP (Transport Control Protocol) is a connection-oriented protocol that provides a reliable flow of data between two computers.
- Example applications:
  - HTTP
  - FTP
  - Telnet

TCP/IP Stack:



Networking Basics

- UDP (User Datagram Protocol) is a protocol that sends independent packets of data, called *datagrams*, from one computer to another with no guarantees about arrival.
- Example applications:
  - Clock server
  - Ping

TCP/IP Stack: