

# Client-Server Communications

- Servers provide services to clients over a network
- Most LANs have file servers to manage common disk space
- This makes it easier to share files and perform backups
- Mail and ftp use client-server paradigm

# Universal Internet Communication Interface (UICI)

- Simplifies client-server programming
- UICI is implemented in this chapter with Sockets

# Network Applications

- ftp – file transfer
- Kerberos – authentication
- telnet – remote login
- NFS – remote filesystems

# Connectionless Protocol

- The client sends a single message to the server
- The server performs a service and sends a reply

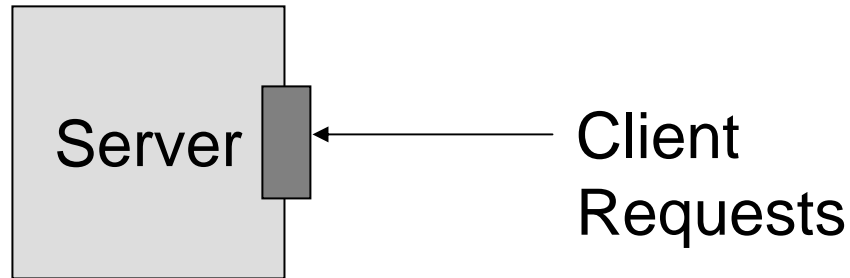
# Connection-Oriented Protocol

- Server waits for a connection request from a client
- Once the connection is established, communication takes place using handles (file descriptors)
- The server address is not included in the user message
- Connection-oriented protocol has setup overhead
- This chapter emphasizes connection-oriented protocol

# Naming of Servers

- Option 1 – Designate server by process ID and host ID
  - Process ID is assigned chronologically at the time process starts running
  - It is difficult to know process ID in advance on a host
- Option2 – Name servers with small integers called ports
  - Server “listens” at a well-known port that has been designated in advance for a particular service
  - Client explicitly specifies a host address and a port number on the host when setting up a connection

# Single Port Strategy

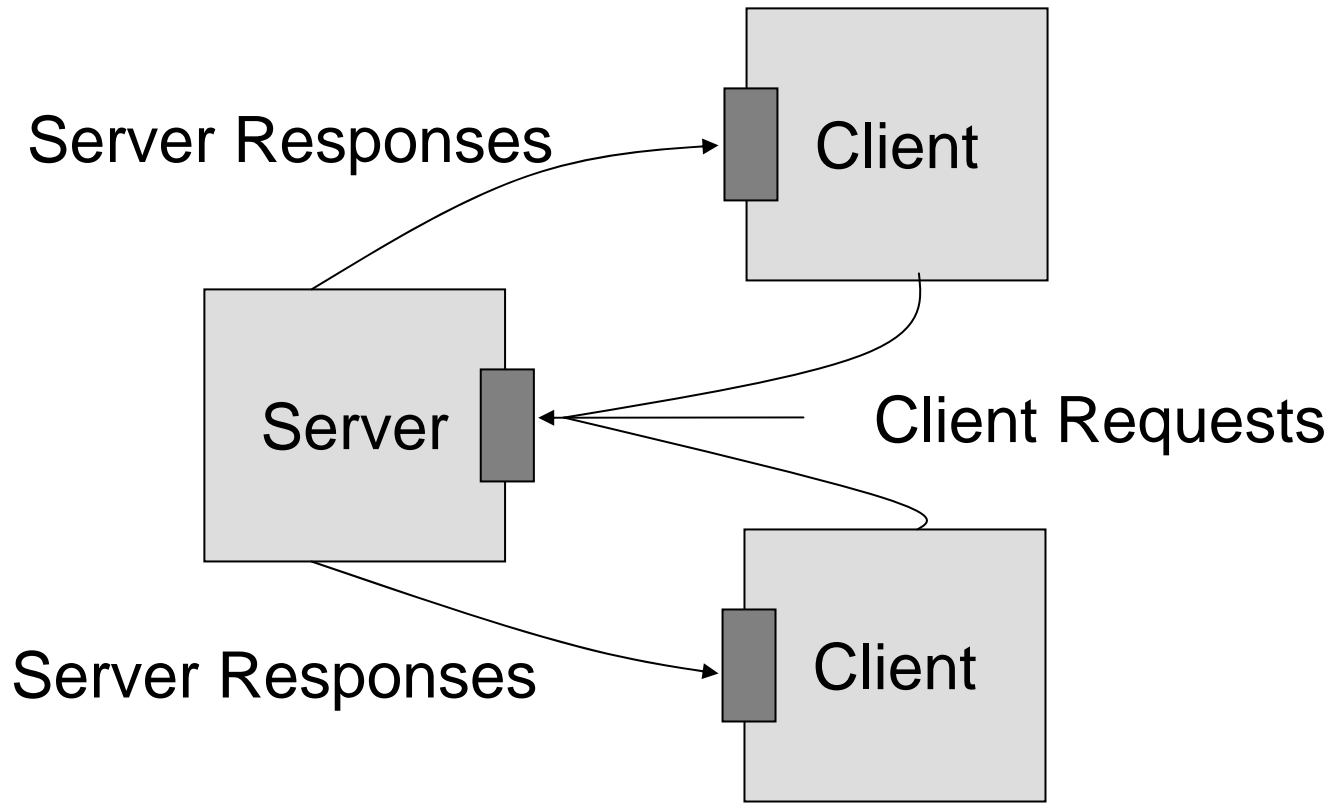


- Simplest client-server communication takes place over a single communication port
- If client and server are on the same machine, the single port can be a FIFO
- On a network port can be socket or TLI connection
- When server starts up, it opens FIFO (or Socket) and waits for client requests
- When client needs service, it opens FIFO (or Socket) and writes its request
- Server then performs the service

# Single Port Problems

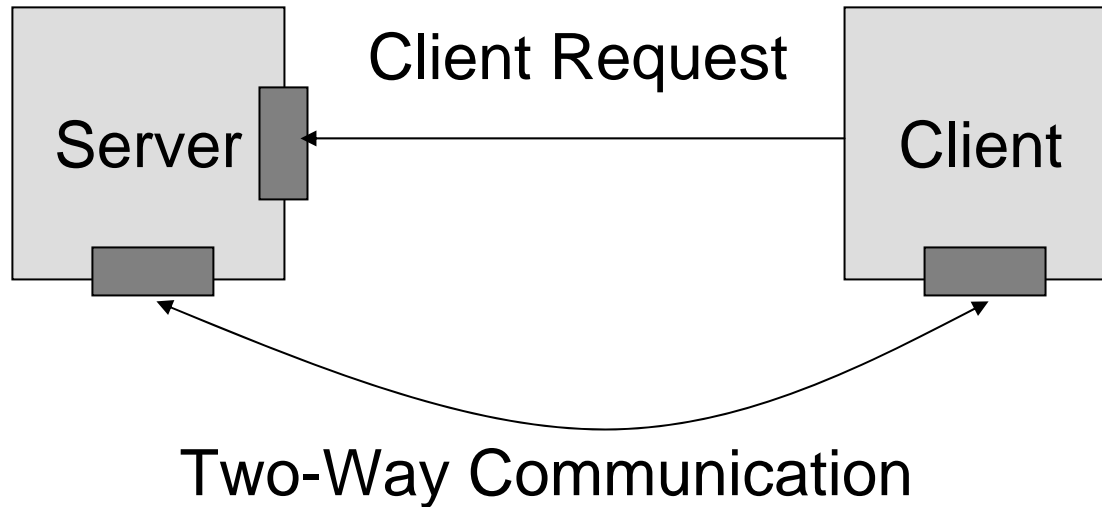
- There is no mechanism for the server to respond to the client
- Since items are removed from the port when they are read, there is nothing to prevent one client from taking another's request

# Connectionless Protocol



- Each client has its own channel for receiving responses from the server
- *sendto* and *receivefrom* form the basis of connectionless protocol

# Serial-Server Strategy

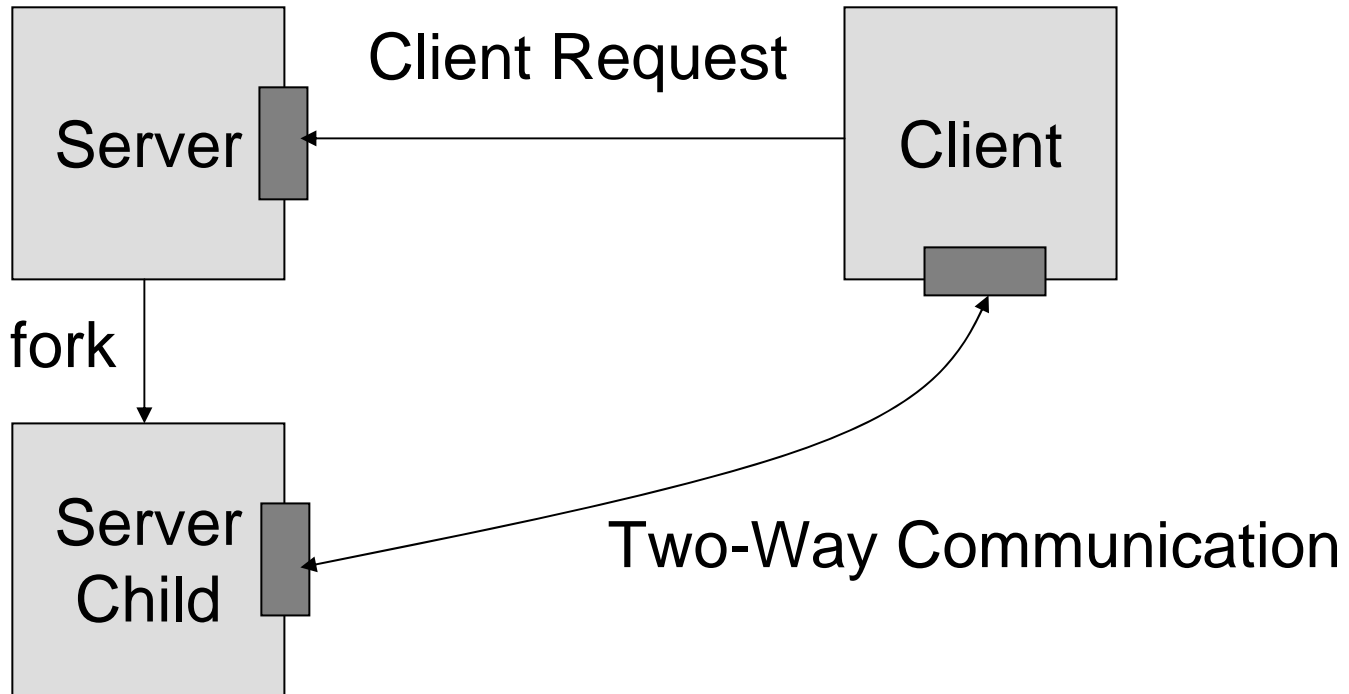


- Two way channel is for additional interaction between client and server
- Initial client request serves to establish two way communication channel
- Communication channel is private – not accessible to other processes
- A busy server handling long-lived requests cannot use serial server strategy

# Serial-Server Pseudocode

```
for (;;) {  
    listen for client request;  
    create private two-way communication channel;  
    while (no error on communication channel)  
        read client request;  
        handle request and respond to client;  
    close communication channel  
}
```

# Parent-Server Strategy

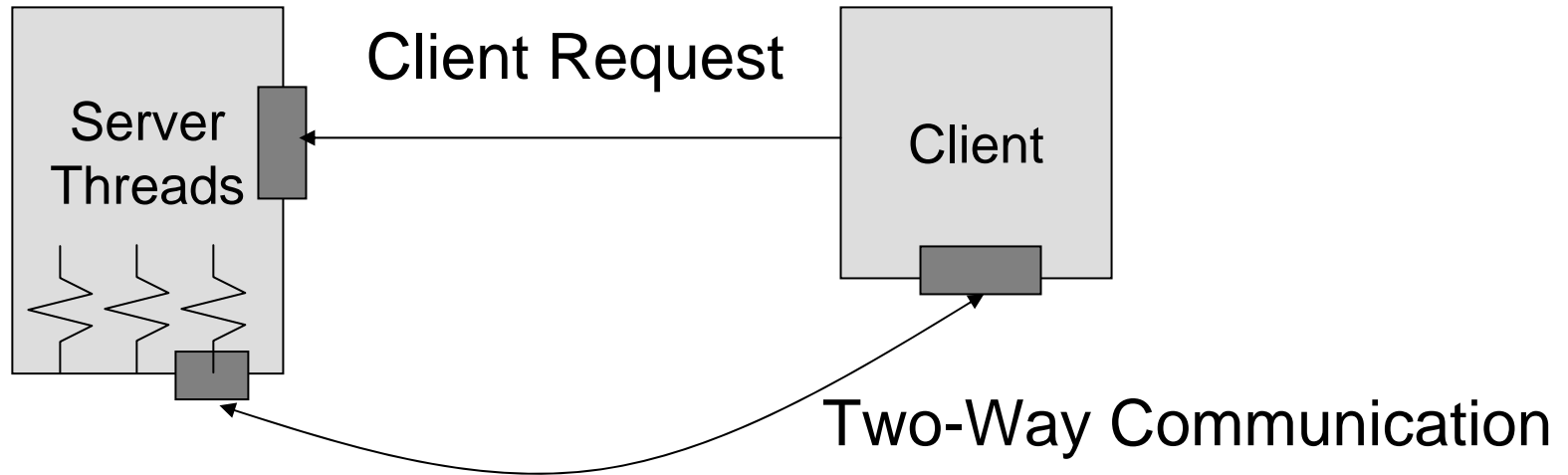


- Server forks a child to handle actual service
- Server resumes listening for more requests
- Similar to switchboard at hotel
- Server can accept multiple client requests

# Parent-Server Pseudocode

```
for (;;) {  
    listen for client request;  
    create a private two-way communication channel;  
    fork a child to handle the request;  
    close the communication channel;  
    clean up zombies;  
}
```

# Thread-Server Strategy



- Low overhead alternative to parent-server strategy
- Create thread to handle request instead of forking child – threads have less overhead
- Efficient if request is small or I/O intensive
- Possible interference among multiple requests due to shared address space
- For computationally intensive services, additional threads may reduce efficiency of or block the main server thread
- Requires good kernel-level parallelism

# UICI Summary

UICI Prototype	Description
<code>int u_open(u_port_t port)</code>	Opens file descriptor bound to port. Returns listening fd
<code>int u_listen(int fd, char *hostn)</code>	Listens for connection request on fd. Returns communication fd
<code>int u_connect(u_port_t port, char *the_host)</code>	Initiates connection to server on port <i>port</i> and host <i>the_host</i> . Returns communication fd
<code>int u_close(fd)</code>	Closes file descriptor fd
<code>ssize_t u_read(int fd, char *buf, size_t nbyte)</code>	Reads up to <i>nbyte</i> bytes from fd into <i>buf</i> . Returns number of bytes actually read
<code>ssize_t u_write(int fd, char *buf, size_t nbyte)</code>	Writes <i>nbyte</i> bytes from <i>buf</i> to fd. Returns number of bytes written
<code>void u_error(char *errmsg)</code>	Outputs <i>errmsg</i> followed by a UICI error message
<code>int u_sync(int fd)</code>	Updates relevant kernel info after calls to <i>exec</i>

# copy\_from\_network\_to\_file

```
int copy_from_network_to_file(int communfd, int filefd) {  
...  
for ( ; ; ) {  
    if ((bytes_read = u_read(communfd, buf, BLKSIZE)) < 0) {  
        u_error("Server read error");  
        break;  
    } else if (bytes_read == 0) {  
        fprintf(stderr, "Network end-of-file\n");  
        break;  
    } else { /* allow for interruption of write by signal */  
        for (bufp = buf, bytes_to_write = bytes_read;  
            bytes_to_write > 0;  
            bufp += bytes_written, bytes_to_write -= bytes_written){  
            bytes_written = write(filefd, bufp, bytes_to_write);  
            if ((bytes_written) == -1 && (errno != EINTR)){  
                perror("Server write error");  
                break;  
            } else if (bytes_written == -1)  
                bytes_written = 0;  
            total_bytes += bytes_written; }  
        if (bytes_written == -1)  
            break; } }  
return total_bytes; }
```

# Serial Server

```
void main(int argc, char *argv[]) {
    u_port_t portnumber;
    int listenfd;
    int communfd;
    char client[MAX_CANON];
    int bytes_copied;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s port\n", argv[0]);
        exit(1); }
    portnumber = (u_port_t) atoi(argv[1]);
    if ((listenfd = u_open(portnumber)) == -1) {
        u_error("Unable to establish a port connection");
        exit(1); }
    while ((communfd = u_listen(listenfd, client)) != -1) {
        fprintf(stderr, "A connection has been made to %s\n", client);
        bytes_copied = copy_from_network_to_file(communfd, STDOUT_FILENO);
        fprintf(stderr, "Bytes transferred = %d\n", bytes_copied);
        u_close(communfd); }
    exit(0); }
```

# Parent-Server (top)

```
void main(int argc, char *argv[])
{
    u_port_t portnumber;
    int listenfd;
    int communfd;
    char client[MAX_CANON];
    int bytes_copied;
    int child;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s port\n", argv[0]);
        exit(1);
    }

    portnumber = (u_port_t) atoi(argv[1]);
    if ((listenfd = u_open(portnumber)) == -1) {
        u_error("Unable to establish a port connection");
        exit(1);
    }
}
```

```

while ((communfd = u_listen(listenfd, client)) != -1) {
    fprintf(stderr, "[%ld]: A connection has been made to %s\n",
        (long) getpid(), client);
    if ((child = fork()) == -1) {
        fprintf(stderr, "Could not fork a child\n");
        break; }
    if (child == 0) { /* child code */
        close(listenfd);
        fprintf(stderr, "[%ld]: A connection has been made to %s\n",
            (long) getpid(), client);
        bytes_copied =
            copy_from_network_to_file(communfd, STDOUT_FILENO);
        close(communfd);
        fprintf(stderr, "[%ld]: Bytes transferred = %d\n",
            (long) getpid(), bytes_copied);
        exit(0);
    } else { /* parent code */
        u_close(communfd);
        while (waitpid(-1, NULL, WNOHANG) > 0) ; }
}
exit(0); }

```

# Parent-Server (bottom)

# Client (top)

```
void main(int argc, char *argv[])
{
    u_port_t portnumber;
    int communfd;
    ssize_t bytesread;
    char buf[BLKSIZE];

    if (argc != 3) {
        fprintf(stderr, "Usage: %s host port\n", argv[0]);
        exit(1);
    }
    portnumber = (u_port_t)atoi(argv[2]);
    if ((communfd = u_connect(portnumber, argv[1])) < 0) {
        u_error("Unable to establish an Internet connection");
        exit(1);
    }
}
```

# Client (bottom)

```
fprintf(stderr, "A connection has been made to %s\n", argv[1]);
for ( ; ; ) {
    if ((bytesread = read(STDIN_FILENO, buf, BLKSIZE)) < 0) {
        perror("Client read error");
        break;
    } else if (bytesread == 0) {
        fprintf(stderr, "Client detected end-of-file on input\n");
        break;
    } else if (bytesread !=
                u_write(communfd, buf, (size_t)bytesread)) {
        u_error("Client write_error");
        break;
    }
}
u_close(communfd);
exit(0);
}
```

# UICI Implementation

UICI	Sockets	TLI	Streams
u_open	socket bind listen	t_open t_bind	create pipe push connd fattach
u_listen	accept	t_alloc t_listen t_bind t_accept	ioctl of I_RECVFD
u_connect	socket connect	t_open t_bind t_alloc t_connect	open
u_read	read	t_rcv	read
u_write	write	t_snd	write
u_sync		t_sync	

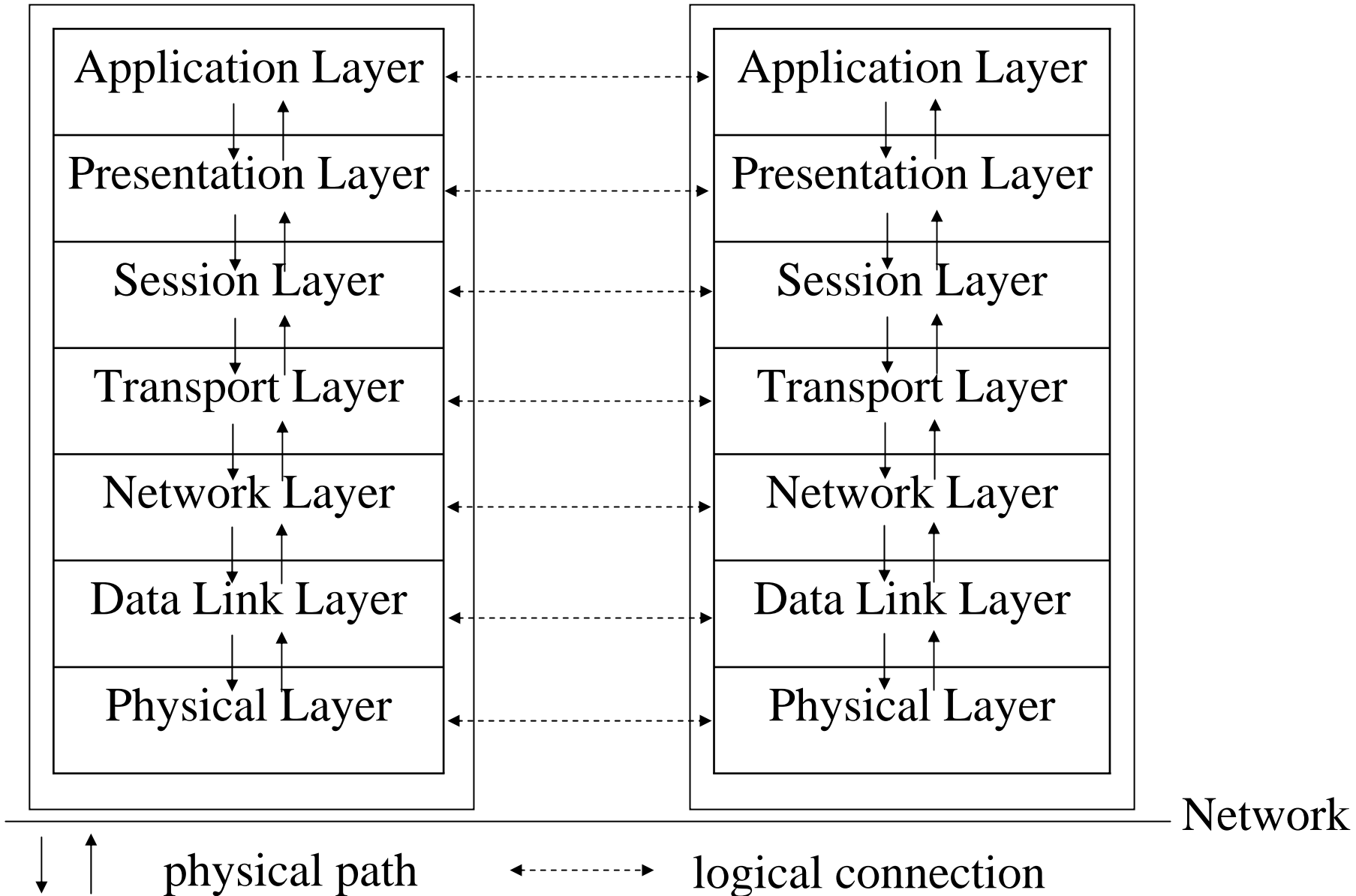
# u\_read an u\_write

- All implementations allow for non-blocking I/O, but UICI interface allows only blocking I/O
- u\_read and u\_write cause caller to block
- u\_read blocks until information is available from network connection
- u\_write blocks on a write, on message delivery problems in lower protocol layers, or when all buffers for the network protocols are full
- u\_write returns when output has been transferred – it does not mean the message has been delivered

# Open Systems Interconnection (OSI)

- OSI is International Standards Organization (ISO) standard for network design
- OSI model consists of 7 protocol layers
- Each layer consists of a set of functions to handle a particular aspect of network communication
- Functions in a particular layer communicate only with layers directly above and below
- Low layers are closely related to hardware, high layers are closely related with the user interface

# Peer-to-Peer OSI Model



# Physical/Data-Link Layers

- Concerned with point-to-point transmission of data
- Ethernet is common low-cost implementation of these layers
- Each host on network has a hardware Ethernet adapter that is connected to a communication link consisting of coaxial cable or twisted pair wire
- Host is identified by 6-byte Ethernet address that is hard-wired into adapter
- Other common implementations of this layer are: token ring, token bus, ISDN, ATM, and FDDI

# Network Layer

- Handles network addressing and routing through bridges and router interconnecting networks
- Most common network layer protocol used by UNIX is called IP – Internet Protocol
- Every host has one or more IP addresses consisting of 4 bytes
- Common to separate bytes with decimal points (e.g., 198.86.40.81)
- Users typically refer to machine by name (e.g., sunsite.unc.edu)
- Convert from name to address with *gethostbyname* and convert from address to name with *gethostbyaddr*

# Transport Layer

- Handles end-to-end communication between hosts
- Two main protocols at this layer are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP)
- UNIX networks use both TCP and UDP
- UDP is connectionless protocol without guarantee of delivery
- TCP is reliable connection-oriented protocol

# Transport Layer Ports

- More than one user at a time may be using TCP or UDP between a given pair of machines
- To distinguish between various communicating processes, they use 16 bit integers called ports
- Well-known addresses
  - telnet – 23
  - tftp – 69
  - finger – 79
  - User processes – above 7,000 so as not to interfere with system services and X

# Session Layer

- Session layer interfaces with the transport layer
- Two standard interfaces are sockets and Transportation Layer Interface (TLI)

# Presentation/Application Layers

- Consists of general utilities and applications
- Presentation layer may handle compression or encryption

# OSI Drawbacks

- Many networks were established before OSI reference model was accepted
- Therefore, the organization of some networks does not fit the model
- It is still a valuable frame of reference

# Sockets

- First socket implementation was in 4.1cBSD UNIX in early 80s
- Compile socket programs with library options
  - lsocket and -lnsl
- All socket system calls return -1 on failure and set errno

# Socket Server

- *socket* – Creates a handle (file descriptor)
- *bind* – Associates handle with a physical location (port) on the network
- *listen* – Sets up a queue size for pending requests
- *accept* – Listens for client requests

# Socket Client

- *socket* – Creates a handle (file descriptor)
- *connect* – Associates handle with the location (port) of the network server

# Socket Communications

- The client and server handles are sometimes called *transmission endpoints*
- Once connection is established, client and server can communicate using ordinary read and write calls

# socket

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

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

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

- *domain* – selects protocol family to be used
  - AF\_UNIX* – Used on a single UNIX system
  - AF\_INET* – Used on internet and between remote hosts
- *type* –
  - SOCK\_STREAM* – Reliable 2-way connection oriented typically implemented with TCP
  - SOCK\_DGRAM* – Connectionless communication using unreliable messages of fixed length, typically implemented with UDP
- *Protocol* – Specifies protocol to be used – There is usually only one type so it is usually 0

# bind

## SYNOPSIS

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

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

```
int bind (int s, const struct sockaddr *address, size_t  
address_len);
```

- *s* – File descriptor returned by the socket call
- *address\_len* – Number of bytes returned in the \*address structure
- \**address* – Contains family name and protocol-specific info

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

# bind (continued)

- Bind associates a socket endpoint or handle with a specific network connection
- Internet domain protocols specify the physical connection by a port number
- UNIX domain protocols specify the connection by a pathname

# struct sockaddr

Contains family name and protocol  
specific  
information

# struct sockaddr\_in

- The internet domain uses struct sockaddr\_in for struct sockaddr
- *sin\_family* – AF\_INET
- *sin\_port* – is the port number using the network byte ordering
- *sin\_addr* – INADDR\_ANY allows communication from any host
- *sin\_zero* – an array that fills out the structure so that it has the same size as struct sockaddr

# listen

## SYNOPSIS

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

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

```
int listen (int s, int backlog);
```

- *s* – File descriptor returned by socket
- *backlog* – number of pending client requests allowed

# sin\_port field

- Represents the port using network byte ordering
- Machines that use a different byte ordering must do a conversion
- The macro host to network short (htons) can be used to convert port numbers
- htons should be used even when not necessary to maintain portability

# u\_open

- The combination of socket, bind, and listen establishes a handle to monitor communication requests from a well-known port
- If an attempt is made to write to a pipe or socket that no process has open for read, the write generates a SIGPIPE signal and sets errno to EPIPE
- The default action of SIGPIPE is to terminate the process – This prevents graceful shutdown
- Socket implementation of UICI ignores SIGPIPE if default handler is active

# u\_open socket Implementation

```
int u_open(u_port_t port)
{ int sock;
  struct sockaddr_in server;
  if (( u_ignore_sigpipe() != 0) ||
      ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0))
    return -1;
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = INADDR_ANY;
  server.sin_port = htons((short)port);
  if((bind(sock, (struct sockaddr *)&server, sizeof(server)) < 0) ||
      (listen(sock, MAXBACKLOG < 0)))
    return -1;
  return sock; }
```

# accept

## SYNOPSIS

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

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

```
int accept(int s, struct sockaddr *address, int *address_len);
```

- Parameters are similar to bind, except that accept fills in \*address with info about the client making the connection
- *sin\_addr* holds Internet address of client
- \**address\_len* – contains number of bytes of the buffer actually filled in by the accept call
- accept returns file descriptor for communicating with client
- In parent-server model, server forks a child to handle the request and resumes monitoring the file descriptor
- Convert *sin\_addr* to a name by calling gethostbyaddr

# gethostbyaddr

## SYNOPSIS

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

```
struct hostent *gethostbyaddr(const void *addr, size_t len, int  
    type);
```

- struct hostent includes a field h\_name that is a pointer to the official host name
- On error, gethostbyaddr returns NULL and sets the external integer h\_error
- gethostbyaddr is not thread safe, but gethostbyaddr\_r is

# u\_listen socket Implementation

```
int u_listen(int fd, char *hostn)
{ struct sockaddr_in net_client;
  int len = sizeof(struct sockaddr);
  int retval;
  struct hostent *hostptr;
  while ( ((retval =
           accept(fd, (struct sockaddr *)&net_client, &len)) == -1) &&
          (errno == EINTR) );
  if (retval == -1)
    return retval;
  hostptr =
    gethostbyaddr((char *)&(net_client.sin_addr.s_addr), 4, AF_INET);
  if (hostptr == NULL)
    strcpy(hostn, "unknown");
  else
    strcpy(hostn, (*hostptr).h_name);
  return retval; }
```

# connect

## SYNOPSIS

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

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

```
int connect (int s, struct sockaddr *address, size_t  
    adres_len);
```

- Establishes a link between file descriptor `s` and a well-known port on the remote server
- The `sockaddr_in` structure is filled in as it is with `bind`

# gethostbyname

## SYNOPSIS

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

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

- The host name string is converted to an appropriate Internet address by calling `gethostbyname`
- `struct hostent` includes two members of interest:
  - `h_addr_list` is an array of pointers to network addresses used by this host – use the first one `h_addr_list[0]`
  - `h_length` is filled with the number of bytes in the address
- On error `gethostbyname` returns `NULL` and sets the external integer `h_error` – macros exist to determine error
- `gethostbyname_r` is thread safe

# u\_connect socket Implementation (top)

```
int u_connect(u_port_t port, char *hostn)
{ struct sockaddr_in server;
  struct hostent *hp;
  int sock;
  int retval;
  if ( (u_ignore_sigpipe() != 0) ||
        !(hp = gethostbyname(hostn)) ||
        ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) )
    return -1;
  memcpy((char *)&server.sin_addr, hp->h_addr_list[0],
         hp->h_length);
  server.sin_port = htons((short)port);
  server.sin_family = AF_INET;
```

# u\_connect Implementation (bottom)

```
while ( ((retval =
        connect(sock, (struct sockaddr *)&server, sizeof(server))) == -1)
        && (errno == EINTR) );
if (retval == -1) {
    close(sock);
    return -1;
}
return sock; }
```

- Client creates a socket and makes the connection request
- Client can be interrupted by a signal and loop reinitiates the call
- Program does not use strncpy for server.sin\_addr since the source may have an embedded zero byte
- Once client and server establish a connection, they exchange information using u\_read (read) and u\_write (write)