Client/Server

Networking Approach

andrei.doncescu@laas.fr

CLIENT/SERVER COMPUTING

(THE WAVE OF THE FUTURE)

OBJECTIVES

Goal:

- How application programs use protocol software to communicate across networks and internets

Introduction to Client-Server paradigm of interaction

Agenda

□ Why client server ?

□ Models

□ Architecture

□ Tools

□ Applications

□ Conclusions

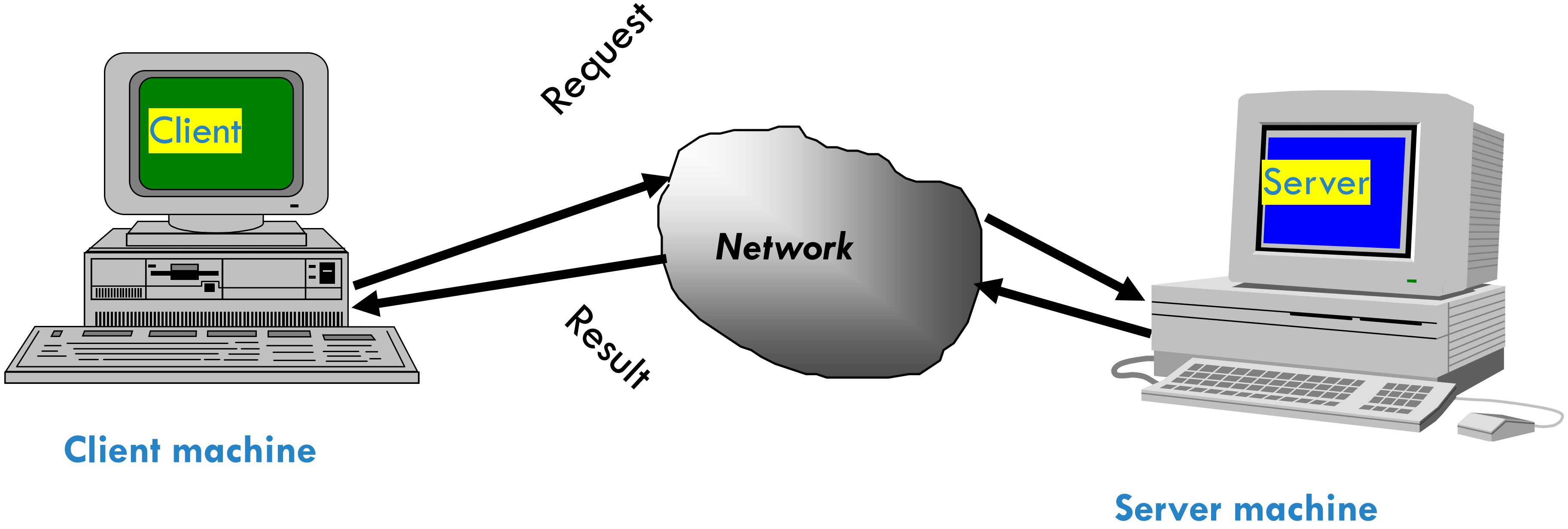
A simple definition

A simple definition of CS is

“ server software accepts requests for data from client software and returns the results to the client”

Elements of C-S Computing

a client, a server, and network



Network

- transfers bits
- operates at application’s request

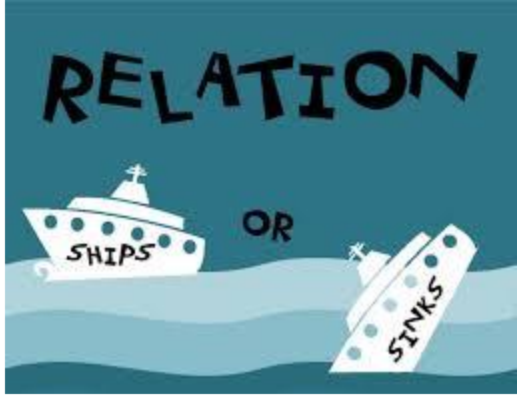
Application determines

- what/when/where to send

- Meaning of bits
  => Application programs are the entities that communicate with each other, not the computers or users.

Important point: For 2 application programs to communicate with each other, one application initiates communication and the other accepts.

WHERE OPERATIONS ARE DONE

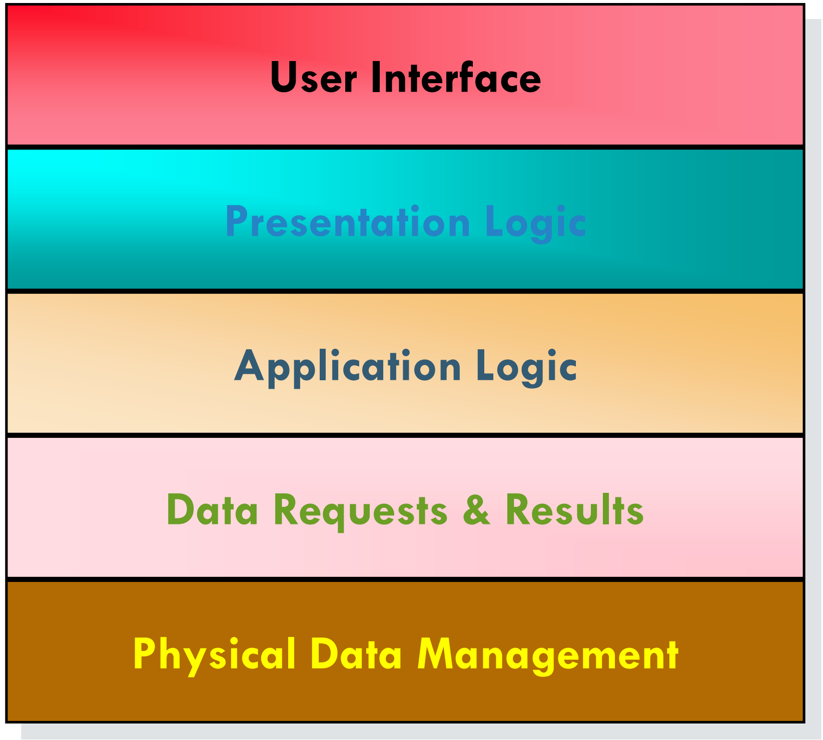


In CS Relationship “most of the application processing is done on a computer (client side), which obtains application services (such as database services) from another computer (server side) in a master slave configuration

CS-Focus is on

In client-server computing major focus is on SOFTWARE

Application Tasks



User Interface

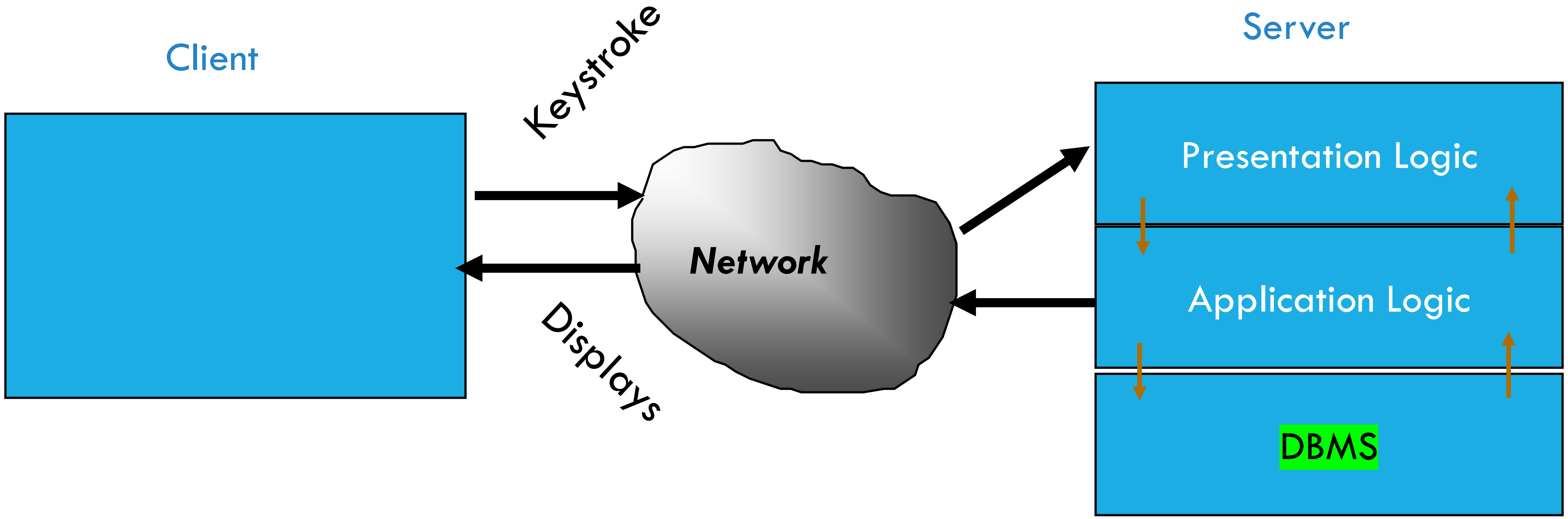
Presentation Logic

Application Logic

Data Requests & Results

Physical Data Management

Client (dumb) - Server Model



CHARACTERISTICS OF A CLIENT

Arbitrary application program

Becomes client temporarily

Can also perform other computations

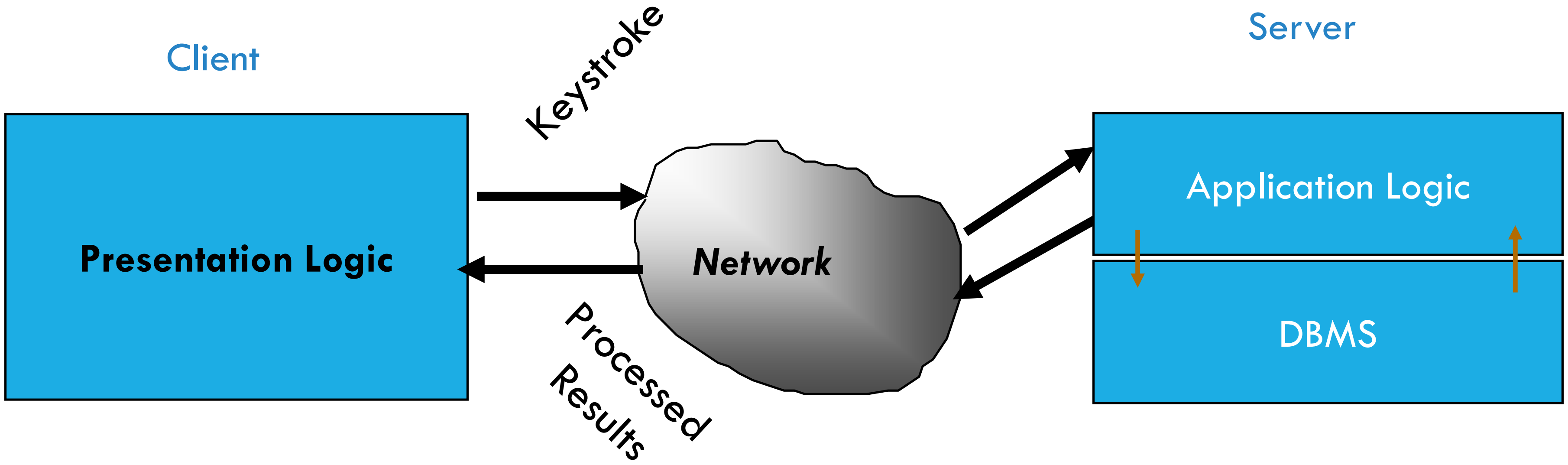
Invoked directly by user

Runs locally on user’s computer

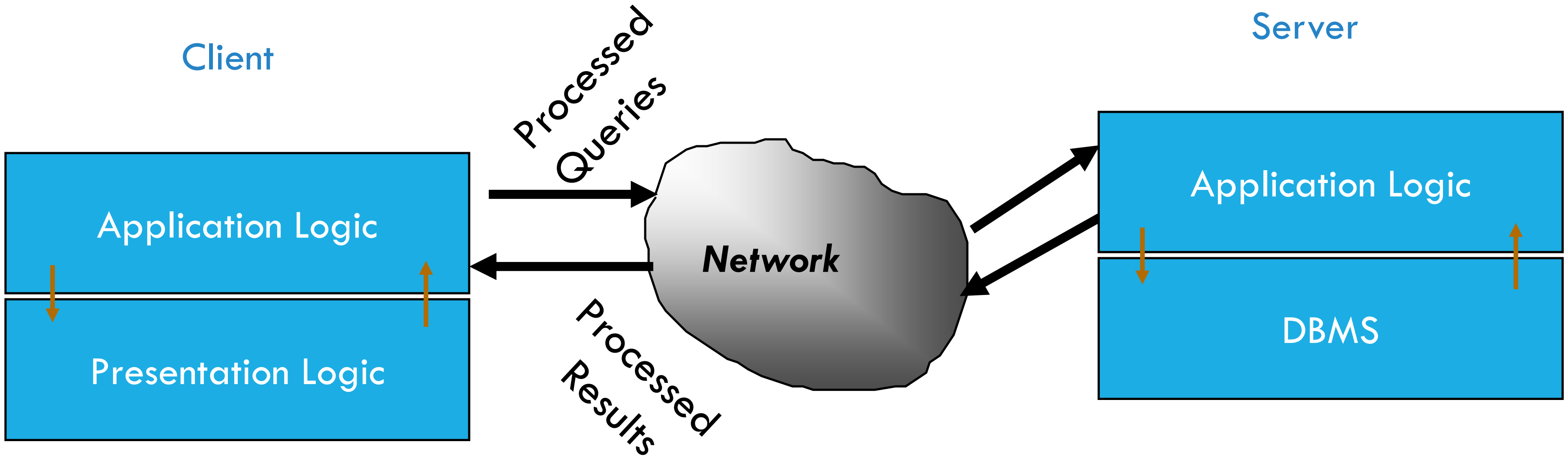
Actively initiates contact with a server

Contacts one server at a time

True Client-Server Model



Distributed Client-Server Model



CHARACTERISTICS OF A SERVER

Special-purpose, privileged program

Dedicated to providing one service

Can handle multiple remote clients simultaneously

Invoked automatically when system boots

Executes forever

Needs powerful computer and operating system

Waits passively for client contact

Accepts requests from arbitrary clients

TERMINOLOGY

Server

- An executing program that accepts contact over the network

server-class computer

- Hardware sufficient to execute a server

Informally

- Term “server” often applied to computer

DIRECTION OF DATA FLOW

Data can flow

- from client to server only
- from server to client only
- in both directions

Application protocol determines flow

Typical scenario

- Client sends request(s)
- Server sends responses(s)

SERVER CPU USE

Facts

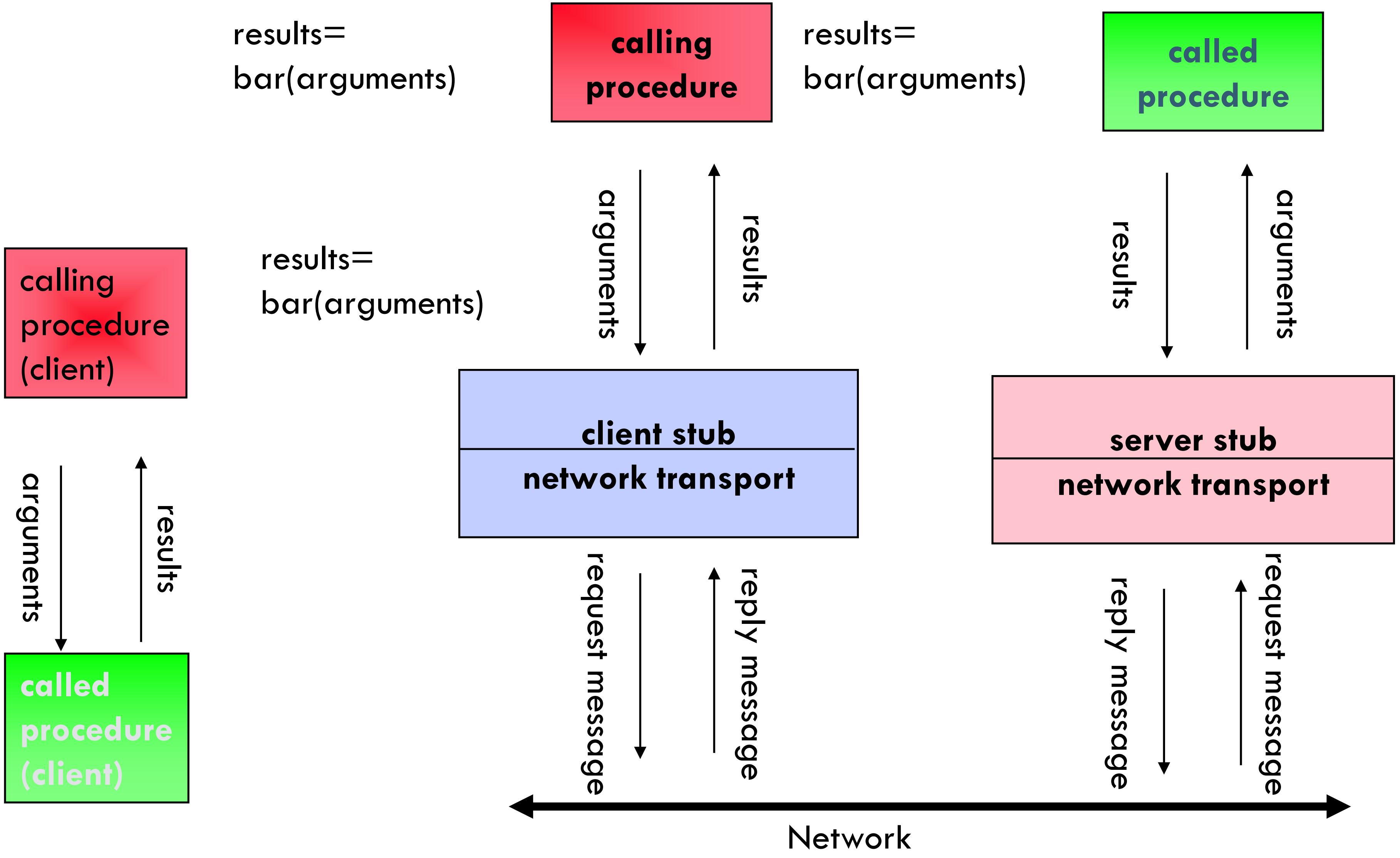
- Server operates like other applications
  - uses CPU to execute instructions
  - Performs I/O operations
- Waiting for data to arrive over a network does not require CPU time

Consequence

- Server program uses only CPU when servicing a request

CLIENT-SERVER COMPUTING IS DISTRIBUTED ACCESS, NOT A DISTRIBUTED COMPUTING.

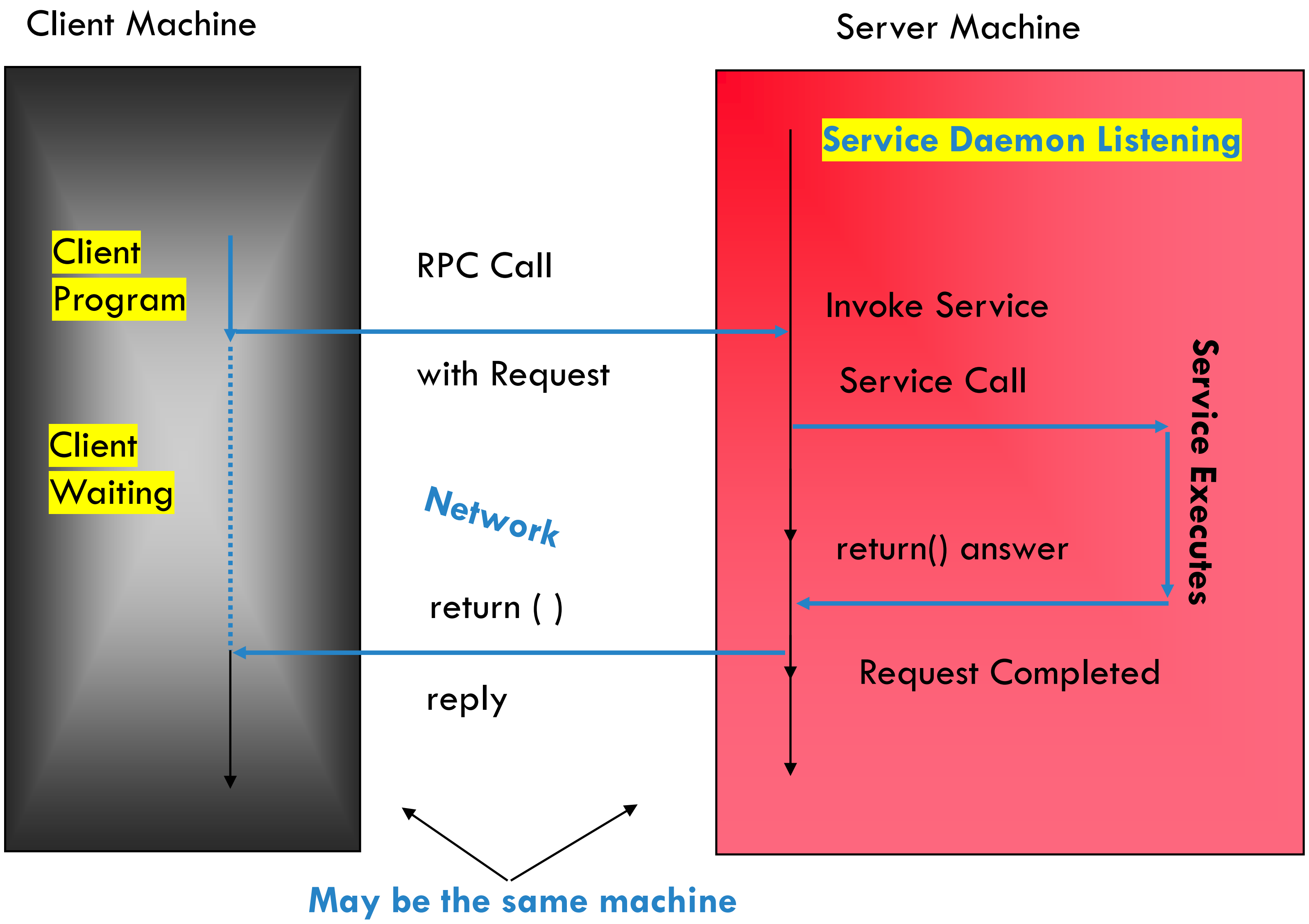
RPC Look and Feel like Local Calls



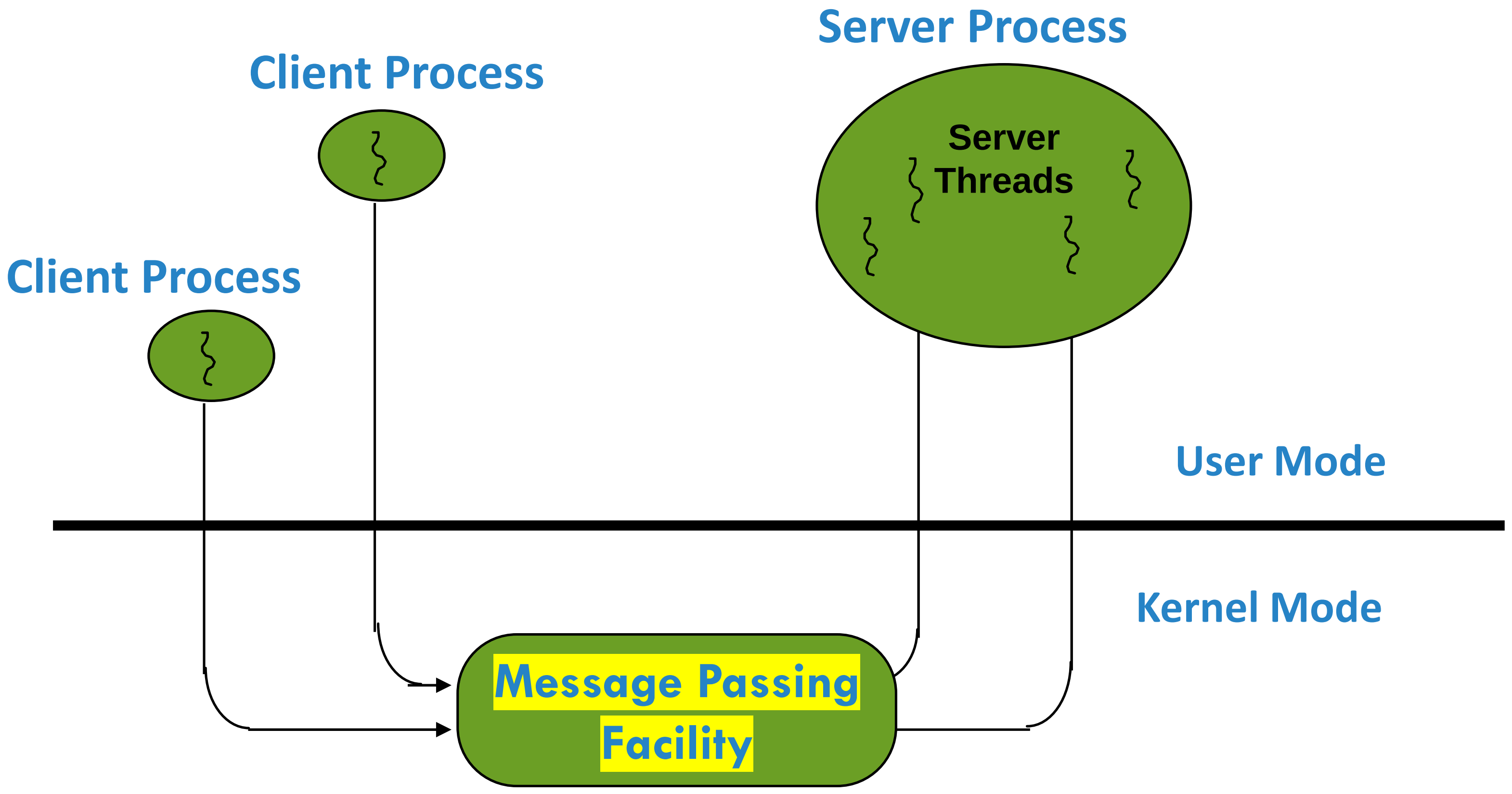
Local Procedure Call

Remote Procedure Call

Flow Control in a Sychronous RPC



Multithreaded Server



CATEGORIES OF SERVERS

File Server

Data Server

Compute Server

Database Server

Communication Server

Video Server

FILE SERVER

File Servers manage a work group's application and data files, so that they may be shared by the group.

Very I/O oriented

Pull large amount of data off the storage subsystem and pass the data over the network

Requires many slots for network connections and a large-capacity, fast hard disk subsystem.

COMPUTE SERVER

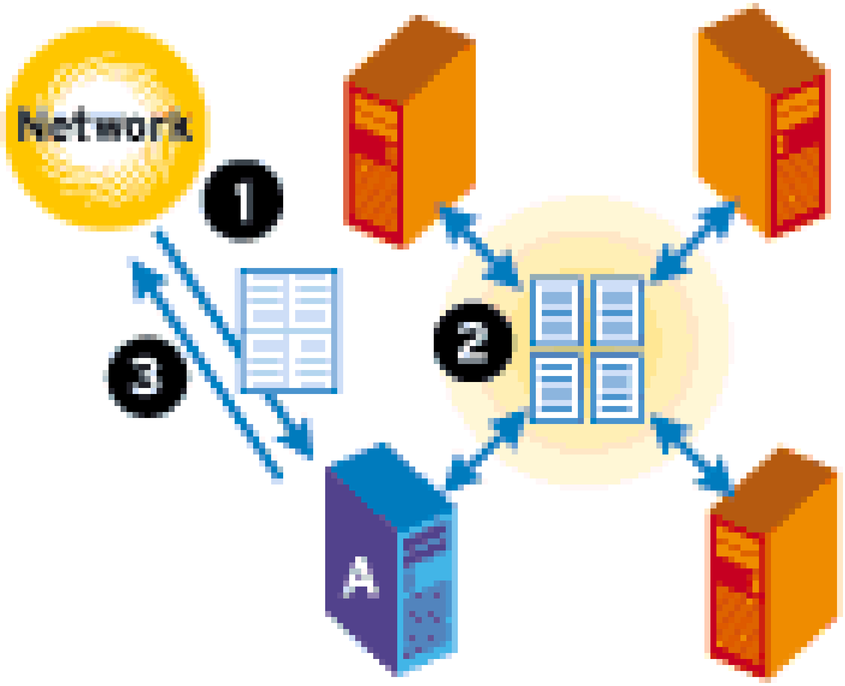
Performs Application logic processing

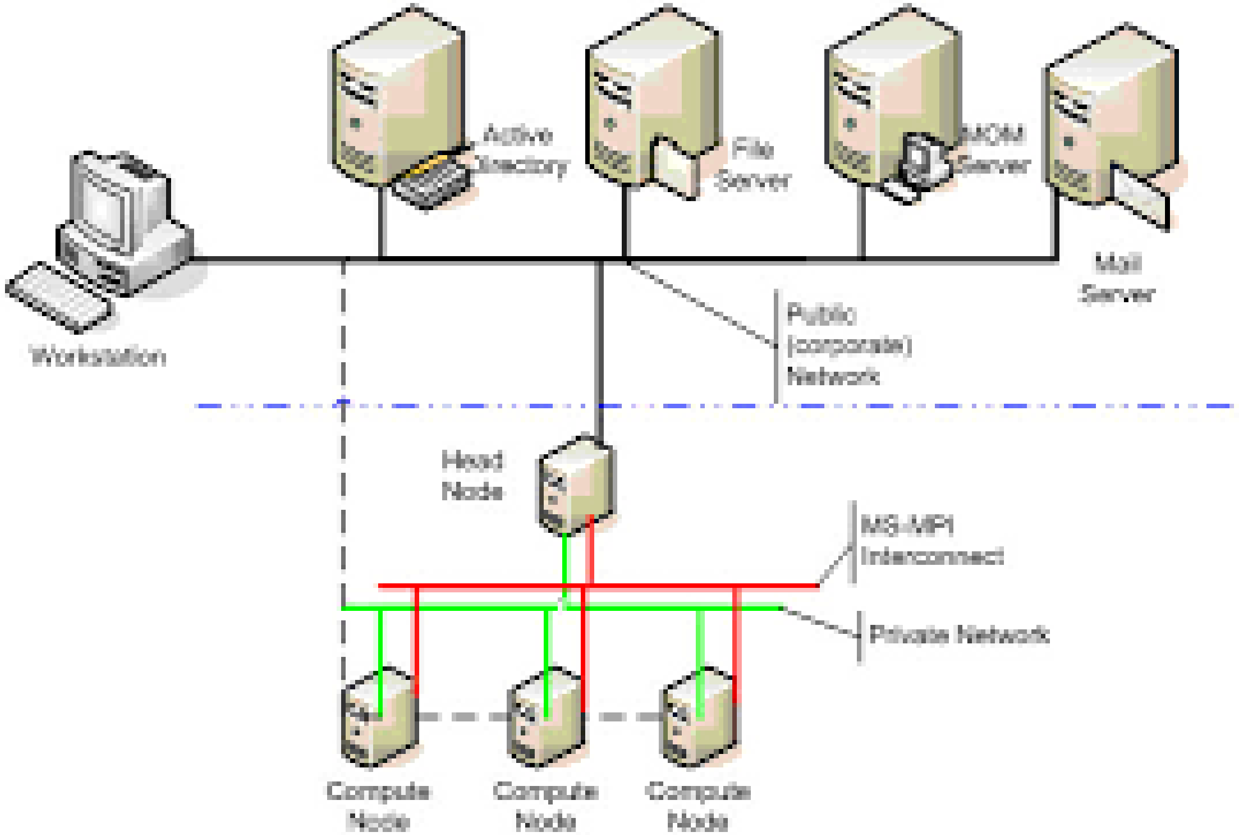
Compute Servers requires

- processors with high performance capabilities
- large amounts of memory
- relatively low disk subsystems

By separating data from the computation processing, the compute server's processing capabilities can be optimized

CLUSTER AS COMPUTE SERVER





DATA SERVER

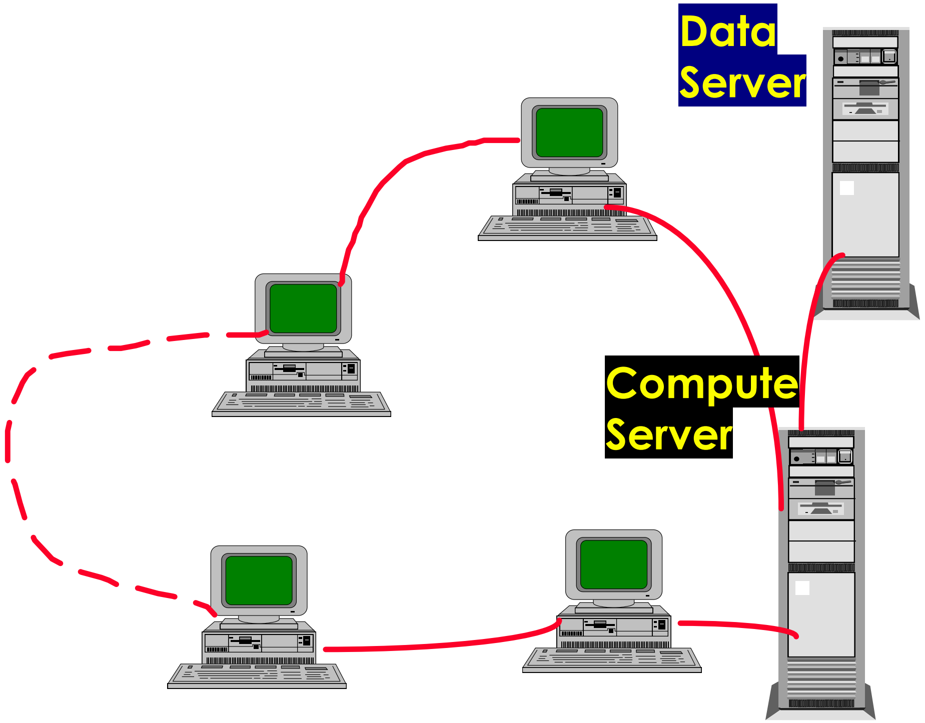
Data-oriented; used only for data storage and management

Since a data server can serve more than one compute server, compute-intensive applications can be spread among multiple severs

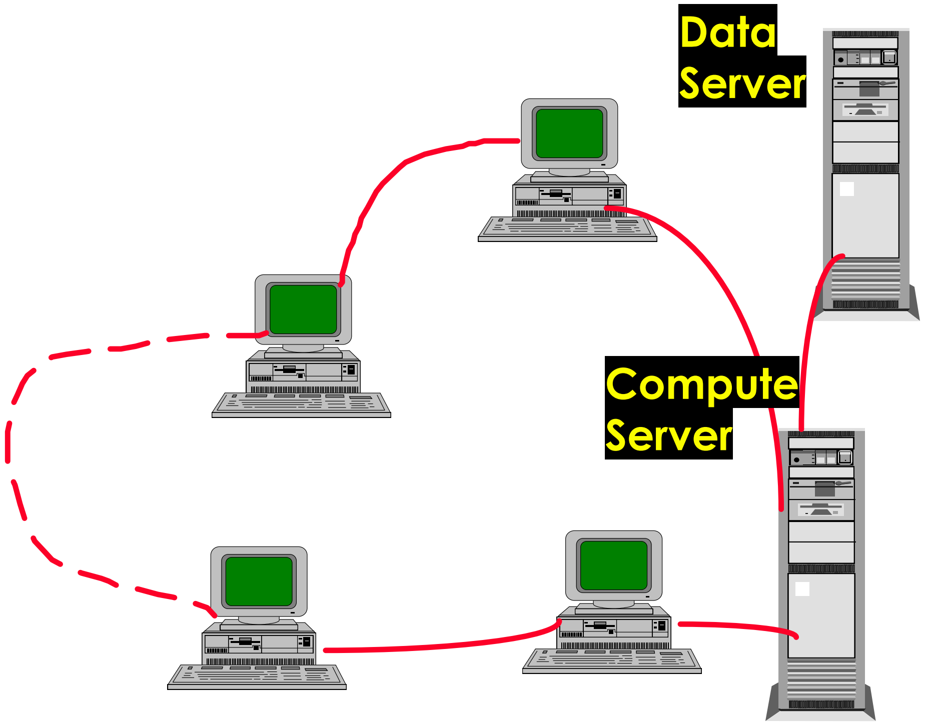
Does not prefer any application logic processing

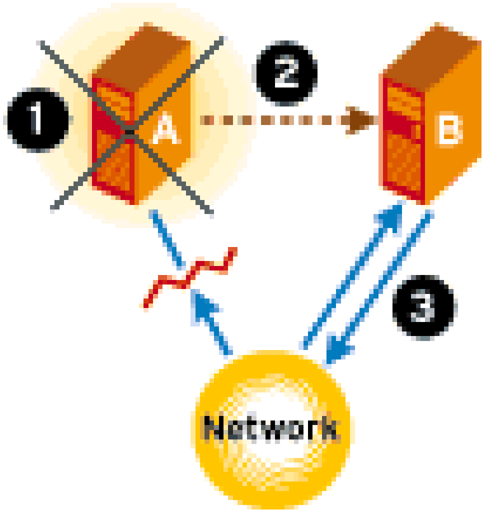
Performs processes such as data validation, required as part of the data management function.

Requires fast processor, large amount of memory and substantial Hard disk capacity.



CLUSTER AS HIGH AVAILABLITY DATA SERVER





DATABASE SERVER

Most typical use of technology in client-server

Accepts requests for data, retrieves the data from its database(or requests data from another node)and passes the results back.

Compute server with data server provides the same functionality.

The server requirement depends on the size of database, speed with which the database must be updated, number of users and type of network used.

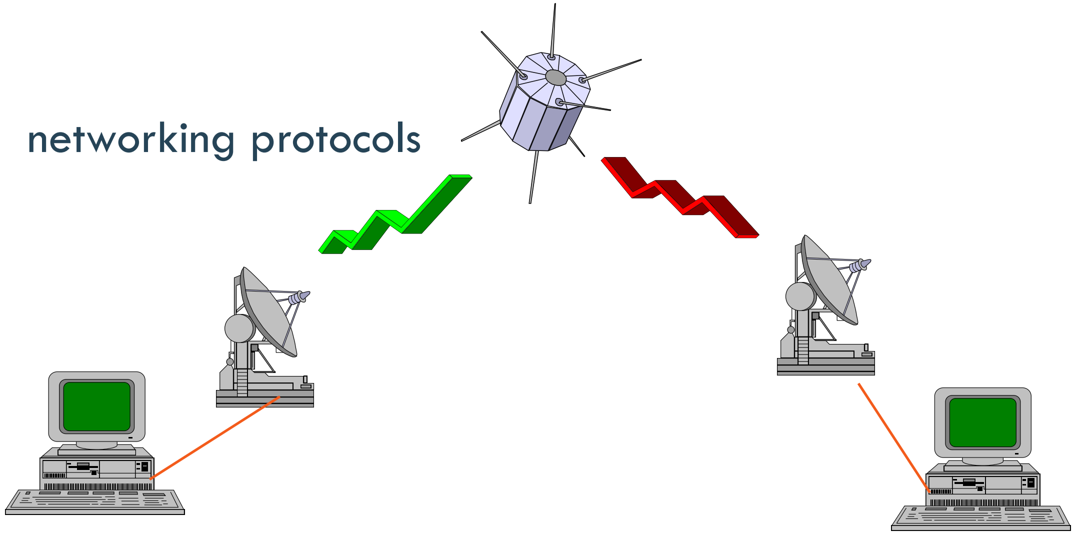
COMMUNICATION SERVER

□Provides gateway to other LANs, networks & Computers

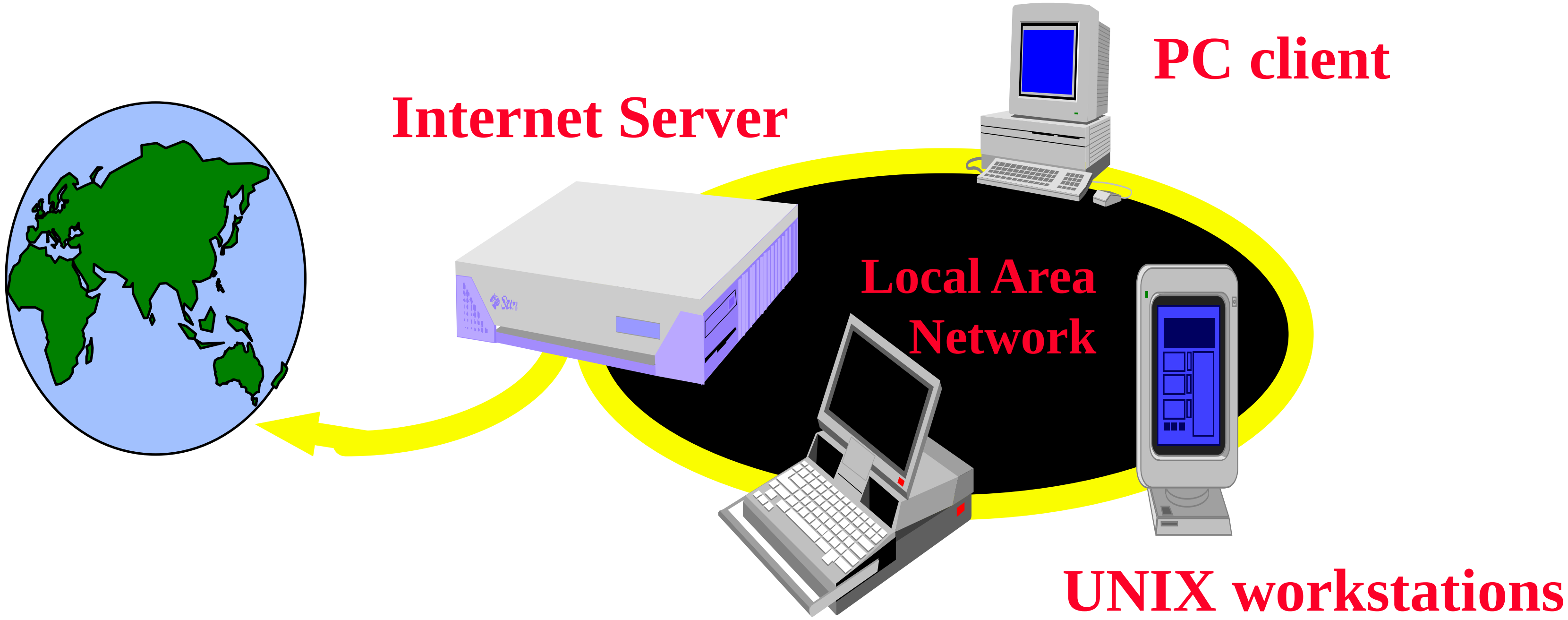
□E-mail Server & internet server

□Modest system requirements

  - □ multiple slots
  - □ fast processor to translate networking protocols



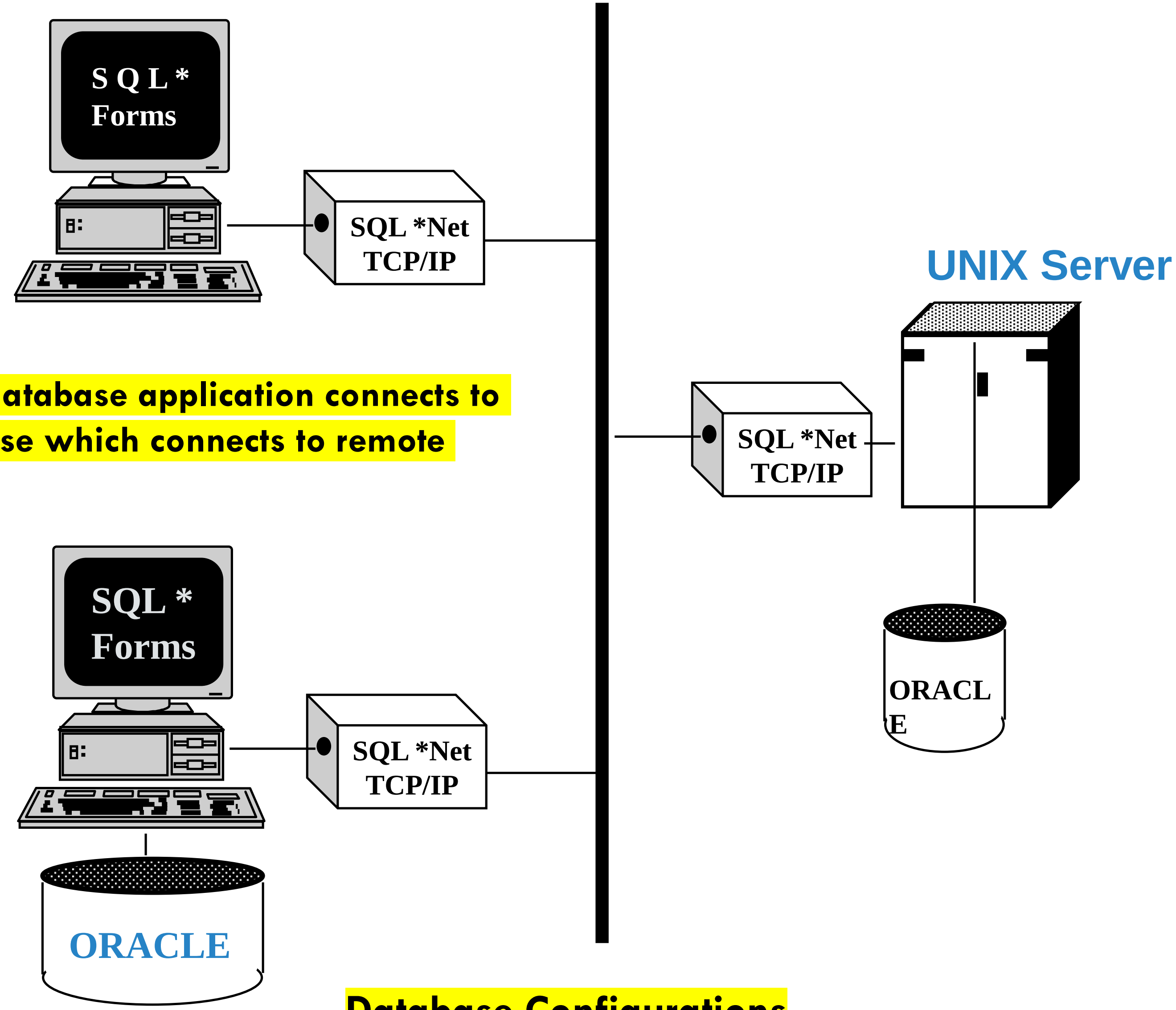
INTERNET SERVER

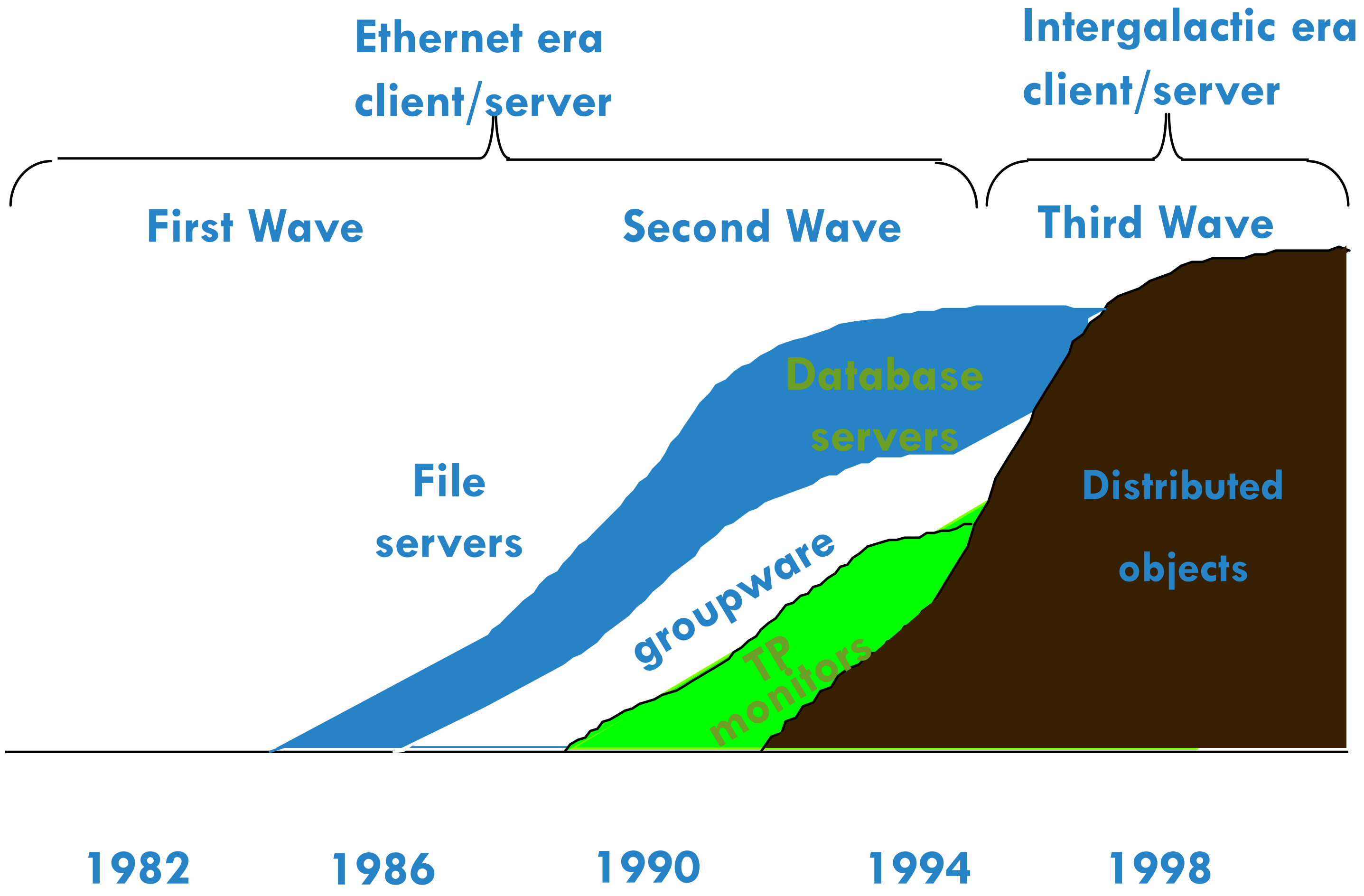


Database Configurations

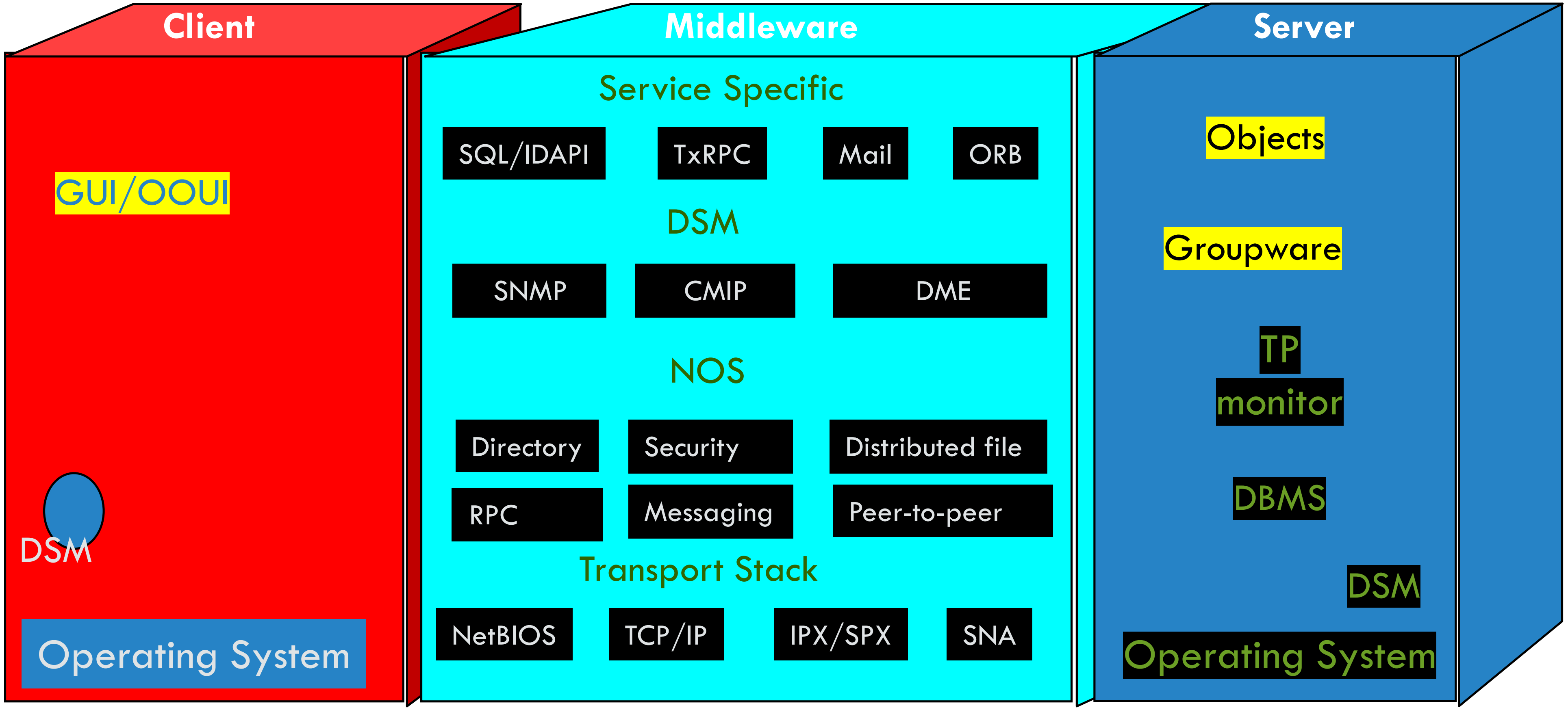
Distributed processing application connects to remote database

Distributed database application connects to local database which connects to remote database




Ethernet era
client/server
Intergalactic era
client/server
First Wave
Second Wave
Third Wave
File
servers
Database
servers
groupware
TP
monitors
Distributed
objects
1982
1986
1990
1994
1998

THE CLIENT/SERVER INFRASTRUCTURE



THE SOCKET INTERFACE

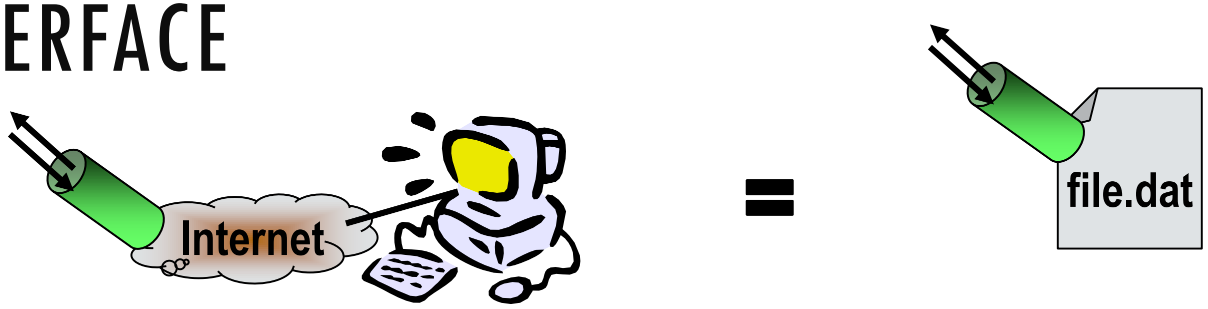
The ***Berkeley Sockets API***

- Originally developed as part of BSD Unix (under gov't grant)
  - BSD = Berkeley Software Distribution
  - API=Application Program Interface
- Now the most popular API for C/C++ programmers writing applications over TCP/IP
  - Also emulated in other languages: Perl, Tcl/Tk, etc.
  - Also emulated on other operating systems: Windows, etc.

THE SOCKET INTERFACE

The basic ideas:

- a ***socket*** is like a file:



  - you can read/write to/from the network just like you would a file
- For connection-oriented communication (e.g. TCP)
  - servers (passive open) do **listen** and **accept** operations
  - clients (active open) do **connect** operations
  - both sides can then do **read** and/or **write** (or **send** and **recv**)
  - then each side must **close**
  - There are more details, but those are the most important ideas
- Connectionless (e.g. UDP): uses **sendto** and **recvfrom**

SOCKETS AND SOCKET LIBRARIES

In Unix, socket procedures (e.g. listen, connect, etc.) **are *system calls***

- part of the operating system
- implemented in the "top half" of the kernel
- when you call the function, control moves to the operating system, and you are using "system" CPU time

SOCKETS AND SOCKET LIBRARIES

On some other systems, socket procedures are *not* part of the OS

- instead, they are implemented as a library, linked into the application object code (e.g. a DLL under Windows)

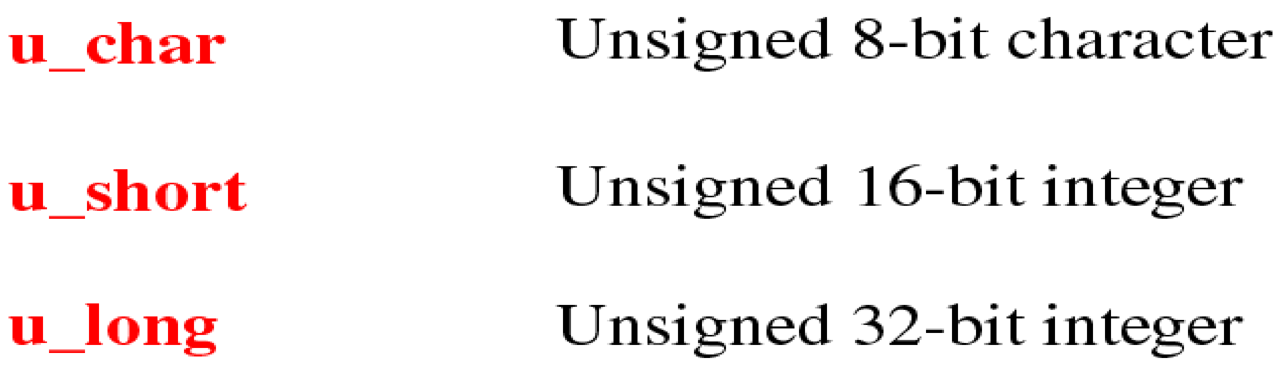
- Typically, this DLL makes calls to similar procedures that ***are*** part of the native operating system.

- This is what the Comer text calls a ***socket library***
  - A socket library simulates Berkeley sockets on OS's where the underlying OS networking calls are different from Berkeley sockets

SOME DEFINITIONS

Data types

int8_t	signed 8-bit integer
int16_t	signed 16-bit integer
int32_t	signed 32-bit integer
uint8_t	unsigned 8-bit integer
uint16_t	unsigned 16-bit integer
uint32_t	unsigned 32-bit integer

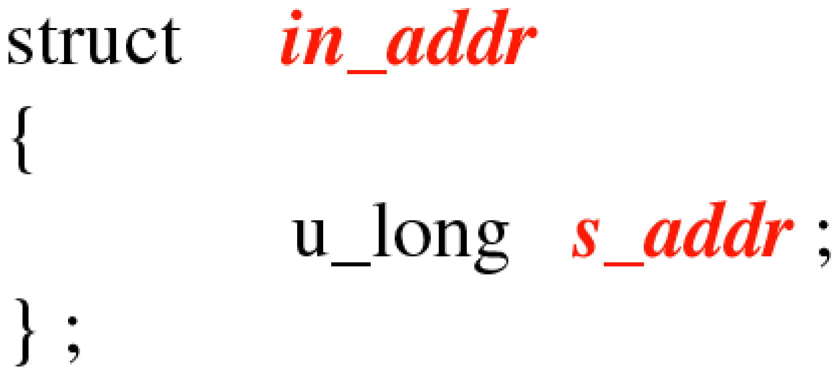


u_char	Unsigned 8-bit character
u_short	Unsigned 16-bit integer
u_long	Unsigned 32-bit integer

MORE DEFINITIONS

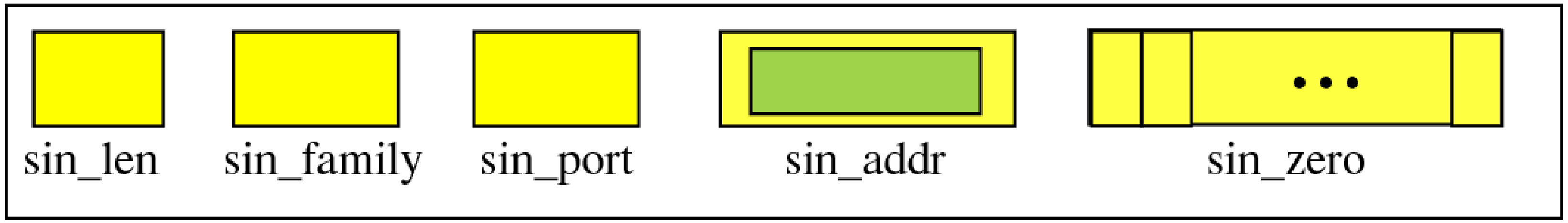
Internet Address Structure

```
struct in_addr
{
        in_addr_t          s_addr;
};
```



```
struct   in_addr
{
            u_long   s_addr ;
} ;
```

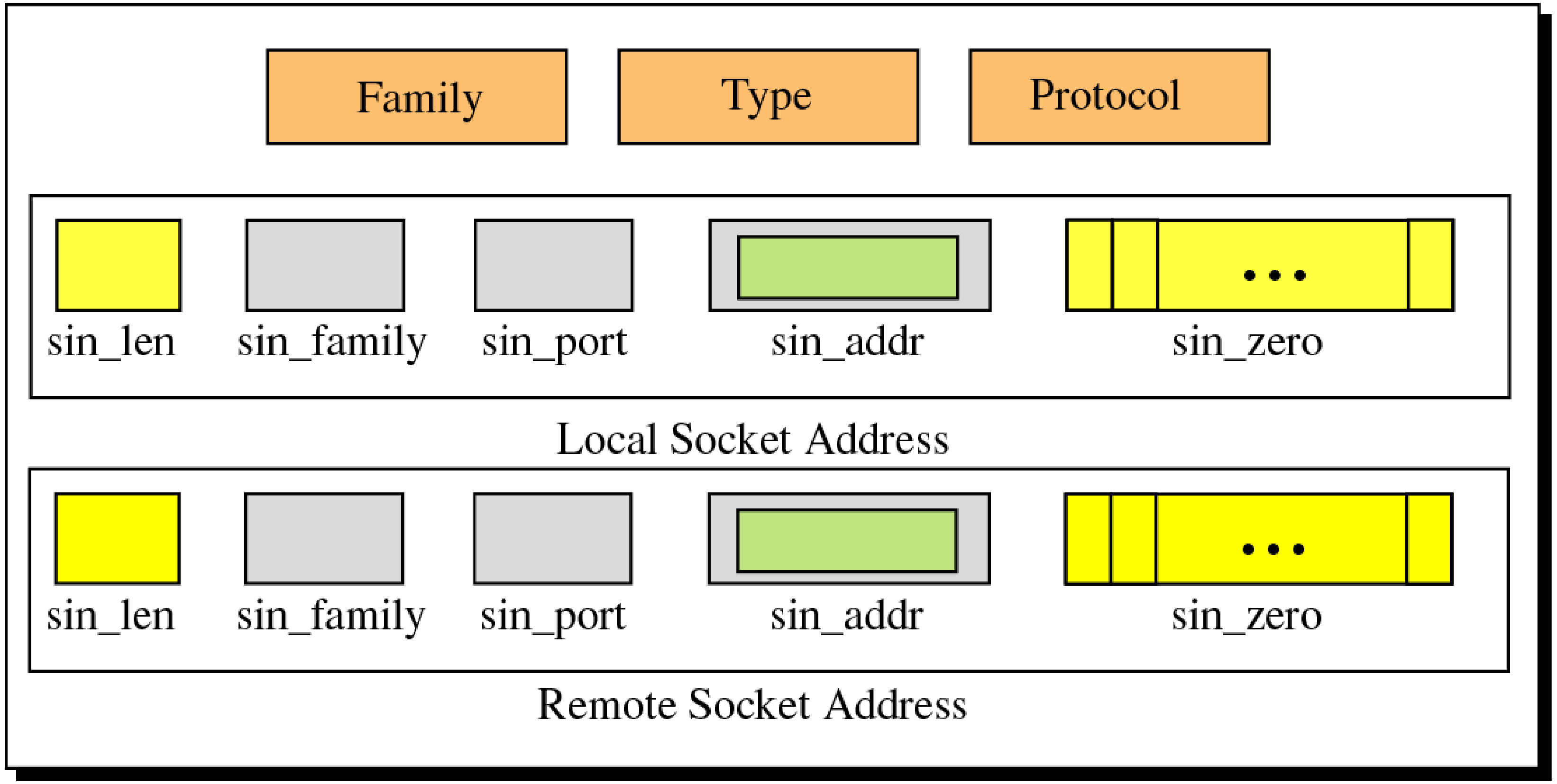
SOCKET ADDRESS STRUCTURE



sockaddr_in

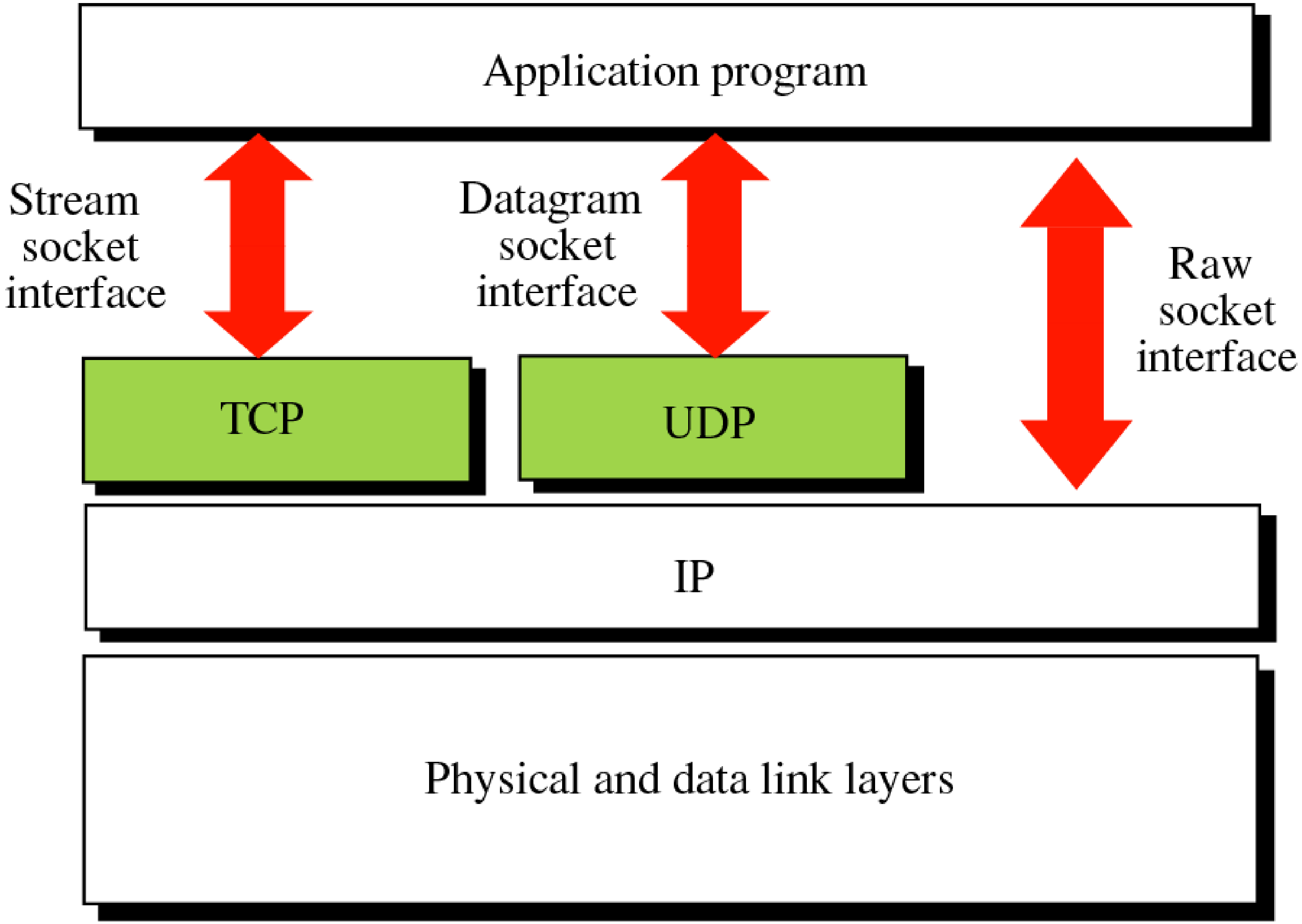
```
struct    sockaddr_in
{
          u_char                      sin_len ;
          u_short                     sin_family ;
          u_short                     sin_port ;
          struct  in_addr             sin_addr ;
          char                        sin_zero [8] ;
} ;
```

SOCKET STRUCTURE



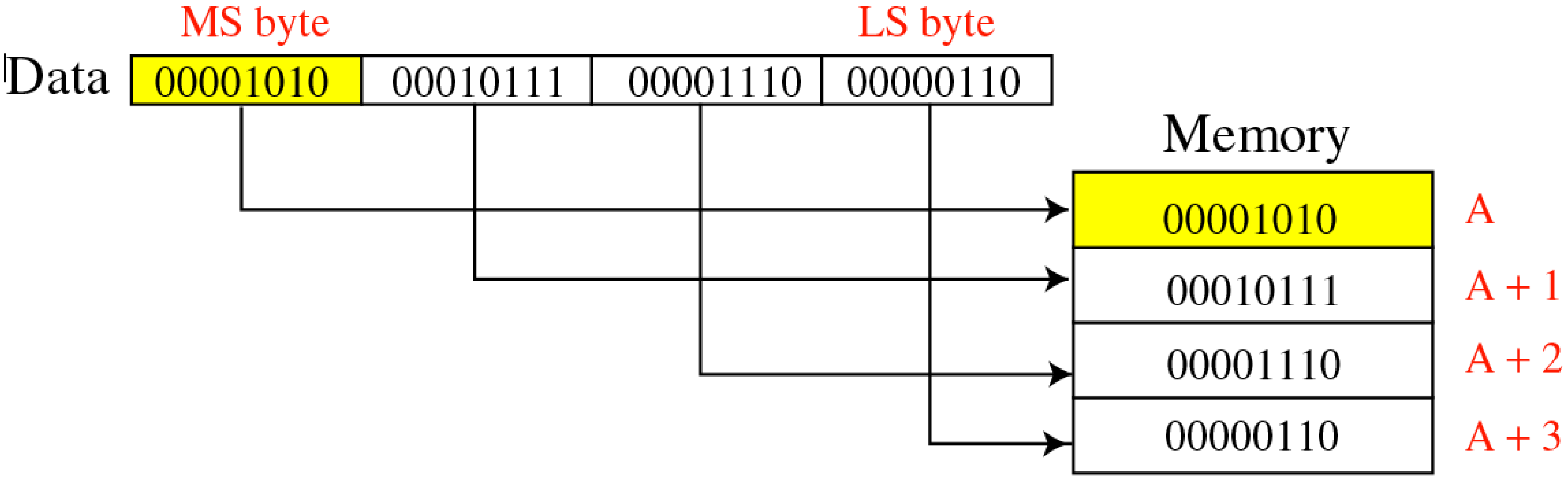
Socket

SOCKET


Application program
Stream socket interface
Datagram socket interface
Raw socket interface
TCP
UDP
IP
Physical and data link layers

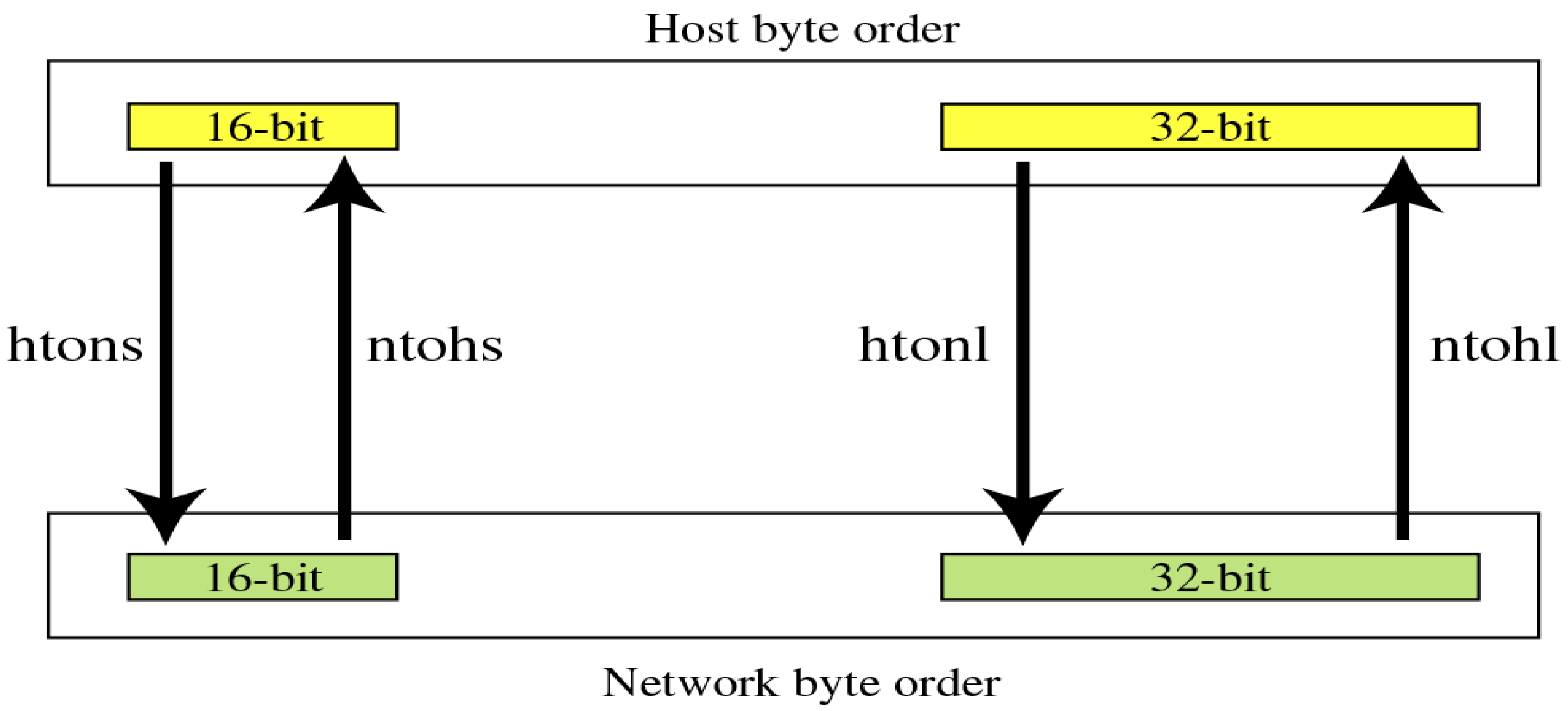
BYTE ORDERING

Big Endian



The byte order for the TCP/IP protocol suite is big endian.

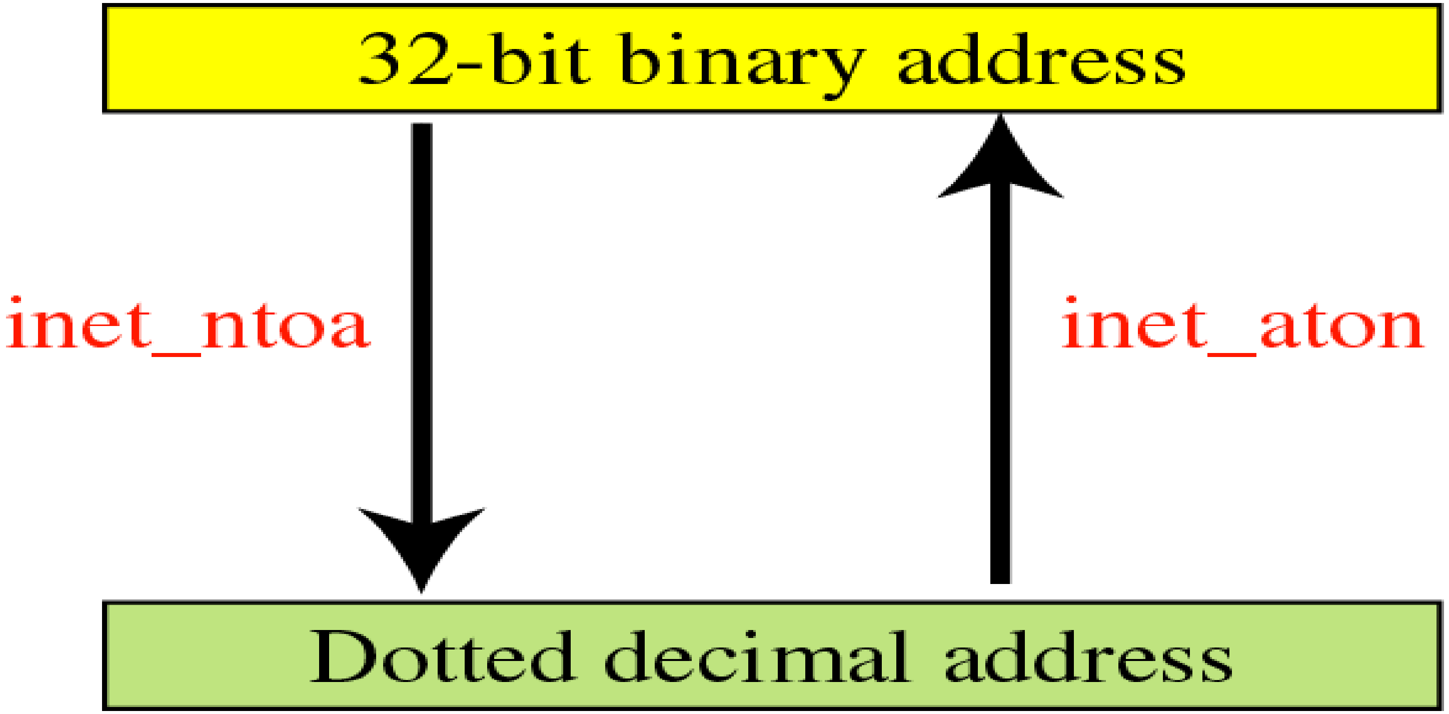
BYTE-ORDER TRANSFORMATION



```
u_short  htons  ( u_short  host_short ) ;

u_short  ntohs  ( u_short  network_short ) ;

u_long   htonl  ( u_long  host_long ) ;

u_long   ntohl  ( u_long  network_long ) ;
```

ADDRI

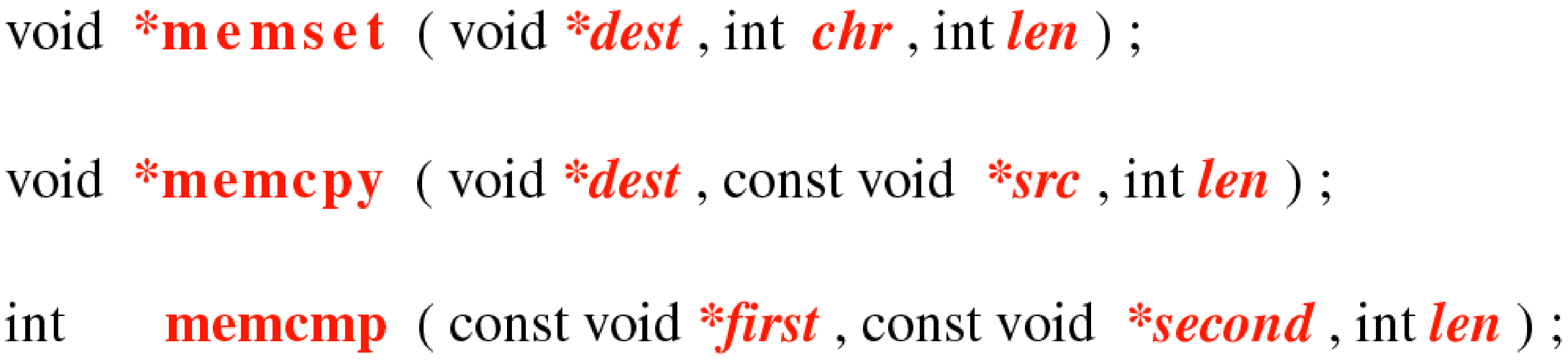
```
int        inet_aton  ( const char  *strptr , struct in_addr  *addrptr ) ;
char      *inet_ntoa  (struct in_addr  inaddr ) ;
```



BYTE-MANIPULATION FUNCTIONS

In network programming, we often need to initialize a field, copy the contents of one field to another, or compare the contents of two fields.

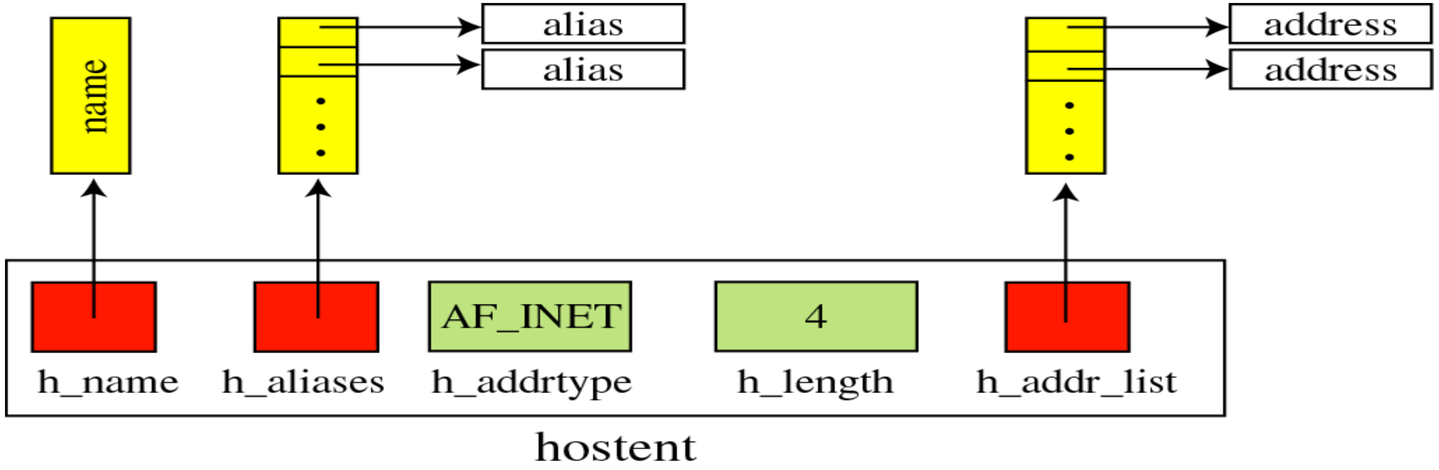
- Cannot use string functions (strcpy, strcmp, …) which assume null character termination.



```
void  *memset  ( void *dest , int  chr , int len ) ;

void  *memcpy  ( void *dest , const void  *src , int len ) ;

int      memcmp  ( const void *first , const void  *second , int len ) ;
```

INFORMATION ABOUT REMOTE HOST

struct hostent ***gethostbyname*** (const char ****hostname***) ;



```
struct     hostent
{
           char                             *h_name ;
           char                            **h_aliases ;
           int                               h_addrtype ;
           int                               h_length ;
           char                            **h_addr_list ;
} ;
```

PROCEDURES THAT IMPLEMENT THE SOCKET API

Creating and Deleting Sockets

`fd=`**`socket`**`(protofamily, type, protocol)`

Creates a new socket. Returns a file descriptor (fd). Must specify:

- the protocol family (e.g. TCP/IP)
- the type of service (e.g. STREAM or DGRAM)
- the protocol (e.g. TCP or UDP)

`close`(fd)

Deletes socket.

For connected STREAM sockets, sends EOF to close connection.

PROCEDURES THAT IMPLEMENT THE SOCKET API

Putting Servers "on the Air"

bind(fd)

Used by server to establish port to listen on.

When server has >1 IP addrs, can specify "ANY", or a specific one

listen (fd, queuesize)

Used by connection-oriented servers only, to put server "on the air"

Queuesize parameter: how many pending connections can be waiting

afd = **accept** (lfd, caddress, caddresslen)

Used by connection-oriented servers to accept one new connection

- There must already be a listening socket (lfd)
- Returns afd, a new socket for the new connection, and
- The address of the caller (e.g. for security, log keeping. etc.)

PROCEDURES THAT IMPLEMENT THE SOCKET API

How Clients Communicate with Servers

`connect` (fd, saddress, saddreslen)

Used by connection-oriented clients to connect to server

- There must already be a socket bound to a connection-oriented service on the fd
- There must already be a listening socket on the server
- You pass in the address (IP address, and port number) of the server.

Used by connectionless clients to specify a "default send to address"

- Subsequent "writes" or "sends" don't have to specify a destination address
- BUT, there really ISN'T any connection established… this is a bad choice of names!

PROCEDURES THAT IMPLEMENT THE SOCKET API

How Clients Communicate with Servers

`send` (fd, data, length, flags)

`sendto` (fd, data, length, flags, destaddress, addresslen)

`sendmsg` (fd, msgstruct, flags)

`write` (fd, data, length)

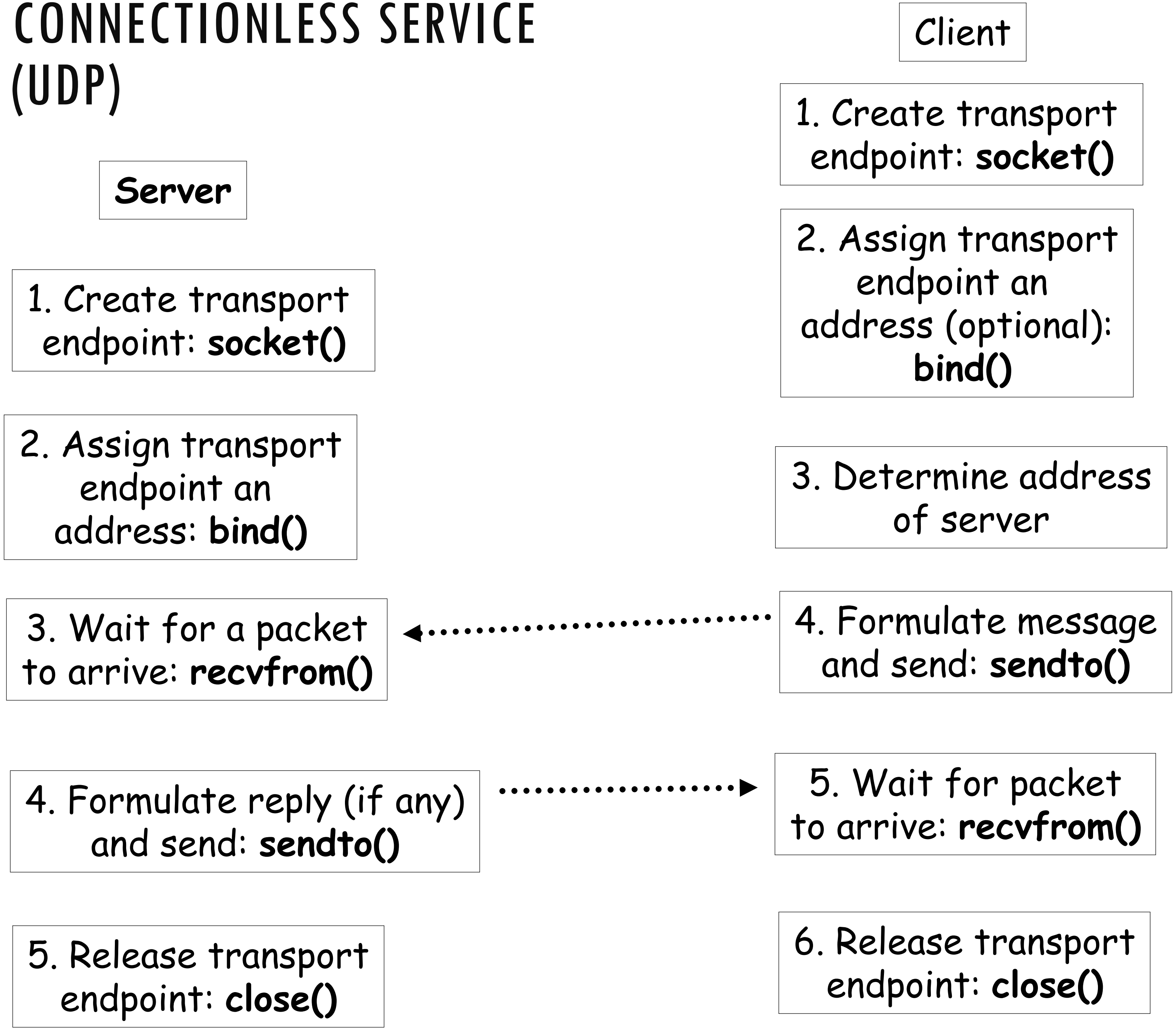
Used to send data.

- **send** requires a connection (or for UDP, default send address) be already established
- **sendto** used when we need to specify the dest address (for UDP only)
- sendmsg is an alternative version of sendto that uses a struct to pass parameters
- write is the "normal" write function; can be used with both files and sockets

`recv` (...) **`recvfrom`** (...) **`recvmsg`** (...) **`read`** (...)

Used to receive data… parameters are similar, but in reverse (destination => source, etc…)

CONNECTIONLESS SERVICE (UDP)

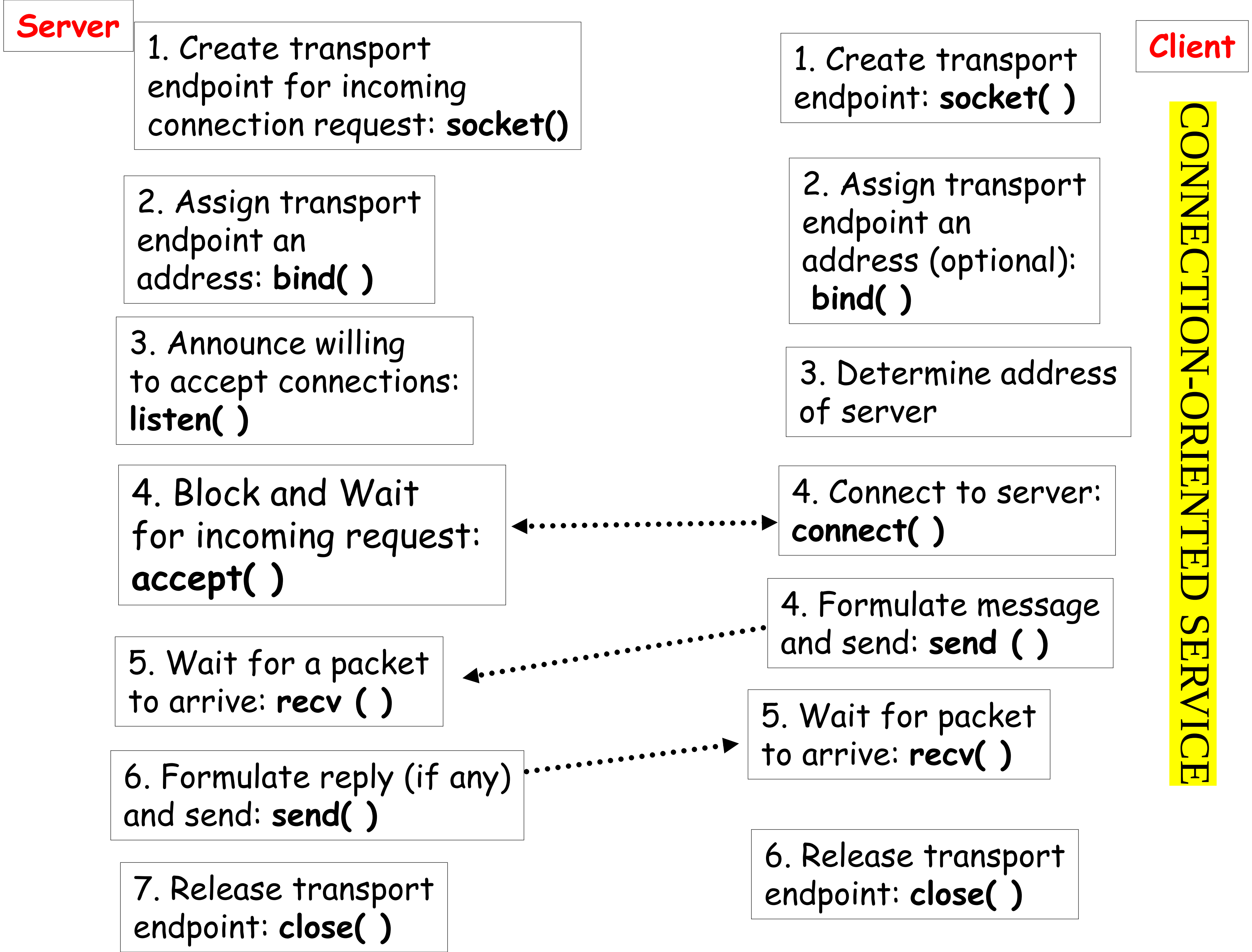


Server

1. Create transport endpoint: **socket()**
2. Assign transport endpoint an address: **bind()**
3. Wait for a packet to arrive: **recvfrom()**
4. Formulate reply (if any) and send: **sendto()**
5. Release transport endpoint: **close()**

Client

1. Create transport endpoint: **socket()**
2. Assign transport endpoint an address (optional): **bind()**
3. Determine address of server
4. Formulate message and send: **sendto()**
5. Wait for packet to arrive: **recvfrom()**
6. Release transport endpoint: **close()**



CONNECTION-ORIENTED SERVICE

Server

1. Create transport endpoint for incoming connection request: **socket()**

2. Assign transport endpoint an address: **bind( )**

3. Announce willing to accept connections: **listen( )**

4. Block and Wait for incoming request: **accept( )**

5. Wait for a packet to arrive: **recv ( )**

6. Formulate reply (if any) and send: **send( )**

7. Release transport endpoint: **close( )**

Client

1. Create transport endpoint: **socket( )**

2. Assign transport endpoint an address (optional): **bind( )**

3. Determine address of server

4. Connect to server: **connect( )**

4. Formulate message and send: **send ( )**

5. Wait for packet to arrive: **recv( )**

6. Release transport endpoint: **close( )**

AN EXAMPLE SERVICE

Connection-oriented service

- Server: keeps a count of number of clients that have accessed its service, then reports the count when a client contacts the server.
- Client: displays the data it receives from the server

Example output:
This server has been contacted 10 times

```
/* To compile me in Solaris, type:  gcc -o client client.c -lsocket -lnsl */
/* To compile me in Linux, type:    gcc -o client client.c  */

/* client.c - code for example client that uses TCP         */
/* From Computer Networks and Internets by Douglas F. Comer */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <stdio.h>
#include <string.h>

#define closesocket      close
#define PROTOPORT        5193        /* default protocol port number */
```

```
extern int          errno;
char   localhost[] = "localhost";    /* default host name          */
/*------------------------------------------------------------------
 * Program:   client
 *
 * Purpose:   allocate a socket, connect to a server, and print all output
 *
 * Syntax:    client [ host [port] ]
 *
 *            host - name of a computer on which server is executing
 *            port - protocol port number server is using
 *
 * Note:      Both arguments are optional.  If no host name is specified,
 *           the client uses "localhost";  if no protocol port is
 *           specified, the client uses the default given by PROTOPORT.
 *
 *------------------------------------------------------------------ */
```

Example Client

```
main(int argc, char *argv[])
{
  struct  hostent  *ptrh;   /* pointer to a host table entry       */
  struct  protoent *ptrp;   /* point to a protocol table entry     */
  struct  sockaddr_in sad;  /* structure to hold server's address  */
  int     sd;               /* socket descriptor                   */
  int     port;             /* protocol port number                */
  char    *host;            /* pointer to host name                */
  int     n;                /* number of characters read           */
  char    buf[1000];        /* buffer for data from the server     */

  memset((char *)&sad, 0, sizeof(sad));  /* clear sockaddr structure */
  sad.sin_family = AF_INET;         /* set family to Internet   */

  /* Check command-line argument for protocol port and extract     */
  /* port number if on is specified.  Otherwise, use the default   */
  /* port value given by constant PROTOPORT                        */
```

Example Client

```
if (argc > 2) port = atoi(argv[2]);
else port = PROTOPORT;

if (port > 0) sad.sin_port = htons((u_short)port);
else
 { fprintf( stderr,"bad port number %s\n", argv[2]);
    exit(1);
 }

if (argc > 1 ) host = argv[1];
else host = localhost;

ptrh = gethostbyname(host);
if( ((char *) ptrh) == NULL)
 { fprintf( stderr, "invalid host:  %s\n", host);
  exit(1);
 }
```

Example Client

```
memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);

if ( ((int)(ptrp = getprotobyname("tcp"))) == 0)
  { fprintf( stderr, "cannot map \"tcp\" to protocol number\n");
    exit(1);
  }

sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0)
  { fprintf( stderr, "socket creation failed\n");
    exit(1);
  }

if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0)
  { fprintf( stderr, "connect failed\n");
    exit(1);
  }
```

Example Client

```
  n = recv(sd, buf, sizeof(buf), 0);
  while(n > 0)
   {
    buf[n] = '\0';
    printf("CLIENT: %s", buf); /* or also write(1, buf, n)
    n = recv(sd, buf, sizeof(buf), 0);
   }

  closesocket(sd);
  exit(0);
}
```

Example Client

```
/* to compile me on Solaris, type:   gcc -o server server.c -lsocket -lnsl */
/* to compile me in Linux, type:     gcc -o server server.c  */

/* server.c - code for example server program that uses TCP */
/* From Computer Networks and Internets by Douglas F. Comer */
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#include <stdio.h>
#include <string.h>

#define PROTOPORT       5193        /* default protocol port number */
#define QLEN             6          /* size of request queue        */

int visits =  0;                    /* counts client connections    */
```

Example Server

```
/*------------------------------------------------------------
* Program:     server
*
* Purpose:       allocate a socket and then repeatedly execute the folllowing:
*                  (1) wait for the next connection from a client
*                  (2) send a short message to the client
*                  (3) close the connection
*                  (4) go back to step (1)
*
* Syntax:        server [ port ]
*
*                    port  - protocol port number to use
*
* Note:          The port argument is optional. If no port is specified,
*                the server uses the default given by PROTOPORT.
*
*------------------------------------------------------------*/
```

Example Server

```
main (argc, argv)
int         argc;
char         *argv[];
{
   struct   hostent   *ptrh;     /* pointer to a host table entry */
   struct   protoent  *ptrp;     /* pointer to a protocol table entry */
   struct   sockaddr_in sad;     /* structure to hold server's address */
   struct   sockaddr_in cad;     /* structure to hold client's address */
   int      sd, sd2;             /* socket descriptors */
   int      port;                /* protocol port number */
   int      alen;                /* length of address */
   char     buf[1000];           /* buffer for string the server sends */

   memset((char  *)&sad,0,sizeof(sad)); /* clear sockaddr structure   */
   sad.sin_family = AF_INET;            /* set family to Internet    */
   sad.sin_addr.s_addr = INADDR_ANY;    /* set the local IP address */
```

Example Server

```
/* Check  command-line argument for protocol port and extract      */
/* port number if one is specfied.  Otherwise, use the default     */
/* port value given by constant PROTOPORT                           */

if (argc > 1) {                        /* if argument specified     */
                port = atoi (argv[1]); /* convert argument to binary*/
} else {
                port = PROTOPORT;      /* use default port number   */
}
if (port > 0)                          /* test for illegal value    */
                sad.sin_port = htons((u_short)port);
else {                                 /* print error message and exit */
                fprintf (stderr, "bad port number %s/n",argv[1]);
                exit (1);
}
```

Example Server

```
/* Map TCP transport protocol name to protocol number */

  if ( ((int)(ptrp = getprotobyname("tcp"))) == 0)  {
            fprintf(stderr, "cannot map \"tcp\" to protocol number");
            exit (1);
  }

  /* Create a socket */
  sd = socket (PF_INET, SOCK_STREAM, ptrp->p_proto);
  if (sd < 0) {
             fprintf(stderr, "socket creation failed\n");
             exit(1);
  }
```

Example Server

```
/* Bind a local address to the socket */
    if (bind(sd, (struct sockaddr *)&sad, sizeof (sad)) < 0) {
                    fprintf(stderr,"bind failed\n");
                    exit(1);
    }
/* Specify a size of request queue */
    if (listen(sd, QLEN) < 0) {
                    fprintf(stderr,"listen failed\n");
                     exit(1);
    }
```

Example Server

```
/* Main server loop - accept and handle requests */
    printf("Server up and running.\n");
    while (1) {
        alen = sizeof(cad);
        fprintf( stderr, "SERVER: Waiting for contact ...\n");

        if (  (sd2=accept(sd, (struct sockaddr *)&cad, &alen)) < 0) {
                        fprintf(stderr, "accept failed\n");
                        exit (1);
         }
         visits++;
         sprintf(buf,"This server has been contacted %d time%s\n",
                    visits, visits==1?".":"s.");
         printf("SERVER: %s", buf);
         send(sd2,buf,strlen(buf),0);
         closesocket (sd2);
         }
}
```

Example Server

ANOTHER EXAMPLE: ECHO SERVICE

```
/* TCP echo service on the server side */
int TCPechoServer (int fd)
{       char buf[BUFSIZE];
    int cc;

    while ( cc = read(fd, buf, sizeof(buf)) ) {
        if (cc < 0)
            errexit(“echo read: %s\n”, strerror(errno));
        if (write(fd, buf, cc) <0)
            errexit(“echo write: %s\n”, sterror(errno));
}
```

```
/* TCP echo service on the client side */
int TCPechoClient (int fd)
{ char buf[BUFSIZE+1]; /* buffer for one line of text */
  int n, outchars, inchars;
  while ( fgets(buf, sizeof(buf), stdin) ) {
          buf[BUFSIZE] = ‘\0’;
          outchars = strlen(buf);
          (void) write(fd, buf, outchars);
          /* read it back*/
          for (inchars = 0; inchars < outchars; inchars+=n ) {
                  n = read( fd, &buf[inchars], outchars-inchars);
                  if (n <0)
                          errexit(“socket read failed: %s\n”, strerror(errno));
                  }
                  fputs(buf, stdout);
          }
}
```