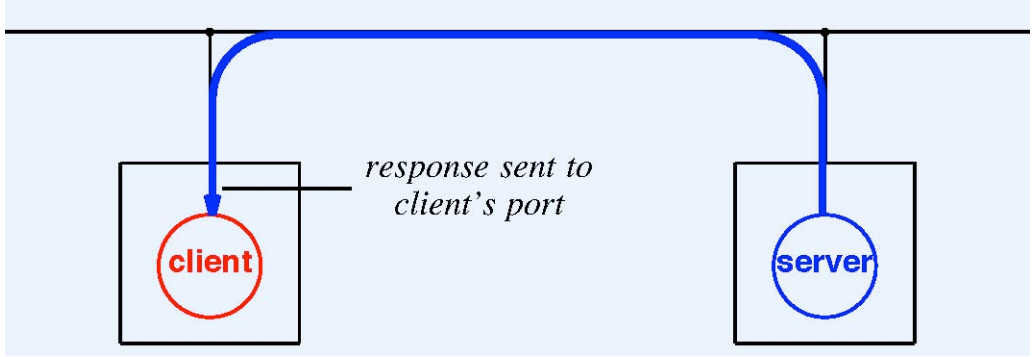
Client Server Interactions

Client Server (C/S) Paradigm

- Conceptual basis for virtually all distributed applications
- One program initiates interaction to which another program responds
- Note: “peer-to-peer” applications use client-server paradigm internally

C/S Paradigm


request sent to
well-known port
client
server


response sent to
client's port
client
server

Definition of C/S

- Client
  - Any application program
  - Contacts a server
  - Forms and sends a request
  - Awaits a response
- Server
  - Usually a specialized program that offers a service
  - Awaits a request
  - Computes an answer
  - Issues a response

Server Persistence (Always present)

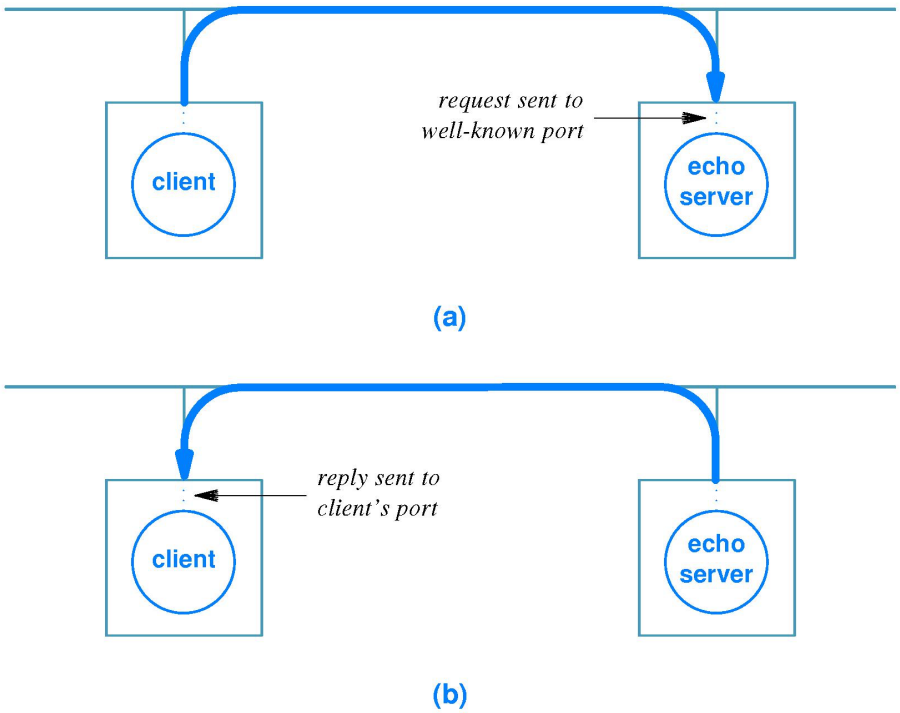
A server starts execution before interaction begins and (usually) continues to accept requests and send responses without ever terminating.

A client is any program that makes a request and awaits a response; it (usually) terminates after using a server a finite number of times.

Use of Ports

A server waits for requests at a well-known port that has been reserved for the service it offers. A client allocates an arbitrary, unused, nonreserved port for its communication.

C/S Model Example



The UDP echo service as an example of the client-server model. In (a), the client sends a request to the server's IP address and UDP port. In (b), the server returns a response.

Client Side

- Any application program can become a client
- Must know how to reach the server
  - Server's Internet address
  - Server's protocol port number d Usually easy to build

Server Side

- Finds client's location from incoming request
- Can be implemented with application program or in operating system
- Starts execution before requests arrive
- Must ensure client is authorized
- Must uphold protection rules
- Must handle multiple, concurrent requests
- Usually complex to design and build

Server Algorithm

- Open well-known port
- Wait for next client request
- Create a new socket for the client
- Create thread / process to handle request
- Continue with *wait* step

Server Complexity

- Servers are usually more difficult to build than clients because:
  - servers must enforce all the access and protection policies of the computer system on which they run
  - must protect themselves against all possible errors
  - Are generally concurrent (handle multiple requests)

Socket Programming

Using the TCP/IP model

Using Protocols

- Protocol software usually embedded in OS
- Applications run outside OS
- Need an *Application Program Interface* (*API*) to allow application to access protocols

API

- TCP/IP standards
  - Describe general functionality needed
  - Do not give details such as function names and arguments
- Each OS free to define its own API
- In practice: *socket interface* has become the de facto standard API

Socket API

- Defined by U.C. Berkeley as part of BSD Unix
- Adopted (with minor changes) by Microsoft as *Windows Sockets*

C/Cs of Socket API

- Follows Unix's open-read-write-close paradigm
- Uses Unix's *descriptor* abstraction
  - First, create a socket and receive an integer descriptor
  - Second, call a set of functions that specify all the details for the socket (descriptor is argument to each function)
- Once socket has been established, use *read* and *write* or equivalent functions to transfer data
- When finished, close the socket

Steps to Socket Programming

- Creating a Socket:
  - The analogy of creating a socket is that of requesting a telephone line from the phone company.
- Identifying/Naming a Socket:
  - The analogy is that of assigning a phone number to the line that you requested from the phone company in step 1 or that of assigning an address to a mailbox.
- Connecting from a Client to a Server
- Accepting Connections on a Server
- Communicate – Exchange Data
- Close the Connection

Creating a Socket

- A socket, s, is created with the socket system call:

  int s = socket(*domain, type, protocol*)

- All the parameters as well as the return value are integers:
  - *domain, or address family* —

    communication domain in which the socket should be created. Example of address families are AF_INET (IP), AF_INET6 (IPv6), AF_UNIX (local channel, similar to pipes), AF_ISO (ISO protocols).

  - *type* —

    type of service. This is selected according to the properties required by the application: SOCK_STREAM (virtual circuit service), SOCK_DGRAM (datagram service), SOCK_RAW (direct IP service). Check with your address family to see whether a particular service is available.

  - *protocol* —

    indicates a specific protocol to use for the sockets operation. This is useful in cases where some families may have more than one protocol to support a given type of service.

- The return value is a file descriptor (*s* is a small integer).

Demo Code: Creating a Socket

```
/*
    demo-01: create a TCP/IP socket
    usage:  demo-01
    create a socket. Doesn't do much
    */
#include <stdio.h>/* defines printf */
#include <stdlib.h>     /* defines exit and other sys calls */
#include <sys/socket.h>
```

Demo Contd. Returns fd: a file descriptor

```
main(int argc, char **argv)
{
        int fd;

        /* create a tcp/ip socket */
        /* request the Internet address protocol */
        /* and a reliable 2-way byte stream (TCP/IP) */

        if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                perror("cannot create socket");
                return 0;
        }

        printf("created socket: descriptor=%d\n", fd);
        exit(0);
}
```

Identify/Name a Socket

- When we talk about naming a socket, we are talking about assigning a transport address to the socket (a port number in IP networking). In sockets, this operation is called binding an address and the bind system call is used for this.

Assigning an Address

- You can explicitly assign an address or allow the system to assign one.
- The address is defined in a socket address structure.
- Applications find addresses of well-known services by looking up their names in a database (e.g., the file /etc/services. E.g., ftp 23).
- The system call for binding is:

int fx = bind(*s, addr, addrlen*)

- *s* is the socket descriptor obtained in step 1,
- *addr* is the address structure (*struct sockaddr **)
- *addrlen* is an integer containing the address length

Socket Address (*struct sockaddr **)

- The address family determines what variant of the sockaddr struct * to use that contains elements that make sense for that specific communication type.
- For IP networking, we use struct sockaddr_in, which is defined in the header netinet/in.h.
- This structure defines:

```
struct sockaddr_in {
        len                 sin_len;
        sa_family           sin_family;
        in_port             sin_port;
        struct in_addr      sin_addr;
        Char                sin_zero[8];
        };
```

Before calling bind, we need to fill out this structure.

The struct *sockaddr_in*

The three key parts we need to set are:

- *sin_family*

The address family we used when we set up the socket. In our case, it's AF_INET.

- *sin_port*

The port number (the transport address).

  - You can explicitly assign a transport address (port) or allow the operating system to assign one.
  - A client that won't be receiving incoming connections, lets the operating system pick any available port number by specifying port 0.
  - A server generally picks a specific number since clients will need to know a port number to connect to.

struct *sockaddr_in* contd.

sin_addr

The address for this socket - machine's IP address.

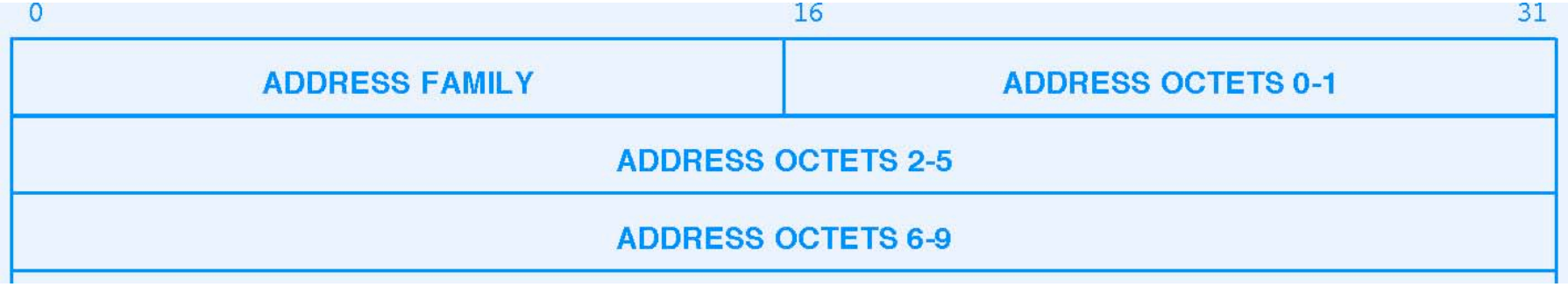
- For IP, your machine will have one IP address for each network interface.
- E.g., if your machine has both Wi-Fi and Ethernet connections, machine will have two addresses, one for each interface.
- Generally not important to specify a specific interface
- Let the operating system use whatever it wants.
- The special address for this is 0.0.0.0, defined by the symbolic constant INADDR_ANY.

sin_zero[8]

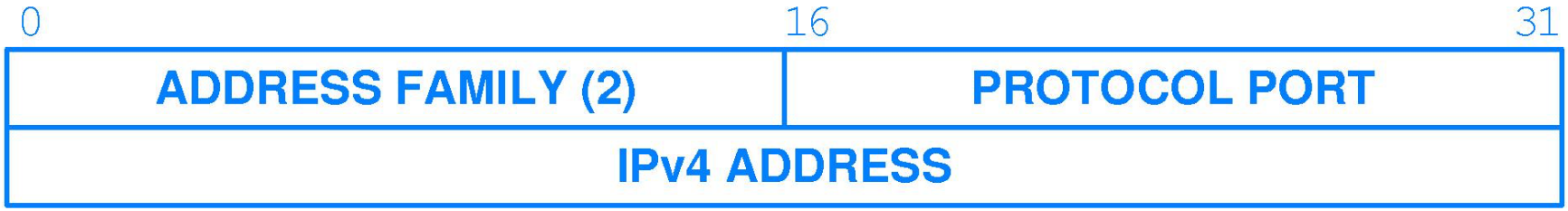
Specifies the length of that structure.

- This is simply sizeof(struct sockaddr_in).
- The address structure may differ based on the type of transport used

Format of a Sockaddr Structure



Sockaddr Structure when used with TCP/Ip



The *sockaddr_in* structure used when passing an IPv4 endpoint to a socket function.

Demo Code: Binding a Socket

```
/*
    demo-01: create a TCP/IP socket

    usage:  demo-01

    Create a socket. Bind it to any available port.
    Then use getsockname to print the port.
    */
#include <stdlib.h>     /* defines system calls */
#include <stdio.h>      /* needed for printf */
#include <string.h>     /* needed for memset */
#include <sys/socket.h>
#include <netinet/in.h>    /* needed for sockaddr_in */
```

Demo Contd.

```
main(int argc, char **argv)
{
    struct sockaddr_in myaddr;    /* our address */
    int fd;   /* our socket */
    unsigned int alen; /* length of address (for getsockname) */

    /* create a tcp/ip socket */
    /* request the Internet address protocol */
    /* and SPECIFY a reliable 2-way byte stream (TCP/IP) */

    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("cannot create socket");
        return 0;
    }
```

Demo Contd.

```
printf("created socket: descriptor = %d\n", fd);

    /* bind to an arbitrary return address */
    /* in this case, assume it's the client side, so we won't
    /* care about the port number as no application will connect here */
    /* INADDR_ANY is the IP address and 0 is the socket */
    /* htonl converts a long integer (e.g. address) to a network */
    /* representation (IP-standard byte ordering) */

    memset((void *)&myaddr, 0, sizeof(myaddr));
    myaddr.sin_family = AF_INET;
    myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    myaddr.sin_port = htons(0);
```

Dem Contd. - Printing

```
    if (bind(fd, (struct sockaddr *)&myaddr, sizeof(myaddr)) < 0) {
        perror("bind failed");
        return 0;
    }
    alen = sizeof(myaddr);
    if (getsockname(fd, (struct sockaddr *)&myaddr, &alen) < 0) {
        perror("getsockname failed");
        return 0;
    }
    printf("bind complete. Port number = %d\n",
ntohs(myaddr.sin_port));
    exit(0);
}
```

Connecting to a Socket (Client side)

- For connection-based communication, the client initiates a connection with the connect system call:

  int fx = connect(*s, serveraddr, serveraddrlen*)

Where:

s is the socket (type int)

serveraddr is a pointer to a structure containing the address of the server (struct *sockaddr **)

Serveraddrlen is a parameter containing the size of struct *sockaddr **. Since the structure may vary with different transports.

- For connectionless service (UDP), the operating system will send datagrams and maintain an association between the socket and the remote destination address.

Accepting a Connection (on Server)

- For connection-based communication (TCP), the server has to first state its willingness to accept connections. This is done with the listen system call:

  int fx = listen(s, backlog)

- The *backlog* is an integer specifying the upper bound on the number of pending connections that should be queued for acceptance.
- After a *listen*, the system is listening for connections to that socket.
- The connections can now be accepted with the *accept* system call, which extracts the first connection request on the queue of pending connections.
- It creates a new socket with the same properties as the listening socket and allocates a new file descriptor for it.
- By default, socket operations are synchronous (or blocking) and *accept* will block until a connection is present on the queue.

Accepting

- The syntax of *accept* is:

```
int s;
struct sockaddr *clientaddr;
int clientaddrlen = sizeof(struct sockaddr);

int snew = accept(s, clientaddr, &clientaddrlen);
```

- The *clientaddr* structure allows a server to obtain the client address.
- *accept* returns a *new* file descriptor *snew* that is associated with a *new* socket.
- The address length field initially contains the size of the address structure
- On return, it contains the actual size of the address.
- Communication takes place on this *new* socket.
- The original socket is used for managing a queue of connection requests (concurrent server - *listen* for other requests on the original socket *s*).

Exchanging Data

- Data can now be exchanged with the regular file system read and write system calls (referring to the socket descriptor).
- The *send/recv* calls are similar to *read/write* but support an extra flags parameter that lets one peek at incoming data and to send out-of-band data.
- The *sendto/recvfrom* system calls are like *send/recv* but also allow callers to specify or receive addresses of the peer with whom they are communicating (most useful for connectionless sockets).

Exchanging Data Contd.

- Summary:
  - read/write or send/recv calls must be used for connection-oriented communication
  - sendto/recvfrom or sendmsg/recvmsg must be used for connectionless communication.
- Note that:
  - With stream virtual circuit service (SOCK_STREAM) and with datagram service (SOCK_DGRAM) the other side may have to perform multiple reads to get results from a single write (because of fragmentation of packets)
  - or vice versa (a client may perform two writes and the server may read the data via a single read) depending on how receiving end is processing the data.

Close Connection

- The system call *close (s)* will shutdown a socket *s*.
- The *shutdown* system call may be used to stop all further read and write operations on a socket *s*:

  shutdown(s);
- The shutdown system call with "how".

```
#include <sys/socket.h>
int shutdown(int s, int how);
```

- Where the second parameter, *how*, allows us to tell the socket *s* what part of the full-duplex connection to shut down:
  - A value of SHUT_RD will disallow further receives on that socket.
  - A value of SHUT_WR will disallow further sends on that socket.
  - A value of SHUT_RDWR will disallow both further sends and receives on that socket.

Synchronous and Asynchronous

- Network communication (or file system access in general) system calls may operate in two modes:
  synchronous or asynchronous.
- In the synchronous mode, socket routines return only when the operation is complete. For example, *accept* returns only when a connection arrives.
- In the asynchronous mode, socket routines return immediately: system calls become non-blocking calls (e.g., *read* does not block).
- You can change the mode with the *fcntl* system call. For example:

```
fcntl(s, F_SETFF, FNDELAY);
```

sets the socket *s* to operate in asynchronous mode.

Byte Order Conversion Routines

- Convert between network byte order and local host byte order
- If local host uses big-endian, routines have no effect

```
localshort = ntohs(netshort)
locallong = ntohl(netlong)
netshort = htons(localshort)
netlong = htonl(locallong)
```

IP Address Manipulation

- Convert from dotted decimal (ASCII string) to 32-bit binary value

- Example:

```
address = inet_addr(string)
```

Example Client Program Part 1

Example Client Program

```
/* whoisclient.c -main */
#include <stdio.h> #include <sys/types.h> #include <sys/socket.h>
#include <netinet/in.h> #include <netdb.h>
/*-----------------------------------------------------------------
Program: whoisclient *
Purpose: UNIX application program that becomes a client for the
Internet "whois" service. *
Use: whois hostname username *
Author: Barry Shein, Boston University *
Date: Long ago in a universe far, far away *
*-----------------------------------------------------------------*/
```

Part 2

```
main(argc, argv)
int argc;                       /* standard UNIX argument declarations */
char *argv[];
{
    int s;                      /* socket descriptor */
    int len;                    /* length of received data */
    struct sockaddr_in sa;      /* Internet socket addr. structure */
    struct hostent *hp;         /* result of host name lookup */
    struct servent *sp;         /* result of service lookup */
    char buf[BUFSIZ+1];         /* buffer to read whois information */
    char *myname;               /* pointer to name of this program */
    char *host;                 /* pointer to remote host name */
    char *user;                 /* pointer to remote user name */
    myname = argv[0];
```

Part 3

```
/*  Check that there are two command line arguments */
if(argc != 3) {
        fprintf(stderr, "Usage: %s host username\n", myname);
        exit(1);
}
host = argv[1]; user = argv[2];

/*  Look up the specified hostname */
if((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr,"%s: %s: no such host?\n", myname, host);
exit(1);
}

/* Put host's address and address type into socket structure */
bcopy((char *)hp->h_addr, (char *)&sa.sin_addr, hp->h_length);
sa.sin_family = hp->h_addrtype;
```

Part 4

```
/* Look up the socket number for the WHOIS service */
if((sp = getservbyname("whois","tcp")) == NULL) {
	fprintf(stderr,"%s: No whois service on this host\n", myname);
	exit(1);
}

/* Put the whois socket number into the socket structure. */
sa.sin_port = sp->s_port;

/* Allocate an open socket */
if((s = socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0) {
	perror("socket");
	exit(1);
}
```

Part 5

```
    /* Connect to the remote server  */
    if(connect(s, &sa, sizeof sa) < 0) {
     perror("connect");
     exit(1);
    }

    /* Send the request*/
    if(write(s, user, strlen(user)) != strlen(user)) {
     fprintf(stderr, "%s: write error\n", myname);
     exit(1);
    }

    /* Read the reply and put to user's output */
    while( (len = read(s, buf, BUFSIZ)) > 0)
     write(1, buf, len);
    close(s);
    exit(0);
}
```

Example Server Program Part 1

```
/* whoisserver.c - main */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <pwd.h>
/*------------------------------------------------------------------
* Program: whoisserver
*
* Purpose: UNIX application program that acts as a server for
* the "whois" service on the local machine. It listens
* on well-known WHOIS port (43) and answers queries from
* clients. This program requires super-user privilege to
* run.
* Use: whois hostname username
```

Part 2

```
* Author: Barry Shein, Boston University
*
* Date: Long ago in a universe far, far away
*-----------------------------------------------------------------
*/
#define BACKLOG 5                /* # of requests we're willing to queue */
#define MAXHOSTNAME 32           /* maximum host name length we tolerate */
main(argc, argv)
int argc;                   /* standard UNIX argument declarations */
char *argv[];
{
int s, t;                   /* socket descriptors */
int i;                      /* general purpose integer */
struct sockaddr_in sa, isa;      /* Internet socket address structure */
struct hostent *hp;              /* result of host name lookup */
char *myname;                    /* pointer to name of this program */
struct servent *sp;              /* result of service lookup */
char localhost[MAXHOSTNAME+1];       /* local host name as character string */
```

Part 3

```
myname = argv[0];

/* Look up the WHOIS service entry */
if((sp = getservbyname("whois","tcp")) == NULL) {
	fprintf(stderr, "%s: No whois service on this host\n", myname);
	exit(1);
}

/*Get our own host information */
gethostname(localhost, MAXHOSTNAME);
if((hp = gethostbyname(localhost)) == NULL) {
	fprintf(stderr, "%s: cannot get local host info?\n", myname);
	exit(1);
}
```

Part 4

```
/*Put the WHOIS socket number and our address info into the socket
structure */
sa.sin_port = sp->s_port;
bcopy((char *)hp->h_addr, (char *)&sa.sin_addr, hp->h_length);
sa.sin_family = hp->h_addrtype;

/* Allocate an open socket for incoming connections */
if((s = socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0) {
	perror("socket");
	exit(1);
}

/* Bind the socket to the service port so we hear incoming connections */
if(bind(s, &sa, sizeof sa) < 0) {
perror("bind");
exit(1);
}
```

Part 5

```
/* Set maximum connections we will fall behind */
listen(s, BACKLOG);

/* Go into an infinite loop waiting for new connections */
while(1) {
      i = sizeof isa;

      /* We hang in accept() while waiting for new customers */
      if((t = accept(s, &isa, &i)) < 0) {
      perror("accept");
      exit(1);
      }
      whois(t); /* perform the actual WHOIS service */
      close(t);
  }
}
```

Part 6

```
/* Get the WHOIS request from remote host and format a
reply. */
whois(sock)
int sock;
{
    struct passwd *p;
    char buf[BUFSIZ+1];
    int i;

/* Get one line request */
if( (i = read(sock, buf, BUFSIZ)) <= 0)
     return;
buf[i] = '\0';                /* Null terminate */
```

Part 7

```
/* Look up the requested user and format reply */
if((p = getpwnam(buf)) == NULL)
    strcpy(buf,"User not found\n");
else
    sprintf(buf, "%s: %s\n", p->pw_name, p->pw_gecos);

/*Return reply */
write(sock, buf, strlen(buf));
return;
}
```