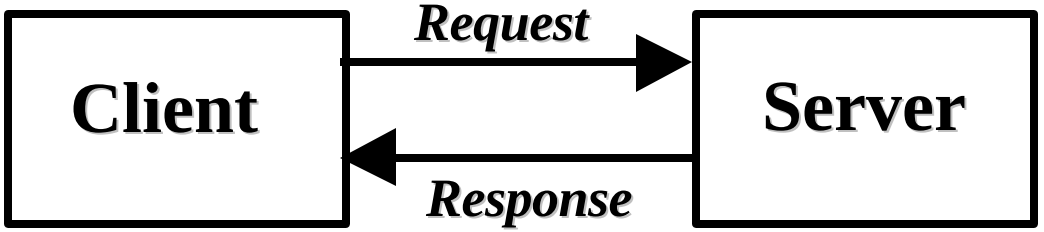
The Application Layer

- **Client/Server Computing, Basic Approaches:**
  - **Passing Messages.**
    - **Example: Communication through sockets (socket programming).**
  - **Remote Procedure Call (RPC).**
    - **Examples: Sun RPC, Distributed Computing Environment (DCE) RPC.**
- **Application/Transport Layer Interface:**
  - **Example: Berkeley Sockets.**
- **Application Layer Protocols:**
  - **Examples: Telnet, FTP, HTTP, etc.**
- **Network Security:**
  - **Encryption basics,**
  - **DES,**
  - **Public-Key Encryption.**
- **Domain Name System (DNS).**
- **Data Compression Basic Techniques.**
  - **Digital Image Compression Example: JPEG.**

Client/Server Computing

- **Client:** A program that initiates a connection request for data from another program using a specific high-level protocol.
- **Server or daemon:** A program running on a machine that responds to communication requests from clients by listening to a specific local address or port and responds with the required data or services using the same specific high-level protocol as the clients.



Realizing Client/Server Communication:

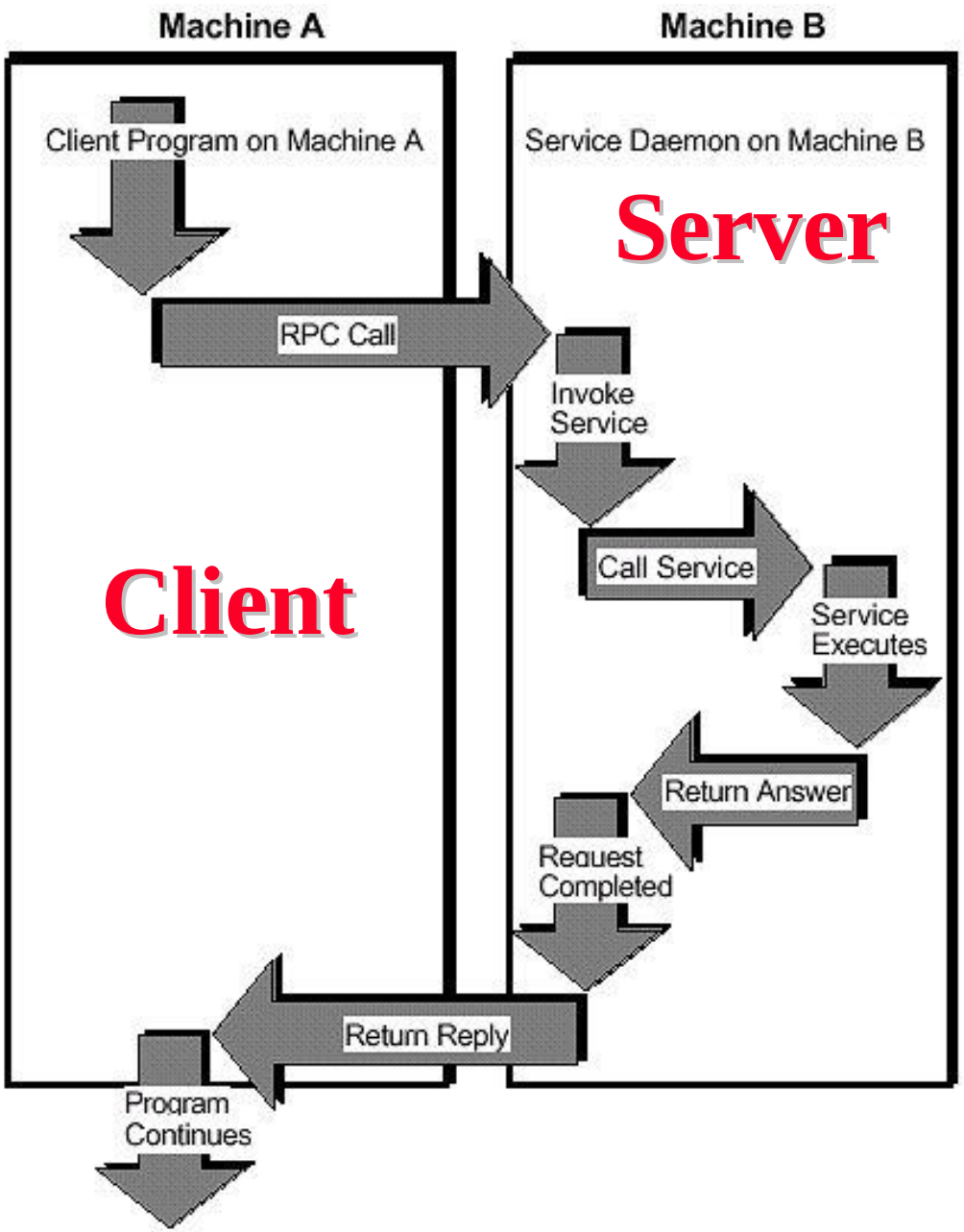
Remote Procedure Calls (RPC)

- **Client/Server application development is possible using remote procedure calls (RPC). This approach:**
  - **Provides the required logical client/server communication.**
  - **Does not require programming most of the interface to the underlying network (more specifically the interface to the transport layer).**
- **With RPC, the client makes a remote procedure call that sends requests to the server, which calls a dispatch routine, performs the requested service, and sends back a reply before the call returns to the client.**
- **Example: a program can simply call rnusers (a C routine that returns the number of users on a remote machine) much like making a local system call (i.e. to malloc).**

The RPC Model

The remote procedure call model is similar to that of the local procedure calls model, which works as follows:

1. **The caller places arguments to a procedure in a specific location (such as a result register).**
2. **The caller temporarily transfers control to the procedure.**
3. **When the caller gains control again, it obtains the results of the procedure from the specified location.**
4. **The caller then continues program execution.**



Basic Client/Server Network Communication Using RPC:

Synchronous RPC call:

Only one of the two processes is active at a given time.

Realizing Client/Server Communication:

Berkeley Sockets

- Sockets, introduced in Berkeley Unix, are a basic mechanism for Inter-Process Communication (IPC) on the same computer system, or on different computer systems connected over a network.
- A socket is an endpoint used by a process for bidirectional communication with a socket associated with another process.
- The communication channel created with sockets can be connection-oriented or connectionless datagram oriented, with the sockets serving as mailboxes.
- IPC using sockets is similar to file I/O operation:
  - A socket appears to the user program to be like a file descriptor on which one can read, write, etc.
  - In the connection-oriented mode, the socket acts like a regular file where a sequence of characters can be read using many read operations.
  - In the connectionless mode, one must get the whole message in a single read operation. Otherwise the rest of the message is lost.

Berkeley Sockets Primary Functions

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

A Server Application Using TCP Socket Primitives

socket => bind => listen => {accept => {read | recvfrom => write | sendto}* }* => close | shutdown

- **Create a socket,**
- **Bind it to a local port,**
- **Set up service with indication of maximum number of concurrent services,**
- **Accept requests from connection oriented clients,**
- **receive messages and reply to them,**
- **Terminate.**

A Client Application Using TCP Socket Primitives

socket => [bind =>] connect => {write | sendto => read | recvfrom }* => close | shutdown

- **Create a socket,**
- **Bind it to a local port,**
- **Establish the address of the server,**
- **Communicate with it,**
- **Terminate.**
- **If bind is not used, the kernel will select a free local port.**

Socket Functions: Creating A Socket socket

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol)
```

- **domain is either AF_UNIX, AF_INET, or AF_OSI, or ..**
  - **AF_UNIX is the Unix domain, it is used for communication within a single computer system.**
  - **AF_INET is for communication on the internet to IP addresses.**
- **type is either:**
  - **SOCK_STREAM (TCP, connection oriented, reliable), or**
  - **SOCK_DGRAM (UDP, datagram, unreliable), or**
  - **SOCK_RAW (IP level).**
  - **It is the name of a file if the domain is AF_UNIX.**
- **protocol specifies the protocol used. Usually 0 to use the default protocol for the chosen domain and type.**
- **If successful, the function returns a socket descriptor which is an int. Returns -1 in case of failure.**

Socket Addresses

- **Typical structures used to store socket addresses as used in the domain AF_INET:**

```
struct in_addr {
        u_long s_addr;
      };

    struct sockaddr_in {
        u_short         sin_family; /*protocol identifier; usually AF_INET */
        u_short         sin_port;   /*port number. 0 means let kernel choose */
        struct in_addr sin_addr;    /*the IP address. INADDR_ANY refers */
                                    /*IP addresses of the current host.*/
                                    /*It is considered a wildcard IP address.*/
        char            sin_zero[8];}; /*Unused, always zero */
```

Socket Functions: Binding to A Local Port

- **Bind is used to specify for a socket the protocol port number where it will wait for messages.**

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

  - **sd: File descriptor of local socket, as created by the socket function.**
  - **addr: Pointer to protocol address structure of this socket.**
  - **addrlen: Length in bytes of structure referenced by addr.**
  - **It returns an integer, the return code (0=success, -1=failure).**

- **A call to bind is optional on the client side, required on the server side.**

A Call to Bind Example

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

#define MYPORT 3490

main()
{
    int sockfd;
    struct sockaddr_in my_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    my_addr.sin_family = AF_INET;     /* host byte order */
    my_addr.sin_port = htons(MYPORT); /* short, network byte order */
    my_addr.sin_addr.s_addr = inet_addr("129.21.195.74");
    bzero(&(my_addr.sin_zero), 8);    /* zero the rest of the struct
*/

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct
        sockaddr)) < 0 {
           perror("bind"); exit(1);}
       …...
```

Socket Functions: connect

- The connect operation is used on the client side to identify and, possibly, start the connection to the server.
- Required in the case of connection oriented communication, but not in the datagram case.

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

 int connect(int sd, struct sockaddr *addr, int addrlen)
```

  - sd: file descriptor of local socket.
  - Addr: pointer to protocol address of other socket.
  - Addrlen: length in bytes of address structure.
  - It returns an integer (0=success, -1=failure).

Connect Example

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

 #define DEST_IP   "132.241.5.10"
 #define DEST_PORT 23

 main()
 {
     int sockfd;
     struct sockaddr_in dest_addr;   /* will hold the destination addr */

     sockfd = socket(AF_INET, SOCK_STREAM, 0); /* create socket */

     dest_addr.sin_family = AF_INET;        /* host byte order */
     dest_addr.sin_port = htons(DEST_PORT); /* short, network byte order */
     dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
     bzero(&(dest_addr.sin_zero), 8);       /* zero the rest of the struct */
    connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
     .
     .
```

Socket Functions: listen

- The listen function is used on the server in the case of connection oriented communication to prepare a socket to accept messages from clients. It has the form:

```
int listen(int fd, int qlen)
```

  - fd: file descriptor of a socket that has already been bound
  - qlen: specifies the maximum number of messages that can wait to be processed by the server while the server is busy servicing another request.
  - It returns an integer (0=success, -1=failure)
- Here is a typical call to listen:

```
if (listen(sd, 3) < 0) {
     {perror("listen"); exit(1);}
```

Socket Functions: accept

- **The accept function is used on the server in the case of connection oriented communication (after a call to listen) to accept a connection request from a client.**

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

int accept(int fd, struct sockaddr *addressp, int *addrlen)
```

  - **fd: the file descriptor of the socket the server was listening on [in fact it is called the listening socket].**
  - **Addressp: points to an address. It will be filled with address of the calling client. We can use this address to determine the IP address and port of the client.**
  - **Addrlen: integer that will contain the actual length of address structure of client.**
  - **It returns an integer representing a new socket (-1 in case of failure).**
  - **It is the socket that the server will use from now on to communicate with the client that requested connection [in fact it is called the connected socket].**

Accept Example

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

#define MYPORT 3490    /* the port users will be connecting to */

#define BACKLOG 10     /* how many pending connections queue will hold */

main()
{
    int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr;    /* my address information */
    struct sockaddr_in their_addr; /* connector's address information */
    int sin_size;

    sockfd = socket(AF_INET, SOCK_STREAM, 0); /* do some error checking! */

    my_addr.sin_family = AF_INET;         /* host byte order */
    my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
    my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
    bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

   bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

    listen(sockfd, BACKLOG);

    sin_size = sizeof(struct sockaddr_in);
    new_fd = accept(sockfd, &their_addr, &sin_size);
    .
```

Socket Functions: send

- Send information over stream sockets or connected datagram sockets.
- sendto() used for unconnected datagram sockets.

```
int send(int sockfd, const void *msg, int len, int flags);
```

  - sockfd: is the socket descriptor you want to send data to. (whether it's the one returned by socket() or the one you got with accept().)
  - msg: is a pointer to the data you want to send, and
  - len: the length of that data in bytes.
  - Flags: Options. Usually set to 0.

- Example code:

```
char *msg = ”EECC 694";
int len, bytes_sent;
.
.
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
```

- send() returns the number of bytes actually sent out which might be less than the number requested.
- Again, -1 is returned on error, and errno is set to the error number.

Socket Functions: recv

- Receive information from stream sockets or connected datagram sockets.
- recvfrom() used for unconnected datagram sockets.

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

  - sockfd: the socket descriptor to read from,
  - buf: the buffer to read the information into,
  - len is the maximum length of the buffer, and
  - flags: can again be set to 0.
  - recv() returns the number of bytes actually read into the buffer, or -1 on error (with errno set, accordingly.)

Socket Functions: sendto and recvfrom

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
                const struct sockaddr *to, int tolen);
```

- **Similar to send() with the addition of two other pieces of information.**
  - **to: a pointer to a struct sockaddr which contains the destination IP address and port.**
  - **tolen can simply be set to sizeof(struct sockaddr).**
  - **Just like with send(), sendto() returns the number of bytes actually sent (which, again, might be less than the number of bytes requested, or -1 on error.**

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags
                  struct sockaddr *from, int *fromlen);
```

- **Similar to recv() with the addition of a two fields.**
  - **from is a pointer to a local struct sockaddr that will be filled with the IP address and port of the originating machine.**
  - **fromlen is a pointer to a local int that should be initialized to sizeof(struct sockaddr). When the function returns, fromlen will contain the length of the address actually stored in from.**
  - **recvfrom() returns the number of bytes received, or -1 on error.**

Socket Functions: close, shutdown

- To close the connection on a socket descriptor, the regular Unix file descriptor close() function may be used.

```
close(sockfd);
```

- For more control over how the socket closes, shutdown() may be used. It allows communication to be cut off in a certain direction, or both ways (just like close() does.)

```
int shutdown(int sockfd, int how);
```

  - sockfd: socket file descriptor to be shutdown, and how is one of the following:
    - 0 - Further receives are disallowed
    - 1 - Further sends are disallowed
    - 2 - Further sends and receives are disallowed (like close())
  - shutdown() returns 0 on success, and -1 on error.

- If shutdown() is used on unconnected datagram sockets, it will simply make the socket unavailable for further send() and recv() calls (they can only be used if connect() is used with the datagram socket.)

Socket Functions: getpeername

- The function getpeername() will indicate who is at the other end of a connected stream socket. The synopsis:

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

  - sockfd:is the descriptor of the connected stream socket,
  - addr: a pointer to a struct sockaddr (or a struct sockaddr_in) that will hold the information about the other side of the connection, and
  - addrlen is a pointer to an int, that should be initialized to sizeof(struct sockaddr).
  - The function returns -1 on error and sets errno accordingly.

- Here is an example of use of getpeername:

```
struct sockaddr_in name;
in namelen = sizeof(name);
.......
if (getpeername(sd, (struct sockaddr *)&name, &namelen) < 0) {
   perror("getpeername"); exit(1);}
printf("Connection from %s\n", inet_ntoa(name.sin_addr));
```

Socket Functions: getsockname

- **Used to determine the address to which a socket is bound.**

int getsockname(int sd, struct sockaddr *addrp, int *addrlen)

  - **sd: socket descriptor of a bound socket.**
  - **addrp: points to a buffer. After the call it will have the address associated to the socket.**
  - **addrlen: gives the size of the buffer. After the call gives size of address.**
  - **It returns an integer (0=success, -1=failure).**

Socket Functions: Gethostbyname

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

- **The function gethostbyname using DNS and given a host name, like beast.rit.edu, returns 0 in case of failure, or a pointer to a struct hostent object which gives information about the host names+aliases+IPaddresses:**

```
struct  hostent {
     char    *h_name;        /* official name of host */
     char    **h_aliases;    /* null terminated list of aliases*/
     int     h_addrtype;     /* host address type */
     int     h_length;       /* length of address structure */
     char    **h_addr_list;  /* null terminated list of addresses */
                             /* from name server */
 #define     h_addr  h_addr_list[0] /*address,for backward
compatibility*/};
```

  - **h_addr_list[0] is the first IP address associated with the host.**
- **In order to use this structure one must include:**

```
#include <netdb.h>
```

Stream Server Example: “Hello, World!” Server

- **All this server does is send the string "Hello, World!\n" out over a stream connection. All you need to do to test this server is run it in one window, and telnet to it from another with:**

$ telnet remotehostname 3490

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

#define MYPORT 3490    /* the port users will be connecting to */

#define BACKLOG 10     /* how many pending connections queue will hold */

main()
{
    int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr;    /* my address information */
    struct sockaddr_in their_addr; /* connector's address information */
    int sin_size;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET;         /* host byte order */
    my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
    my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
    bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */
```

Stream Server Example: “Hello, World!” Server (Continued)

```
if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) \
                                                                    == -1) {
        perror("bind");
        exit(1);
    }

    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

    while(1) {  /* main accept() loop */
        sin_size = sizeof(struct sockaddr_in);
        if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, \
                                                    &sin_size)) == -1) {
            perror("accept");
            continue;
        }
        printf("server: got connection from %s\n", \
                                            inet_ntoa(their_addr.sin_addr));
        if (!fork()) { /* this is the child process */
            if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd);  /* parent doesn't need this */

        while(waitpid(-1,NULL,WNOHANG) > 0); /* clean up child processes */
    }
}
```

A Simple Stream Client

- **The client connect to port 3490 of the host specified in the command and gets the string that the server sends.**

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

#define PORT 3490    /* the port client will be connecting to */

#define MAXDATASIZE 100 /* max number of bytes we can get at once */

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in their_addr; /* connector's address information */

    if (argc != 2) {
        fprintf(stderr,"usage: client hostname\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  /* get the host info */
        herror("gethostbyname");
        exit(1);
    }
```

A Simple Stream Client (continued)

```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }

        their_addr.sin_family = AF_INET;      /* host byte order */
        their_addr.sin_port = htons(PORT);    /* short, network byte order */
        their_addr.sin_addr = *((struct in_addr *)he->h_addr);
        bzero(&(their_addr.sin_zero), 8);     /* zero the rest of the struct */

        if (connect(sockfd, (struct sockaddr *)&their_addr, \
                                              sizeof(struct sockaddr)) == -1) {
            perror("connect");
            exit(1);
        }

        if ((numbytes=recv(sockfd, buf, MAXDATASIZE, 0)) == -1) {
            perror("recv");
            exit(1);
        }

        buf[numbytes] = '\0';

        printf("Received: %s",buf);

        close(sockfd);

        return 0;
    }
```