Network Programming

Compiled By:

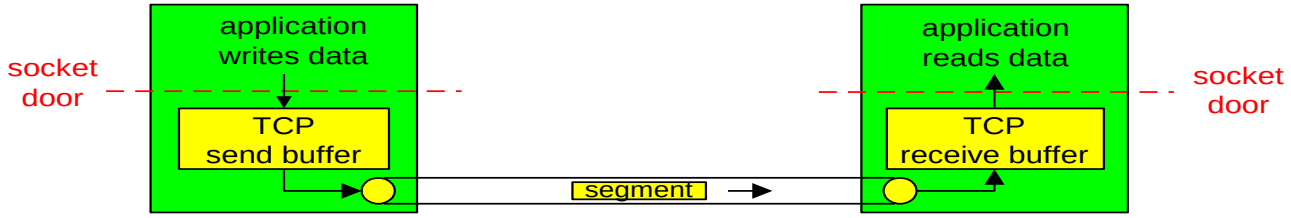
Madan Kadariya

NCIT

Transmission Control Protocol (TCP)

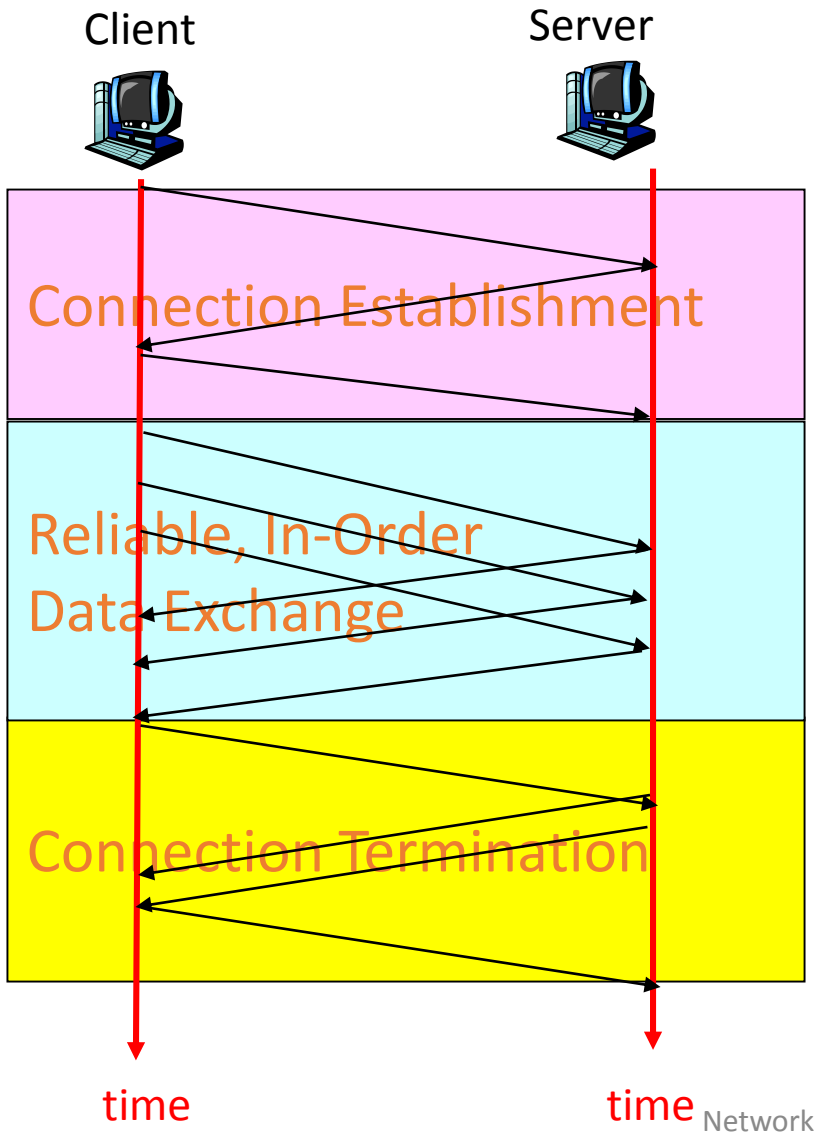
TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581



- point-to-point (unicast):
  - one sender, one receiver
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
  - State resides only at the END systems – Not a virtual circuit!
- full duplex data:
  - bi-directional data flow in same connection (A->B & B->A in the same connection)
  - MSS: maximum segment size
- reliable, in-order *byte steam:*
  - no "message boundaries"
- send & receive buffers
  - buffer incoming & outgoing data
- flow controlled:
  - sender will not overwhelm receiver
- congestion controlled:
  - sender will not overwhelm network

Typical TCP Transaction



- A TCP Transaction consists of 3 Phases
  1. Connection Establishment
     - Handshaking between client and server
  2. Reliable, In-Order Data Exchange
     - Recover any lost data through retransmissions and ACKs
  3. Connection Termination
     - Closing the connection

Protocol Comparison

Services/Features	SCTP	TCP	UDP
Connection-oriented	yes	yes	no
Full duplex	yes	yes	yes
Reliable data transfer	yes	yes	no
Partial-reliable data transfer	optional	no	no
Ordered data delivery	yes	yes	no
Unordered data delivery	yes	no	yes
Flow control	yes	yes	no
Congestion control	yes	yes	no
ECN capable	yes	yes	no
Selective ACKs	yes	optional	no
Preservation of message boundaries	yes	no	yes
Path MTU discovery	yes	yes	no
Application PDU fragmentation	yes	yes	no
Application PDU bundling	yes	yes	no
Multistreaming	yes	no	no
Multihoming	yes	no	no
Protection against SYN flooding attacks	yes	no	n/a
Allows half-closed connections	no	yes	n/a
Reachability check	yes	yes	no
Psuedo-header for checksum	no (uses vtags)	yes	yes
Time wait state	for vtags	for 4-tuple	n/a

TCP Connection Establishment

The following scenario occurs when a TCP connection is established.

1. The server must be prepared to accept an incoming connection. This is normally done by calling **socket, bind**, and **listen** and is called a passive open.
2. The client issues an active open by calling **connect**. This causes the client TCP to send a "**synchronize**" (**SYN**) segment, which tells the server the client's initial sequence number for the data that the client will send on the connection. Normally, there is no data sent with the **SYN**; it just contains an IP header, a TCP header, and possible TCP options.
3. The server must acknowledge (**ACK**) the client's **SYN** and the server must also send its own **SYN** containing the initial sequence number for the data that the server will send on the connection. The server sends its **SYN** and the **ACK** of the client's **SYN** is a single segment.
4. The client must acknowledge the server's SYN.

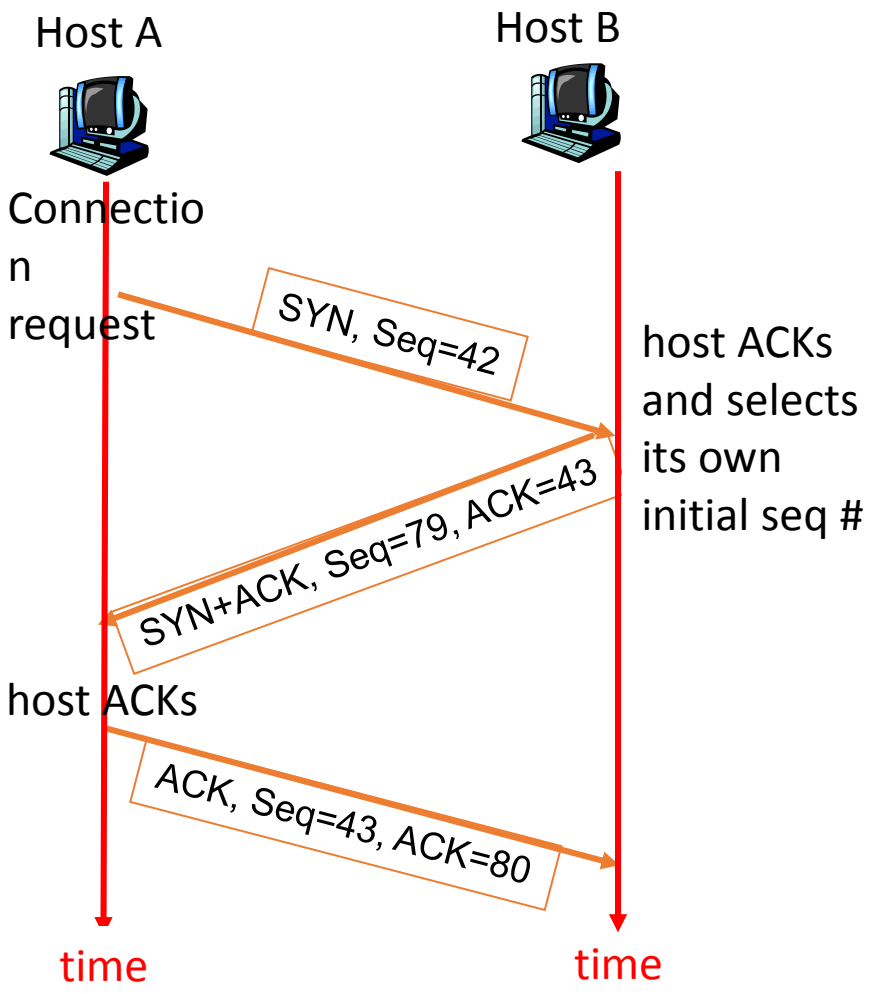
Connection Establishment (cont)

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies a random initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data



Three-way handshake

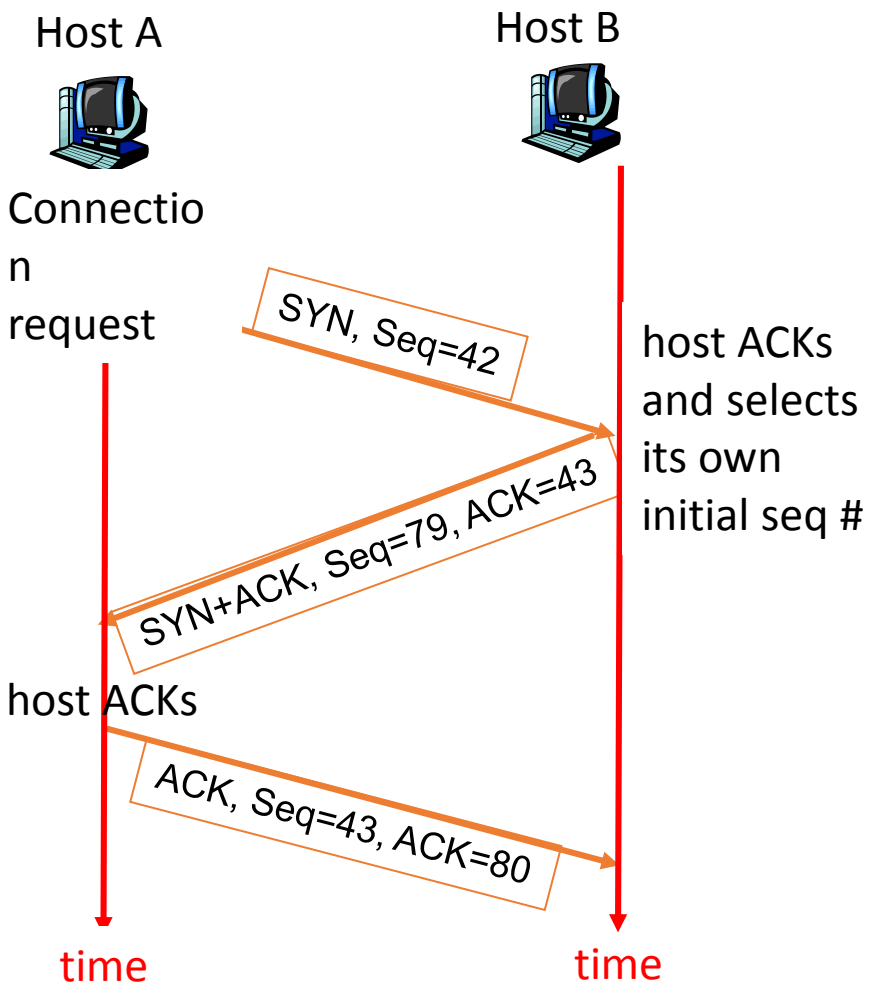
Connection Establishment (cont)

Seq. #'s:

- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK



Three-way handshake

TCP Starting Sequence Number Selection

- Why a random starting sequence #? Why not simply choose 0?
  - To protect against two incarnations of the same connection reusing the same sequence numbers too soon
  - That is, while there is still a chance that a segment from an earlier incarnation of a connection will interfere with a later incarnation of the connection
- How?
  - Client machine seq #0, initiates connection to server with seq #0.
  - Client sends one byte and client machine crashes
  - Client reboots and initiates connection again
  - Server thinks new incarnation is the same as old connection

TCP Connection Termination

1. One application calls close first, and we say that this end performs the active close. This end's TCP sends a FIN segment, which means it is finished sending data.
2. The other end that receives the FIN performs the passive close. The received FIN is acknowledged by TCP. The receipt of the FIN is also passed to the application as an end-of-file, since the receipt of the FIN means the application will not receive any additional data on the connection.
3. Sometime later, the application that received the end-of-file will close its socket. This causes its TCP to send a FIN.
4. The TCP on the system that receives this final FIN (the end that did the active close) acknowledges the FIN.

TCP Connection Termination

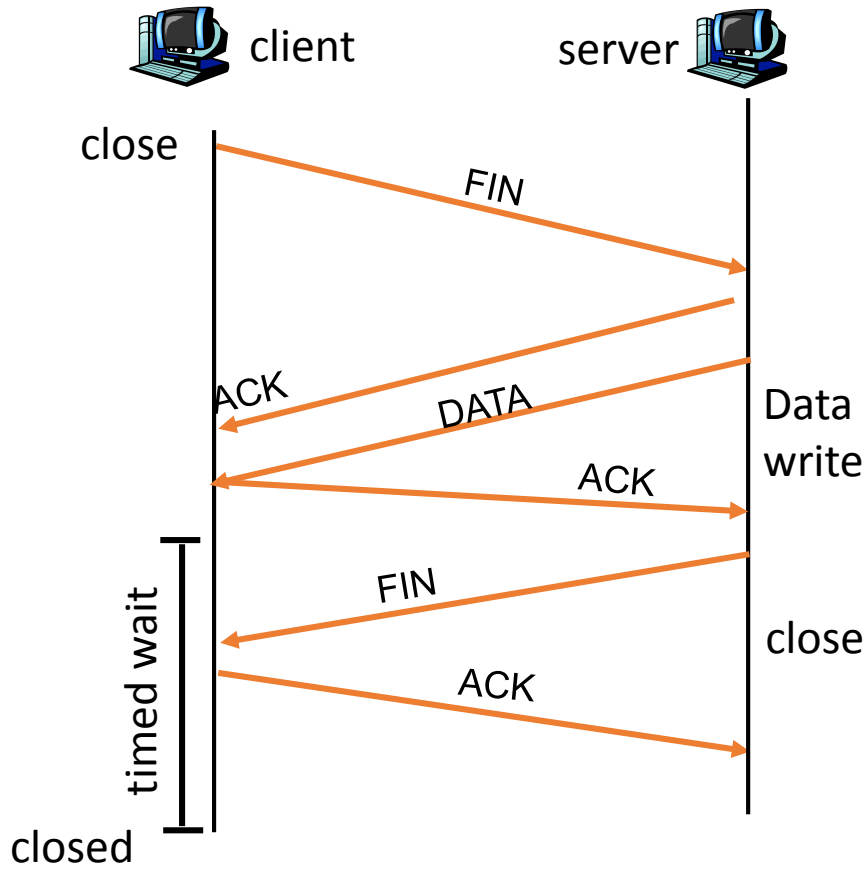
Closing a connection:

client closes socket:

```
clientSocket.close();
```

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Server might send some buffered but not sent data before closing the connection. Server then sends FIN and moves to Closing state.



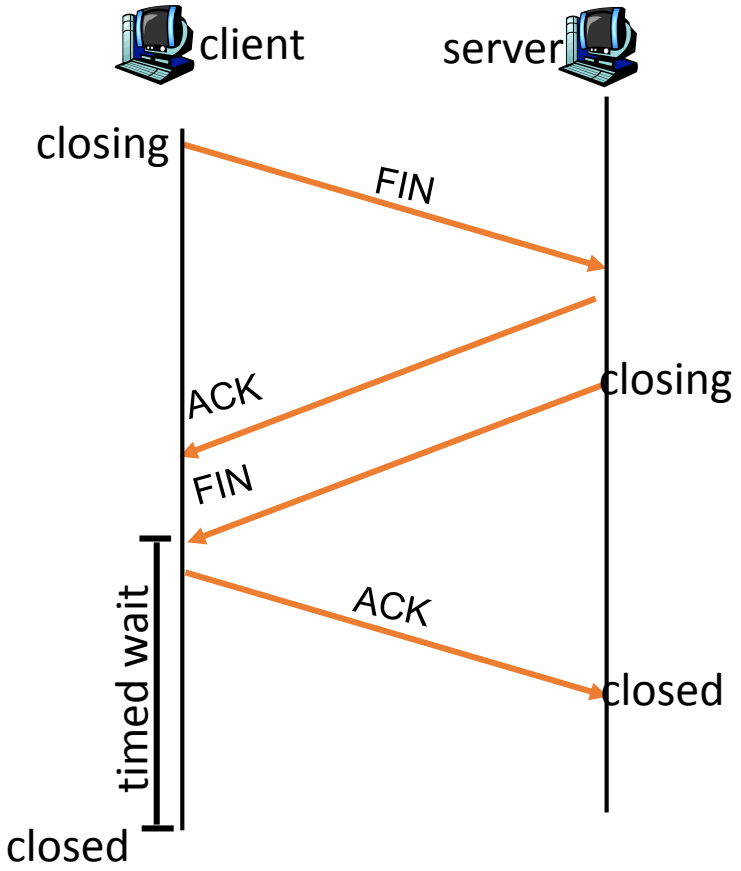
TCP Connection Termination

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

- Why wait before closing the connection?
  - If the connection were allowed to move to CLOSED state, then another pair of application processes might come along and open the same connection (use the same port #s) and a delayed FIN from an earlier incarnation would terminate the connection.

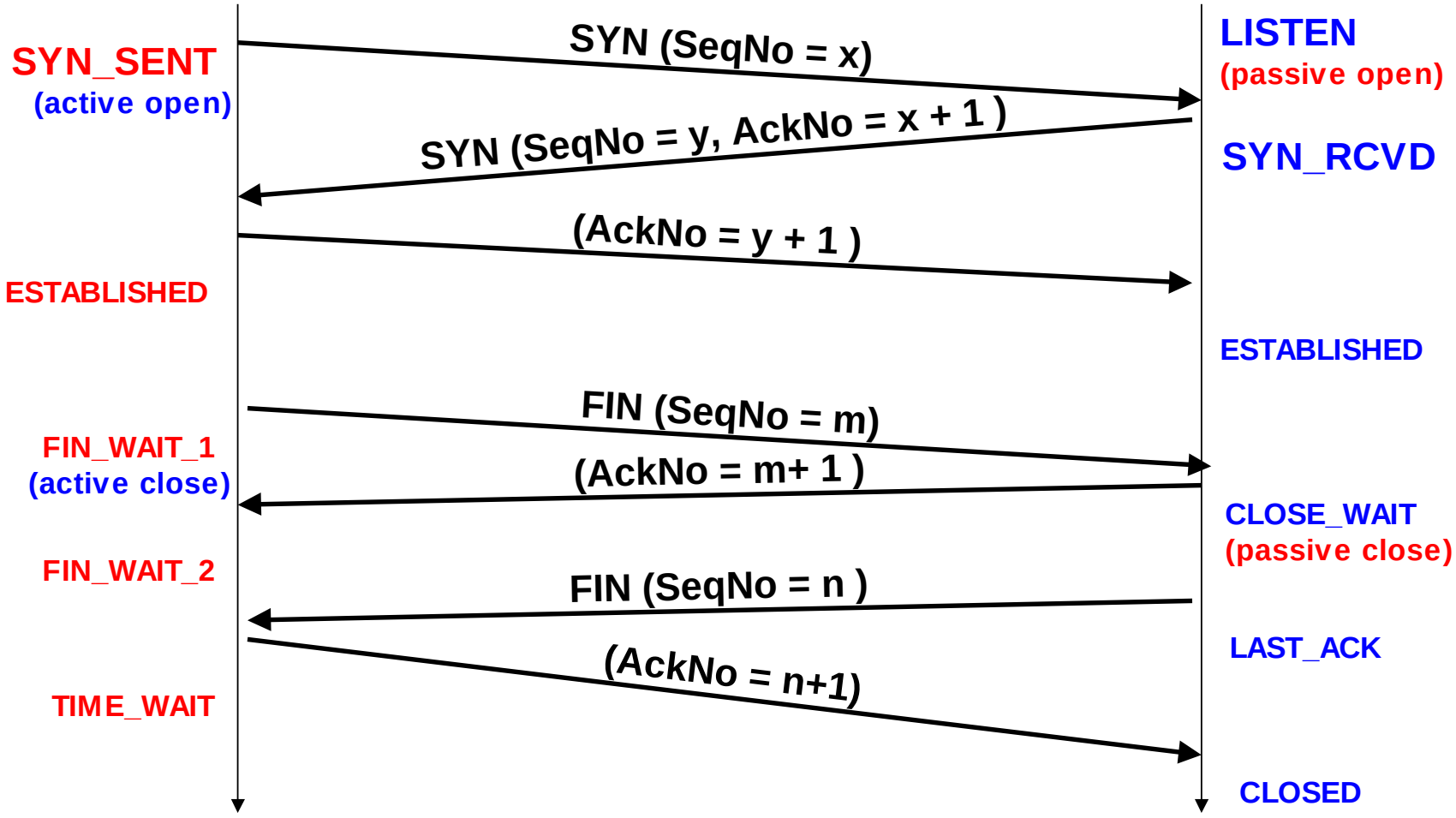


TCP/IP State Transition Diagram

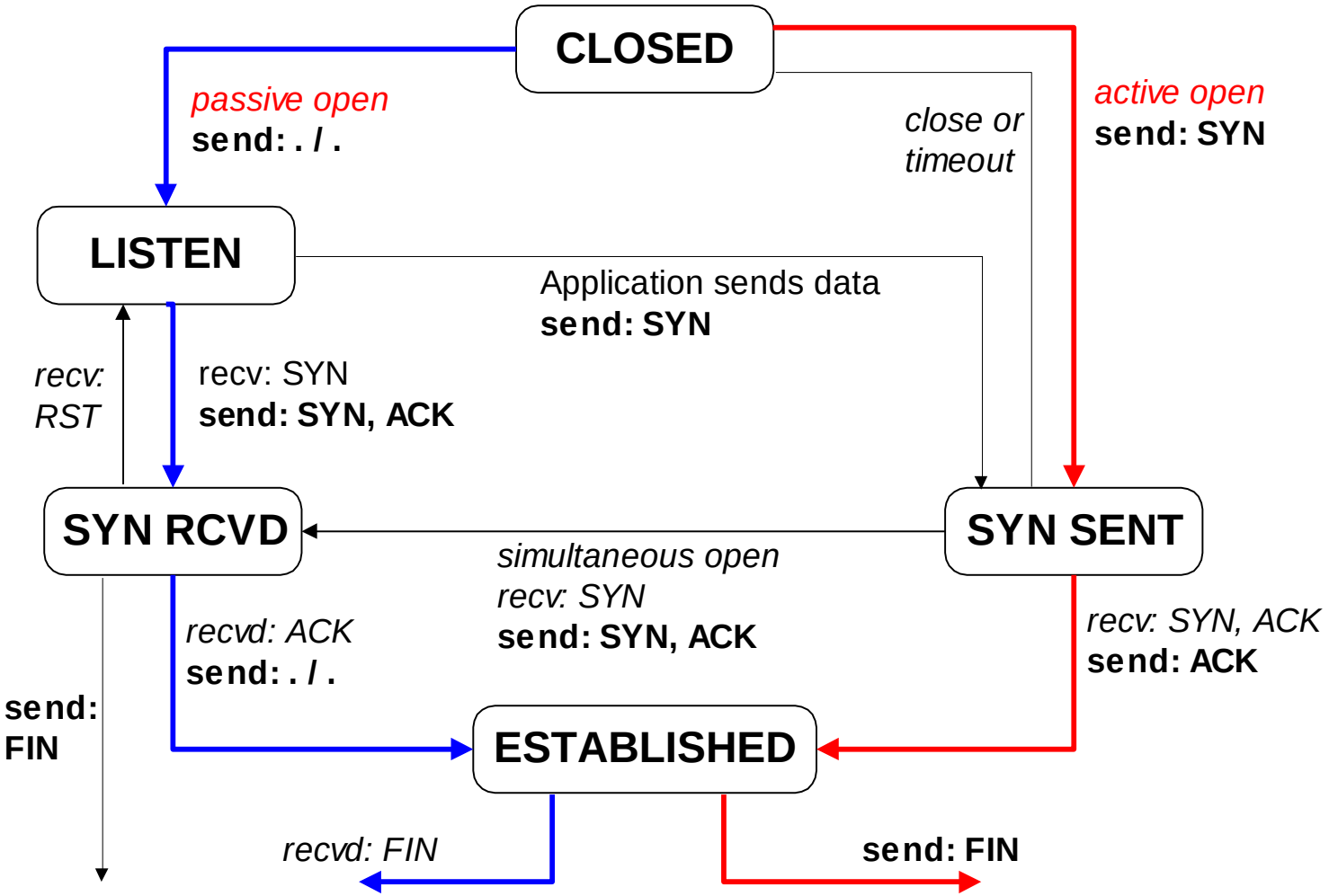
TCP States

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for Ack
SYN SENT	The client has started to open a connection
ESTABLISHED	Normal data transfer state
FIN WAIT 1	Client has said it is finished
FIN WAIT 2	Server has agreed to release
TIMED WAIT	Wait for pending packets (“2MSL wait state”)
CLOSING	Both Sides have tried to close simultanesously
CLOSE WAIT	Server has initiated a release
LAST ACK	Wait for pending packets

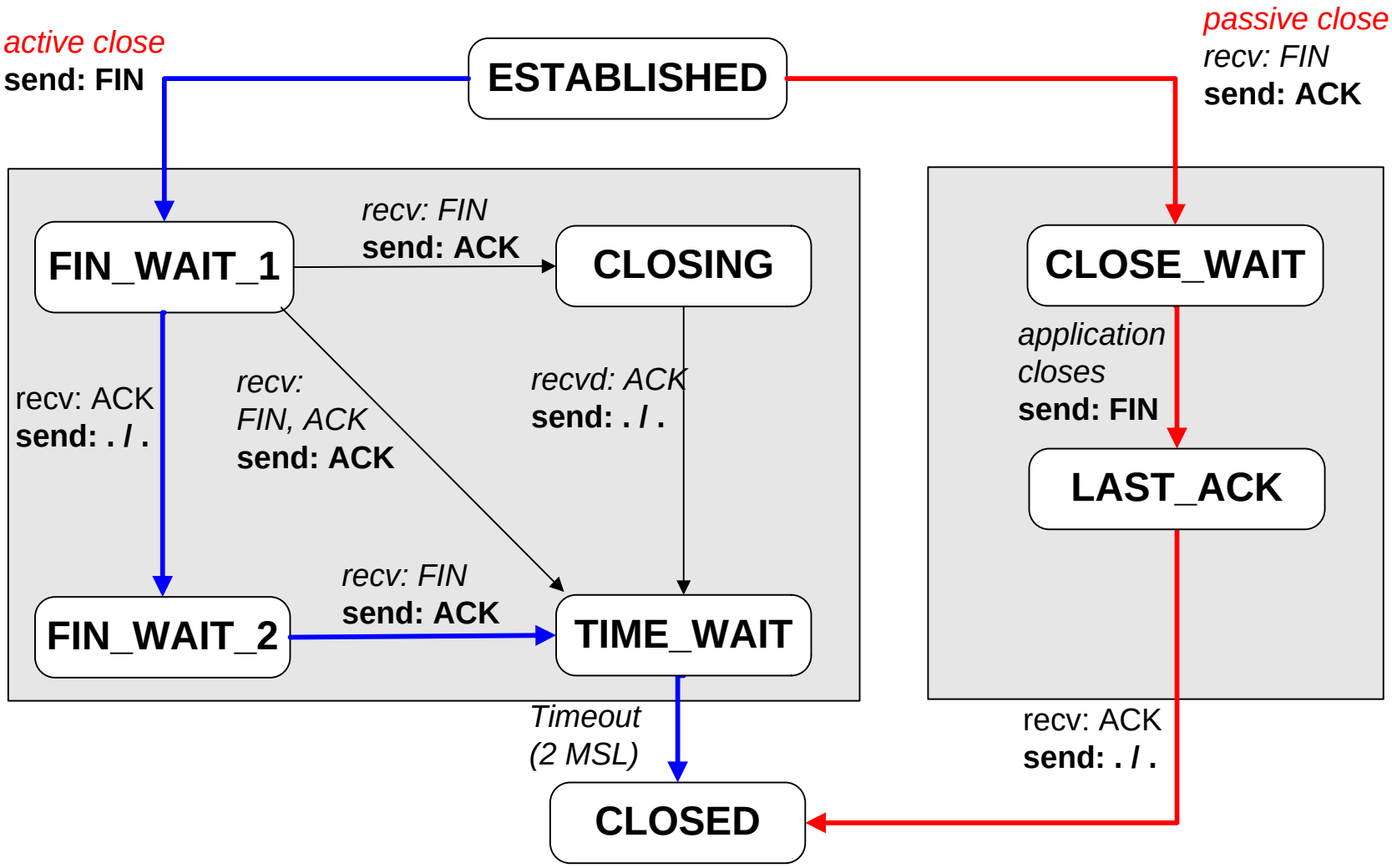
TCP States in “Normal” Connection Lifetime



TCP State Transition Diagram Opening A Connection



TCP State Transition Diagram Closing A Connection



2MSL Wait State

2MSL Wait State = TIME_WAIT

- When TCP does an active close, and sends the final ACK, the connection **must stay in in the TIME_WAIT state for twice the maximum segment lifetime**.

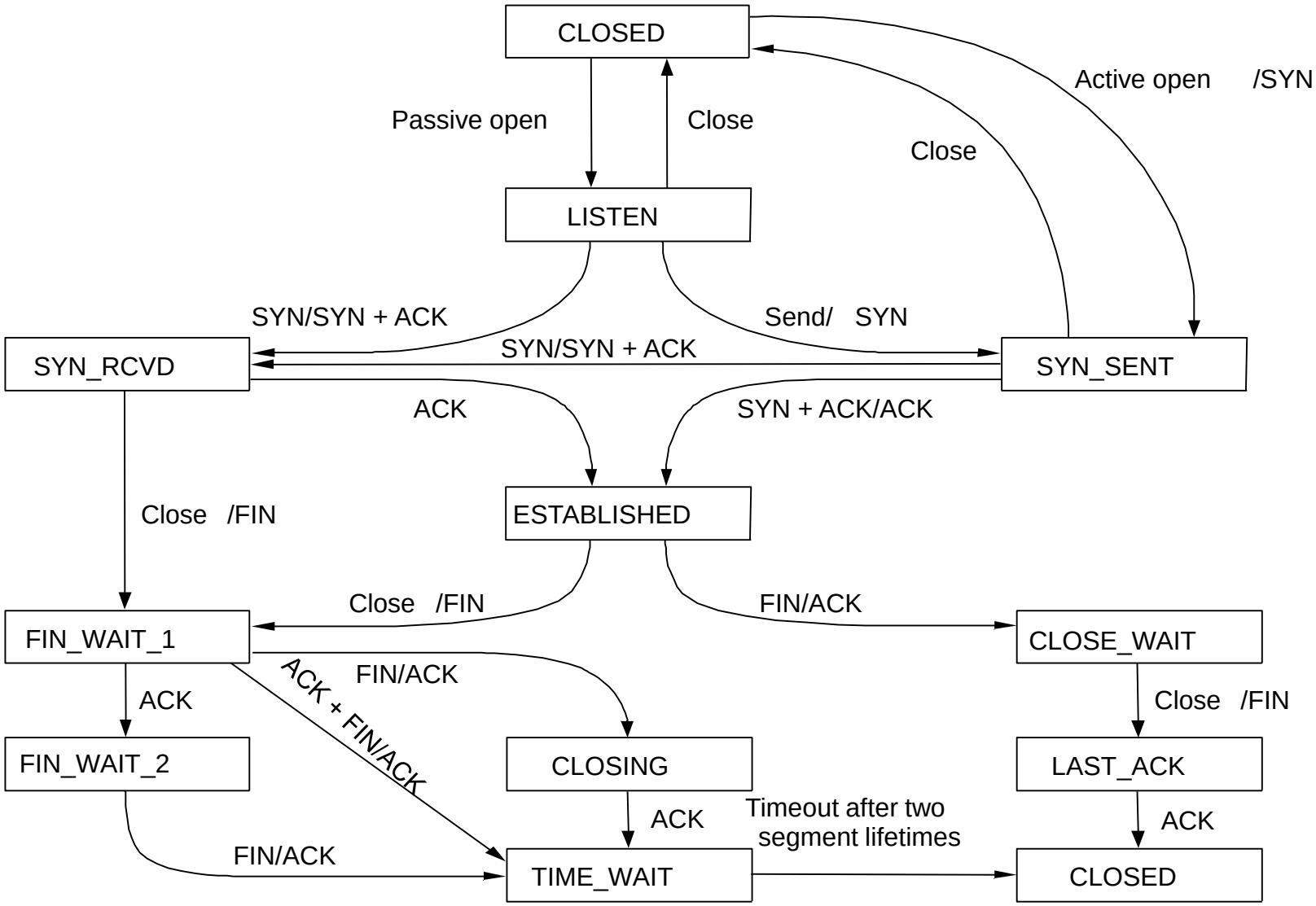
2MSL= 2 * Maximum Segment Lifetime

- Why?
  TCP is given a chance to resent the final ACK. (Server will timeout after sending the FIN segment and resend the FIN)
- The MSL is set to 2 minutes or 1 minute or 30 seconds.

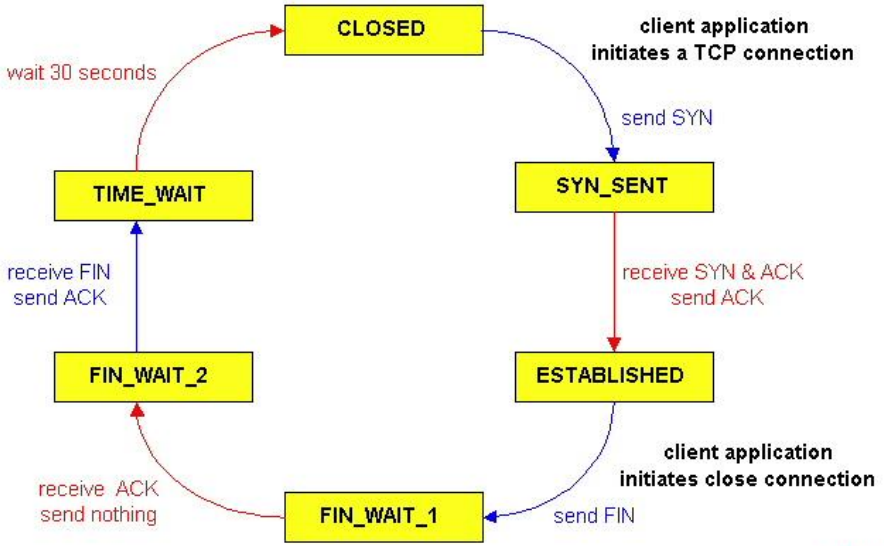
Resetting Connections

- Resetting connections is done by setting the RST flag
- **When is the RST flag set?**
  - Connection request arrives and no server process is waiting on the destination port
  - Abort (Terminate) a connection
    Causes the receiver to throw away buffered data. Receiver does not acknowledge the RST segment

TCP State-Transition Diagram

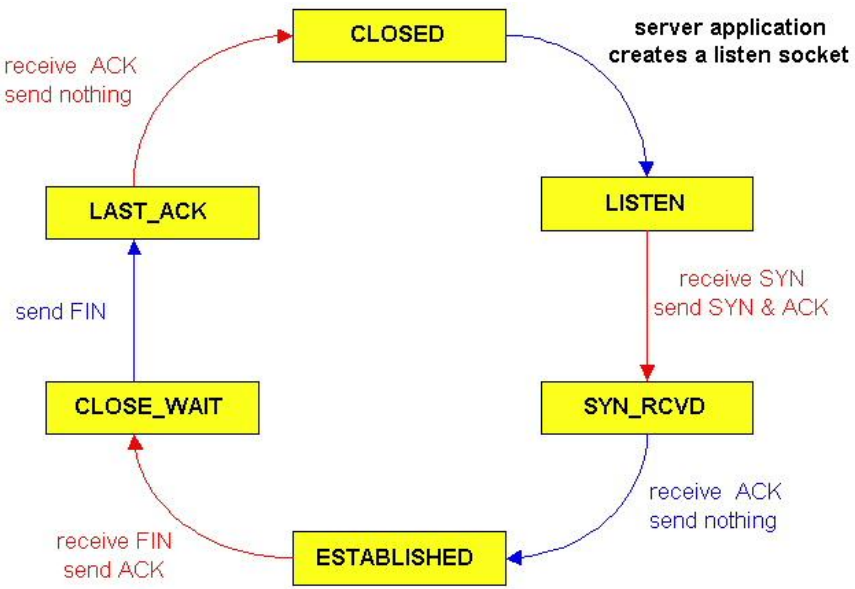


Typical TCP Client/Server Transitions



TCP client lifecycle

TCP server lifecycle

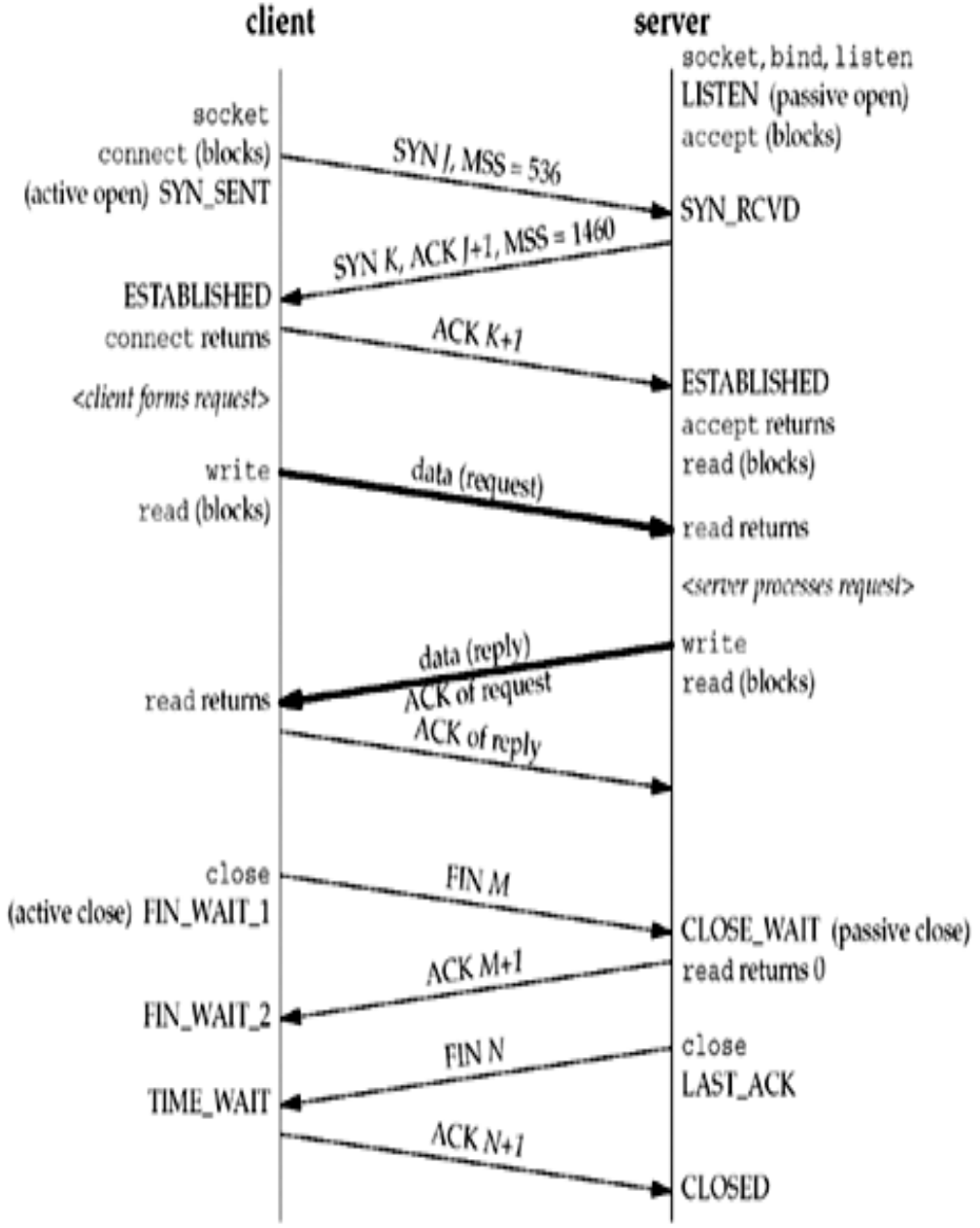


Watching the Packets

The client in this example announces an MSS of 536 (**minimum reassembly buffer size**) and the server announces an MSS of 1,460 (typical for IPv4 on an Ethernet). It is okay for the MSS to be different in each direction. The acknowledgment of the client's request is sent with the server's reply. This is called **piggybacking** and will normally happen when the time it takes the server to process the request and generate the reply is less than around 200 ms.

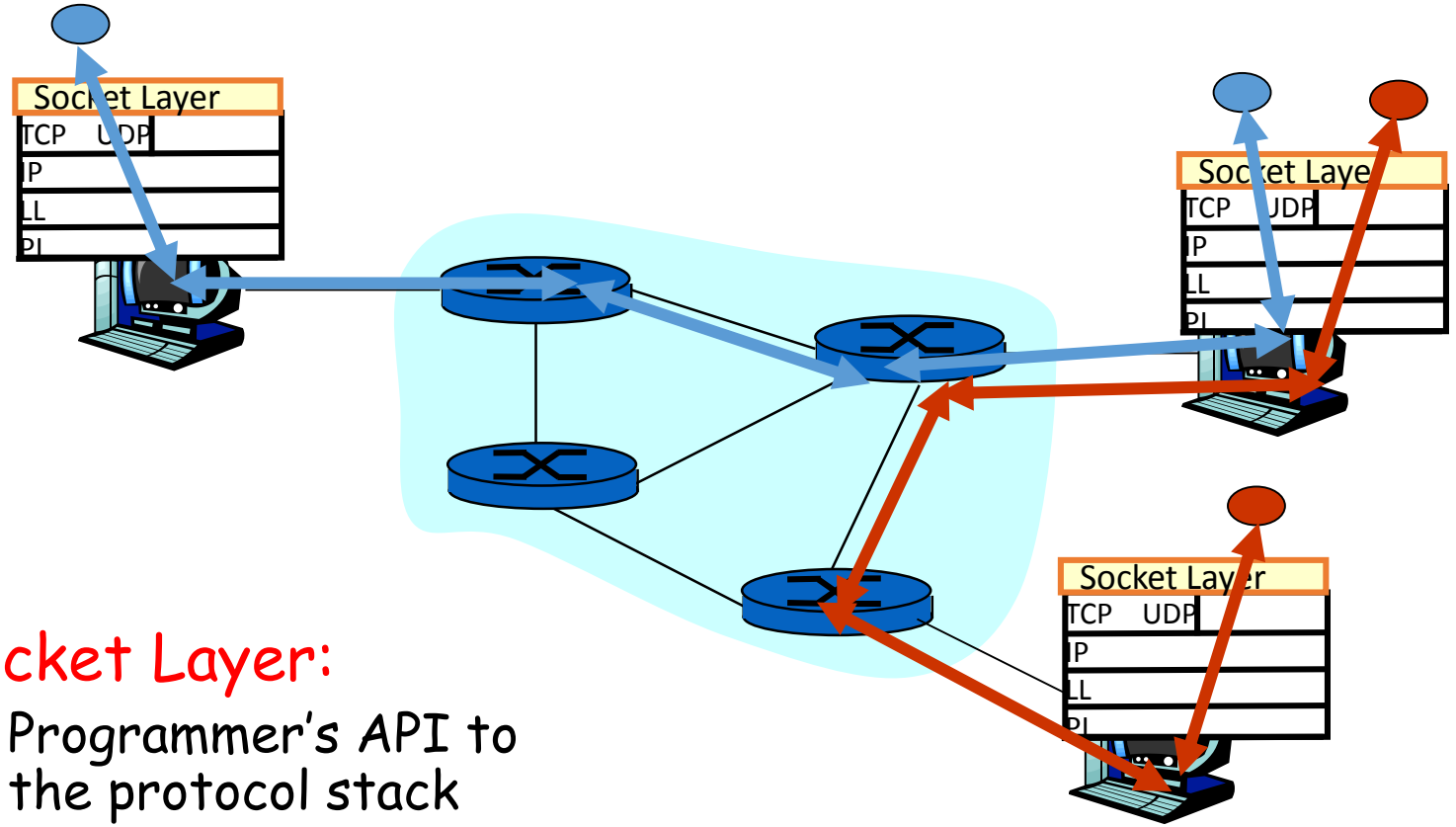
With TCP, there would be eight segments of overhead. If UDP was used, only two packets would be exchanged. UDP removes all the reliability that TCP provides to the application.

UDP avoids the overhead of TCP connection establishment and connection termination.



Socket

How to program using the TCP?



- **Socket Layer:**
  - Programmer's API to the protocol stack
- Typical network app has two pieces: *client* and *server*
- **Server:** Passive entity. Provides service to clients
  - e.g., Web server responds with the requested Web page
- **Client:** initiates contact with server ("speaks first")
  - typically requests service from server, e.g., Web Browser

Socket Creation

- mySock = socket(family, type, protocol);
- UDP/TCP/IP-specific sockets

	Family	Type	Protocol
TCP	PF_INET	SOCK_STREAM	IPPROTO_TCP
UDP		SOCK_DGRAM	IPPROTO_UDP

- Socket reference
  - File (socket) descriptor in UNIX
  - Socket handle in WinSock

TCP Client/Server Interaction

Server starts by getting ready to receive client connections…

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Create socket for incoming connections */
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
 echoServAddr.sin_family = AF_INET;                 /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);/* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);         /* Local port */

if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
   DieWithError("bind() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
 /* Mark the socket so it will listen for incoming connections */
if (listen(servSock, MAXPENDING) < 0)
   DieWithError("listen() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
for (;;) /* Run forever */
{
   clntLen = sizeof(echoClntAddr);

   if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
      DieWithError("accept() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

Server is now blocked waiting for connection from a client

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

Later, a client decides to talk to the server...

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Create a reliable, stream socket using TCP */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
 echoServAddr.sin_family      = AF_INET;                /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP);   /* Server IP address */
echoServAddr.sin_port        = htons(echoServPort);    /* Server port */

if (connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
   DieWithError("connect() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
echoStringLen = strlen(echoString);          /* Determine input length */

/* Send the string to the server */
  if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
     DieWithError("send() sent a different number of bytes than expected");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Receive message from client */
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
   DieWithError("recv() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

close(sock);

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

close(clntSocket)

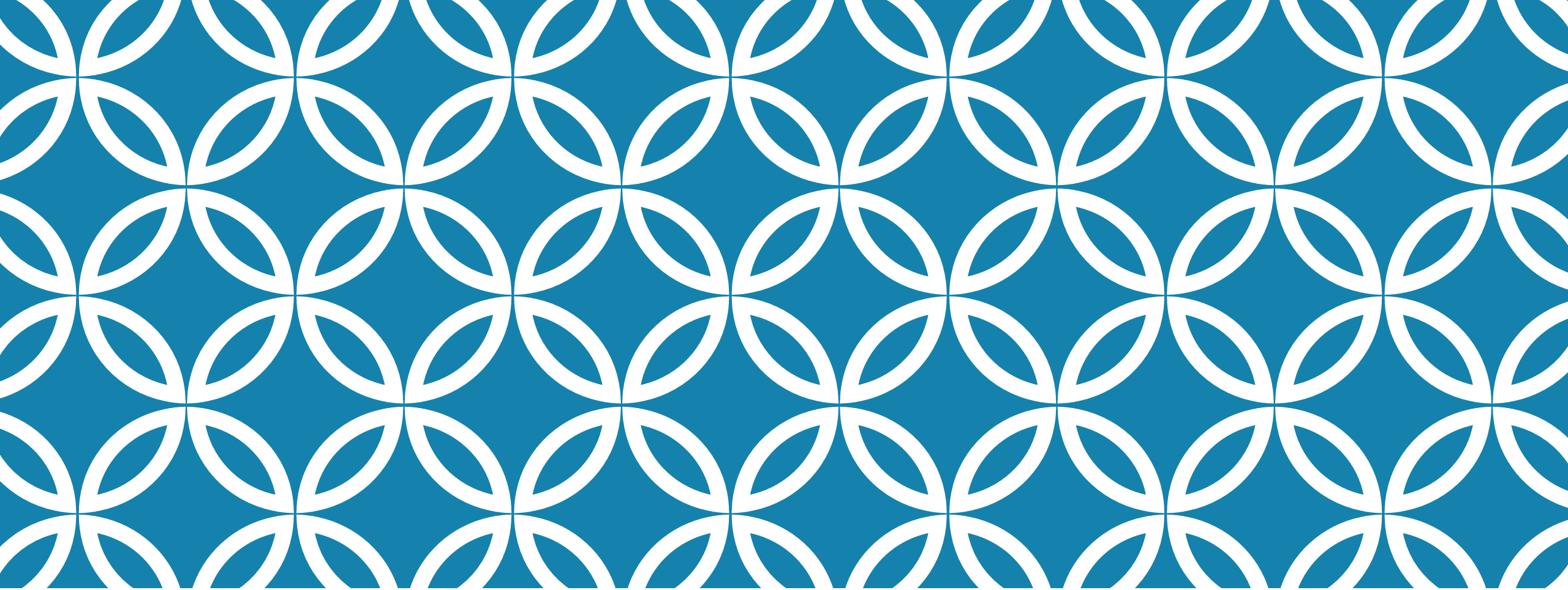
Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Tidbits

- Client knows server address and port
- No correlation between `send()` and `recv()`

Client	Server
send(“Hello Bob”)	
	recv() -> “Hello ”
	recv() -> “Bob”
	send(“Hi ”)
	send(“Jane”)
recv() -> “Hi Jane”	



NETWORK PROGRAMMING CHAPTER 2: UNIX NETWORK PROGRAMMING

Compiled By:
Madan Kadariya
HoD, Department of IT, NCIT

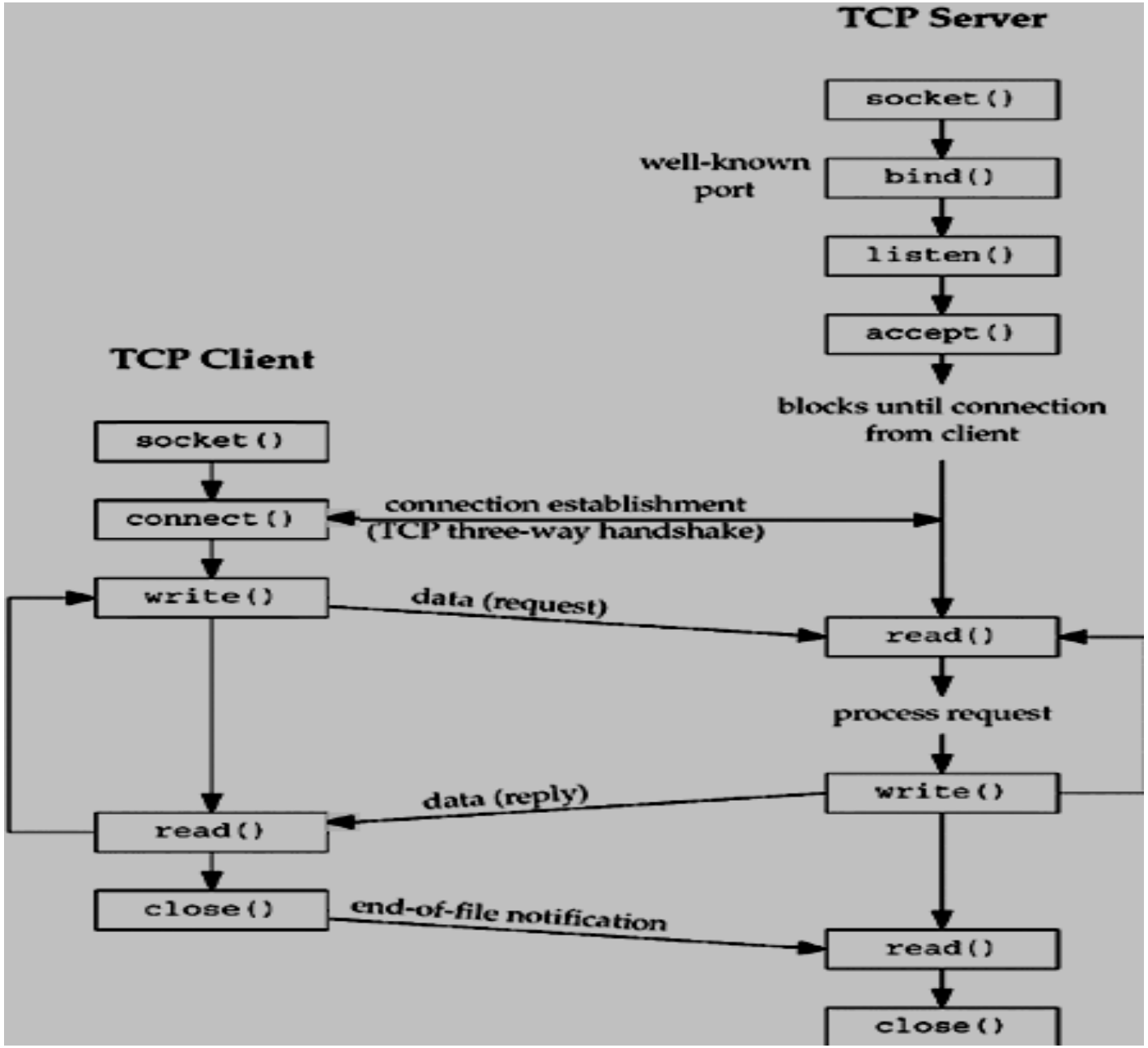
SOCKETS

- Socket is an abstraction that is provided to an application programmer to send or receive data to another process.
- Data can be sent to or received from another process running on the same machine or a different machine.
- What is a socket?
  - To the kernel, a socket is an endpoint of communication.
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network.
    - Remember: All Unix I/O devices, including networks, are modeled as files.
- Clients and servers communicate with each by reading from and writing to socket descriptors.
- The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors.

SOCKET API

- A socket API is an application-programming interface (API), usually provided by the operating system, that allows application programs to control and use network sockets.
- Internet socket APIs are usually based on the Berkeley sockets standard.
- In the Berkeley sockets standard, sockets are a form of file descriptor (a file handle)
- In inter-process communication, each end will generally have its own socket, but these may use different APIs: they are abstracted by the network protocol.
- A socket address is the combination of an IP address and a port number.

SOCKET FUNCTIONS FOR TCP CLIENT/SERVER



SOCKET ADDRESS STRUCTURES

- Various structures are used in Unix Socket Programming to hold information about the address and port, and other information.
- Most socket functions require a **pointer to a socket address structure** as an argument.
- Generic socket address structure to hold socket information. It is passed in most of the socket function calls. Unix Socket Programming provides IP version specific socket address structures.

SOCKET ADDRESS STRUCTURES

Generic socket address:

- For address arguments to `connect`, `bind`, and `accept`.

```
struct sockaddr {
  unsigned short  sa_family;    /* protocol family */
  char            sa_data[14];  /* address data.  */
};
```

Internet-specific socket address (IPv4 socket address structure) :

- Must cast (`sockaddr_in *`) to (`sockaddr *`) for `connect`, `bind`, and `accept`.

```
struct sockaddr_in  {
  unsigned short  sin_family;  /* address family (always AF_INET) */
  unsigned short  sin_port;    /* port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```

```
struct in_addr {
    unsigned long s_addr;              // this holds IPv4 address
};
```

IPV6 SOCKET ADDRESS STRUCTURE

```
struct sockaddr_in6 {
    u_int16_t          sin6_family;     // AF_INET6
    u_int16_t          sin6_port;        // this holds port number
    u_int32_t          sin6_flowinfo;  // this holds IPv6 flow information
    struct in6_addr    sin6_addr;   // used to hold IPv6 address
    u_int32_t          sin6_scope_id;// scope id
};

strcut in6_addr {
              unsigned char s6_addr[16]; // this holds IPv6 address
};
```

DATATYPES REQUIRED BY POSIX

Datatype	Description	Header
int8_t	Signed 8-bit integer	<sys/types.h>
uint8_t	Unsigned 8-bit integer	<sys/types.h>
int16_t	Signed 16-bit integer	<sys/types.h>
uint16_t	Unsigned 16-bit integer	<sys/types.h>
int32_t	Signed 32-bit integer	<sys/types.h>
uint32_t	Unsigned 32-bit integer	<sys/types.h>
sa_family_t	Address family of socket address structure	<sys/socket.h>
socklen_t	Length of socket address structure, normally uint32_t	<sys/socket.h>
in_addr_t	IPv4 address, normally uint32_t	<netinet/in.h>
in_port_t	TCP or UDP port, normally uint16_t	<netinet/in.h>

VALUE-RESULT ARGUMENTS

- When a **socket address structure** is passed to any socket function, it is always passed by reference. That is, a **pointer to the structure** is passed.
- The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed:

- From the process to the kernel
- From the kernel to the process

From process to kernel

bind(), *connect()*, and *sendto()* functions pass a **socket address structure** from the process to the kernel

Arumgents to these functions:

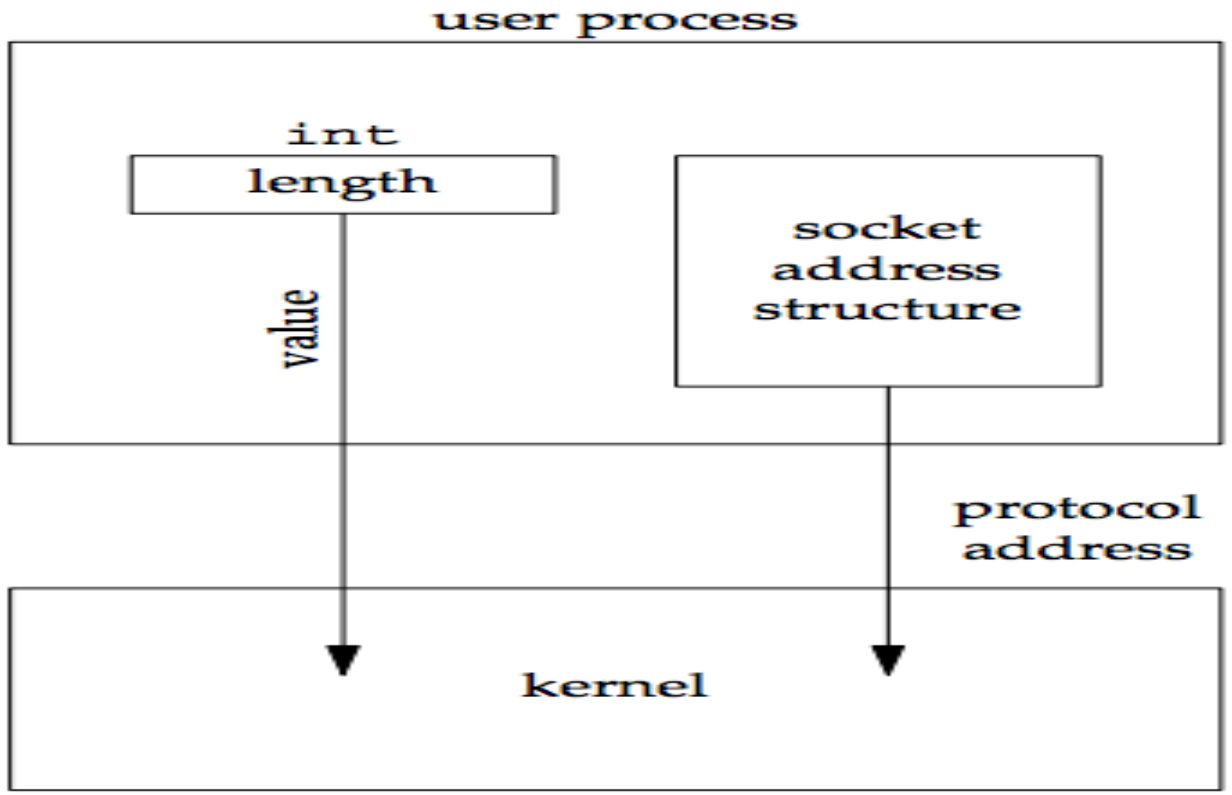
The pointer to the socket address structure

The integer size of the structure

```
struct sockaddr_in serv;
/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```

The datatype for the size of a socket address structure is actually *socklen_t* and not *int*, but the POSIX specification recommends that *socklen_t* be defined as *uint32_t*

VALUE-RESULT ARGUMENTS



VALUE-RESULT ARGUMENTS

Kernel to Process

accept(), *recvfrom()*, *getsockname()*, and *getpeername()* functions pass a **socket address structure** from the kernel to the process.

Arguments to these functions:

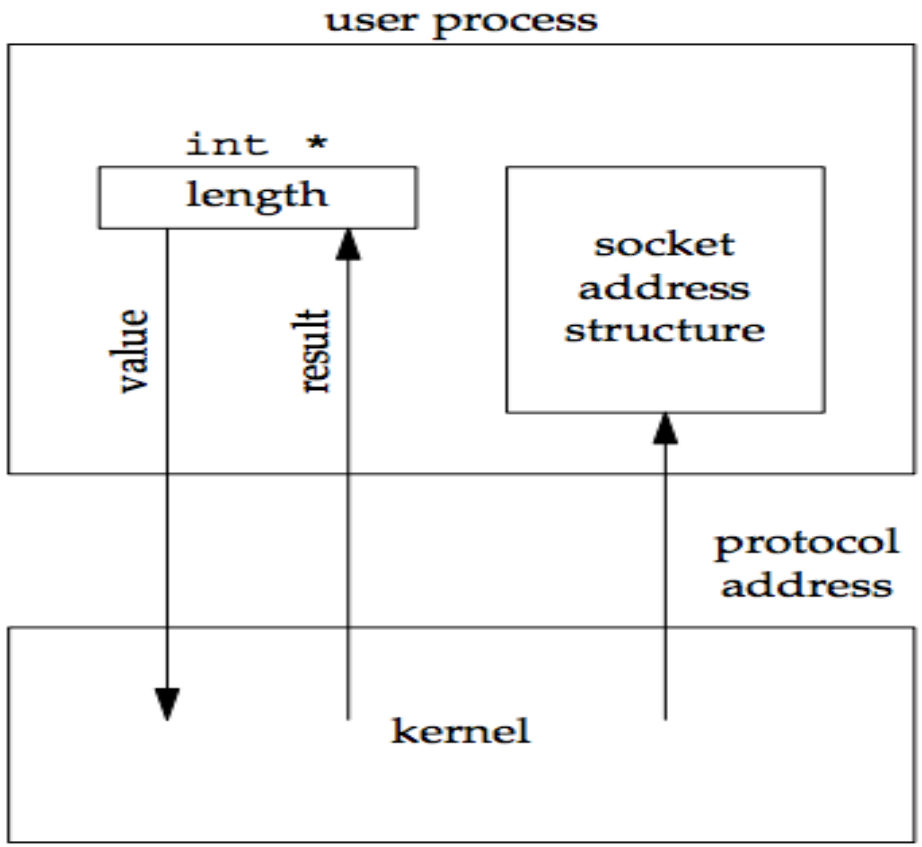
The pointer to the socket address structure

The pointer to an integer containing the size of the structure

```
struct sockaddr_un  cli;    /* Unix domain */
socklen_t  len;
len = sizeof(cli);          /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```

VALUE-RESULT ARGUMENTS

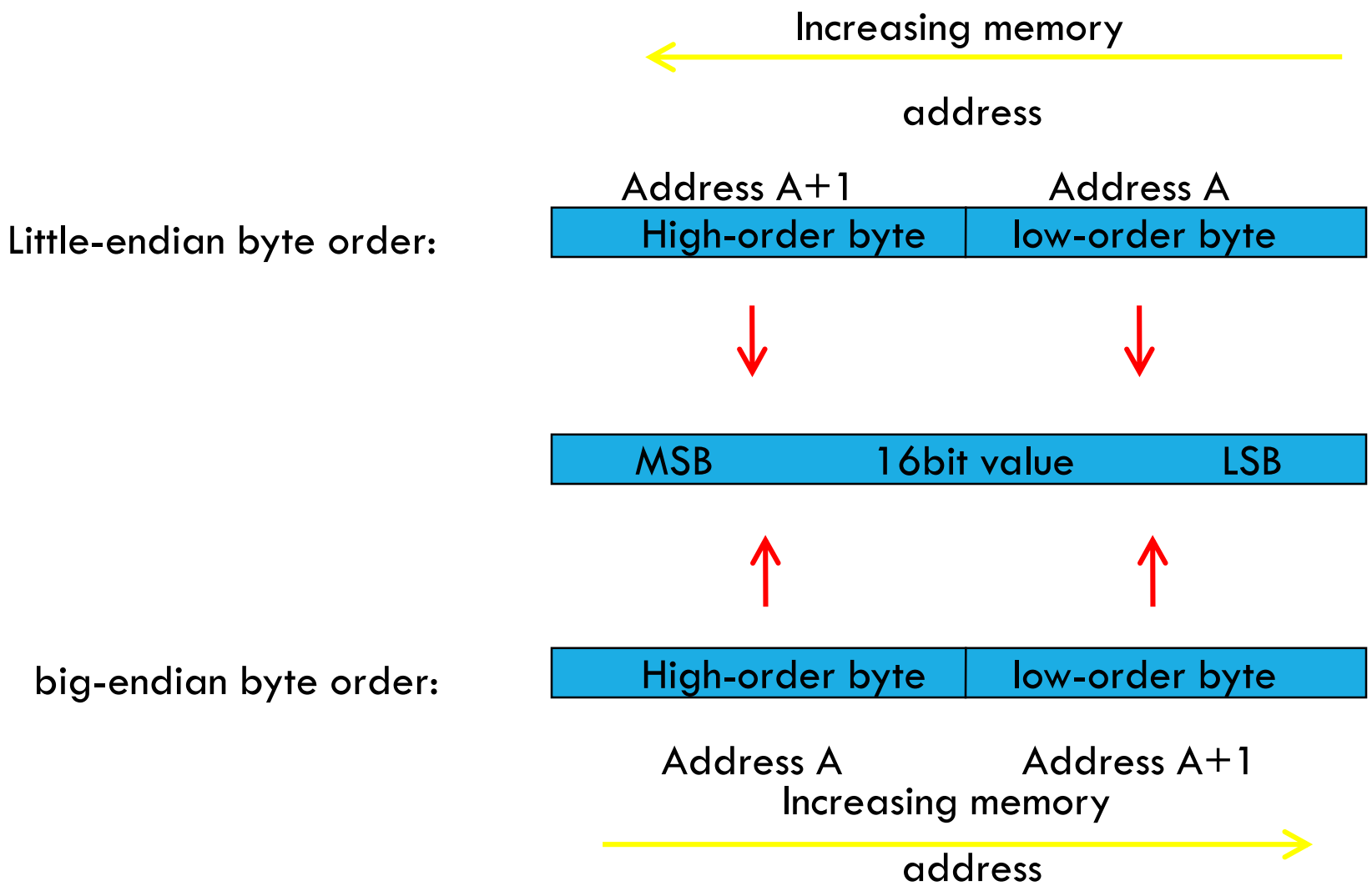
- Value-only: ***bind, connect, sendto*** (from process to kernel)
- Value-Result: ***accept, recvfrom, getsockname, getpeername*** (from kernel to process, pass a pointer to an integer containing size)
  - *Tells process how much information kernel actually stored*



BYTE ORDERING FUNCTIONS

- Two ways to store 2 bytes (16-bit integer) in memory
  - Low-order byte at starting address → little-endian byte order
  - High-order byte at starting address → big-endian byte order
- *in a big-endian computer* → store 4F52
  - Stored as 4F52 → 4F is stored at storage address 1000, 52 will be at address 1001, for example
- *In a little-endian system* → store 4F52
  - it would be stored as 524F (52 at address 1000, 4F at 1001)
- Byte order used by a given system known as *host byte order*
- Network programmers use *network byte order*
- Internet protocol uses big-endian byte ordering for integers (port number and IP address)

BYTE ORDERING FUNCTIONS



BYTE ORDERING FUNCTIONS

```
#include        "unp.h"
int main(int argc, char **argv)
{
        union {
                short  s;
                char   c[sizeof(short)];
                  } un;

        un.s = 0x0102;
        //printf("%s: ", CPU_VENDOR_OS);
        if (sizeof(short) == 2) {
                if (un.c[0] == 1 && un.c[1] == 2)
                        printf("big-endian\n");
                else if (un.c[0] == 2 && un.c[1] == 1)
                        printf("little-endian\n");
                else
                        printf("unknown\n");
        } else
                printf("sizeof(short) = %d\n", sizeof(short));

        exit(0);
}
```

- Sample program to figure out little-endian or big-endian machine
- Source code in byteorder.c

BYTE ORDERING FUNCTIONS

Converts between **host byte order** and **network byte order**

- 'h' = host byte order
- 'n' = network byte order
- 'l' = long (4 bytes), converts IP addresses
- 's' = short (2 bytes), converts port numbers

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int
hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int
netshort);
```

BYTE ORDERING FUNCTIONS 4/4

- To convert between byte orders
  - Return value in network byte order
    - ✓ htons (s for short word 2 bytes)
    - ✓ htonl (l for long word 4 bytes)
  - Return value in host byte order
    - ✓ ntohs
    - ✓ ntohl
- Must call appropriate function to convert between host and network byte order
- On systems that have the same ordering as the Internet protocols, four functions usually defined as null macros

```
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(13);
```

DEALING WITH IP ADDRESSES

IP Addresses are commonly written as strings (“128.2.35.50”), but programs deal with IP addresses as integers.

Converting strings to numerical address:

```
struct sockaddr_in srv;

srv.sin_addr.s_addr = inet_addr(“128.2.35.50”);
if(srv.sin_addr.s_addr == (in_addr_t) -1) {
      fprintf(stderr, "inet_addr failed!\n"); exit(1);
}
```

Converting a numerical address to a string:

```
struct sockaddr_in srv;
char *t = inet_ntoa(srv.sin_addr);
if(t == 0) {
      fprintf(stderr, “inet_ntoa failed!\n”); exit(1);
}
```

BYTE MANIPULATION FUNCTIONS

#include <strings.h>

void bzero (void *dest, size_t nbytes);

// sets specified number of bytes to 0 in the destination

void bcopy (const void *src,void * dest, size_t nbytes);

// moves specified number of bytes from source to destination

void bcmp (const void *ptr1, const void *ptr2,size_t nbytes)

//compares two arbitrary byte strings, return value is zero if two byte strings are identical, otherwise, nonzero

ADDRESS CONVERSION FUNCTIONS 1/2

Convert an IPv4 address from a dotted-decimal string “206.168.112.96” to a 32-bit network byte order binary value

```
#include <arpa/inet.h>
int inet_aton (const char* strptr, struct in_addr *addrptr);
// return 1 if string was valid, 0 on error. Address stored in *addrptr

in_addr_t inet_addr (const char * strptr);
// returns 32 bit binary network byte order IPv4 address, currently deprecated

char * inet_nota (struct in_addr inaddr);
//returns pointer to dotted-decimal string
```

ADDRESS CONVERSION FUNCTIONS 2/2

To handle both IPv4 and IPv6 addresses

```
#include <arpa/inet.h>
int inet_pton (int family, const char* strptr, void *addrptr);
```

// return 1 if OK, 0 on error. 0 if not a valid presentation, -1 on error, Address stored in *addrptr
[Here p:presentation, n: Numeric]

```
Const char * inet_ntop (int family, const void* addrptr, char *strptr, size_t len);
```

// return pointer to result if OK, NULL on error

```
if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
                    err_quit("inet_pton error for %s", argv[1]);

ptr = inet_ntop (AF_INET,&addr.sin_addr,str,sizeof(str));
```

The family argument for both functions is either AF_INET or AF_INET6. If family is not supported, both functions return an error with errno set to EAFNOSUPPORT.

FORK AND EXEC

FORK AND EXEC FUNCTIONS

- fork is called once but it returns twice.
- The creation of a new process is done using the **fork()** system call.
- A new program is run using the **exec**(l,lp,le,v,vp) family of system calls.
- These are two separate functions which may be used independently.

THE FORK() SYSTEM CALL

1. A call to fork() will create a completely separate sub-process which will be exactly the same as the parent.
2. The process that initiates the call to fork is called the parent process.
3. The new process created by fork is called the child process.
4. The child gets a copy of the parent's text and memory space.
5. They do not share the same memory .

FORK RETURN VALUES

fork() system call returns an integer to both the parent and child processes:

- -1 this indicates an error with no child process created.
- A value of zero indicates that the child process code is being executed.
- Positive integers represent the child's process identifier (PID) and the code being executed is in the parent's process.

```
if ( (pid = fork()) == 0)
        printf("I am the child\n");
else
        printf("I am the parent\n");
```

THE EXEC() SYSTEM CALL

- Calling one of the **exec()** family will terminate the currently running program and starts executing a new one which is specified in the parameters of exec in the context of the existing process. The process id is not changed.

EXEC Family of Functions

```
int execl( const char *path, const char *arg, ...);

int execle( const char *path, const char *arg , ..., char * const envp[]);

int execv( const char *path, char *const argv[]);

int execv( const char *path, char *const argv[], char * const envp[]);

int execlp( const char *file, const char *arg, ...);

int execvp( const char *file, char *const argv[]);
```

- The first difference in these functions is that the first four take a pathname argument while the last two take a filename argument. When a filename argument is specified:
  - if filename contains a slash, it is taken as a pathname.
  - otherwise the executable file is searched for in the directories specified by the PATH environment variable.

SIMPLE EXECLP EXAMPLE

```
#include <sys/type.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid_t pid;
        /* fork a child process */
        pid = fork();
        if (pid < 0){ /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0){ /* child process */
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for child to complete */
                wait(NULL);
                printf("Child Complete");
                exit(0);
        }
}
```

FORK() AND EXEC()

- When a program wants to have another program running in parallel, it will typically first use **fork**, then the child process will use **exec** to actually run the desired program.
- The **fork-and-exec** mechanism switches an old command with a new, while the environment in which the new program is executed remains the same, including configuration of input and output devices, and environment variables.

CONCURRENT SERVER

- When a client request can take longer to service, we do not want to tie up a single server with one client; we want to handle multiple clients at the same time. The server that handles multiple clients simultaneously is a concurrent server.
- Outline of typical Concurrent Server

```
pid_t pid;
int listenfd, connfd;
listenfd = Socket(…);
/* fill in sockaddr_in{} with server's well-know port */
Bind(listenfd, …);
Listen(listenfd, LISTENQ);
for(;;) {
                connfd = Accept(listenfd, …); // probably blocks
                if( (pid = Fork()) ==0) {
                                Close(listenfd); // child closes listening socket
                                doit(connfd); // process the request
                                Close(connfd); // done with this client
                                exit(0); //child terminates
        }
    Close(connfd);
}
```

- When a connection is established, accept returns, the server calls fork, and the child process services the client (on connfd, the connected socket) and the parent process waits for another connection (on listenfd, the listening socket).
- The parent closes the connected socket since the child handles the new client. The function do it does whatever is required to service the client.
- Calling close on a TCP socket causes a FIN to be sent, followed by the normal TCP connection termination sequence. However, the close of connfd by the parent (in the outline) doesn't terminate its connection with the client. This is because every file or socket has a reference count.
- The reference count is maintained in the file table entry. This is a count of the number of descriptors that are currently open that refer to this file or socket. In the outline, after socket returns, the file table entry associated with listenfd has a reference count of 1.
- After accept returns, the file table entry associated with connfd has a reference count of 1. But, after fork returns, both descriptors are shared between the parent and child, so the file table entries associated with both sockets now have a reference count of 2.
- Therefore, when the parent closes connfd, it just decrements the reference count from 2 to 1 and that is all. The actual cleanup and de-allocation of the socket does not happen until the reference count reaches 0. This will occur at some time later when the child closes connfd.

CONCURRENT SERVER

Visualization of sockets and connection that occur in the outline code

1. The server is blocked in the call to accept and the connection request arrives from the client.
2. The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.
3. The fork is called. After fork returns, listenfd and connfd are shared between the parent and child.
4. The parent closes the connected socket and the child closes the listening socket.

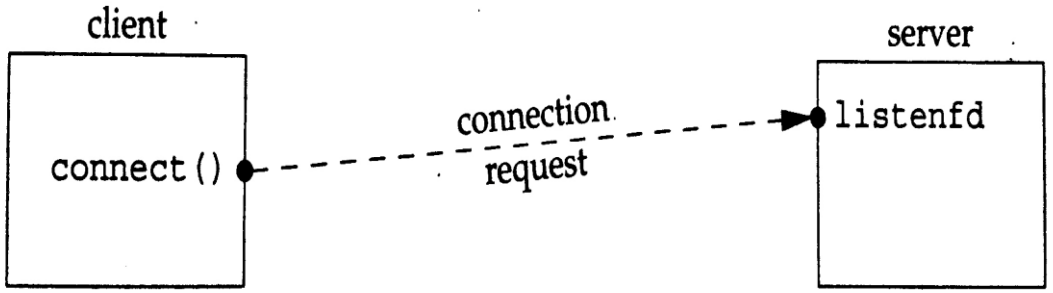


Figure 4.14 Status of client/server before call to accept returns.

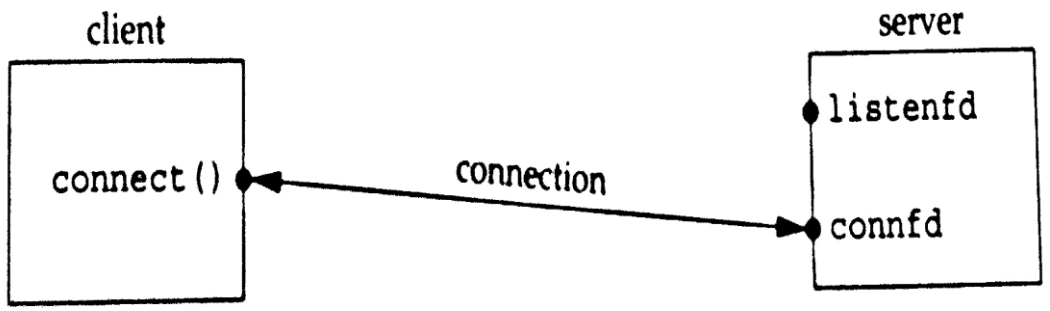


Figure 4.15 Status of client/server after return from accept.

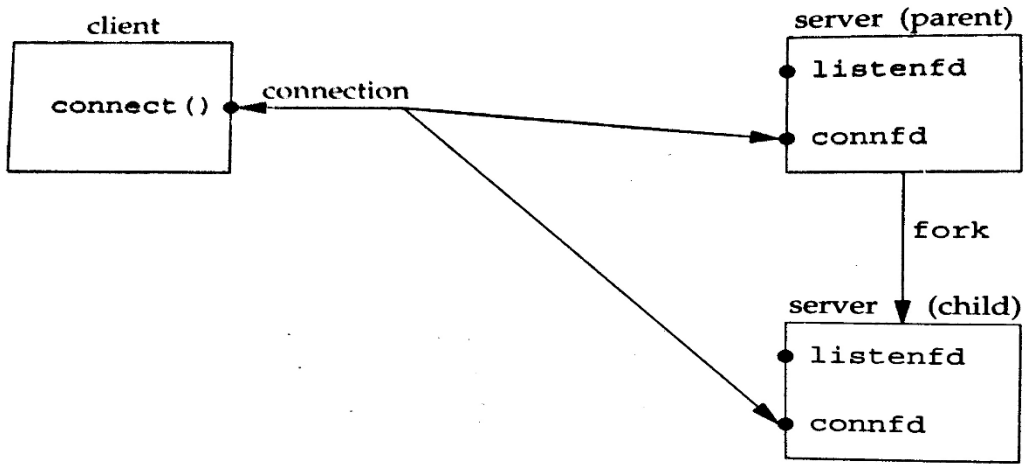


Figure 4.16 Status of client/server after `fork` returns.

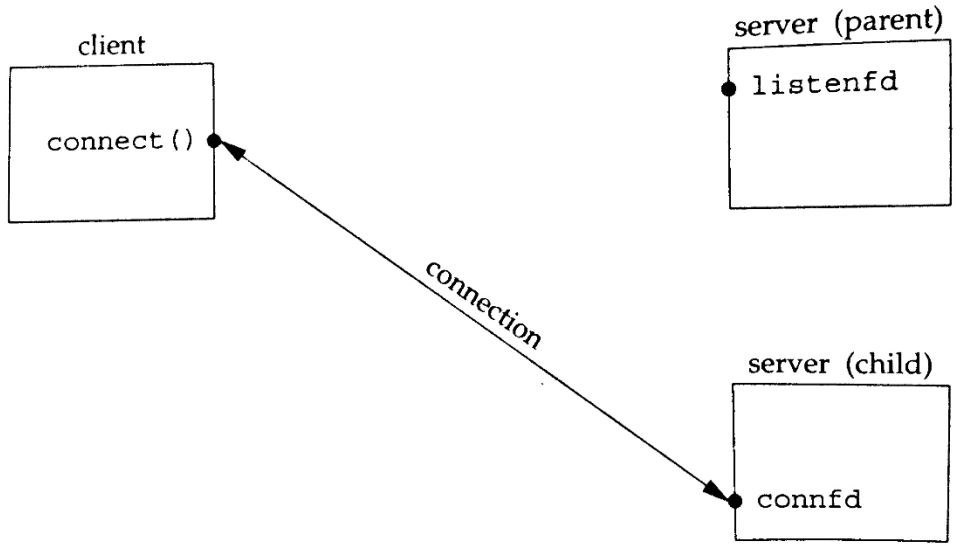
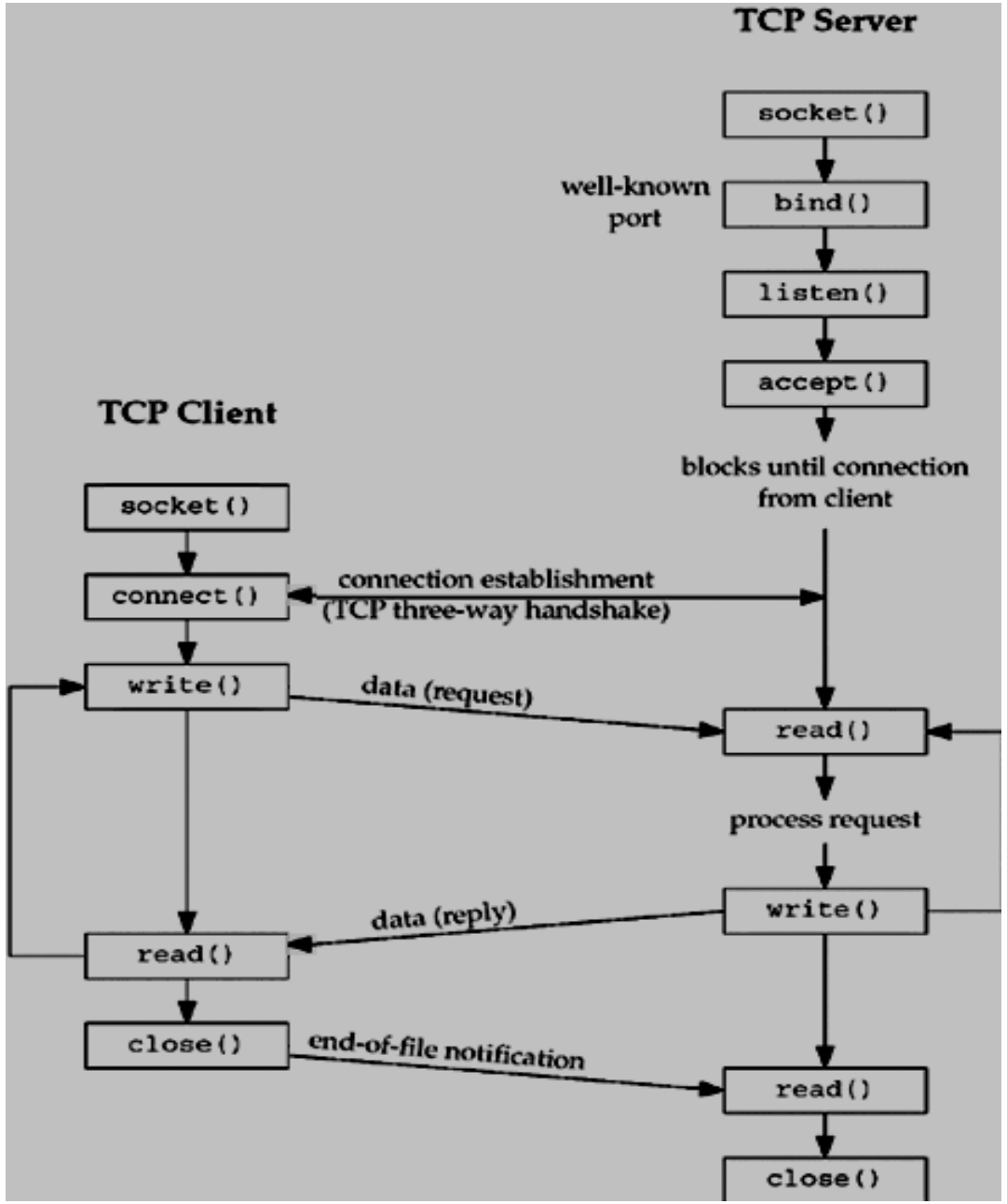


Figure 4.17 Status of client/server after parent and child close appropriate sockets.

UNIX/INTERNET DOMAIN SOCKET (TCP CONNECTION)



SOCKET ( )

To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

int socket(int *family*, int *type*, int *protocol*);

Returns: non-negative descriptor if OK, -1 on error

family is one of

- AF_INET (IPv4), AF_INET6 (IPv6), AF_LOCAL (local Unix),
- AF_ROUTE (access to routing tables), AF_KEY (new, for encryption)

type is one of

- SOCK_STREAM (TCP), SOCK_DGRAM (UDP)
- SOCK_RAW (for special IP packets, PING, etc. Must be root)
- SOCK_SEQPACKET (Sequenced packet socket)

Protocol is one of

- IPPROTO_TCP
- IPPROTO_UDP
- IPPROTO_SCTP
- *protocol* is 0 (used for some raw socket options)

> Not all combinations of socket *family* and *type* are valid. The table below shows the valid combinations, along with the actual protocols that are valid for each pair. The boxes marked "Yes" are valid but do not have handy acronyms. The blank boxes are not supported.

	AF_INET	**AF_INET6**	**AF_LOCAL**	**AF_ROUTE**	**AF_KEY**
SOCK_STREAM	TCP/SCTP	TCP/SCTP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_SEQPACKET	SCTP	SCTP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

The key socket, AF_KEY, is newer than the others. It provides support for cryptographic security. Similar to the way that a routing socket (AF_ROUTE) is an interface to the kernel's routing table, the key socket is an interface into the kernel's key table.

BIND()

- The bind function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

int bind(int *sockfd*, const struct sockaddr **myaddr*,socklen_t *addrlen*);

- Returns: 0 if OK,-1 on error

sockfd is socket descriptor from `socket()`

myaddr is a pointer to address struct with:

- *port number* and *IP address*
- if port is 0, then host will pick ephemeral port
  - not usually for server (exception RPC port-map)
- IP address != INADDR_ANY (unless multiple nics)

addrlen is length of structure

returns 0 if ok, -1 on error

- EADDRINUSE ("Address already in use")

- Calling **bind** lets us specify the IP address, the port, both, or neither. The following table summarizes the values to which we set **sin_addr** and **sin_port**, or **sin6_addr** and **sin6_port**, depending on the desired result.

IP address	Port	Result
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

- If we specify a port number of 0, the kernel chooses an ephemeral port when bind is called.
- If we specify a wildcard IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP).

WILDCARD ADDRESS AND INADDR_ANY

- With IPv4, the *wildcard* address is specified by the constant INADDR_ANY, whose value is normally 0. This tells the kernel to choose the IP address.

```
struct sockaddr_in servaddr;

servaddr.sin_addr.s_addr = htonl (INADDR_ANY); /* wildcard */
```

- While this works with IPv4, where an IP address is a 32-bit value that can be represented as a simple numeric constant (0 in this case), we cannot use this technique with IPv6, since the 128-bit IPv6 address is stored in a structure.

```
struct sockaddr_in6 serv;

serv.sin6_addr = in6addr_any; /* wildcard */
```

- The system allocates and initializes the **in6addr_any** variable to the constant IN6ADDR_ANY_INIT.
- The value of INADDR_ANY (0) is the same in either network or host byte order, so the use of **htonl** is not really required. But, since all the **INADDR_constants** defined by the <netinet/in.h> header are defined in host byte order, we should use **htonl** with any of these constants.

LISTEN()

The connect function is used by a TCP client to establish a connection with a TCP server
Int listen(int sockfd, int backlog);
Change socket state for TCP server.

- *sockfd* is socket descriptor from socket()
- *backlog* is maximum number of *incomplete* connections
  - historically *5*
  - rarely above 15 on a even moderate Web server!

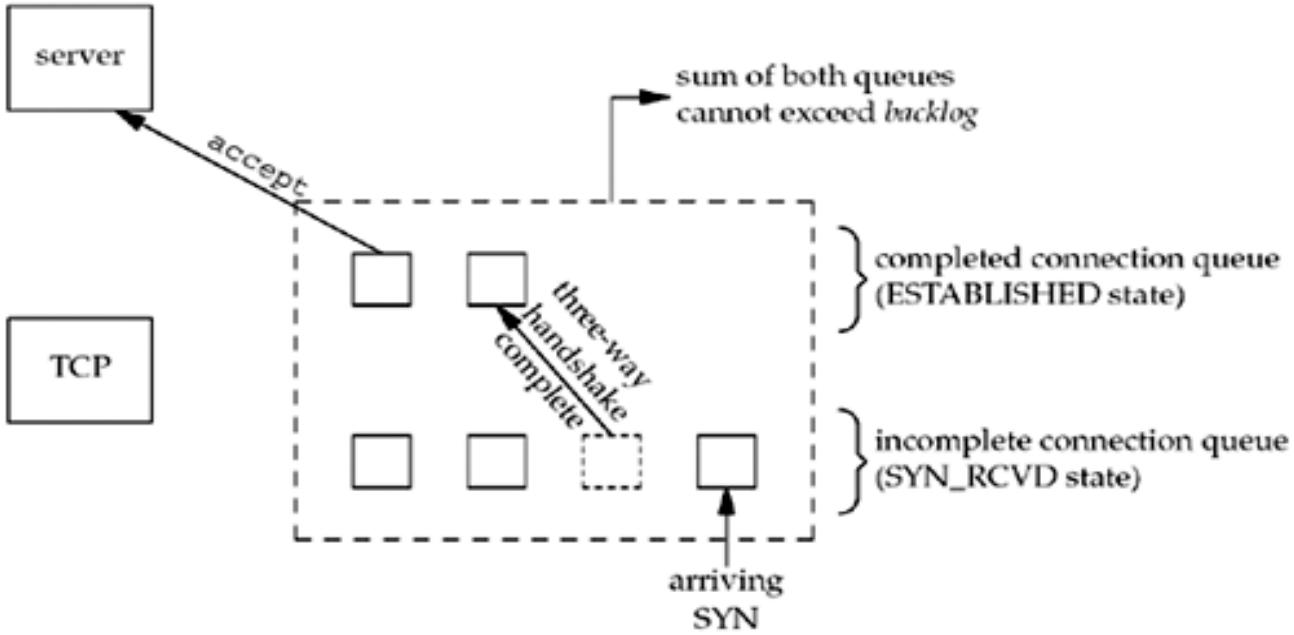
Sockets default to active (for a client)

- change to passive so OS will accept connection

- An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake.
- A completed connection queue, which contains an entry for each client with whom the TCP three-way handshake has completed.
- In terms of the TCP state transition diagram, the call to listen moves the socket from the CLOSED state to the LISTEN state.

CONNECTION QUEUES

- For ***backlog*** argument, we must realize that for a given listening socket, the kernel maintains two queues:
  - An **incomplete connection queue,** which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state.
  - A **completed connection queue,** which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state.



CONNECT ( )

```
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

Connect to server.

- *sockfd* is socket descriptor from socket()
- *servaddr* is a pointer to a structure with:
  - *port number* and *IP address*
  - must be specified (unlike bind())
- *addrlen* is length of structure
- returns socket descriptor if ok, -1 on error
- The client does not have to call bind before calling connect, the kernel will choose both an ephemeral port and the source IP address if necessary.
- ETIMEDOUT: host doesn't exist (connection timed out)
- ECONNREFUSED: no process is waiting for connections on the server host at the port specified
- EHOSTUNREACH: no route to host

- In the case of a TCP socket, the connect function initiates TCP's three-way handshake.
- The function returns only when the connection is established or an error occurs.

Possible Error:

1. If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned.
   - connect moves from the CLOSED state (the state in which a socket begins when it is created by the socket function) to the SYN_SENT state, and then, on success, to the ESTABLISHED state.
   - If connect fails, the socket is no longer usable and must be closed. We cannot call connect again on the socket.
2. If the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for connections on the server host at the port specified (the server process is probably not running). This is a **hard error** and the error ECONNREFUSED is returned to the client as soon as the RST is received. An RST is a type of TCP segment that is sent by TCP when something is wrong. Three conditions that generate an RST are:
   - When a SYN arrives for a port that has no listening server.
   - When TCP wants to abort an existing connection.
   - When TCP receives a segment for a connection that does not exist.
3. If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered a **soft error**. The client kernel saves the message but keeps sending SYNs with the same time between each SYN as in the first scenario. If no response is received after some fixed amount of time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as either EHOSTUNREACH or ENETUNREACH.

ACCEPT()

```
int accept(int sockfd, struct sockaddr  *cliaddr, socklen_t *addrlen);
```

Return next completed connection.

- *sockfd* is socket descriptor from `socket()`
- *cliaddr* and *addrlen* return protocol address from client
- returns brand new descriptor, created by the kernel. This new descriptor refers to the TCP connection with the client.
- The **listening socket** is the first argument (*sockfd*) to accept (the descriptor created by socket and used as the first argument to both bind and listen).
- The **connected socket** is the return value from accept the connected socket.
- A given server normally creates only one listening socket, which then exists for the lifetime of the server.
- The kernel creates one connected socket for each client connection that is
- When the server is finished serving a given client, the connected socket is closed.

This function returns up to three values:

- An integer return code that is either a new socket descriptor or an error indication,
- The protocol address of the client process (through the *cliaddr* pointer),
- The size of this address (through the *addrlen* pointer).

SENDING AND RECEIVING

```
int recv(int sockfd, void *buff, size_t mbytes, int
 flags);

int send(int sockfd, void *buff, size_t mbytes, int
 flags);
```

Same as `read()` and `write()` but for *flags*

- MSG_DONTWAIT (this send non-blocking)
- MSG_OOB (out of band data, 1 byte sent ahead)
- MSG_PEEK (look, but don't remove)
- MSG_WAITALL (don't give me less than max)
- MSG_DONTROUTE (bypass routing table)

CLOSE()

```
int close(int sockfd);
```

Close socket for use.

sockfd is socket descriptor from socket()

closes socket for reading/writing

- returns (doesn't block)
- attempts to send any unsent data
- socket option SO_LINGER
  - block until data sent
  - or discard any remaining data
- returns -1 if error
- The default action of close with a TCP socket is to mark the socket as closed and return to the process immediately.
- The socket descriptor is no longer usable by the process. It cannot be used as an argument to read or write.
- But ,TCP will try to send any data that is already queued to be sent to the other end, and after this occurs, the normal TCP connection termination sequence takes place.

GETSOCKNAME AND GETPEERNAME FUNCTIONS

#include <sys/socket.h>

int getsockname (int sockfd, struct sockaddr* localaddr, socklen_t * addrlen)

Int getpeername (int sockfd, struct sockaddr* peeraddr, socklen_t * addrlen)

- **getsockname** returns local protocol address associated with a socket
- **getpeername** returns the foreign protocol address associated with a socket
- **getsockname** will return local IP/Port if unknown (TCP client calling connect without a bind, calling a bind with port 0, after accept to know the connection local IP address, but use connected socket)

GETSOCKNAME AND GETPEERNAME FUNCTIONS

Why getsockname() and getpeername() is required?

- After connect successfully returns in a TCP client that does not call bind, **getsockname**() returns the local IP address and local port number assigned to the connection by the kernel.
- After calling bind with a port number of 0 (telling the kernel to choose the local port number), **getsockname** returns the local port number that was assigned.
- **getsockname** can be called to obtain the address family of a socket.
- In a TCP server that binds the wildcard IP address, once a connection is established with a client, the server can call **getsockname** to obtain the local IP address assigned to the connection. The socket descriptor argument in this call must be that of the connected socket, and not the listening socket.
- When a server is execed by the process that calls accept, the only way the server can obtain the identity of the client is to call **getpeername**.

EXAMPLE

Daytime server and daytime client - DONE

RWServer and RWClient – Next day

Concurrent Eco server and Client - DONE

Homework1: modify concurrent server and client to make chat application.

HOMEWORK-2 (TCPQUIZ)

Implement TCP quiz for the users. Users can create question/answer pairs and submit those to server. Server saves submitted pairs to a buffer where it randomly selects next one to be asked. All clients, except the one who have entered the current question try to answer it. In every questions there should be 2 minutes limit after which the correct answers is shown to users and new questions will be drawn from the the buffer, or if the buffer is empty it will be asked from the clients
The program gets it's parameters from the command line

Server is started with:
tcpquizclientserver –p *port*

Client is started with:

tcpquizclientserver –h *serveraddress* **-p** *port* **-n** *nickname*

The client sends the nickname as the first message to the server. The server remembers the nickname, and TCP file descriptor so it knows later on which nickname belongs to which client. The server should be able to keep track of 10 simultaneous users. Length of the nickname should be limited to 10 characters.

Use select() statement to check if input is coming from the keyboard or from which of the clients.

MESSAGE RULES

Messages between client and server must follow these rules. These messages also must trigger the actions listed

Information messages from server to client

Must begin with 0 e.g. *0Welcome to the game*

Server asks for new questions from client

Must begin with 1 e.g. *1Enter new question*

This should be done only if there is no questions in the buffer

Client sends new question to server

Must begin with 2

Question part must end with ?

Must contain correct answer after the ? inside ( )

e.g. *2Who is your network programming instructor? (Madan)*

Client should be able to enter new questions at any time. If the current questions is still open, this newly submitted question should be stored into a buffer and used later on

Server sends question to other clients

Must begin with 3

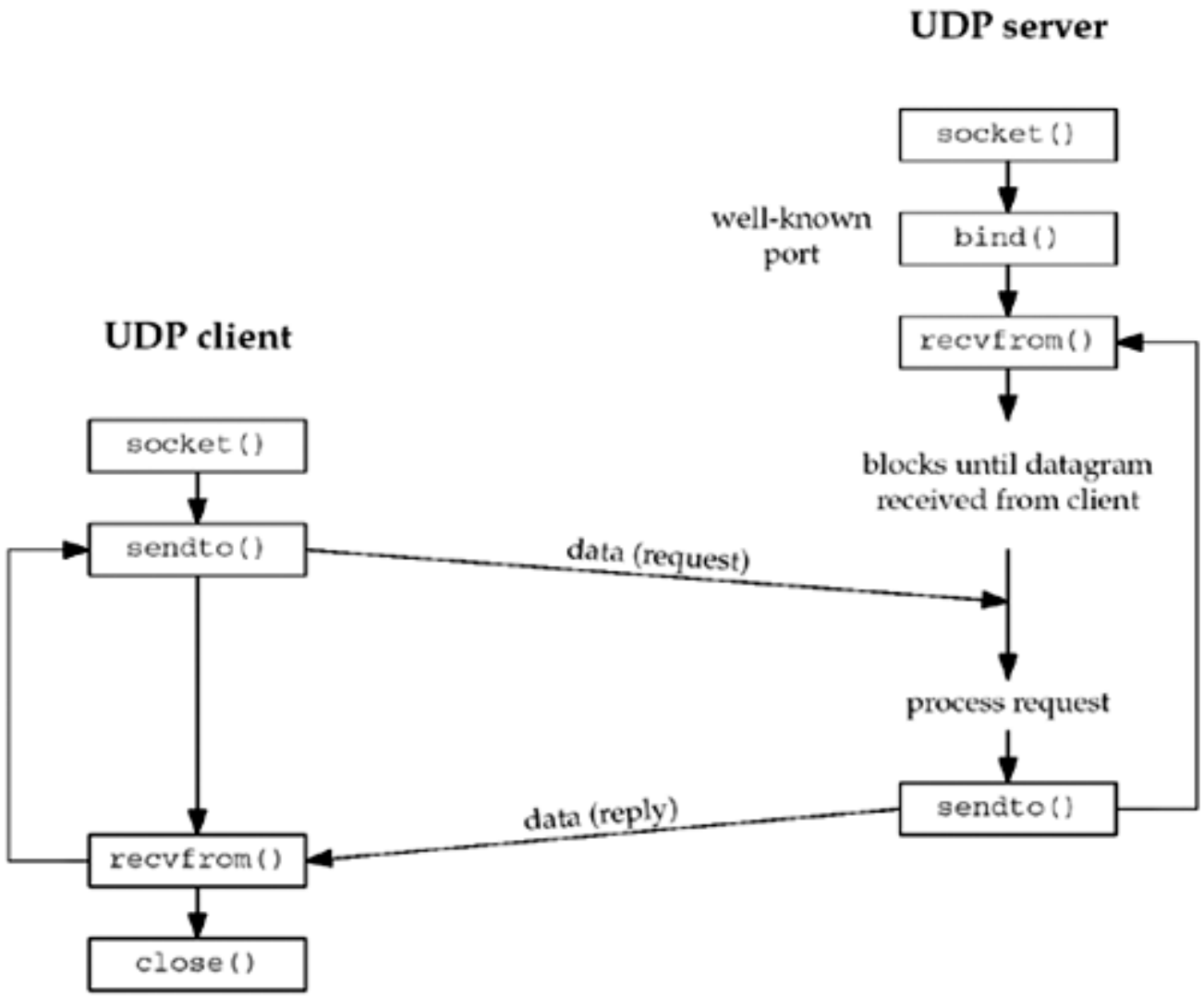
First question from the buffer should be used. If the buffer is empty --> ask for new questions from the clients

Client answers to the received question

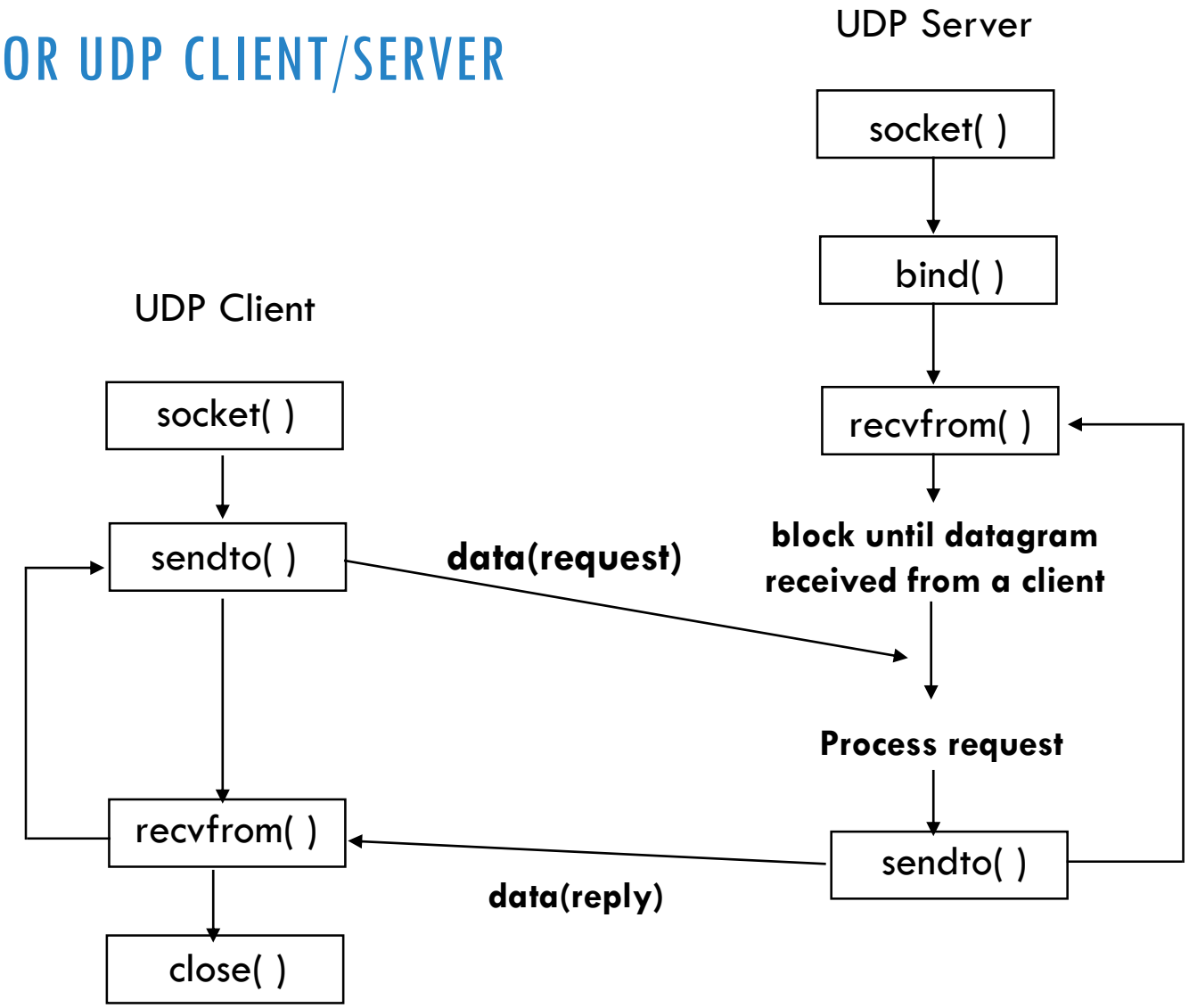
Must begin with 4

If correct answer is not found until timer expires server should tell the correct answer to clients and take a new question from the buffer

UNIX/INTERNET DOMAIN SOCKET (UDP CONNECTION)



SOCKET FUNCTIONS FOR UDP CLIENT/SERVER



RECVFROM AND SENDTO FUNCTION

```
#include<sys/socket.h>
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
                 struct sockaddr *from, socklen_t *addrlen);
ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
               const struct sockaddr *to, socklen_t addrlen);
//Both return: number of bytes read or written if OK,-1 on error
```

- The first three arguments, *sockfd, buff*, and *nbytes*, are identical to the first three arguments for **read** and : descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.
- The ***to*** argument for sendto is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is to be sent.
- The final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value-result argument).
- The final two arguments to recvfrom are similar to the final two arguments to accept: The contents of the socket address structure upon return tell us who sent the datagram (in the case of UDP) or who initiated the connection (in the case of TCP). The final two arguments to sendto are similar to the final two arguments to connect: We fill in the socket address structure with the protocol address of where to send the datagram (in the case of UDP) or with whom to establish a connection (in the case of TCP).
- Both functions return the length of the data that was read or written as the value of the function. In the typical use of recvfrom, with a datagram protocol, the return value is the amount of user data in the datagram received.

CONNECT FUNCTION WITH UDP

- We can call connect for a UDP socket. The kernel just checks for any immediate errors (e.g. an obviously unreachable destination), records the IP address and port number of the peer, and returns immediately to the calling process. Obviously, there is no three-way handshake.

- With this capability, we must now distinguish between
  - An unconnected UDP socket, the default when we create a UDP socket
  - A connected UDP socket, the result of calling connect on a UDP socket

- With a connected UDP socket, three things change, compared to the default unconnected UDP socket:
  - We can no longer specify the destination IP address and port for an output operation. We do not use **sendto**, but **write** or **send** instead. Anything written to a connected UDP socket is automatically sent to the protocol address (e.g., IP address and port) specified by connect.
    - Similar to TCP, we can call sendto for a connected UDP socket, but we cannot specify a destination address. The fifth argument to sendto (the pointer to the socket address structure) must be a null pointer, and the sixth argument (the size of the socket address structure) should be 0. The POSIX specification states that when the fifth argument is a null pointer, the sixth argument is ignored.

CONNECT FUNCTION WITH UDP…

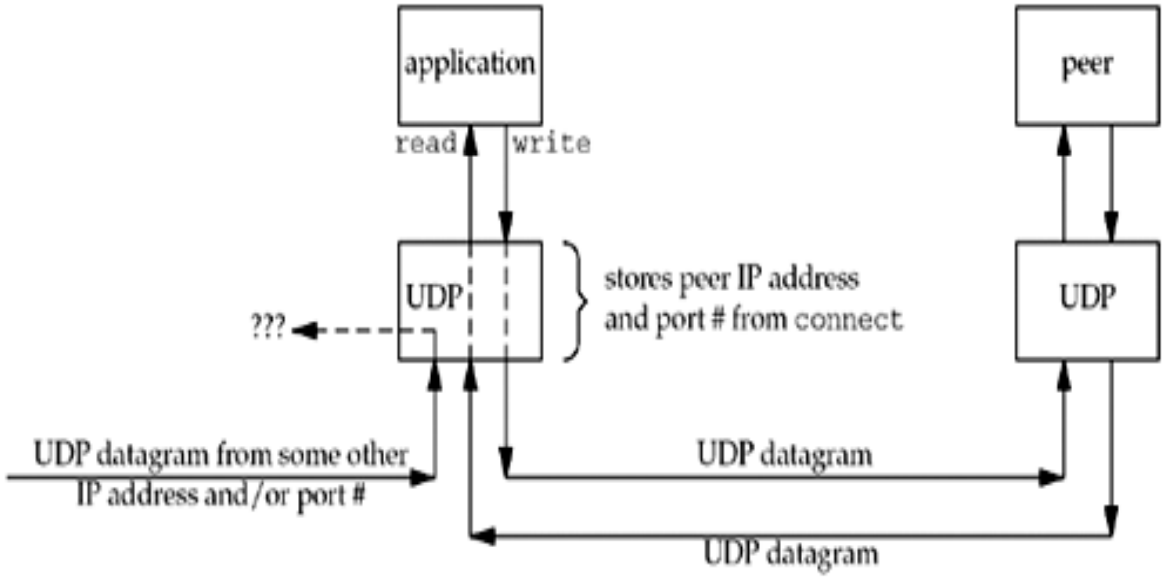
- We do not need to use **recvfrom** to learn the sender of a datagram, but **read**, **recv**, or **recvmsg** instead. The only datagrams returned by the kernel for an input operation on a connected UDP socket are those arriving from the protocol address specified in connect. Datagrams destined to the connected UDP socket's local protocol address (e.g., IP address and port) but arriving from a protocol address other than the one to which the socket was connected are not passed to the connected socket.
  - Technically, a connected UDP socket exchanges datagrams with only one IP address, because it is possible to connect to a multicast or broadcast address.
- Asynchronous errors are returned to the process for connected UDP sockets.

Type of socket	write or send	sendto that does not specify a destination	sendto that specifies a destination
TCP socket	OK	OK	EISCONN
UDP socket, connected	OK	OK	EISCONN
UDP socket, unconnected	EDESTADDRREQ	EDESTADDRREQ	OK

Table: TCP and UDP sockets: can a destination protocol address be specified?

CONNECT FUNCTION WITH UDP…

- The application calls connect, specifying the IP address and port number of its peer. It then uses read and write to exchange data with the peer.
- Datagrams arriving from any other IP address or port (??? in Figure) are not passed to the connected socket because either the source IP address or source UDP port does not match the protocol address to which the socket is connected. These datagrams could be delivered to some other UDP socket on the host. If there is no other matching socket for the arriving datagram, UDP will discard it and generate an ICMP "port unreachable" error.
- In summary, UDP client or server can call connect only if that process uses the UDP socket to communicate with exactly one peer. Normally, it is a UDP client that calls connect, but there are applications in which the UDP server communicates with a single client for a long duration (e.g., TFTP); in this case, both the client and server can call connect.



UNIX DOMAIN SOCKETS/PROTOCOLS

- The Unix domain protocols are not an actual protocol suite, but a way of performing client/server communication on a single host using the same API that is used for clients and servers on different hosts.
- Two types of sockets are provided in the Unix domain: **stream sockets** (similar to TCP) and **datagram sockets** (similar to UDP).

Unix domain sockets are used for three reasons:

- On Berkeley-derived implementations, Unix domain sockets are often twice as fast as a TCP socket when both peers are on the same host.
- Unix domain sockets are used when passing descriptors between processes on the same host.
- Newer implementations of Unix domain sockets provide the client's credentials to the server, which can provide additional security checking.

UNIX DOMAIN SOCKET ADDRESS STRUCTURE

The Unix domain socket address structure, which is defined by including the <sys/un.h> header, is:

```
struct sockaddr_un {
        sa_family_t sun_family; /* AF_LOCAL */
        char sun_path[104]; /* null-terminated pathname */
};
```

- The pathname stored in the sun_path array must be null-terminated. The macro SUN_LEN is provided and it takes a pointer to a sockaddr_un structure and returns the length of the structure, including the number of non-null bytes in the pathname.

- The unspecified address is indicated by a null string as the pathname, that is, a structure with sun_path[0] equal to 0. This is the Unix domain equivalent of the IPv4 INADDR_ANY constant and the IPv6 IN6ADDR_ANY_INIT constant.

EXAMPLE: BIND OF UNIX DOMAIN SOCKET

```
int main(int argc, char ** argv[]) {
            int sockfd;
            socklen_t len;
            struct sockaddr_un addr1, addr2;
            if (argc != 2)
                        err_quit("usage: unixbind <pathname>");
            sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);
unlink (argv[1]); // OK if this fails
bzero(&addr,sizeof(addr1));
addr1.sun_family = AF_LOCAL;
strncpy(addr1.sun_path, argv[1], sizeof(addr1.sun_path) -1);
bind(sockfd, (SA *) &addr1, SUN_LEN(&addr1));
len = sizeof(addr2);
getsockname(sokfd, (SA *)  &addr2, &len);
printf("bound name = %s, returned len = %d\n", addr2_sun_path, len);
exit(0);
}
```

SOCKETPAIR() FUNCTION

- The **socketpair()** function creates two sockets that are connected together. This function applies only to Unix domain sockets.

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

  int socketpair(int family, int type, int protocol, int sockfd[2]);

  Returns: nonzero if OK, -1 on error
```

- Family: AF_LOCAL and the protocol must be 0.
- Type: either SOCK_STREAM or SOCK_DGRAM. The two socket descriptors that are created are returned as sockfd[0] and sockfd[1];
- The two created sockets are unnamed; that is; there is no implicit bind involved.
- The result of **scoketpair()** with a type of SOCK_STREAM is called a stream pipe. The stream pipe is full-duplex; that is, both descriptors can be read and written.

SOCKET FUNCTIONS

Differences and restrictions in the socket functions when using Unix domain sockets.

- The default file access permissions for a pathname created by bind should be 0777 (read, write, and execute by user group, group, and other), modified by the current umask value.
- The pathname associated with a Unix domain socket should be an absolute pathname, not a relative pathname.
- The pathname specified in a call to connect must be a pathname that is currently bound to an open Unix domain socket of the same type (stream or datagram).
- Unix domain stream sockets are similar to TCP sockets. They provide a byte stream interface to the process with no record boundaries.
- If a call to connect for a Unix domain stream socket finds that the listening socket's queue is full, ECONNREFUSED is returned immediately.
- Unix domain datagram sockets are similar to UDP sockets. They provide an unreliable datagram service that preserves record boundaries.
- Sending a datagram on an unbound Unix domain datagram socket does not bind a pathname to the socket. Calling connect for a Unix domain datagram socket does not bind a pathname to the socket.

NAME AND ADDRESS CONVERSION
DOMAIN NAME SYSTEM(DNS)

- The DNS is used primarily to map between hostnames and IP addresses.
- A hostname can be either a simple name, such as solaris or freebsd, or a fully qualified domain name (FQDN), such as solaris.unpbook.com.
- Entries in the DNS are known as resource records (RRs).
- Domain name is converted to IP address by contacting a DNS server by calling functions in a library known as the resolver.

GETHOSTBYNAME() FUNCTION

- This function is used to convert hostname to IP address.

  #include <netdb.h>

  struct hostent * gethostbyname(const char * hostname);

  Returns: non-null pointer if OK, NULL on error with h_errno set

- The non-null pointer returned by this function points to the following **hostent** structure.

```
struct hostent {
 char  *h_name;       /* official (canonical) name of host */
 char **h_aliases;    /* pointer to array of pointers to alias names */
 int    h_addrtype;   /* host address type: AF_INET */
 int    h_length;     /* length of address: 4 */
 char **h_addr_list;  /* ptr to array of ptrs with IPv4 addrs */
};
```

- This function can return only IPv4 addresses.

GETHOSTBYNAME() FUNCTION ...

- Gethostbyname() differs from the other socket functions that it does not set errno when an error occurs. Instead, it sets the global integer h_errno to one of the following constants defined by including <netdb.h>:

  - HOST_NOT_FOUND
  - TRY_AGAIN
  - NO_RECOVERY
  - NO_DATA (identical to NO_ADDRESS)

- The **NO_DATA** error means the specified name is valid, but it does not have an A record.

GETHOSTBYADDR() FUNCTION

- The function **gethostbyaddr()** takes a binary IPv4 address and tries to find the host-name corresponding to that address. This is the reverse of **gethostbyname**.

```
#include <netdb.h>

struct hostent *gethostbyaddr (const char *addr, socklen_t len, int family);

                    Returns: non-null pointer if OK, NULL on error with h_errno set
```

- The addr argument is not a char *, but is really a pointer to an in_addr structure containing the IPv4 address.
- The len is the size of this structure: 4 for an IPv4 address. The family argument is AF_INET.

GETSERVBYNAME() AND GETSERVBYPORT() FUNCTIONS

Services, like hosts, are often known by names, too.

```
#include <netdb.h>
struct servent *getservbyname (const char *servname, const char *protoname);
```

Returns: non-null pointer if OK, NULL on error
This function returns a pointer to the following structure.

```
struct servent {
char *s_name; /* official service name */
char **s_aliases; /* alias list */
int s-port; /* port number, network-byte order */
char *s_proto; /* protocol to use */
};
```

- The service name servname must be specified. If a protocol is also specified (protoname is a non-null pointer), then the entry must also have a matching protocol.

GETSERVBYPORT()

Getservbyport(), looks up a service given its port number and an optional protocol.

```
#include <netdb.h>
struct servent *getservbyport (int port, const char *protoname);
```

Returns: non-null pointer if OK, NULL on error

- The *port* value must be network byte ordered. Typical calls to this function could be as follows:

```
struct servent *sptr;
sptr = getservbyport (htons (53), "udp"); /* DNS using UDP */
sptr = getservbyport (htons (21), "tcp"); /* FTP using TCP */
 sptr = getservbyport (htons (21), NULL); /* FTP using TCP */
sptr = getservbyport (htons (21), "udp"); /* this call will fail */
```

GETADDRINFO() FUNCTION

- The gethostbyname and gethostbyaddr functions only support IPv4.
- The getaddrinfo supports both IPv4 and IPv6.
- The getaddrinfo function handles both name-to-address and service-to-port translation, and returns sockaddr structures instead of a list of addresses.
- These sockaddr structures can then be used by the socket functions directly.

```
#include <netdb.h>

int getaddrinfo (const char *hostname, const char *service, const struct addrinfo *hints, struct addrinfo **result) ;
```

Returns: 0 if OK, nonzero on error

- This function returns through the result pointer a pointer to a linked list of addrinfo structures, which is defined by including <netdb.h>.

```
struct addrinfo {
int ai_flags; /* AI_PASSIVE, AI_CANONNAME */
int ai_family; /* AF_xxx */
int ai_socktype; /* SOCK_xxx */
int ai_protocol; /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
socklen_t ai_addrlen; /* length of ai_addr */
char *ai_canonname; /* ptr to canonical name for host */
struct sockaddr *ai_addr; /* ptr to socket address structure */
struct addrinfo *ai_next; /* ptr to next structure in linked list */
};
```

- The hostname is either a hostname or an address string (dotted-decimal for IPv4 or a hex string for IPv6). The service is either a service name or a decimal port number string. hints is either a null pointer or a pointer to an addrinfo structure that the caller fills in with hints about the types of information the caller wants returned.

GAI_STRERROR FUNCTION

- The non zero error return values from getaddrinfo have specific names and meanings.
- The function gai_strerror takes one of these values (int) as an argument and returns a pointer to the corresponding error string.

```
#include <netdb.h>

const char *gai_strerror (int error);
```

Returns: pointer to string describing error message

Nonzero error return constants from getaddrinfo.

Constant	Description
EAI_AGAIN	Temporary failure in name resolution
EAI_BADFLAGS	Invalid value for ai_flags
EAI_FAIL	Unrecoverable failure in name resolution
EAI_FAMILY	ai_family not supported
EAI_MEMORY	Memory allocation failure
EAI_NONAME	*hostname* or *service* not provided, or not known
EAI_OVERFLOW	User argument buffer overflowed (*getnameinfo*() only)
EAI_SERVICE	*service* not supported for ai_socktype
EAI_SOCKTYPE	ai_socktype not supported
EAI_SYSTEM	System error returned in errno

FREEADDRINFO FUNCTION

- All the storage returned by getaddrinfo is obtained dynamically. This storage is returned by calling freeaddrinfo.

```
#include <netdb.h>

void freeaddrinfo (struct addrinfo *ai);
```

- ai should point to the first addrinfo structure returned by getaddrinfo.
- All the structures in the linked list are freed.

SIGNAL HANDLING

(POSIX) SIGNAL HANDLING

- A signal is a notification to a process that an event has occurred. Signals are sometimes called software interrupts. Signals usually occur asynchronously.
- Signals can be sent
  - By one process to another process (or to itself)
  - By the kernel to a process
- The **SIGCHLD** signal is one that is sent by the kernel whenever a process terminates, to the parent of the terminating process.
- Every signal has a disposition, which is also called the action associated with the signal. We set the disposition of a signal by calling the **sigaction** function. We have three choices for the disposition.
  1. We can provide a function that is called whenever a specific signal occurs. This function is called a signal handler and the action is called catching a signal. The two signals **SIGKILL** & **SIGSTOP** cannot be caught.
  2. We can ignore a signal by setting its disposition to **SIG_IGN**. The two signals **SIGKILL** and **SIGSTOP** cannot be ignored.
  3. We can set the default disposition for a signal by setting its disposition to **SIG_DFL**. There are few signals whose default disposition is to be ignored. **SIGCHLD** and **SIGURG**.

Function prototypes for signal handler functions

ANSI C:

```
void handler(int);
```

POSIX SA_SIGINFO:

```
void handler(int, siginfo_t *info, ucontext_t *uap);
```

SIGACTION FUNCTION

- sigaction returns the old action for the signal as the return value of the signal function.

#include<signal.h>

int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact);

The struct sigaction is

```
struct  sigaction {
union __sigaction_u __sigaction_u;  // signal handler
              sigset_t sa_mask;     // signal mask to apply
              int      sa_flags;    // signal options
};
union __sigaction_u {
              void    (*___sa_handler)(int);
              void    (*___sa_sigaction)(int, siginfo_t *, void *);
};
#define sa_handler      __sigaction_u.__sa_handler
#define sa_sigaction    __sigaction_u.___sa_sigaction
```

- The sig is the signal to be captured. The act is the information about signal handling function, masked signal and flags. The oact is the information about the previous signal handling function, masked signal and flags.

- The sa_mask in struct sigaction takes the signal to be masked when the handler function is called on arrival of the signal. The sa_flags in struct sigaction sets flags. E.g., Setting sa_flags to **SA_SIGINFO** uses **POSIX** signal handler function.

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal

16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor
24	SIGXCPU	terminate process	cpu time limit exceeded (see
25	SIGXFSZ	terminate process	file size limit exceeded (see
26	SIGVTALRM	terminate process	virtual time alarm (see
27	SIGPROF	terminate process	profiling timer alarm (see
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

POSIX SIGNAL SEMANTICS

- Once a signal handler is installed, it remains installed.
- While a signal handler is executing, the signal being delivered is blocked. Furthermore, any additional signals that were specified in the **sa_mask** signal set passed to **sigaction** when the handler was installed are also blocked.
- If a signal is generated one or more times while it is blocked, it is normally delivered only one time after the signal is unblocked. That is, by default, Unix signals are not queued.
- It is possible to selectively block and unblock a set of signals using the **sigprocmask** function.

WAIT AND WAITPID FUNCTION

- The **wait()** and **waitpid()** functions are called to handle the terminated child.
- **wait()** and **waitpid()** both return two values: the return value of the function is the process ID of the terminated child, and the termination status of the child (an integer) is returned through the **statloc** pointer.

```
#include <sys/wait.h>
 pid_t wait(int *statloc);
 pid_t waitpid(pid_t pid, int *statloc,int options);
 Both return; process ID if OK, or -1 on error
```

- If there are no terminated children for the process calling wait, but the process has one or more children that are still executing, then wait blocks until the first child of the existing children terminates.
- The **waitpid()** gives us more control over which process to wait for and whether or not to block. First, the pid argument lets us specify the proces ID that we want to wait for. A value of -1 says to wait for the first of our children to terminate. The options argument lets us specify additional options.

SOCKET SYSTEM CALLS

Write to socket function variants

```
ssize_t write(int fildes, void *buf, size_t nbyte);
#include <sys/socket.h>
ssize_t send(int socket, const void *buffer, size_t length, int flags);
ssize_t sendmsg(int socket, const struct msghdr *message, int flags);

ssize_t sendto(int socket, const void *buffer, size_t length, int flags,
          const struct sockaddr *dest_addr, socklen_t dest_len);
// sendto =  write + connect
```

- Upon successful completion, the number of bytes that were sent is returned. Otherwise, -1 is returned and the global variable errno is set to indicate the error.

Read from socket function variants

```
ssize_t read(int fildes, void *buf, size_t nbyte);
#include <sys/socket.h>

ssize_t recv(int socket, void *buffer, size_t length, int flags);

ssize_t recvfrom(int socket, void *restrict buffer, size_t length, int flags, struct
sockaddr *restrict address, socklen_t *restrict address_len);
ssize_t recvmsg(int socket, struct msghdr *message, int flags);
```

- These calls return the number of bytes received, or -1 if an error occurred. For TCP sockets, the return value 0 means the peer has closed its half side of the connection.

ERROR RELATED FUNCTIONS

```
void perror(const char *); // print string equivalent of global error number variable

char * strerror(int ); // returns string equivalent of given error number
```

PASSING (FILE) DESCRIPTORS

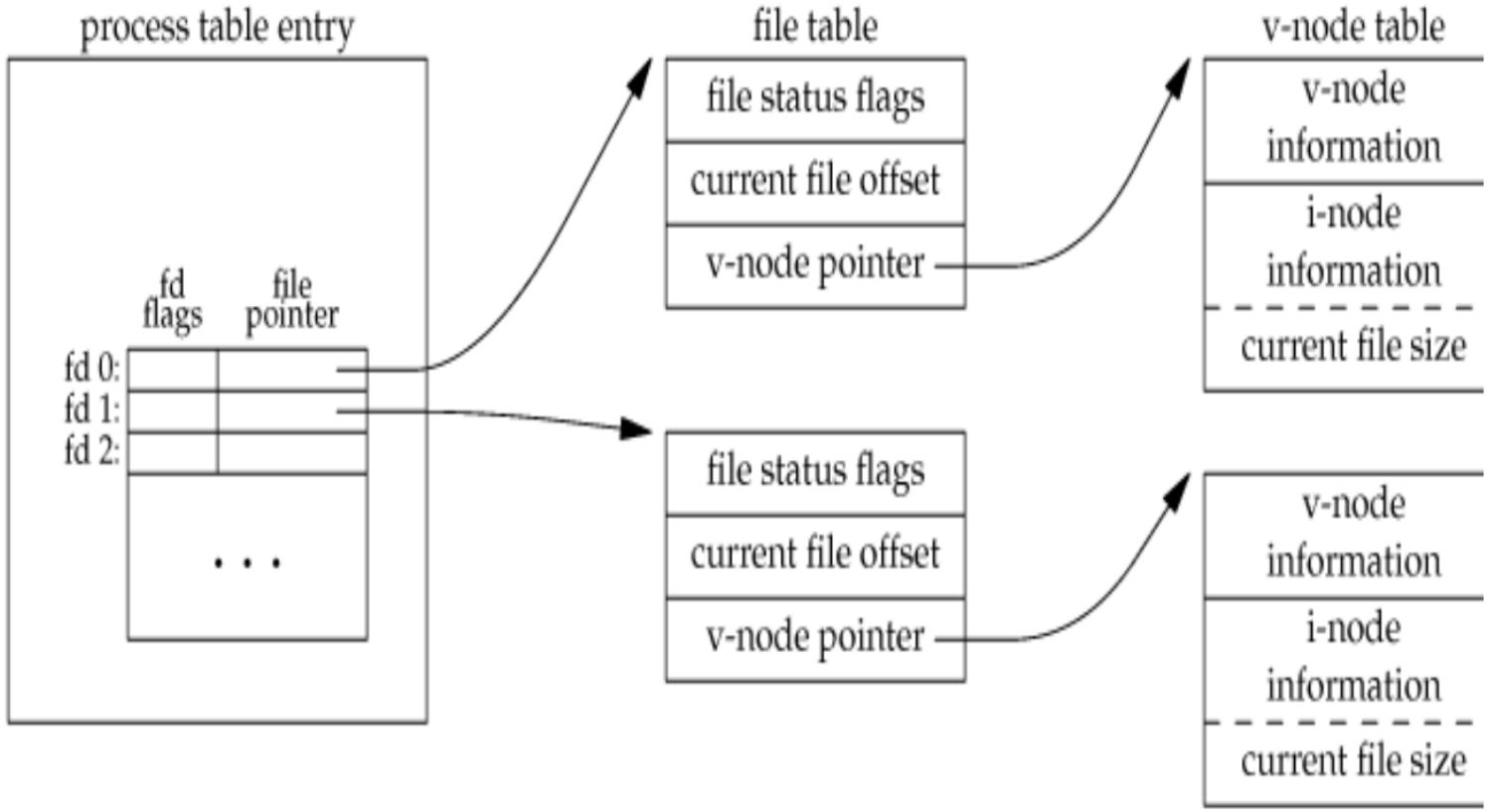
OPEN FILE IN UNIX SYSTEM

The kernel uses three data structures to represent an open file.

1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor. Associated with each file descriptor are
   - The file descriptor flags
   - A pointer to a file table entry

2. The kernel maintains a file table for all open files. Each file table entry contains
   - The file status flags for the file, such as read, write, append, sync, and nonblocking;
   - The current file offset
   - A pointer to the v-node table entry for the file

3. Each open file (or device) has a v-node structure that contains information about the type of file and pointers to functions that operate on the file. For most files, the v-node also contains the i-node for the file. This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available. For example, the i-node contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk, etc.



The figure shows a pictorial arrangement of these three tables for a single process that has two different files open.

PASSING A FILE DESCRIPTOR

- Passing an open descriptor from one process to another takes place as
  - A child sharing all the open descriptors with the parent after a call to fork
  - All descriptors normally remaining open when exec is called.
- Current Unix systems provide a way to pass any open descriptor from one process to any other process.
- The technique requires us to first establish a Unix domain socket between the two processes and then use **sendmsg** to send a special message across the Unix domain socket.
- This message is handled specially by the kernel, passing the open descriptor from the sender to the receiver.

Steps involved in passing a descriptor between two processes

- Create a Unix domain socket, either a stream socket or a datagram socket.
  - If the goal is to fork a child and have the child open the descriptor and pass the descriptor back to the parent, the parent can call **socketpair** to create a stream pipe that can be used to exchange the descriptor.
  - If the processes are unrelated, the server must create a Unix domain stream socket and bind a pathname to it, allowing the client to connect to that socket.

Steps involved in passing a descriptor between two processes(contd...)

- One process opens a descriptor by calling any of Unix functions that returns a descriptor. Any type of descriptor can be passed from one process to another.
- The sending process builds a **msghdr** structure containing the descriptor to be passed. The sending process calls **sendmsg** to send the descriptor across the Unix domain socket.
  - At this point, the descriptor is "in flight". Even if the sending process closes the descriptor after calling **sendmsg**, but before the receiving process calls **recvmsg**, the descriptor remains open for the receiving process. Sending a descriptor increments the descriptor's reference count by one.
- The receiving process calls **recvmsg** to receive the descriptor on the Unix domain socket. It is normal for the descriptor number in the receiving process to differ from the descriptor number in the sending process. Passing a descriptor involves creating a new descriptor in the receiving process that refers to the same file table entry within the kernel as the descriptor that was sent by the sending process.

I/O MODEL

INTRODUCTION 1/2

TCP echo client is handling two inputs at the same time: standard input and a TCP socket

- when the client was blocked in a call to read, the server process was killed
- server TCP sends FIN to the client TCP, but the client never sees FIN since the client is blocked reading from standard input
  - ✓We need the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready.
  - ✓I/O multiplexing (**select**, **poll**, or newer **pselect** functions)

INTRODUCTION 2/2

Scenarios for I/O Multiplexing

- client is handling multiple descriptors (interactive input and a network socket).
- Client to handle multiple sockets (rare)
- TCP server handles both a listening socket and its connected socket.
- Server handle both TCP and UDP.
- Server handles multiple services and multiple protocols

I/O MODELS

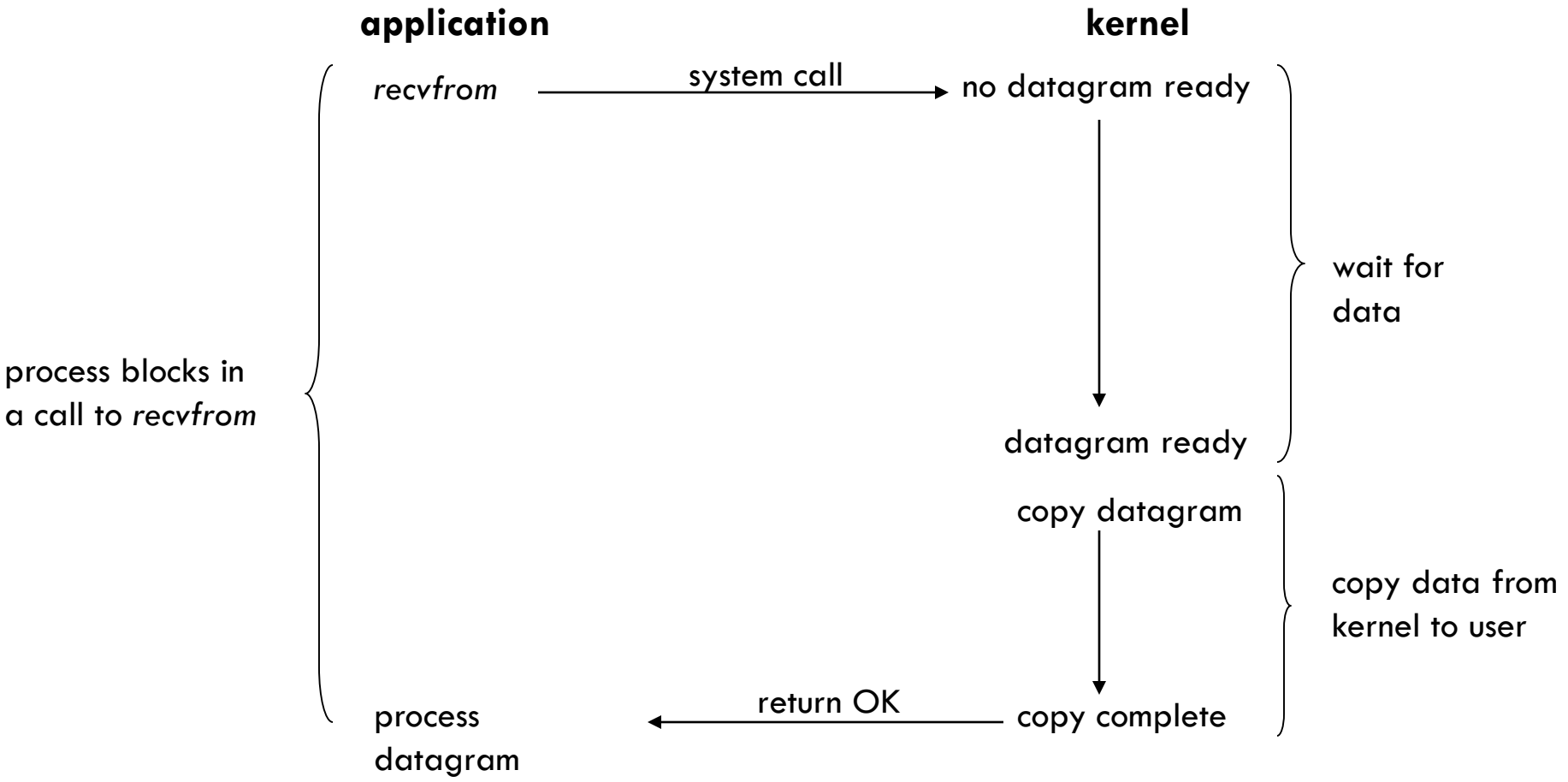
Models

1. Blocking I/O
2. Nonblocking I/O
3. I/O multiplexing(**select** and **poll**)
4. Signal driven I/O (**SIGIO**)
5. Asynchronous I/O

Two *distinct phases* for an input operation

- Waiting for the data to be ready (for a socket, wait for the data to arrive on the network, then copy into a buffer within the kernel)
- Copying the data from the kernel to the process (from kernel buffer into application buffer)

BLOCKING I/O MODEL



BLOCKING I/O MODEL...

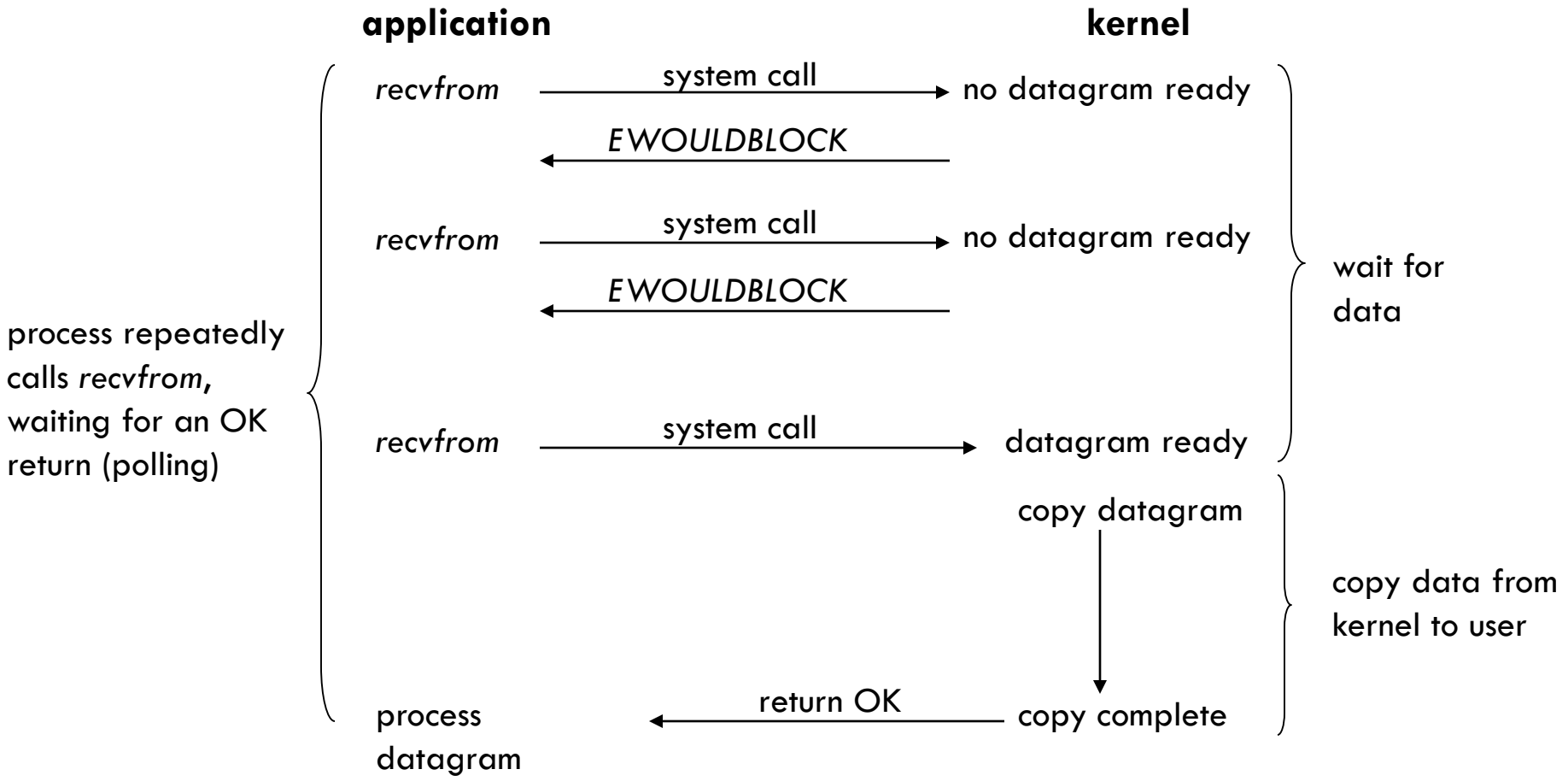
- By default, all sockets are blocking.

- The process calls **recvfrom** and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs.

- We say that our process is blocked the entre time from when it calls **recvfrom** until it returns.

- When **recvfrom** returns successfully, our application process the datagram.

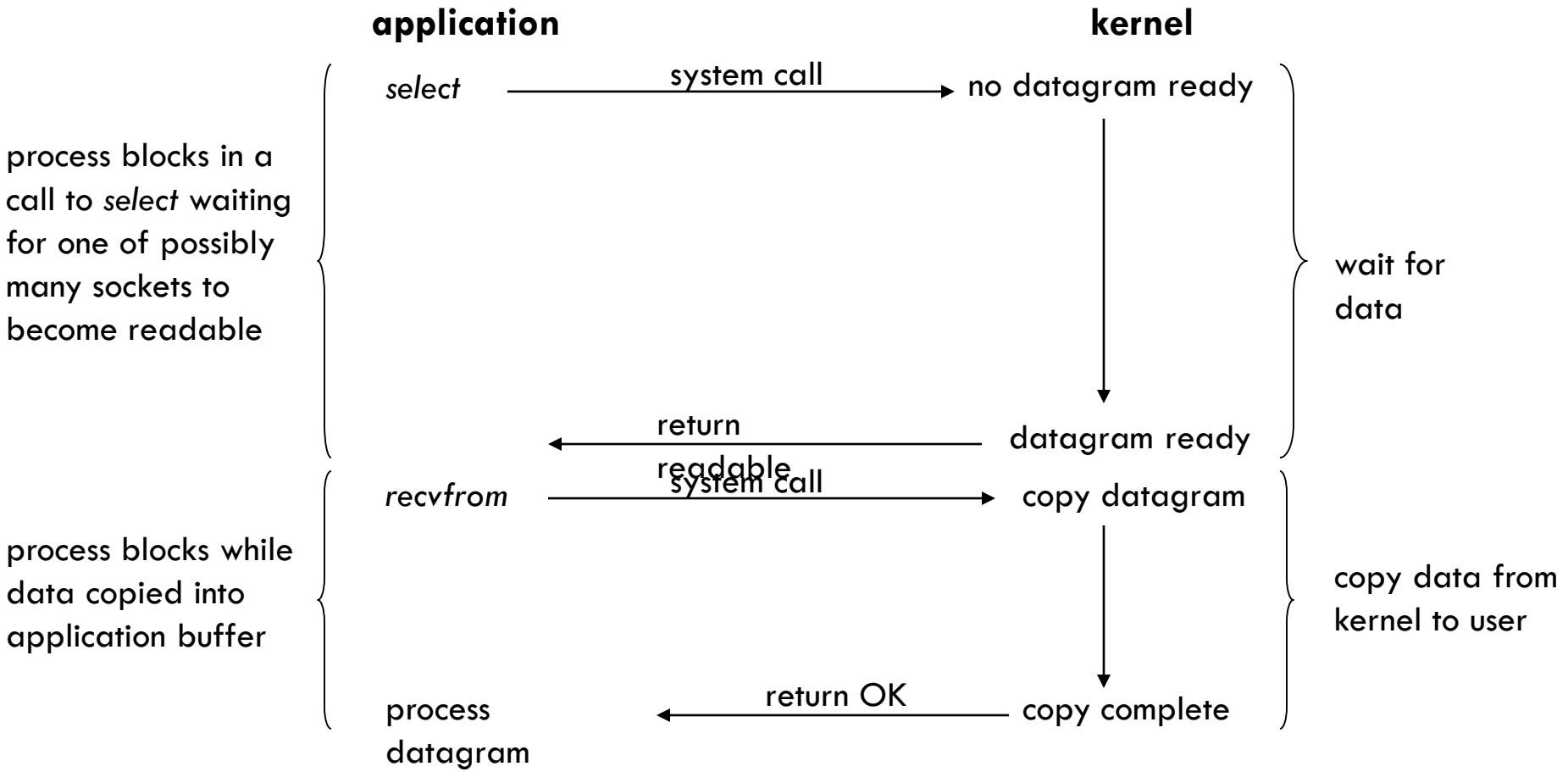
NON-BLOCKING I/O MODEL



NON-BLOCKING I/O MODEL…

- When a socket is non-blocking, It instruct the kernel as "when an I/O operation that the process requests cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead."
- The first three times that we call **recvfrom**, there is no data to return, so the kernel immediately returns an error of EWOULDBLOCK instead.
- The fourth time we call **recvfrom**, a datagram is ready, it is copied into our application buffer, and **recvfrom** returns successfully.
- We then process data. When an application sits in a loop calling **recvfrom** on a non-blocking descriptor like this, it is called polling.
- The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time.

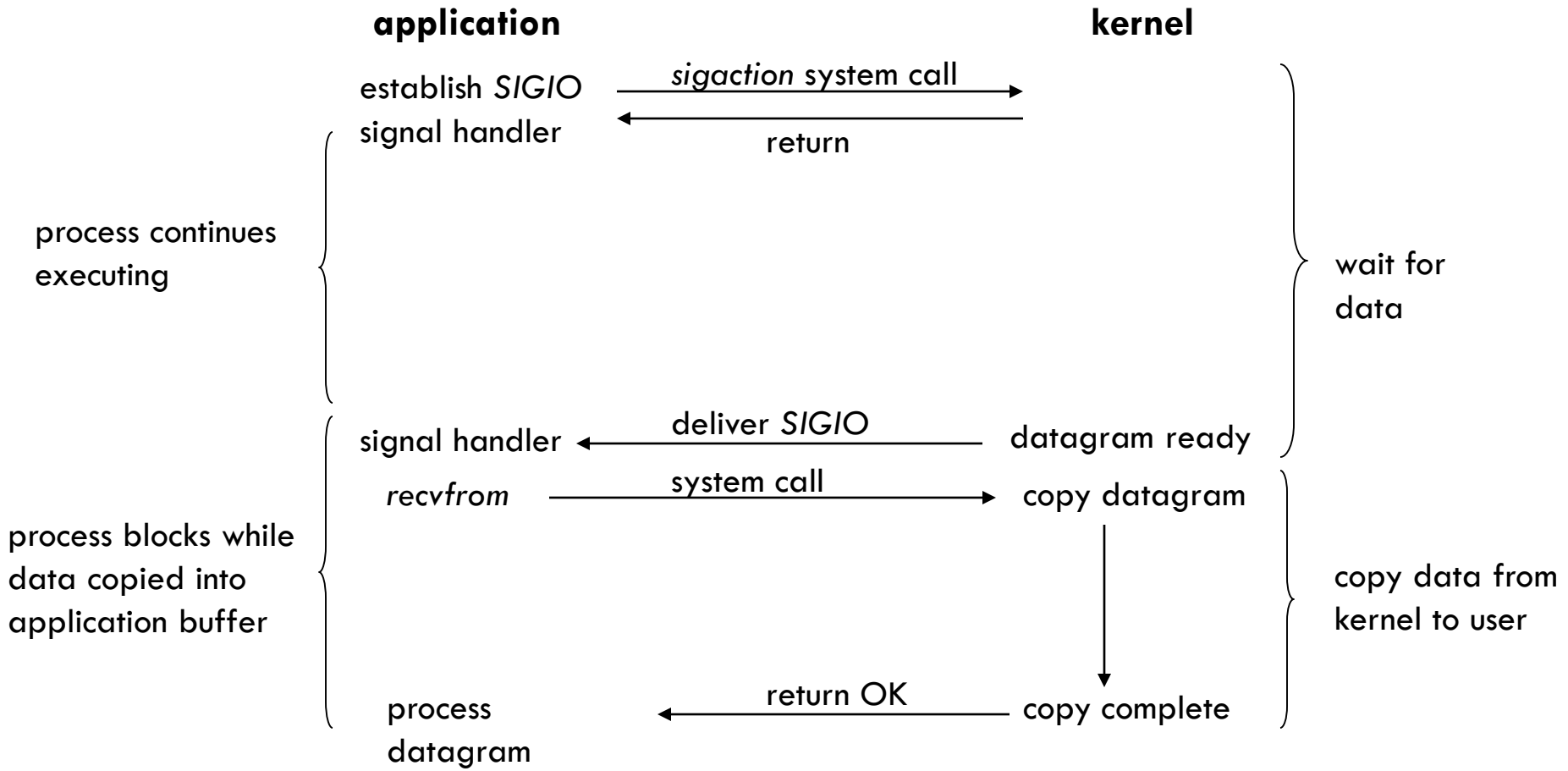
I/O MULTIPLEXING MODEL



I/O MULTIPLEXING MODEL...

- With I/O multiplexing, we call **select** or **poll** and block in one of these two system calls, instead of blocking in the actual I/O system call.
- We block in a call to **select**, waiting for the datagram socket to be readable.
- When **select** returns that the socket is readable, we then call **recvfrom** to copy the datagram into our application buffer.
- With **select**, we can wait for more than one descriptor to be ready.

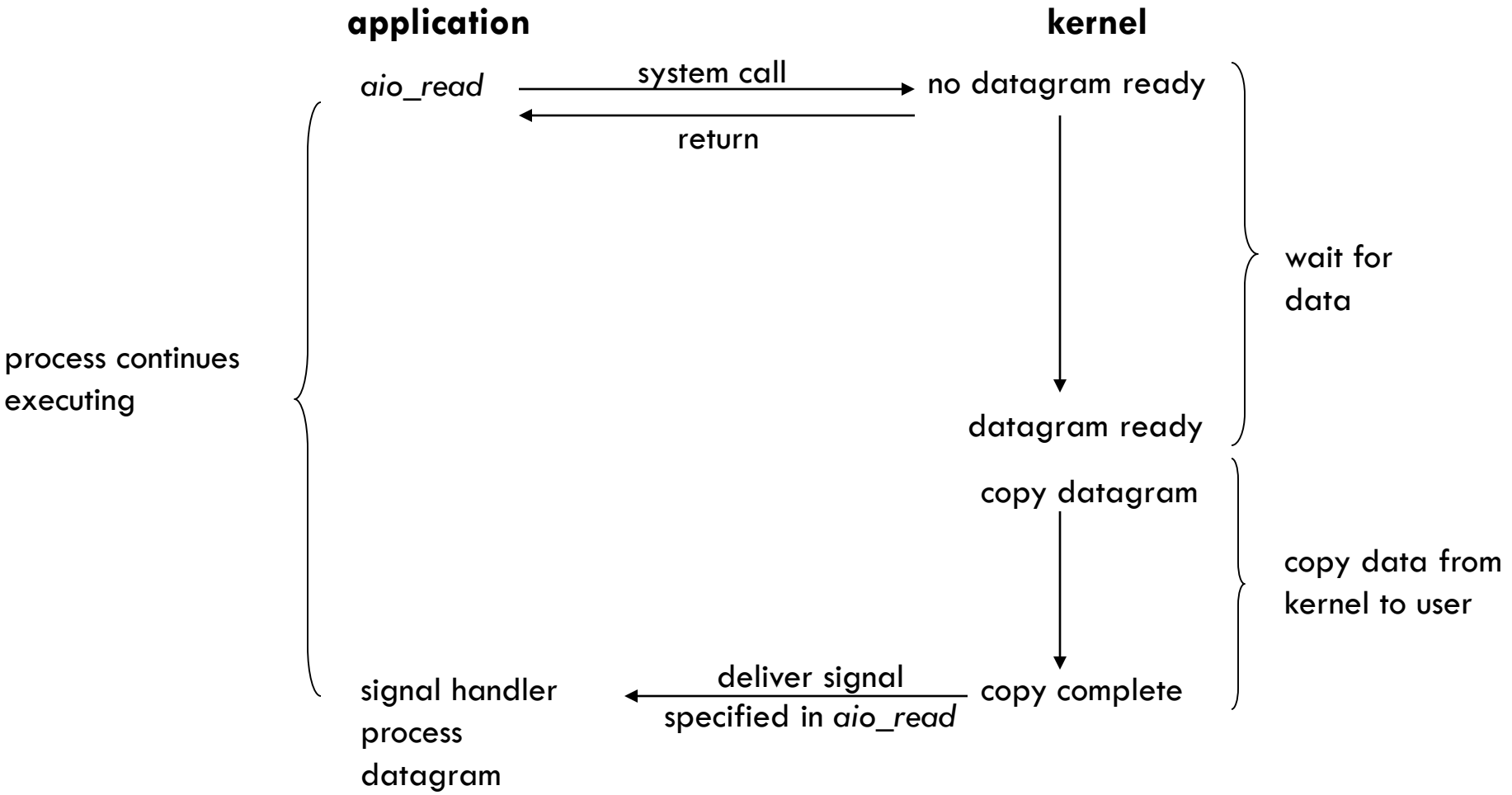
SIGNAL DRIVEN I/O MODEL



SIGNAL DRIVEN I/O MODEL…

- A **SIGIO signal** is used to tell the kernel when the descriptor is ready. We call this signal-driven I/O.
- We first enable the socket for the signal-driven I/O and install a signal handler using the **sigaction** system call.
- The return from this system call is immediate and our process continues; it is not blocked.
- When the datagram is ready to be read, the **SIGIO** signal is generated for our process. We can then read the data.
- The advantage of this model is that we are not blocked while waiting for the datagram to arrive.
- The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

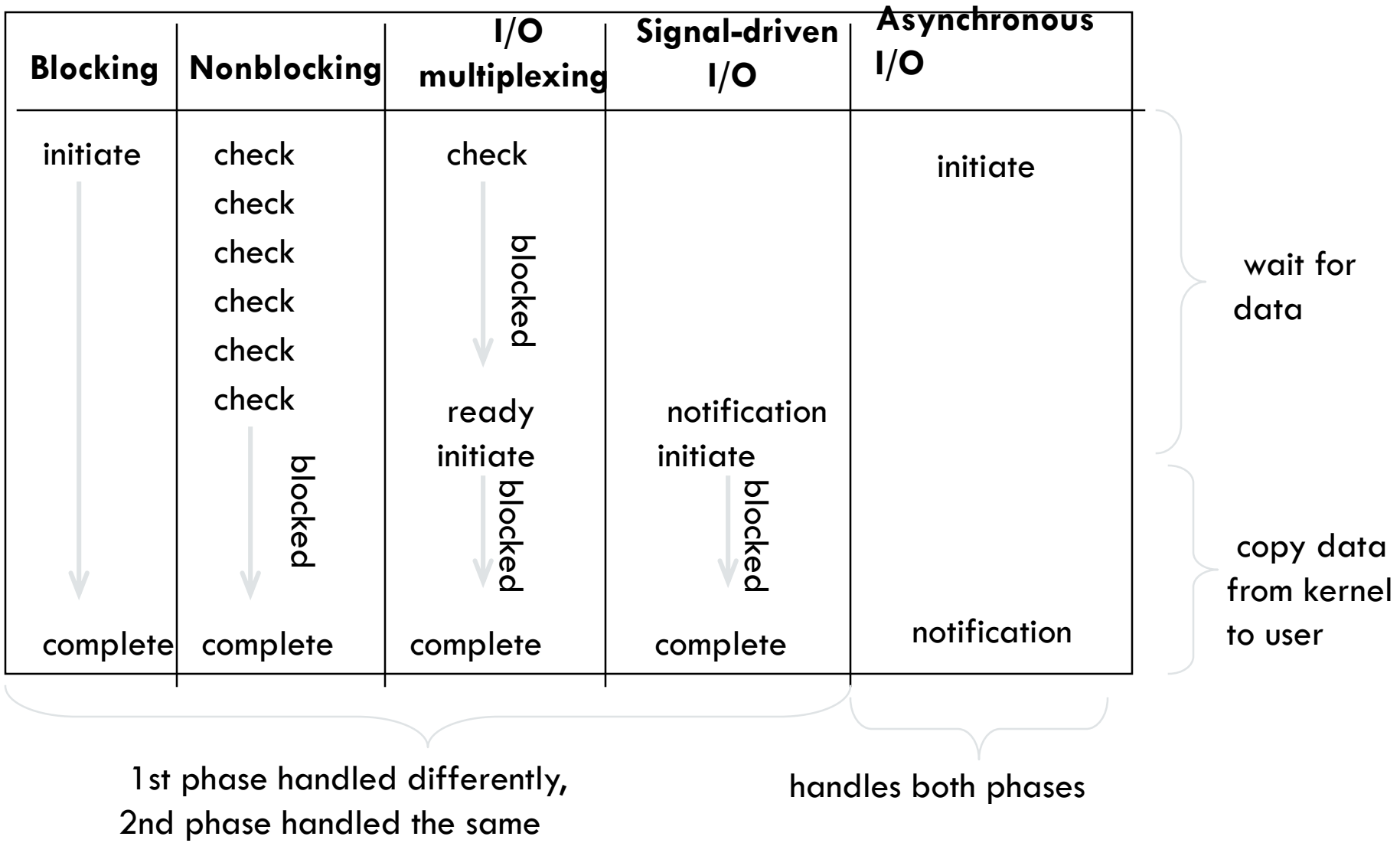
ASYNCHRONOUS I/O MODEL



ASYNCHRONOUS I/O MODEL

- Asynchronous I/O is defined by the POSIX specification.
- These functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete.
- The main difference between this model and the signal-driven I/O model is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/ operation is complete.
- We call **aio_read** and pass the kernel the descriptor, buffer pointer, buffer size, file offset, and how to notify us when the entire operation is complete.
- This system call returns immediately and our process is not blocked while waiting for the I/O to complete.

COMPARISON OF THE I/O MODELS



SYNCHRONOUS I/O , ASYNCHRONOUS I/O

Synchronous I/O

- causes the requesting process to be blocked until that I/O operation (recvfrom) completes. (blocking, nonblocking, I/O multiplexing, signal-driven I/O)

Asynchronous I/O

- does not cause the requesting process to be blocked

SELECT FUNCTION

- Allows the process to instruct the kernel to *wait for any one of multiple events to occur* and to wake up the process only when one or more of these events occurs or *when a specified amount of time has passed*.
- What descriptors we are interested in (readable ,writable , or exception condition) and how long to wait?

```
#include <sys/select.h>
     #include <sys/time.h>
     int select (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set
 *exceptset, const struct timevaI *);
//Returns: +ve count of ready descriptors, 0 on timeout, -1 on error
struct timeval{
          long  tv_sec;  /* seconds */
          long  tv_usec; /* microseconds */ }
```

- The final argument, timeout, tells the kernel how long to wait for one of the specified file descriptors to become ready. A timeval structure specifies the number of seconds and microseconds.

POSSIBILITIES FOR SELECT FUNCTION

1. Wait forever : return only when descriptor (s) is ready (specify **timeout** argument as NULL)
2. wait up to a fixed amount of time: Return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the timeval structure pointed to by the timeout argument.
3. Do not wait at all : return immediately after checking the descriptors(called Polling) (specify **timeout** argument as pointing to a **timeval** structure where the timer value is 0)

- The wait is normally interrupted if the process catches a signal and returns from the signal handler
  - **select** might return an error of **EINTR**
  - Actual return value from function = -1

RETURN VALUE OF SELECT

- Select() returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred. If the time limit expires, select() returns 0. If select() returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified and the global variable errno will be set to indicate the error.

Error	**Description**
[EAGAIN]	The kernel was (perhaps temporarily) unable to allocate the requested number of file descriptors.
[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.
[EINVAL]	ndfs is greater than FD_SETSIZE and _DARWIN_UNLIMITED_SELECT is not defined.

SELECT FUNCTION DESCRIPTOR ARGUMENTS

readset → descriptors for checking readable

writeset → descriptors for checking writable

exceptset → descriptors for checking exception conditions (2 exception conditions)

- ✓arrival of out of band data for a socket
- ✓the presence of control status information to be read from the master side of a pseudo terminal (Ignore)

If you pass the 3 arguments as NULL, you have a high precision timer than the sleep function

DESCRIPTOR SETS

Array of integers : each bit in each integer correspond to a descriptor (**fd_set**)

4 macros

- `void  FD_ZERO(fd_set *fdset);  /* clear all bits in fdset */`
- `void  FD_SET(int fd, fd_set *fdset); /* turn on the bit for fd in fdset */`
- `void  FD_CLR(int fd, fd_set *fdset); /* turn off the bit for fd in fdset*/`
- `int   FD_ISSET(int fd, fd_set *fdset);/* is the bit for fd on in fdset ? */`

EXAMPLE OF DESCRIPTOR SETS MACROS

```
fd_set  rset;


FD_ZERO(&rset);          /*all bits off : initiate*/
FD_SET(1, &rset);        /*turn on bit fd 1*/
FD_SET(4, &rset);        /*turn on bit fd 4*/
FD_SET(5, &rset);        /*turn on bit fd 5*/
```

maxfdp1 argument to select function

- specifies the number of descriptors to be tested.
- Its value is the maximum descriptor to be tested, plus one. (hence maxfdp1)
  - Descriptors 0, 1, 2, up through and including **maxfdp1**-1 are tested
  - example: interested in **fds** 1,2, and 5 → **maxfdp1** = 6
  - Your code has to calculate the **maxfdp1** value constant **FD_SETSIZE** defined by including **<sys/select.h>**
  - is the number of descriptors in the **fd_set** datatype. (often = 1024)

Value-Result arguments in select function

- Select modifies descriptor sets pointed to by **readset**, **writeset**, and **exceptset** pointers
- On function call
  - Specify value of descriptors that we are interested in

On function return

- Result indicates which descriptors are ready

Use **FD_ISSET** macro on return to test a specific descriptor in an **fd_set** structure

- Any descriptor not ready will have its bit cleared
- You need to turn on all the bits in which you are interested on all the descriptor sets each time you call **select**

CONDITION FOR A SOCKET TO BE READY FOR *SELECT*

Condition	*Readable?*	*writable?*	*Exception?*
Data to read	•		
read-half of the connection closed	•		
new connection ready for listening socket	•		
Space available for writing		•	
write-half of the connection closed		•	
Pending error	•	•	
TCP out-of-band data			•

POLL FUNCTION

```
int poll(struct pollfd *fdarray, unsigned long nfds, int
 timeout);
```

returns count of ready descriptors, 0 on timeout, -1 on error.

- Each element of ***fdarray*** is a *pollfd* structure that specifies the condition to be tested for a given descriptor
- ***pollfd*** contains: file descriptor *fd,* events of interest on *fd*, and events that occured on *fd.*
- To switch off a descriptor, set the *fd* member of the *pollfd* structure to a negative value.

SOCKET OPTION

SOCKET OPTIONS

There are 3 ways to get and set options affecting sockets -

- the *getsockopt* and *setsockopt* functions.
- the *fcntl* function
- the *ioctl* function

GETSOCKOPT() AND SETSOCKOPT()

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

int getsockopt(int sockfd, int level, int optname, void * optval, socklen_t *
optlen);

int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t
optlen);

Both functions return 0 if OK else -1 on error.
```

- The **sockfd** must refer to an open socket descriptor.
- The **level** indicates whether the socket option is **general** or **protocol-specific** socket.
- The **optval** is a pointer to a variable from which the new value of the option is fetched by **setsockopt**, or into which the current value of the option is stored by **getsockopt**.
- The size of this variable is specified by the final argument, as a value for **setsockopt** and as a value-result for **getsockopt**.

GETSOCKOPT() AND SETSOCKOPT()...

- There are two basic types of options that can be queried by **getsockopt** or set by **setsockopt**: binary options that enable or disable a certain feature (flags), and options that fetch and return specific values that we can either set or examine (values).
- When calling **getsockopt** for the flag options, * **optval** is an integer.
- The value returned in ***optval** is **zero** if the option is disabled, or nonzero if the option is enabled.
- Similarly, **setsockopt** requires a **nonzero** ***optval** to turn the option on, and a **zero** value to turn the option off.
- For non-flag options, the option is used to pass a value of the specified datatype between the user process and the system.

SOCKET OPTIONS

Two basic type of options -

- Flags - binary options that enable or disable a feature.
- Values - options that fetch and return specific values.

Not supported by all implementations.

Socket option fall into 4 main categories -

- Generic socket options
  - SO_RCVBUF, SO_SNDBUF, SO_BROADCAST, etc.
- IPv4
  - IP_TOS, IP_MULTICAST_IF, etc.
- IPv6
  - IPv6_HOPLIMIT, IPv6_NEXTHOP, etc.
- TCP
  - TCP_MAXSEG, TCP_KEEPALIVE, etc.

SOCKET STATES

- Options have to be set or fetched depending on the state of a socket.
- Some socket options are inherited from a listening socket to the connected sockets on the server side.

  - E.g. SO_RCVBUF and SO_SNDBUF

    These options have to be set on the socket before calling *listen()* on the server side and before calling *connect()* on the client side.

GENERIC SOCKET OPTIONS

SO_BROADCAST

- Enables or disables the ability of a process to send broadcast messages.
- It is supported only for datagram sockets.
- Its default value is ***off***.

SO_ERROR

- *Pending Error* - When an error occurs on a socket, the kernel sets the ***so_error*** variable.
- The process can be notified of the error in two ways -
  - If the process is blocked in ***select*** for either read or write, it returns with either or both conditions set.
  - If the process is using signal driven I/O, the ***SIGIO*** signal is generated for the process.

GENERIC SOCKET OPTIONS CONTD…

SO_KEEPALIVE

- Purpose of this option is to detect if the peer host crashes. The **SO_KEEPALIVE** option will detect half-open connections and terminate them.
- If this option is set and no data has been exchanged for 2 hours, then TCP sends **keepalive** probe to the peer.
  - Peer responds with **ACK**. Another probe will be sent only after 2 hours of inactivity.
  - Peer responds with **RST** (has crashed and rebooted). Error is set to **ECONNRESET** and the socket is closed.
  - No response. 8 more probes are sent after which the socket's pending error is set to either **ETIMEDOUT** or **EHOSTUNREACH** and the socket is closed.

GENERIC SOCKET OPTIONS CONTD…

Receive Low Water Mark -

- Amount of **data** that must be in the socket receive buffer for a socket to become ready for *read.*

Send Low Water Mark -

- Amount of **space** that must be available in the socket send buffer for a socket to become ready for *write*.

SO_RCVLOWAT and **SO_SNDLOWAT**

- These options specify the receive low water mark and send low water mark for TCP and UDP sockets.

GENERIC SOCKET OPTIONS CONTD…

SO_RCVTIMEO and **SO_SNDTIMEO**

- These options place a timeout on socket receives and sends.
- The timeout value is specified in a ***timeval*** structure.

```
struct timeval {
 long tv_sec ;
 long tv_usec ;
}
```

- To disable a timeout, the values in the ***timeval*** structure are set to 0.

GENERIC SOCKET OPTIONS CONTD…

SO_REUSEADDR

- It allows a listening server to restart and bind its well known port even if previously established connections exist.
- It allows multiple instances of the same server to be started on the same port, as long as each instance binds a different local IP address.
- It allows a single process to bind the same port to multiple sockets, as long as each bind specifies a different local IP address.
- It allows completely duplicate bindings only for UDP sockets (broadcasting and multicasting).

GENERIC SOCKET OPTIONS CONTD…

SO_LINGER

- Specifies how *close* operates for a connection-oriented protocol
- The following structure is used:

```
struct linger {
    int l_onoff;
    int l_linger;
}
// l_onoff - 0=off; nonzero=on
// l_linger specifies seconds
```

Three scenarios:

1. If ***l_onoff*** is 0, ***close*** returns immediately. If there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer. The value of ***l_linger*** is ignored.
2. If ***l_onoff*** is **nonzero** and ***linger*** is 0, TCP aborts the connection when *close* is called. TCP discards data in the send buffer and sends **RST** to the peer.

GENERIC SOCKET OPTIONS CONTD…

3. **SO_LINGER (cont)**

If ***l_onoff*** is nonzero and ***linger*** is nonzero, the kernel will linger when *close* is called.

- If there is any data in the send buffer, the process is put to sleep until either:
  - the data is sent and acknowledged
  - Or
  - the linger time expires (for a nonblocking socket the process will not wait for *close* to complete)
- When using this feature, the return value of *close* must be checked. If the linger time expires before the remaining data is send and acknowledged, close returns **EWOULDBLOCK** and any remaining data in the buffer is ignored.

GENERIC SOCKET OPTIONS CONTD…

SO_SNDBUF/SO_RCVBUF

Level: SOL_SOCKET

Get/Set supported

Non-flag option

Datatype of optval: int

Description: Send buffer size/Receive buffer size

GENERIC SOCKET OPTIONS CONTD…

TCP_NODELAY

Level: IPPROTO_TCP

Get/Set supported

Flag option i.e. enable or disable Nagle algorithm

Description: Disable/Enable Nagle algorithm

FCNTL() FUNCTION

- **Fcntl()** stands for "file control" and this function performs various descriptor control operations.
- The fcntl function provides the following features related to network programming.
  - **Non-blocking I/O** – We can set the **O_NONBLOCK** file status flag using the **F_SETFL** command to set a socket as non-blocking.
  - **Signal-driven I/O** – We can set the **O_ASYNC** file status flag using the **F_SETFL** command, which causes the SIGIO signal to be generated when the status of a socket changes.
  - The **F_SETOWN** command lets us set the socket owner (the process ID or process group ID) to receive the **SIGIO** and **SIGURG** signals. The former signal is generated when the signal-driven I/O is enabled for a socket and the latter signal is generated when new out-of-band data arrives for a socket. The **F_GETOWN** command returns the current owner of the

socket.

FCNTL() …

```
int fcntl(int fd, int cmd, long arg);
```

- Each descriptor has a set of file flags that is fetched with the **F_GETFL** command and set with the **F_SETFL** command. The two flags that affect a socket are

O_NONBLOCK - non-blocking I/O

O_ASYNC-signal-driven I/O

Miscellaneous file control operations

- Non-blocking I/O (O_NONBLOCK, F_SETFL)
- Signal-driven I/O (O_ASYNC, F_SETFL)
- Set socket owner (F_SETOWN)

FCNTL AND IOCTL

Operation	fcntl	ioctl	Routing socket	Posix.1g
set socket for nonblocking I/O	F_SETFL, O_NONBLOCK	FIONBIO		fcntl
set socket for signal-driven I/O	F_SETFL, O_ASYNC	FIOASYNC		fcntl
set socket owner	F_SETOWN	SIOCSPGRP or FIOSETOWN		fcntl
get socket owner	F_GETOWN	SIOCGPGRP or FIOGETOWN		fcntl
get #bytes in socket receive buffer		FIONREAD		
test for socket at out-of-band mark		SIOCATMARK		sockatmark
obtain interface list		SIOCGIFCONF	sysctl	
interface operations		SIOC[GS]IF*xxx*		
ARP cache operations		SIOC*x*ARP	RTM_*xxx*	
routing table operations		SIOC*xxx*RT	RTM_*xxx*	

Figure 7.15 Summary of fcntl, ioctl, and routing socket operations.

FCNTL() FUNCTION

- *fcntl* provides the following features related to network programming
  - **Nonblocking I/O** (be aware of error-handling in the following code)

```
int flags=fcntl(fd, F_GETFL, 0);
flags |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

  - **Signal driven I/O**

```
int flags=fcntl(fd, F_GETFL, 0);
flags |= O_ASYNC;
fcntl(fd, F_SETFL, flags);
```

  - Set socket owner to receive SIGIO signals

```
fcntl(fd, F_SETOWN, getpid());
```

FCNTL() FUNCTION

- The signals **SIGIO** and **SIGURG** are generated for a socket only if the socket has been assigned an owner with the **F_SETOWN** command. The integer arg value for the **F_SETOWN** command can be either positive integer, specifying the process ID to receive the signal, or a negative integer whose absolute value is the process group ID to receive the signal.
- The **F_GETOWN** command returns the socket owner as the return value from the fcntl function, either the process ID (a positive return value) or the process group ID (a negative value other than -1).
- The difference between specifying a process or a process group to receive the signal is that the former causes only a single process to receive the signal, while the latter causes all processes in the process group to receive the signal.

IOCTL OPERATIONS

- The common use of **ioctl** by network programs (typically servers) is to obtain information on all the host's interfaces when the program starts: the interface addresses, whether interface supports broadcasting, whether the interface supports multicasting, and so on.

Ioctl() Function

- This function affects an open file referenced by the **fd** argument.

```
#include <unistd.h>

 int ioctl(int fd, int request, …/* void *arg */);

 Returns: 0 if OK, -1 on error
```

- The third argument is always a pointer, but the type of pointer depends on the request.

IOCTL() FUNCTION

We can divide the requests related to networking into six categories.

1. Socket operations
2. File operations
3. Interface operations
4. ARP cache operations
5. Routing table operations
6. STREAMS system

Note that not only do some of the **ioctl** operations overlap some of the **fcntl** operations (e.g. setting a socket to non-blocking), but there are also some operations that can be specified more than one way using **ioctl** (e.g., setting the process group ownership of a socket).

Category	*request*	Description	Datatype
Socket	SIOCATMARK	At out-of-band mark ?	int
	SIOCSPGRP	Set process ID or process group ID of socket	int
	SIOCGPGRP	Get process ID or process group ID of socket	int
File	FIONBIO	Set/clear nonblocking flag	int
	FIOASYNC	Set/clear asynchronous I/O flag	int
	FIONREAD	Get # bytes in receive buffer	int
	FIOSETOWN	Set process ID or process group ID of file	int
	FIOGETOWN	Get process ID or process group ID of file	int
Interface	SIOCGIFCONF	Get list of all interfaces	struct ifconf
	SIOCSIFADDR	Set interface address	struct ifreq
	SIOCGIFADDR	Get interface address	struct ifreq
	SIOCSIFFLAGS	Set interface flags	struct ifreq
	SIOCGIFFLAGS	Get interface flags	struct ifreq
	SIOCSIFDSTADDR	Set point-to-point address	struct ifreq
	SIOCGIFDSTADDR	Get point-to-point address	struct ifreq
	SIOCGIFBRDADDR	Get broadcast address	struct ifreq
	SIOCSIFBRDADDR	Set broadcast address	struct ifreq
	SIOCGIFNETMASK	Get subnet mask	struct ifreq
	SIOCSIFNETMASK	Set subnet mask	struct ifreq
	SIOCGIFMETRIC	Get interface metric	struct ifreq
	SIOCSIFMETRIC	Set interface metric	struct ifreq
	SIOCGIFMTU	Get interface MTU	struct ifreq
	SIOC*xxx*	(many more; implementation-dependent)	
ARP	SIOCSARP	Create/modify ARP entry	struct arpreq
	SIOCGARP	Get ARP entry	struct arpreq
	SIOCDARP	Delete ARP entry	struct arpreq
Routing	SIOCADDRT	Add route	struct rtentry
	SIOCDELRT	Delete route	struct rtentry
STREAMS	I_*xxx*	(see Section 31.5)	

SOCKET OPERATIONS

- Three **ioctl** requests are explicitly used for sockets. All three require that the third argument to **ioctl** be a pointer to an integer.

- **SIOCATMARK**: Return through the integer pointed to by the third argument a non-zero value if the socket's read pointer is currently at the out-of-band mark, or a zero value if the read pointer is not at the out-of-band mark.

- **SIOCGPGRP**: Return through the integer pointed to by the third argument either the process ID or the process group ID that is set to receive **SIGIO** or SIGURG signal for this socket. This request is identical to an fcntl of **F_GETOWN**, note that POSIX standardizes the fcntl.

- **SIOCSPGRP:** Set either the process ID or process group ID to receive the **SIGIO** or **SIGURG** signal for this socket from the integer pointed to by the third argument. This request is identical to an fnctl of **F_SETOWN**, note that POSIX standardizes the **fcntl**.

FILE OPERATIONS

The next group of requests begin with FIO and may apply to certain types of files, in addition to sockets.

FIONBIO	The nonblocking flag for the socket is cleared or turned on, depending on whether the third argument to ioctl points to a zero or nonzero value, respectively. This request has the same effect as the **O_NONBLOCK** file status flag, which can be set and cleared with the **F_SETFL** command to the fcntl function.
FIOASYNC	The flag that governs the receipt of asynchronous I/O signals (SIGIO) for the socket is cleared or turned on, depending on whether the third argument to ioctl points to a zero or nonzero value, respectively. This flag has the same effect as the O_ASYNC file status flag, which can be set and cleared with the F_SETFL command to the fcntl function.
FIONREAD	Return in the integer pointed to by the third argument to ioctl the number of bytes currently in the socket receive buffer. This feature also works for files, pipes, and terminals.
FIOSETOWN	Equivalent to SIOCSPGRP for a socket.
FIOGETOWN	Equivalent to SIOCGPGRP for a socket.

INTERFACE OPERATIONS

The **SIOCGIFCONF** request returns the name and a socket address structure for each interface that is configured. Many of requests use a socket address structure to specify or return an IP address or address mask with the application.

SIOCGIFADDR	Return the unicast address in the ifr_addr member.
SIOCSIFADDR	Set the interface address from the ifr_addr member. The initialization function for the interface is also called.
SIOCGIFFLAGS	Return the interface flags in the ifr_flags member. The names of the various flags are IFF_*xxx* and are defined by including the <net/if.h> header.
SIOCSIFFLAGS	Set the interface flags from the ifr_flags member.
SIOCGIFDSTADDR	Return the point-to-point address in the ifr_dstaddr member.
SIOCSIFDSTADDR	Set the point-to-point address from the ifr_dstaddr member.
SIOCGIFBRDADDR	Return the broadcast address in the ifr_broadaddr member. The application must first fetch the interface flags and then issue the correct request: SIOCGIFBRDADDR for a broadcast interface or SIOCGIFDSTADDR for a point-to-point interface.
SIOCSIFBRDADDR	Set the broadcast address from the ifr_broadaddr member.
SIOCGIFNETMASK	Return the subnet mask in the ifr_addr member.
SIOCSIFNETMASK	Set the subnet mask from the ifr_addr member.
SIOCGIFMETRIC	Return the interface metric in the ifr_metric member. The interface metric is maintained by the kernel for each interface but is used by the routing daemon routed. The interface metric is added to the hop count (to make an interface less favorable).
SIOCSIFMETRIC	Set the interface routing metric from the ifr_metric member.

ARP CACHE OPERATIONS

- On some systems, the ARP cache is also manipulated with the **ioctl** function. Systems that use routing sockets usually use routing sockets instead of **ioctl** to access the ARP cache. These requests use an **arpreq** structure, shown below and defined by including the <net/if_arp.h> header.

```
<net/if_arp.h>

struct arpreq {

struct sockaddr arp_pa; /* protocol address */

struct sockaddr arp_ha; /* hardware address */

int arp_flags; /* flags */

};

#define ATF_INUSE 0x01 /* entry in use */

#define ATF_COM 0x02 /* completed entry (hardware addr valid) */

#define ATF_PERM 0x04 /* permanent entry */

#define ATF_PUBL 0x08 /* published entry (respond for other host) */
```

ARP CACHE OPERATIONS

The third argument to ioctl must point to one of these structures. The following three *requests* are supported

SIOCSARP	Add a new entry to the ARP cache or modify an existing entry. arp_pa is an Internet socket address structure containing the IP address, and arp_ha is a generic socket address structure with sa_family set to AF_UNSPEC and sa_data containing the hardware address (e.g., the 6-byte Ethernet address). The two flags, ATF_PERM and ATF_PUBL, can be specified by the application. The other two flags, ATF_INUSE and ATF_COM, are set by the kernel.
SIOCDARP	Delete an entry from the ARP cache. The caller specifies the Internet address for the entry to be deleted.
SIOCGARP	Get an entry from the ARP cache. The caller specifies the Internet address, and the corresponding Ethernet address is returned along with the flags.

ROUTING TABLE OPERATIONS

- On some systems, two **ioctl** requests are provided to operate on the routing table. These two requests require that the third argument to **ioctl** be a pointer to an **rtentry** structure, which is defined by including the <net/route.h> header. These requests are normally issued by the route program. Only the superuser can issue these requests. On systems with routing sockets, these requests use routing sockets instead of ioctl.

SIOCADDRT	Add an entry to the routing table.
SIOCDELRT	Delete an entry from the routing table.

- There is no way with **ioctl** to list all the entries in the routing table. This operation is usually performed by the **netstat** program when invoked with the -r flag. This program obtains the routing table by reading the kernel's memory (/dev/kmem).

IOCTL SUMMARY

The ioctl commands that are used in network programs can be divided into six categories:

1. Socket operations (Are we at the out-of-band mark?)
2. File operations (set or clear the nonblocking flag)
3. Interface operations (return interface list, obtain broadcast address)
4. ARP table operations (create, modify, get, delete)
5. Routing table operations (add or delete)
6. STREAMS system

DAEMON PROCESS, SYSLOGD DAEMON, SYSLOG FUNCTION, IOCTL OPERATION, IOCTL FUNCTION

DAEMON PROCESS

- A daemon is a process that runs in the background and is not associated with a controlling terminal. Unix systems typically have many processes that are daemons, running in the background, performing different administrative tasks.

- The lack of a controlling terminal is typically a side effect of being started by a system initialization script. But if a daemon is started by a user typing to a shell prompt, it is important for the daemon to disassociate itself from the controlling terminal to avoid any unwanted interaction with job control, terminal session management, or simply to avoid unexpected output to the terminal from the daemon as it runs in the background.

HOW TO START DAEMON PROCESS?

- During system startup, many daemons are started by the system initialization scripts. Daemons started by these scripts begin with superuser privileges.
- Many network servers are started by the **inetd superserver**. The **inetd** itself is started from one of the scripts in Step 1. The inetd listens for network requests, and when a request arrives, it invokes the actual server.
- The execution of programs on a regular basis is performed by the **cron** daemon, and programs that it invokes run as daemons. The cron daemon itself is started in Step 1 during system startup.
- The execution of a program at one time in the future is specified by the "**at**" command. The cron daemon normally initiates these programs when their time arrives, so these programs run as daemons.
- Daemons can be started from user terminals, either in the foreground or in the background. This is often done when testing a daemon, or restarting a daemon that was terminated for some reason.
- Since a daemon does not have a controlling terminal, it needs some way to output messages when something happens, either normal informational messages or emergency messages that need to be handled by an administrator.

HOW TO CREATE A DAEMON IN UNIX (HOW TO DAEMONIZE A PROCESS)

1. fork

We first call fork and then the parent terminates, and the child continues. If the process was started as a shell command in the foreground, when the parent terminates the shell think the command is done. This automatically runs the child process in the background.

2. setsid

setsid is a POSIX function that creates a new session. The process becomes the session leader of the new session, becomes the process group leader of a new process group, and has no controlling terminal.

3. Ignore SIGHUP and fork again

We ignore SIGUP and call fork again. When this function returns, the parent is really the first child and it terminates, leaving the second child running. The purpose of this second fork is to guarantee that the daemon cannot automatically acquire a controlling terminal should it open a terminal device in the future. We must ignore SIGHUP because when the session leader terminates (the first child), all processes in the session (our second child) receive the SIGHUP signal.

4. Change working directory

We change the working directory to the root directory. The file system cannot be un-mounted if working directory is not changed.

5. Close any open descriptors

We open any open descriptors that are inherited from the process that executed the daemon (normally a shell).

6. Redirect stdin, stdout, and stderr to /dev/null

We open /dev/null for standard input, standard output, and standard error. This guarantees that these common descriptors are open, and a read from any of these descriptors returns 0 (EOF), and the kernel just discards anything written to them.

7. Use syslogd for errors

The syslogd daemon is used to log errors.

SYSLOGD DAEMON

- Unix systems normally start a daemon named **syslogd** from one of the system initialization scripts, and it runs as long as the system is up. The **syslogd** perform the following actions on startup.
  - The configuration file, normally /etc/syslog.conf, is read, specifying what to do with each type of log messages that the daemon can receive.
  - A Unix domain socket is created and bound to the pathname /var/run/log (/dev/log on some systems).
  - A UDP socket is created and bound to port 514 (the syslog service).
  - The pathname /dev/klog is opened. Any error messages from within the kernel appear as input on this device.
- The **syslogd** daemon runs in an infinite loop that calls **select**, waiting for any one of its three descriptors (last three of above bullets) to be readable; it reads the log message and does what the configuration file says to do with that message. If the daemon receives the SIGUP signal, it reads its configuration file

SYSLOG FUNTION

- Since a daemon does not have a controlling terminal, it cannot just **fprintf** to **stderr**. The common technique for logging messages from a daemon is to call the syslog function.

```
#include <syslog.h>

 void syslog(int priority, const char * message,…);
```

- The priority argument is a combination of a level and a facility. The message is like a format string to printf, with addition of a %m specification, which is replaced with error message corresponding to the current value of **errno**.
- For example, the following call could be issued by a daemon when a call to the rename function unexpectedly fails:

```
syslog(LOG_INFO| LOG_LOCAL2, "rename (%s, %s): %m", file1,
file2);
```

Priority order (higher to lower)

- LOG_EMERG: A panic condition. This is normally broadcast to all users.
- LOG_ALERT:A condition that should be corrected immediately, such as a corrupted system database.
- LOG_CRIT: Critical conditions, e.g., hard device errors.
- LOG_ERR:Errors.
- LOG_WARNING:Warning messages.
- LOG_NOTICE: Conditions that are not error conditions, but should possibly be handled specially.
- LOG_INFO:Informational messages.
- LOG_DEBUG:Messages that contain information normally of use only when debugging a program.

- **Facility parameter**
- LOG_AUTH : The authorization system: login(1), su(1), getty(8), etc.
- LOG_AUTHPRIV : The same as LOG_AUTH, but logged to a file readable only by selected individuals.
- LOG_CRON : The cron daemon: cron(8).
- LOG_DAEMON : System daemons, such as routed(8), that are not provided for explicitly by other facilities.
- LOG_FTP : The file transfer protocol daemons: ftpd(8), tftpd(8).
- LOG_KERN : Messages generated by the kernel. These cannot be generated by any user processes.
- LOG_LPR : The line printer spooling system: cups-lpd(8), cupsd(8), etc.
- LOG_MAIL : The mail system.
- LOG_NEWS : The network news system.
- LOG_SECURITY : Security subsystems, such as ipfw(4).
- LOG_SYSLOG : Messages generated internally by syslogd(8).
- LOG_USER : Messages generated by random user processes. This is the default facility identifier if none is specified.
- LOG_UUCP : The uucp system.
- LOG_LOCAL0 : Reserved for local use.
- Similarly for LOG_LOCAL1 through LOG_LOCAL7.

- When the application calls syslog the first time, it creates a Unix domain datagram socket and then calls connect to the well-known pathname of the socket created by the syslogd daemon. This socket remains open until the process terminates. Alternatively, the process can call openlog and closelog.

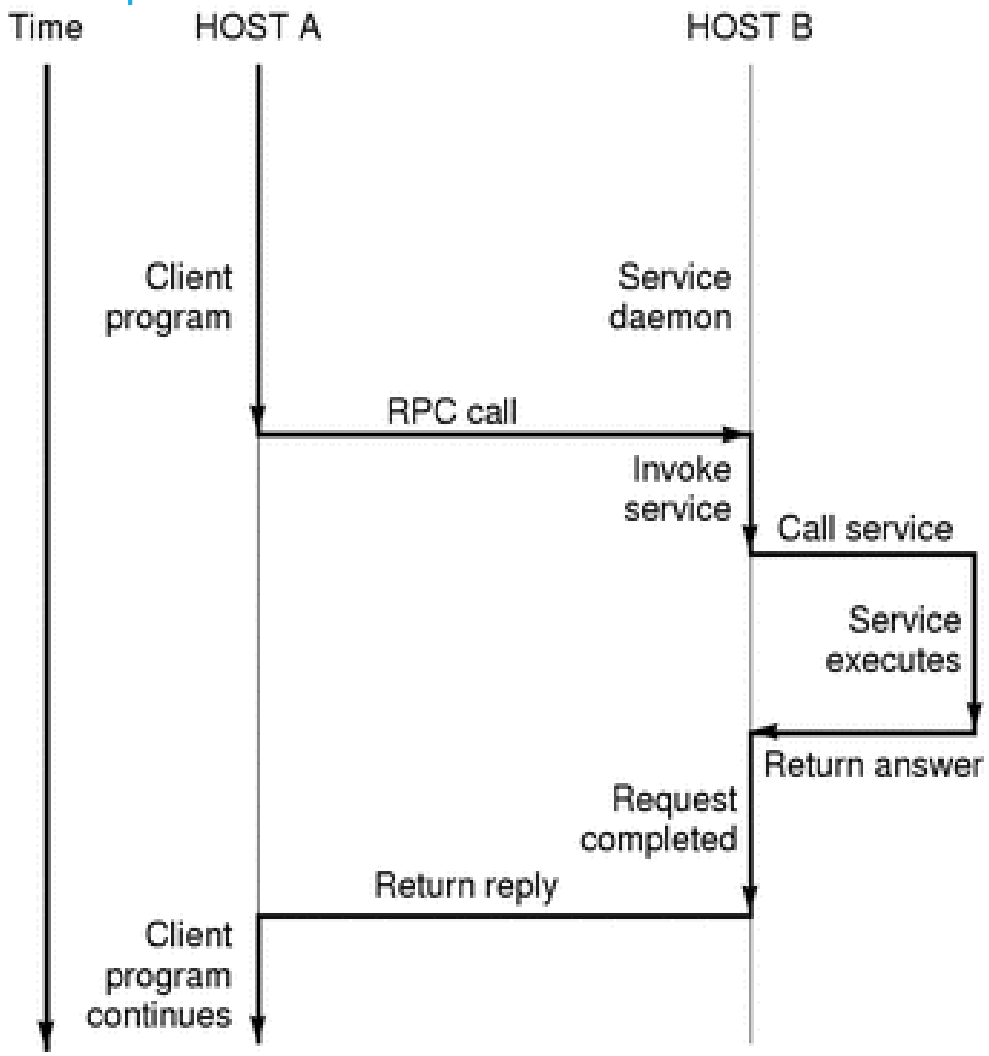
```
#include <syslog.h>

 void openlog(const char *ident, int options, int facility);

void closelog();
```

- The openlog can be called before the first call to syslog and closelog can be called when the application is finished sending log messages.

REMOTE PROCEDURAL CALL

- Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details.
- RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure.
- The two processes may be on the same system, or they may be on different systems with a network connecting them.
- By using RPC, programmers of distributed applications avoid the details of the interface with the network.
- The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.
- RPC makes the client/server model of computing more powerful and easier to program.

REMOTE PROCEDURAL CALL



When making a remote procedure call:

1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.
2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

The main goal of RPC is to hide the **existence of the network** from a program.

1. The message-passing nature of network communication is hidden from the user. The user doesn't first open a connection, read and write data, and then close the connection. Indeed, a client often does not even know they are using the network.

REMOTE PROCEDURAL CALL

- RPC is especially well suited for client-server (e.g., query-response) interaction in which the flow of control alternates between the caller and callee.
- Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

1. A client invokes a *client stub* procedure, passing parameters in the usual way. The client stub resides within the client's own address space.
2. The client stub *marshalls (arranges)* the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.
3. The client stub passes the message to the transport layer(**TCP/UDP or UNIX/Local Domain in case of IPC**), which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a *server stub*, which demarshalls the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

CRASHING OF SERVER HOST

1. When the server host crashes, nothing is sent out on the existing network connections from the server. We are assuming that the host crashes and is not shut down by an operator.
2. **The (client) application doesn't know about the status of server until it sends data to the server**. When the (client) application sends data to the server, the transport layer expects acknowledgement from the server.
3. In the absence of acknowledgements, the transport layer retransmits the data segment. After specific number of retransmission, the transport layer gives up and notifies the (client) application. In this case (the server host crashed and there was no response at all to the client's data segment), the error ETIMEDOUT is returned.

   However, if the server host has not crashed but was unreachable on the network, assuming the host was still unreachable, and some intermediate router determined that the server host was unreachable and responded with an ICMP "destination unreachable" message, the error is either EHOSTUNREACH or ENETUNREACH.

   If we want to detect the crashing of the server host even if we are not actively sending it data, we must set SO_KEEPALIVE socket option, and send KEEPALIVE packets periodically to the server host.

WHEN SERVER HOST REBOOTS

1. When the server host reboots after crashing, its TCP loses all information about connections that existed before the crash. Therefore, the server TCP respond to the received data segment from the client with an RST packet.

2. The (client) application gets ECONNRESET error from the transport layer.

PROCESS TABLE

- The **process table** is a data structure maintained by the operating system to facilitate context switching and scheduling, etc.
- Each entry in the table, often called a **context block**, contains information about a process such as process name and state, priority, memory state, resource state, registers, and a semaphore it may be waiting on.
- The exact contents of a context block depend on the operating system. For instance, if the OS supports paging, then the context block contains an entry to the page table.

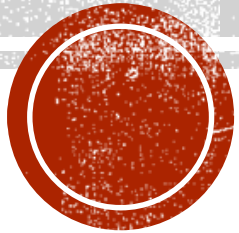
END OF CHAPTER 2

CHAPTER 3: WINSOCK PROGRAMMING

Compiled by:

Madan Kadariya

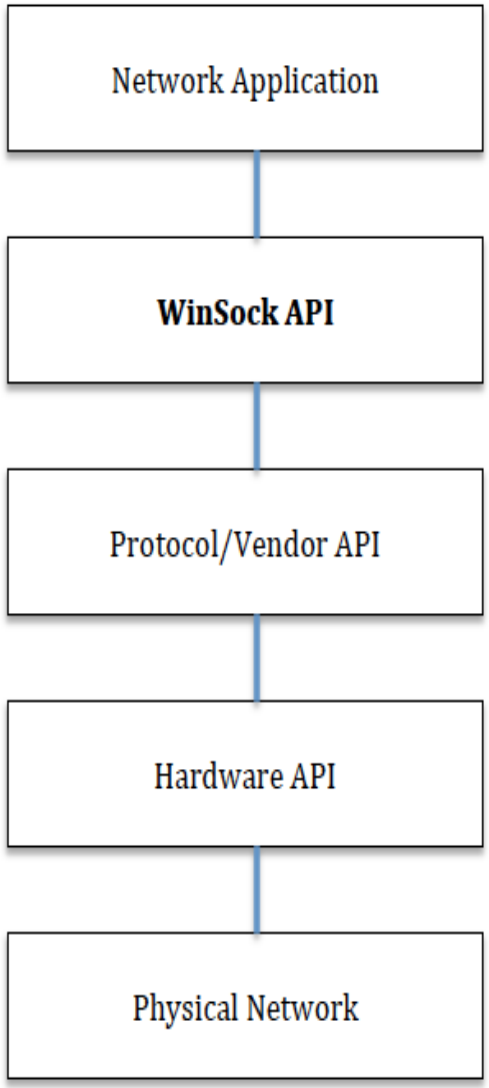
NCIT



INTRODUCTION TO WINSOCK PROGRAMMING

- The Windows Sockets Application Programming Interface (WinSock API) is a library of functions that implements the socket interface.
- Winsock augments the Berkeley socket implementation by adding Windows-specific extension to support the message driven nature of the Windows operating system.
- WinSock is a network application programming interface (API) for Microsoft Windows; a well-defined set of data structures and function calls implemented as a dynamic link library (DLL).
- An application make function calls requesting generic network services (like send() and receive()); Winsock translates these generic requests into protocol- specific requests and performs the necessary tasks.
- Residing between the application and the network implementation, Winsock shields you from the details of low-level network protocols.

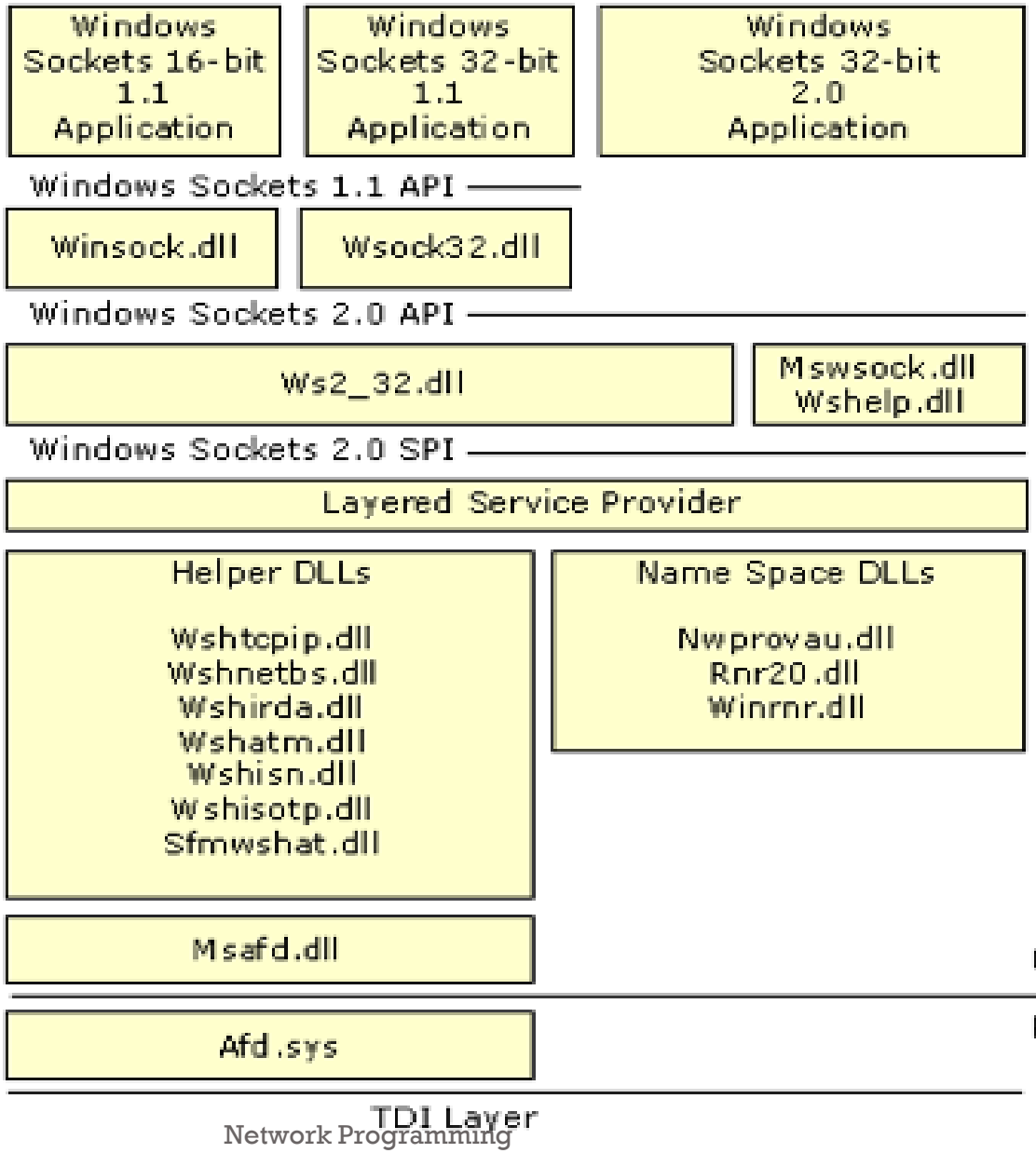
INTRODUCTION TO WINSOCK PROGRAMMING



The WinSock specification allows TCP/IP stack vendors to provide a consistent interface of their stacks so that application developers can write an application to the WinSock specification and have that application run on any vendor's WinSock-compatible TCP/IP protocol stack.

This is contrast to the days before the WinSock standard when software developers had to link their applications with libraries to each TCP/IP vendor's implementation. This limited the number of stacks that most applications ran on because of the difficulty in maintaining an application that used several different implementations of Berkeley sockets.

WINSOCK ARCHITECTURE



The top layer shows different network applications pertaining to different Winsock API versions and Windows architectures.

The two layers below it specify DLLs used for Winsock API version 1.1 and 2.0, respectively.

Winsock provides a Service Provider Interface for creating Winsock services, commonly referred to as the Winsock SPI.

Afd.sys is a Windows driver. A driver is a small software program that allows your computer to communicate with hardware or connected devices.

WINSOCK DLL (CURRENT IS WS2_32.DLL)

- **WINSOCK.DLL** is a dynamic-link library that provides a common application programming interface (API) for developers of network applications that use the Transmission Control Protocol/Internet Protocol (TCP/IP) stack.
- This means that a programmer who develops a Windows-based TCP/IP application, such as an FTP or Telenet client, can write one program that works with any TCP/IP protocol stack that provides Windows Socket Services (**WINSOCK.DLL**).

Some Winsock DLLs and their descriptions.

- **Winsock.dll**
  - 16-bit Winsock 1.1
- **Wsock32.dll**
  - 32-bit Winsock 1.1
- **Ws2_32.dll (this is latest)**
  - Main Winsock 2.0
- **Mswsock.dll**
  - Microsoft extensions to Winsock. Mswsock.dll is an API that supplies services that are not part of Winsock.
- **Ws2help.dll**
  - Platform-specific utilities. Ws2help.dll supplies operating system–specific code that is not part of Winsock.
- **Wshtcpip.dll**
  - Helper for TCP
- Dynamic linking allows a module to include only the information needed to locate an exported DLL function at load time or run time. Dynamic linking differs from the more familiar static linking, in which the linker copies a library function's code into each module that calls it.

TYPES OF DYNAMIC LINKING

There are two methods for calling a function in a DLL:

- In **load-time dynamic linking**, a module makes explicit calls to exported DLL functions as if they were local functions. This requires to link the module with the import library for the DLL that contains the functions. An import library supplies the system with the information needed to load the DLL and locate the exported DLL functions when the application is loaded.

- In **run-time dynamic linking**, a module uses the **LoadLibrary** or **LoadLibraryEx** function to load the DLL at run time. After the DLL is loaded, the module calls the **GetProcAddress** function to get the addresses of the exported DLL functions. The module calls the exported DLL functions using the function pointers returned by **GetProcAddress**. This eliminates the need for an import library.

DLLs and Memory Management

- Every process that loads the DLL maps it into its virtual address space.
- After the process loads the DLL into its virtual address, it can call the exported DLL functions.
- The system maintains a per-process reference count for each DLL.
- When a thread loads the DLL, the reference count is incremented by one. When the process terminates, or when the reference count becomes zero (run-time dynamic linking only), the DLL is unloaded from the virtual address space of the process.
- Like any other function, an exported DLL function runs in the context of the thread that calls it. Therefore, the following conditions apply:
  - The threads of the process that called the DLL can use handles opened by a DLL function. Similarly, handles opened by any thread of the calling process can be used in the DLL function.
  - The DLL uses the stack of the calling thread and the virtual address space of the calling process.
  - The DLL allocates memory from the virtual address space of the calling process.

ADVANTAGES OF DYNAMIC LINKING

- Multiple processes that load the same DLL at the same base address share a single copy of the DLL in physical memory. Doing this saves system memory and reduces swapping.
- When the functions in a DLL change, the applications that use them do not need to be recompiled or relinked as long as the function arguments, calling conventions, and return values do not change. In contrast, statically linked object code requires that the application be relinked when the functions change.
- A DLL can provide after-market support. For example, a display driver DLL can be modified to support a display that was not available when the application was initially shipped.
- Programs written in different programming languages can call the same DLL function as long as the programs follow the same calling convention that the function uses. The calling convention (such as C, Pascal, or standard call) controls the order in which the calling function must push the arguments onto the stack, whether the function or the calling function is responsible for cleaning up the stack, and whether any arguments are passed in registers.
- A potential disadvantage to using DLLs is that the application is not self-contained; it depends on the existence of a separate DLL module. The system terminates processes using load-time dynamic linking if they require a DLL that is not found at process startup and gives an error message to the user. The system does not terminate a process using run-time dynamic linking in this situation, but functions exported by the missing DLL are not available to the program.

Berkeley Socket Versus WinSock

- WinSock supports the TCP/IP domain for inter-process communication on the same computer as well as network communication. In addition to the TCP/IP domain, sockets in most UNIX implementations support the UNIX domain for the inter-process communication on the same computer.

- The return values of certain Berkeley functions are different. For example, the **socket()** function returns **-1** on failure in the UNIX environment; the WinSock implementation returns **INVALID_SOCKET**.

- Certain Berkeley functions have different names in WinSock. For example, in UNIX the **close()** system call is used to close the socket connection. In WinSock, the function is called **closesocket().** It is so because WinSock socket handles may not be UNIX-style file descriptors.

WinSock Extension to Berkeley Sockets

- WinSock has several extensions to Berkeley sockets. Most of these extensions are due to the message-driven architecture of Microsoft Windows. Some extensions are also required to support the non-preemptive nature of the 16-bit Windows operating environment.

Blocking I/O

- The simplest form of I/O in Windows Sockets 2 is blocking I/O. Sockets are created in blocking mode by default.
- Any I/O operation with a blocking socket will not return until the operation has been fully completed.
- Thus, any thread can only execute one I/O operation at a time. For example, if a thread issues a receive operation and no data is currently available, the thread will block until data becomes available and is placed into the thread's buffer. Although this is simple, it is not necessarily the most efficient way to do I/O.

WinSock Asynchronous Functions

- WinSock was originally designed for the non-preemptive Windows architecture. For this reason, several extensions were added to traditional Berkeley sockets.

Blocking Versus Non-blocking

- Many of the Berkeley socket functions take an indeterminate amount of time to execute. When a function exhibits this behavior, it is said to block; calling the function blocks the further execution of the calling program.
- In the Berkeley UNIX environment, for which sockets were originally developed, this didn't pose a serious problem because the UNIX operating system would simply preempt the blocking program and begin running another program.
- Windows (unlike Windows NT) can't preempt a task, so all other programs are put on hold until the blocking call returns.
- The designers of WinSock knew this posed a serious problem, so they added special code in the blocking functions to force the message loop of other applications to be checked. But this was still not the most efficient technique.
- Berkeley sockets already have the notion of blocking versus non-blocking for some operations. For example, the send() function used to send data to a remote host may not return immediately, so the programmer is given the option of creating the socket with blocking or non-blocking sends.
- If the socket is created in blocking mode, it won't return until the data has been delivered. If it's created in non-blocking mode, the call to the send() function returns immediately and the program must call another function called select() to determine the status of the send.
- Windows and Windows NT can use the select() method of non-blocking calls, but the best thing to do in a Windows program is to use the special Windows asynchronous functions.

- The special Windows asynchronous functions begin with the prefix **WSAAsync**. These functions were added to WinSock to make Berkeley sockets better fit the message-driven paradigm of Windows.
- The most common events to use the asynchronous functions for are the sending and receiving of data. Sending data might not happen instantly, and receiving data most certainly will cause a program to wait unless it is receiving a constant stream of bytes.
- By creating a socket for nonblocking sends and receives and using the **WSAAsyncSelect()** function call, an application will receive event notification messages to inform it when it can send data or when data has arrived and needs to be read. In the mean time, when there is no data communications occurring, the rest of the program remains fully responsive to the user's actions.
- WinSock even extends Berkeley's nonblocking support to functions that could still cause a Berkeley UNIX program to block.
- The name server's job is to take as input the plain text representation of a computer's name and return that computer's IP address.
- GetXbyY is used to refer to database functions; function names take the form of get X by Y, or put another way: "Given Y, what is the corresponding X?"
- gethostbyname(): given the computer's name, what is its host information? In Berkeley UNIX, the getXbyY functions may block.
- WinSock adds asynchronous versions of the getXbyY functions called **WSAAsyncGetXbyY**. The gethostbyname() function is complimented by the nonblocking WinSock function called **WSAAsyncGetHostByName**(), for example. A call to a WSAAsyncGetXbyY function returns immediately with an identifying handle. When the actual work performed by the function has completed, a message is sent to the application notifying completion of the function with the specified handle.

- The WinSock asynchronous functions were added primarily for the benefit of the nonpreemptive Windows environment.
- The **WSAAsync** functions have an important use even in Windows NT. They allow an applications to remain responsive to the user.
- Users won't enjoy working with any program if it forces them to wait for completion of a long event.
- Most users expect a way to cancel operations that take a long time. For example, suppose that you have a program that takes as input a computer's plain-text name. The user enters the name and then presses a button labeled Look Up, which causes the gethostbyname() function to be called. Using gethostbyname() will cause the program to hang for an indeterminate amount of time until the request is carried out.
- Under Windows NT other programs would still run, but under Windows the performance of all programs would be degraded. This program could be modified to use **WSAAsyncGetHostByName**() instead of **gethostbyname**(). As soon as users press the Look Up button, the **WSAAsyncGetHostByName**() function is called and returns an identifying handle.
- If users wish to cancel the search, they can press the Cancel button, which terminates the request with that identifying handle. Users would maintain full control instead of being at the mercy of the program.

WINDOWS SOCKET EXTENSION; SETUP AND CLEANUP FUNCTION

- The WinSock functions the application needs are located in the dynamic library named **WINSOCK.DLL** or **WSOCK32.DLL** depending on whether the 16-bit or 32- bit version of Windows is being targeted.
- The application is linked with either **WINSOCK.LIB** or **WSOCK32.LIB** as appropriate.
- The include file where the WinSock functions and structures are defined is named **WINSOCK.H** for both the 16-bit and 32-bit environments.
- Before the application uses any WinSock functions, the application must call an initialization routine called **WSAStartup()**.
- Before the application terminates, it should call the **WSACleanup()** function.

WSASTARTUP

- The **WSAStartup**() function initializes the underlying Windows Sockets Dynamic Link Library (WinSock DLL).
- The **WSAStartup**() function gives the TCP/IP stack vendor a chance to do any application-specific initialization that may be necessary.
- **WSAStartup()** is also used to confirm that the version of the WinSock DLL is compatible with the requirements of the application.

The prototype of the WSAStartup() functions follows:

- ***int WSAStartup(WORD wVersionRequired, LPWSADATA lpWSAData);***

- The **wVersionRequired** parameter is the highest version of the WinSock API the calling application can use. The high-order byte specifies the minor version and the low-order byte specifies the major version number. The **lpWSAData** parameter is a pointer to a WSAData structure that receives details of the WinSock implementation.

- **WinSock implementation.**

```
typedef struct WSAData {
        WORD wVersion; // version supplied
        WORD wHighVersion; // highest version that can be supplied
        char   szDescription[WSADESCRIPTION_LEN+1];   /*   a   null   terminated
                ASCII string that describes the WinSock implementation*/
        char   szSystemStatus[WSASYS_STATUS_LEN+1];   /*   a   null   terminated
                ASCII string that contains relevant status or configuration information */
        unsigned  short  iMaxSockets;  /*  the  maximum  number  of  sockets  that  a
                single process can potentially open */
        unsigned  short  iMaxUdpDg;    /*  the  size,  in  bytes,  of  the  largest  UDP
                datagram that can be sent or received */
        char FAR * lpVendorInfo; // a pointer to a vendor specific data structures
} WSADATA;
```

WSACLEANUP

- The **WSACleanup**() function is used to terminate an application's use of WinSock.
- For every call to **WSAStartup**() there has to be a matching call to **WSACleanup**().
- **WSACleanup**() is usually called after the application's message loop has terminated.
- In an MFC application, the **ExitInstance**() member function of the **CWinApp** class provides a convenient location to call **WSACleanup**().
- The prototype follows:

int PASCAL FAR WSACleanup(void);

WSAGETLASTERROR

- The **WSAGetLastError**() function doesn't deal exclusively with startup or shutdown procedures, but it needs to be addressed early. Its function prototype looks like

int WSAGetLastError(void);

- **WSAGetLastError**() returns the last WinSock error that occurred. Because WinSock isn't really part of the operating system but is instead a later add-on, errno (like in UNIX and MS-DOS) couldn't be used.
- As soon as a WinSock API call fails, the application should call **WSAGetLastError**() to retrieve specific details of the error.

FUNCTION FOR HANDLING BLOCKED I/O

- The Berkeley method of using non-blocking sockets involves two functions: ioctl() and select().
- ioctl() is the UNIX function to perform input/output control on a file descriptor or socket.
- Because a WinSock socket descriptor may not be a true operating system file descriptor, ioctl() can't be used, so ioctlsocket() is provided instead. select() is used to determine the status of one or more sockets. The use of ioctlsocket() to convert a socket to nonblocking mode looks like this:

```
// put socket s into nonblocking mode
u_long ulCmdArg = 1; // 1 for non-blocking, 0 for blocking
ioctlsocket(s, FIONBIO, &ulCmdArg);
```

- Once a socket is in its non-blocking mode, calling a normally blocking function simply returns **WSAEWOULDBLOCK** if the function can't immediately complete.

```
SOCKET clientS;
clientS = accept(s, NULL, NULL);
if (clientS == INVALID_SOCKET)
{
int nError = WSAGetLastError();
// if there is no client waiting to connect to this server,
nError will be WSAEWOULDBLOCK
}
```

- Your server application could simply call accept() periodically until the call succeeded, or you could use the select() call to query the status of the socket.
- The select() function checks the readability, writeability, and exception status of one or more sockets.
- WinSock provides a function called **WSAAsyncSelect**() to solve the problem of blocking socket function calls. It is a much more natural solution to the problem than using **ioctlsocket**() and select().
- It works by sending a Windows message to notify a window of a socket event. Its prototype is as follows:

```
int WSAAsyncSelect(SOCKET s, HWND hWnd, u_int wMsg, long lEvent);
```

- **s** is the socket descriptor for which event notification is required. **hWnd** is the Window handle that should receive a message when an event occurs on the socket. **wMsg** is the message to be received by **hWnd** when a socket event occurs on socket **s**. It is usually a user-defined message (WM_USER + n). **lEvent** is a bitmask that specifies the events in which the application is interested.
- **WSAAsyncSelect**() returns 0 (zero) on success and **SOCKET_ERROR** on failure. On failure, **WSAGetLastError**() should be called. **WSAAsyncSelect**() is capable of monitoring several socket events.

Event	**Meaning**
FD_READ	Socket ready for reading
FD_WRITE	Socket ready for writing
FD_OOB	Out-of-band data ready for reading on socket
FD_ACCEPT	Socket ready for accepting a new incoming connection
FD_CONNECT	Connection on socket completed
FD_CLOSE	Connection on socket has been closed

- The lEvent parameter is constructed by doing a logical OR on the events in which you're interested. To cancel all event notifications, call WSAAsyncSelect() with wMsg and lEvent set to 0.

ASYNCHRONOUS DATABASE FUNCTION

- WinSock provides a set of procedures commonly referred to as the database functions. The duty of these database functions is to convert the host and service names that are used by humans into a format usable by the computer.
- The computers on an internetwork also require that certain data transmitted between them be in a common format. WinSock provides several conversion routines to fulfill this requirement.

Conversion Routines and Network Byte Ordering

- There are four primary byte-order conversion routines. They handle the conversions to and from unsigned short integers and unsigned long integers.

Unsigned Short Integer Conversion

- The **htons()** and **ntohs()** functions convert an unsigned short from host-to- network order and from network-to-host order. The prototype looks like.

```
u_short htons(u_short hostshort);
u_short ntohs(u_short netshort);
```

Unsigned Long Integer Conversion

- The **htonl()** and **ntohl()** functions work like **htons()** and **ntohs()** except that they operate on four-byte unsigned longs rather than unsigned shorts. The prototype look like the following:

```
u_long PASCAL htons(u_long hostlong);
u_long PASCAL ntohs(u_long netlong);
```

Converting IP Addresses

- WinSock provides another set of conversion functions that provide a translation between the ASCII representation of a dotted-decimal IP address and the internal 32-bit, byte-ordered number required by other WinSock functions.

Converting an IP Address String to Binary

- **inet_addr()** converts a dotted-decimal IP address string into a number suitable for use as an Internet address. Its function prototype is as follows:

```
unsigned long inet_addr(const char * cp);
```

- cp is a pointer to a string representing an IP address in dotted-decimal notation. The **inet_addr()** function returns a binary representation of the Internet address given. This value is already in network byte order, so there is no need to call **htonl()**. If the cp string doesn't contain a valid IP address, inet_addr() returns **INADDR_NONE**. E.g.

```
u_long ulIPAddress = inet_addr("166.78.16.148");
```

Converting a Binary IP Address to a String

- **inet_ntoa()** performs the opposite job of **inet_addr()**. Its function prototype is as follows:

```
char *  inet_ntoa(struct in_addr in);
```

- **in** is a structure that contains an Internet host addess. On success, the **inet_ntoa()** function returns a pointer to a string with a dotted-decimal representation of the IP address. On error, NULL is returned. A NULL value means that the IP address passed as the in parameter is invalid.E.g.
- // first get an unsigned long with a valid IP address

```
u_long ulIPAddress = inet_addr("166.78.16.148");
```

- // copy the four bytes of the IP address into an in_addr structure IN_ADDR in;

```
memcpy(&in, &ulIPAddress, 4);
```

- // convert the IP address back into a string

```
char lpszIPAddress[16];
lstrcpy(lpszIPAddress, inet_ntoa(in));
```

What's My Name?

- Some applications need to know the name of the computer on which they are running. The **gethostname()** function provides this functionality. The function's prototype looks like the following.

```
int gethostname(char  * name, int namelen);
```

- name is a pointer to a character array that will accept the null-terminated host name, and namelen is the size of that character array. The **gethostname()** function returns 0 (zero) on success and **SOCKET_ERROR** on failure. On a return value of **SOCKET_ERROR**, the application can call **WSAGetLastError()** to determine the specifics of the problem.

```
#define HOST_NAME_LEN 50
char lpszHostName[HOST_NAME_LEN]; // will accept the host name
char lpszMessage[100]; // informational message
if (gethostname(lpszHostName, HOST_NAME_LEN) == 0) {
printf(lpszMessage, "This computer's name is %s", lpszHostName);
} else {
      printf(lpszMessage, "gethostname() generated error %d", WSAGetLastError());
}
```

Finding a Host's IP Address

- The function of **gethostbyname()** is to take a host name and return its IP address. This function, and its asynchronous counterpart named **WSAAsyncGetHostByName()**, may perform a simple table lookup on a host file local to the computer on which the program us running, or it may send the request across the network to a name server.
- The function's prototype looks like the following:

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

- name is a pointer to a null-terminated character array that contains the name of the computer about which you want host information. The **hostent** structure returned has the following format:

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

Asynchronously Finding a Host's IP Address

- In the getXbyY functions, one of which is gethostbyname(), the data retrieved might come from the local host or might come by way of a request over the network to a server of some kind.
- Consequently, the application has to be concerned with response times and the responsiveness of the application to the user while those network requests are taking place.
- The **WSAAsyncGetHostByName**() function is the asynchronous version of gethostbyname().
- The function prototype for WSAAsyncGetHostByName() is as follows:

```
HANDLE WSAAsyncGetHostByName(HWND hwnd, u_int wMsg, const char *name, char *buf, int buflen);
```

- **hWnd** is the handle to the window to which a message will be sent when **WSAAsyncGetHostByName**() has completed its asynchronous operation.
- **wMsg** is the message that will be posted to **hWnd** when the asynchronous operation is complete.
- **name** is a pointer to a string that contains the host name for which information is being requested.
- **buf** is a pointer to an area of memory that, on successful completion of the host name lookup, will contain the **hostent** structure for the desired host.
- **buflen** is the size of the buf buffer. It should be MAXGETHOSTSTRUCT for safety's sake.
- If the asynchronous operation is initiated successfully, the return value of **WSAAsyncGetHostByName**() is a handle to the asynchronous task. On failure of initialization, the function returns 0 (zero), and **WSAGetLastError**() should be called to find out the reason for the error.

Canceling an Outstanding Asynchronous Request

- The handle returned by the asynchronous database functions, such as **WSAAsyncGetHostByName**(), can be used to terminate the database lookup.
- The **WSACancelAsyncRequest**() function performs this task. Its prototype is the following:

```
int WSACancelAsyncRequest(HANDLE hAsyncTaskHandle);
```

- **hAsyncTaskHandle** is the handle to the asynchronous task you wish to abort.
- On success, this function returns 0 (zero).
- On failure, it returns SOCKET_ERROR, and **WSAGetLastError**() can be called.

Finding a Host Name When You Know Its IP Address

- The function of **gethostbyaddr**() is to take the IP address of a host and return its name. This function, and its asynchronous counterpart named **WSAAsyncGetHostByAddr**(), might perform a simple table lookup on a host file local to the computer on which the program is running, or it might send the request across the network to a name server. The function's prototype looks like the following:

```
struct hostent * PASCAL gethostbyaddr(const char * addr, int len, int type);
```

- **addr** is a pointer to the IP address, in network byte order, of the computer about which you want host information.
- **len** is the length of the address to which addr points. In WinSock 1.1, the length is always four because this version of the specification supports only Internet style addressing.
- **type** must always be **PF_INET** for the same reason.
- The **gethostbyaddr**() function returns a pointer to a **hostent** host entry structure on success and NULL on failure.
- Upon a return value of NULL, you can call **WSAGetLastError**() to determine the specifics of the problem.

Asynchronously Finding a Host Name When You Know Its IP Address

- The **WSAAsyncGetHostByAddr**() function is the asynchronous version of gethostbyaddr(). Its function prototype is as follows:

```
HANDLE WSAAsyncGetHostByAddr(HWND hWnd, u_int wMsg, const char * addr,
int len, int type, char * buf, int buflen);
```

- **hWnd** is the handle to the window to which a message will be sent when **WSAAsyncGetHostByAddr**() has completed its asynchronous operation.
- **wMsg** is the user defined message that will be posted to **hWnd** when the asynchronous operation is complete.
- **addr** is a pointer to the IP address, in network byte order, of the computer about which you want host information.
- **len** is the length of the address to which **addr** points and is always 4 (four) for Internet addresses. type must always be PF_INET because WinSock 1.1 supports only Internet-style addressing.
- **buf** is a pointer to an area of memory that, upon successful completion of the address lookup, will contain the hostent structure for the desired host.
- **buflen** is the size of the buf buffer. It should be MAXGETHOSTSTRUCT for safety's sake.
- If the asynchronous operation is initiated successfully, the return value of **WSAAsyncGetHostByAddr**() is a handle to the asynchronous task. On failure of initialization, the function returns 0 (zero) and WSAGetLastError() should be called to find out the reason for the error.

Finding a Service's Port Number

- The **getservbyname**() function gets service information corresponding to a specific service name and protocol. Its function prototype looks like the following:

```
struct servent *  getservbyname(const char * name, const char * proto);
```

- name is a pointer to a string that contains the service for which you are searching. proto is a pointer to a string that contains the transport protocol to use; it's either "udp", "tcp", or NULL. A NULL proto will match on the first service in the services table that has the specified name, regardless of the protocol. The **servent** structure returned has the following format:

```
struct servent {
char * s_name; // official service name
char ** s_aliases; // alias list
short s_port; // port #
char * s_proto; // protocol to use
};
```

- The **getservbyname**() function returns a pointer to a servent structure on success and NULL on failure. On a return value of NULL, you can call WSAGetLastError() to determine the specifics of the problem.

Asynchronously Finding a Service's Port Number

- **WSAAsyncGetServByName**() is the asynchronous counterpart to getservbyname(). Its function prototype is as follows:
- `HANDLE WSAAsyncGetServByName(HWND hWnd, u_int wMsg, const char * name, const char * proto, char * buf, int buflen);`
- **hWnd** is the handle to the window to which a message will be sent when WSAAsyncGetServByName() has completed its asynchronous operation.
- **wMsg** is the user defined message that will be posted to hWnd when the asynchronous operation is complete.
- **name** is a pointer to a service name about which you want service information. proto is a pointer to a protocol name; it is "tcp", "udp", or NULL.
- If **proto** is NULL, the first matching service is returned.
- **buf** is a pointer to an area of memory that, on successful completion of the service lookup, will contain the servent structure for the desired service.
- **buflen** is the size of the buf buffer.
- It should be MAXGETHOSTSTRUCT for safety's sake.

Finding a Service Name When You Know Its Port Number

- The **getservbyport**() function gets service information corresponding to a specific port and protocol. Its function prototype looks like the following:

```
struct servent FAR * PASCAL FAR getservbyport(int port, const char FAR * proto);
```

- **port** is the service port, in network byte order.
- **proto** is a pointer to a protocol name; it is "tcp", "udp", or NULL.
- If **proto** is NULL, the first matching service is returned.
- The **getservbyport**() function returns a pointer to a servent structure on success and NULL on failure.
- On a return value of NULL, you can call WSAGetLastError() to determine the specifics of the problem.

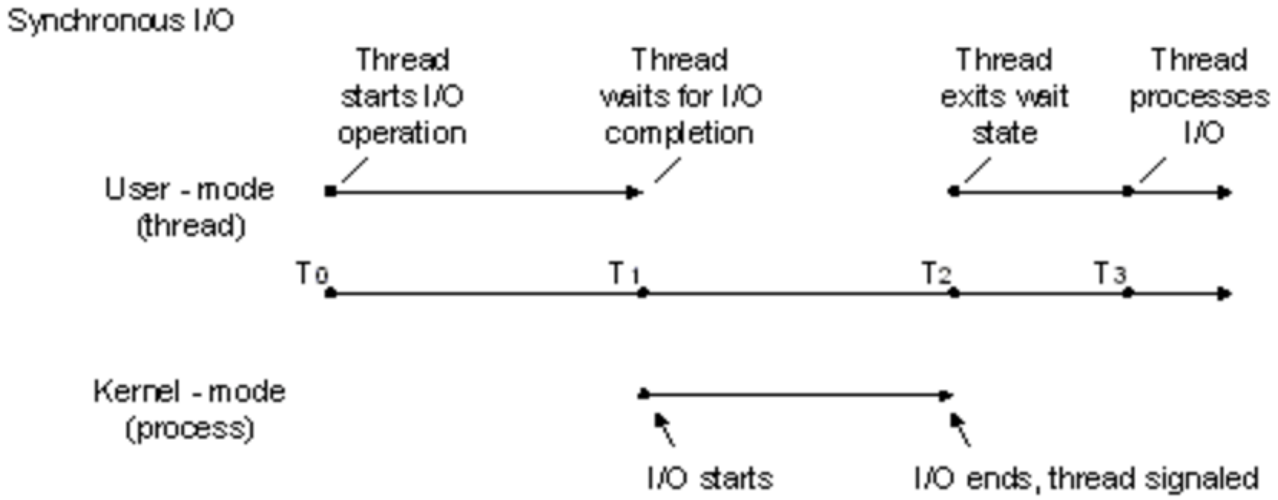
Asynchronously Finding a Service Name When You Know Its Port Number

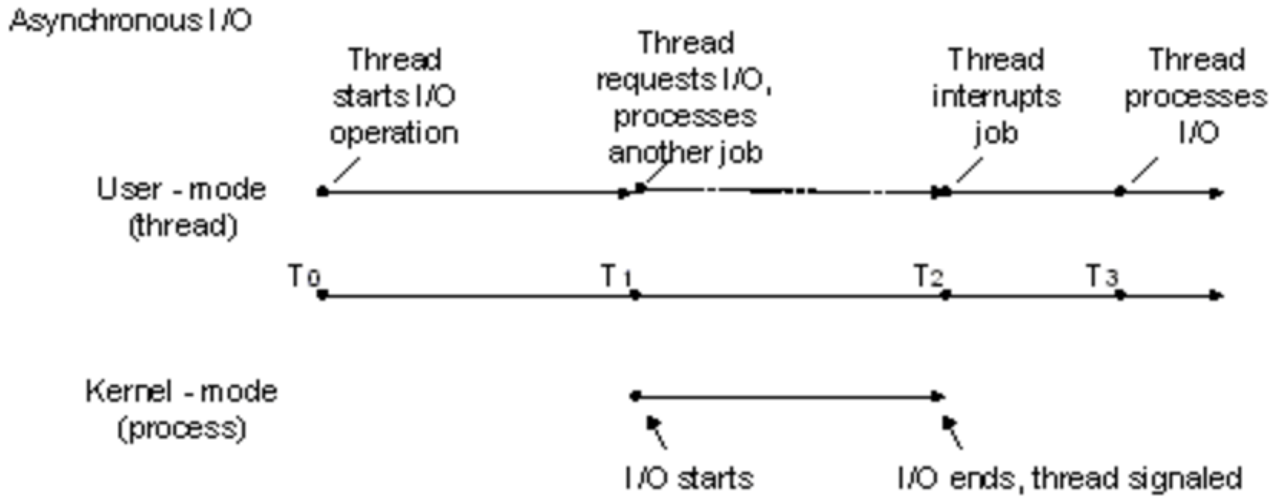
- WSAAsyncGetServByPort() is the asynchronous counterpart to getservbyport(). Its function prototype is as follows:

```
HANDLE PASCAL FAR WSAAsyncGetServByPort(HWND hWnd, u_int wMsg, int port,
const char FAR * proto, char FAR * buf, int buflen);
```

- **hWnd** is the handle to the window to which a message will be sent when WSAAsyncGetServByName() has completed its asynchronous operation.
- **wMsg** is the user defined message that will be posted to hWnd when the asynchronous operation is complete.
- **port** is the service port, in network byte order, of the service about which you want information. proto is a pointer to a protocol name; it is "tcp", "udp", or NULL.
- If **proto** is NULL, the first matching service is returned.
- **buf** is a pointer to an area of memory that, on successful completion of the service lookup, will contain the servent structure for the desired service.
- **buflen** is the size of the buf buffer. It should be MAXGETHOSTSTRUCT for safety's sake.
- If the asynchronous operation is initiated successfully, the return value of WSAAsyncGetServByName() is a handle to the asynchronous task. On failure of initialization, the function returns 0 (zero) and WSAGetLastError() should be called to find out the reason for the error.

SYNCHRONOUS AND ASYNCHRONOUS I/O





ASYNCHRONOUS I/O FUNCTIONS

- There are two types of input/output (I/O) synchronization: **synchronous I/O** and **asynchronous I/O**.
- Asynchronous I/O is also referred to as overlapped I/O.
- In synchronous file I/O, a thread starts an I/O operation and immediately enters a wait state until the I/O request has completed.
- A thread performing asynchronous file I/O sends an I/O request to the kernel by calling an appropriate function.
- If the request is accepted by the kernel, the calling thread continues processing another job until the kernel signals to the thread that the I/O operation is complete.
- It then interrupts its current job and processes the data from the I/O operation as necessary.
- In situations where an I/O request is expected to take a large amount of time, such as a refresh or backup of a large database or a slow communications link, asynchronous I/O is generally a good way to optimize processing efficiency.
- However, for relatively fast I/O operations, the overhead of processing kernel I/O requests and kernel signals may make asynchronous I/O less beneficial, particularly if many fast I/O operations need to be made. In this case, synchronous I/O would be better.
- WinSock provides functions (**WSASend**, **WSASendTo**, **WSARecv**, **WSARecvFrom**, **WSAIoctl**, etc.) that support both synchronous and asynchronous I/O.
- **WSASocket** is used to create overlapped socket.

```
SOCKET WSASocket(
_In_ int                 af,
_In_ int                 type,
_In_ int                 protocol,
_In_ LPWSAPROTOCOL_INFO lpProtocolInfo,
_In_ GROUP               g,
_In_ DWORD               dwFlags);   // Used to create overlapped socket
```

- **af[in]** : address family; AF_INET, AF_INET6, AF_UNSPEC
- **type[in]** : type of the new socket; SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
- **protocol[in]** : the protocol to be used. If a value of 0 is specified, the caller does not wish to specify a protocol and the service provider will choose the protocol to use; else IPPROTO_TCP, IPPROTO_UDP, IPROTO_ICMP
- **lpProtocolInfo[in]** : A pointer to a WSAPROTOCOL_INFO structure that defines the characteristics of the socket to be created.
- **g [in]** : An existing socket group ID.
- **dwFlags[in]** : A set of flags used to specify additional socket attributes.

WSASEND

```
int WsaSend(
_In_  SOCKET                              s,
_In_  LPWSABUF                            lpBuffers,
_In_   DWORD                              dwBufferCount,
_Out_  LPDWORD                            lpNumberOfBytesSent,
_In_   DWORD                              dwFlags,
_In_   LPWSAOVERLAPPED                    lpOverlapped,
_In_   LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine );
```

- **s[in]** : A descriptor that identifies a connected socket.
- **lpBuffers[in]** : A pointer to an array of WSABUF structures. Each WSABUF structure contains a pointer to a buffer and the length, in bytes, of the buffer.
- **dwBufferCount[in]** : The number of WSABUF structures in the lpBuffers array.
- **lpNumberOfBytesSent[out]** : The pointer to the number, in bytes, sent by this call if the I/O operation completes immediately. NULL for overlapped socket.
- **dwFlags[in]** : The flags used to modify the behavior of the WSASend function call.
- **lpOverlapped[in]** : A pointer to a WSAOVERLAPPED structure. This parameter is ignored for non-overlapped sockets.
- **lpCompletionRoutine[in]** : A pointer to the completion routine (function) called when the send operation has been completed. This parameter is ignored for non- overlapped sockets.

WSASENDTO

```
int WSASendTo(
_In_   SOCKET                             s,
_In_   LPWSABUF                           lpBuffers,
_In_   DWORD                              dwBufferCount,
_Out_  LPDWORD                            lpNumberOfBytesSent,
_In_   DWORD                              dwFlags,
_In_   const struct sockaddr             *lpTo,
_In_   int                                iToLen,
_In_   LPWSAOVERLAPPED                    lpOverlapped,
_In_   LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine );
```

- **s[in]** : A descriptor identifying a socket.
- **lpBuffers[in]** : A pointer to an array of WSABUF structures. Each WSABUF structure contains a pointer to a buffer and the length of the buffer, in bytes. This array must remain valid for the duration of the send operation.
- **dwBufferCount[in]** : The number of WSABUF structures in the lpBuffers array.
- **lpOverlapped[in]** : A pointer to a WSAOVERLAPPED structure (ignored for nonoverlapped sockets).

- **lpCompletionRoutine[in]** : A pointer to the completion routine called when the send operation has been completed (ignored for nonoverlapped sockets).
- **lpNumberofBytesSent[out]** : The pointer to the number, in bytes, sent by this call if the I/O operation completes immediately. NULL for overlapped socket.
- **dwFlags[in]** : The flags used to modify the behavior of the WSASendTo call.
- **lpTo[in]** : An optional pointer to the address of the target socket in the SOCKADDR structure.
- **iToLen** [**in**] : The size, in bytes, of the address in the IpTo parameter.

WSARECV

```
int WSARecv(
_In_     SOCKET                             s,
_Inout_  LPWSABUF                           lpBuffers,
_In_     DWORD                              dwBufferCount,
_Out_    LPDWORD                            lpNumberOfBytesRecvd,
_Inout_  LPDWORD                            lpFlags,
_In_     LPWSAOVERLAPPED                    lpOverlapped,
_In_     LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine );
```

- **s[in]** : A descriptor identifying a connected socket.
- **lpBuffers[in, out]** : A pointer to an array of WSABUF strucutres. Each WSABUF structure contains a pointer to a buffer and the length, in bytes, of the buffer.
- **dwBufferCount[in]** : The number of WSABUF structures in the lpBuffers array.
- **lpNumberofBytesRecvd[out]** : A pointer to the number, in bytes, of data received by this call if the receive operation competes immediately. (NULL for overlapped socket)
- **lpFlags [in, out]** : A pointer to flags used to modify the behavior of the WSARecv function call.
- **lpOverlapped[in]** : A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets).
- **lpCompletionRoutine[in]** : A pointer to the completion routine called when the receive operation has been completed (ignored for non-overlapped sockets).

WSARECVFROM

```
int WSARecvFrom(
  _In_    SOCKET                             s,
  _Inout_ LPWSABUF                           lpBuffers,
  _In_    DWORD                              dwBufferCount,
  _Out_   LPDWORD                            lpNumberOfBytesRecvd,
  _Inout_ LPDWORD                            lpFlags,
  _Out_   struct sockaddr                    *lpFrom,
  _Inout_ LPINT                              lpFromlen,
  _In_    LPWSAOVERLAPPED                    lpOverlapped,
  _In_    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

- **s [in]** : A descriptor identifying a socket.
- **lpBuffers [in, out]** : A pointer to an array of WSABUF structures. Each WSABUF structure contains a pointer to a buffer and the length of the buffer.
- **dwBufferCount [in]** : The number of WSABUF structures in the lpBuffers array.
- **lpNumberofBytesRecvd[out]** : A pointer to the number of bytes received by this call if the WSARecvFrom operation completes immediately. (NULL for overlapped socket)
- **lpFlags [in, out]** : A pointer to flags used to modify the behavior of the WSARecvFrom function call.
- **lpFrom [out]** : An optional pointer to a buffer that will hold the source address upon the completion of the overlapped operation.
- **lpFromlen [in, out]** : A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets).
- **lpCompletionRoutine [in]** : A pointer to the completion routine called when the WSARecvFrom operation has been completed (ignored for nonoverlapped sockets).

WSAIOCTL

```
int WSAIoctl(
_In_   SOCKET                              s,
_In_   DWORD                               dwIoControlCode,
_In_   LPVOID                              lpvInBuffer,
_In_   DWORD                               cbInBuffer,
_Out_  LPVOID                              lpvOutBuffer,
_In_   DWORD                               cbOutBuffer,
_Out_  LPDWORD                             lpcbBytesReturned,
_In_   LPWSAOVERLAPPED                     lpOverlapped,
_In_   LPWSAOVERLAPPED_COMPLETION_ROUTINE  lpCompletionRoutine );
```

- **s [in]** : A descriptor identifying a socket.
- **dwloControlCode [in]** : The control code of operation to perform
- **lpInBuffer [in]** : A pointer to the input buffer
- **cbInBuffer [in]** : The size, in bytes, of the input buffer
- **lpOutBuffer [out]** : A pointer to the output buffer
- **cbOutBuffer [in]** : The size, in bytes, of the output buffer
- **lpcbBytesReturned [out]** :A pointer to actual number of bytes of output
- **lpOverlapped [in]** : A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets)
- **lpCompletionRoutine [in]** : A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets)

ERROR HANDLING FUNCTIONS; ASYNCHRONOUS OPERATION

- If asynchronous function returns SOCKET_ERROR, and the specific error code retrieved by calling WSAGetLastError() is WSA_ISO_PENDING, then it means the operlapped operation has been successfully initiated and the completion will be indicated at a later time.
- Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.
- When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine(if specified), or through the *lpcbTransfer* parameter in **WSAGetOverlappedResult**.
- ***BOOL WSAAPI WSAGetOverlappedResult(_In_ SOCKET s,_In_ LPWSAOVERLAPPED lpOverlapped,_Out_ LPDWORD lpcbTransfer,_In_ BOOL fWait,_Out_ LPDWORD lpdwFlags);***
- **s [in]**: A descriptor identifying the socket.
- **lpOverlapped [in]**: A pointer to a WSAOVERLAPPED structure that was specified when the overlapped operation was started. This parameter must not be a NULL pointer.
- **lpcbTransfer [out]**: A pointer to a 32-bit variable that receives the number of bytes that were actually transferred by a send or receive operation, or by the WSAIoctl function.

- **fWait [in]**: A flag that specifies whether the function should wait for the pending overlapped operation to complete.
- **lpdwFlags [out]**: A pointer to a 32-bit variable that will receive one or more flags that supplement the completion status.
- If **WSAGetOverlappedResult** succeeds, the return value is **TRUE.** This means that the overlapped operation has completed successfully and that the value pointed to by *lpcbTransfer* has been updated.
- If **WSAGetOverlappedResult** returns **FALSE**, this means that either the overlapped operation has not completed, the overlapped operation completed but with errors, or the overlapped operation's completion status could not be determined due to errors in one or more parameters to **WSAGetOverlappedResult**.
- On failure, the value pointed to by *lpcbTransfer* will not be updated. Use WSAGetLastError to determine the cause of the failure (either by the WSAGetOverlappedResult function or by the associated overlapped operation).

USING NON-BLOCKING SOCKET, NON-BLOCKING WITH CONNECT

- When connect is called on a blocking socket, the function blocks i.e. the function doesn't return until the TCP's 3-way handshake is completed (actually, until SYN, ACK is received from remote end).
- The **ioctlsocket** function can be used to set socket in non-blocking mode. With a non-blocking socket, the connect returns immediately without completion.
- In this case, connect will return SOCKET_ERROR, and WSAGetLastError will return WSAEWOULDBLOCK.
- In this case, there are three possible scenarios.
  - Use the select function to determine the completion of the connection request by checking to see if the socket is writeable. If connect fails, failure of the connect attempt is indicated in exceptfds (application must then call getsockopt SO_ERROR to determine the error value to describe why the failure occurred).
  - If the application is using WSAAsyncSelect to indicate interest in connect events, then the application will receive an FD_CONNECT notification indicating that the connect operation is complete (successfully or not).
  - If the application is using WSAEventSelect to indicate interest in connection events, then the associated event object will be signaled indicating that the connect operation is complete (successfully or not).

SELECT IN CONJUNCTION WITH ACCEPT, SELECT WITH RECV/RECVFROM AND SEND/SENDTO

Select with accept

- The accept function can block the caller until a connection is present if no pending connections are present on the queue, and the socket is marked as blocking.
- If the socket is marked as non-blocking and no pending connections are present on the queue, accept returns an error.
- When using non-blocking socket, select function can be used to determine if the connecting is present by checking to see if the socket is readable. Then, accept returns successfully, immediately.

Select with recv/recvfrom

- With blocking socket, recv/recvfrom blocks if no data is available. When using non-blocking socket, select function can be used to determine if the data is available by checking to see if the socket is readable. Then, recv/recvfrom returns successfully, immediately.

Select with send/sendto

- With blocking socket, send/sendto blocks if data can't be written to the kernel. When using non-blocking socket, select function can be used to determine if the data can be written to the kernel by checking to see if the socket is writable. Then, send/sendto returns successfully, immediately.

Getting Started with WinSock

General model for creating a streaming TCP/IP Server and Client.

Client

1. Initialize Winsock.
2. Create a socket.
3. Connect to the server.
4. Send and receive data.
5. Disconnect.

Server

1. Initialize Winsock.
2. Create a socket.
3. Bind the socket.
4. Listen on the socket for a client.
5. Accept a connection from a client.
6. Receive and send data.
7. Disconnect.

Initializing Winsock

- All processes (applications or DLLs) that call Winsock functions must initialize the use of the Windows Sockets DLL before making other Winsock functions calls. This also makes certain that Winsock is supported on the system.

To initialize Winsock

1. Create a WSADATA object called wsaData.

```
WSADATA wsaData;
```

2. Call WSAStartup and return its value as an integer and check for errors.

```
int iResult;

// Initialize Winsock

iResult = WSAStartup(MAKEWORD(2,2), &wsaData);

if (iResult != 0) {

printf("WSAStartup failed: %d\n", iResult);

return 1;

}
```

- The **WSAStartup** function is called to initiate use of WS2_32.dll. The WSADATA structure contains information about the Windows Sockets implementation. The MAKEWORD(2,2) parameter of WSAStartup makes a request for version 2.2 of Winsock on the system, and sets the passed version as the highest version of Windows Sockets support that the caller can use.

Creating a Socket for the Client

- After initialization, a SOCKET object must be instantiated for use by the client.

To create a socket

1. Declare an **addrinfo** object that contains a **sockaddr** structure and initialize these values. For this application, the Internet address family is unspecified so that either an IPv6 or IPv4 address can be returned. The application requests the socket type to be a stream socket for the TCP protocol.

```
struct addrinfo *result = NULL,*ptr = NULL, hints;
ZeroMemory( &hints, sizeof(hints) );
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
```

2. Call the **getaddrinfo** function requesting the IP address for the server name passed on the command line. The TCP port on the server that the client will connect to is defined by DEFAULT_PORT as 27015 in this sample. The **getaddrinfo** function returns its value as an integer that is checked for errors.

```
#define DEFAULT_PORT "27015"
// Resolve the server address and port
iResult = getaddrinfo(argv[1], DEFAULT_PORT, &hints, &result);
if (iResult != 0) {
printf("getaddrinfo failed: %d\n", iResult);
WSACleanup();
return 1;
}
```

3. Create a SOCKET object called ConnectSocket.

- SOCKET ConnectSocket = INVALID_SOCKET;

4. Call the socket function and return its value to the ConnectSocket variable. For this application, use the first IP address returned by the call to getaddrinfo that matched the address family, socket type, and protocol specified in the hints parameter. In this example, a TCP stream socket was specified with a socket type of SOCK_STREAM and a protocol of IPPROTO_TCP. The address family was left unspecified (AF_UNSPEC), so the returned IP address could be either an IPv6 or IPv4 address for the server. If the client application wants to connect using only IPv6 or IPv4, then the address family needs to be set to AF_INET6 for IPv6 or AF_INET for IPv4 in the hints parameter.

```
// Attempt to connect to the first address returned by
// the call to getaddrinfo
ptr=result;
// Create a SOCKET for connecting to server
ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
```

5. Check for errors to ensure that the socket is a valid socket.

```
if (ConnectSocket == INVALID_SOCKET) {
printf("Error at socket(): %ld\n", WSAGetLastError());
freeaddrinfo(result);
WSACleanup();
return 1;
}
```

- The parameters passed to the socket function can be changed for different implementations. Error detection is a key part of successful networking code. If the socket call fails, it returns INVALID_SOCKET. The if statement in the previous code is used to catch any errors that may have occurred while creating the socket. WSAGetLastError returns an error number associated with the last error that occurred.

Connecting to a Socket

- For a client to communicate on a network, it must connect to a server.
- **To connect to a socket**
- Call the connect function, passing the created socket and the sockaddr structure as parameters. Check for general errors.

```
// Connect to server.

iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);

if (iResult == SOCKET_ERROR) {

closesocket(ConnectSocket);

ConnectSocket = INVALID_SOCKET;

}

// Should really try the next address returned by getaddrinfo

// if the connect call failed

freeaddrinfo(result);

if (ConnectSocket == INVALID_SOCKET) {

printf("Unable to connect to server!\n");

WSACleanup();

return 1;
}
```

- The getaddrinfo function is used to determine the values in the sockaddr structure.
- In this example, the first IP address returned by the getaddrinfo function is used to specify the sockaddr structure passed to the connect.
- If the connect call fails to the first IP address, then try the next addrinfo structure in the linked list returned from the getaddrinfo function.
- The information specified in the sockaddr structure includes:
  - the IP address of the server that the client will try to connect to.
  - the port number on the server that the client will connect to. This port was specified as port 27015 when the client called the getaddrinfo function.

SENDING AND RECEIVING DATA ON THE CLIENT

Client

```
#define DEFAULT_BUFLEN 512

int recvbuflen = DEFAULT_BUFLEN;

char *sendbuf = "this is a test";

char recvbuf[DEFAULT_BUFLEN];

int iResult;

// Send an initial buffer

iResult = send(ConnectSocket, sendbuf, (int)
strlen(sendbuf), 0);

if (iResult == SOCKET_ERROR) {

printf("send failed: %d\n", WSAGetLastError());

closesocket(ConnectSocket);

WSACleanup();

return 1;

}

printf("Bytes Sent: %ld\n", iResult);

// shutdown the connection for sending since no more data will be sent
// the client can still use the ConnectSocket for receiving data
iResult = shutdown(ConnectSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
printf("shutdown failed: %d\n", WSAGetLastError());
closesocket(ConnectSocket);
WSACleanup();
return 1;
}
// Receive data until the server closes the connection
do {
iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
if (iResult > 0)
printf("Bytes received: %d\n", iResult);
else if (iResult == 0)
printf("Connection closed\n");
else
printf("recv failed: %d\n", WSAGetLastError());
} while (iResult > 0);
```

Disconnecting the Client

- Once the client is completed sending and receiving data, the client disconnects from the server and shutdowns the socket.

To disconnect and shutdown a socket

1. When the client is done sending data to the server, the shutdown function can be called specifying SD_SEND to shutdown the sending side of the socket. This allows the server to release some of the resources for this socket. The client application can still receive data on the socket.

```
// shutdown the send half of the connection since no more data will be sent
iResult = shutdown(ConnectSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
printf("shutdown failed: %d\n", WSAGetLastError());
closesocket(ConnectSocket);
WSACleanup();
return 1;
}
```

2. When the client application is done receiving data, the closesocket function is called to close the socket. When the client application is completed using the Windows Sockets DLL, the WSACleanup function is called to release resources.

```
// cleanup

closesocket(ConnectSocket);

WSACleanup();

return 0;
```

CREATING A SOCKET FOR THE SERVER

- After initialization, a SOCKET object must be instantiated for use by the server.

To create a socket for the server

1. The getaddrinfo function is used to determine the values in the sockaddr structure:
   - AF_INET is used to specify the IPv4 address family.
   - SOCK_STREAM is used to specify a stream socket.
   - IPPROTO_TCP is used to specify the TCP protocol.
   - AI_PASSIVE flag indicates the caller intends to use the returned socket address structure in a call to the bind function.
   - When the AI_PASSIVE flag is set and nodename parameter to the getaddrinfo function is a NULL pointer, the IP address portion of the socket address structure is set to INADDR_ANY for IPv4 addresses or IN6ADDR_ANY_INIT for IPv6 addresses.

   - 27015 is the port number associated with the server that the client will connect to.

- The addrinfo structure is used by the getaddrinfo function.

```
#define DEFAULT_PORT "27015"
struct addrinfo *result = NULL, *ptr = NULL, hints;
Compiled by Chandra Mohan Jayaswal
ZeroMemory(&hints, sizeof (hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE;
// Resolve the local address and port to be used by the server
iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
if (iResult != 0) {
printf("getaddrinfo failed: %d\n", iResult);
WSACleanup();
return 1;
}
```

2. Create a SOCKET object called ListenSocket for the server to listen for client connections.

SOCKET ListenSocket = INVALID_SOCKET;

3. Call the socket function and return its value to the ListenSocket variable. For this server application, use the first IP address returned by the call to getaddrinfo that matched the address family, socket type, and protocol specified in the hints parameter.
   - If the server application wants to listen on IPv6, then the address family needs to be set to AF_INET6 in the hints parameter.
   - If a server wants to listen on both IPv6 and IPv4, two listen sockets must be created, one for IPv6 and one for IPv4.
   - These two sockets must be handled separately by the application. Windows Vista and later offer the ability to create a single IPv6 socket that is put in dual stack mode to listen on both IPv6 and IPv4. For more information on this feature.

// Create a SOCKET for the server to listen for client connections

ListenSocket = socket(result->ai_family, result->ai_socktype, result->ai_protocol);

4. Check for errors to ensure that the socket is a valid socket.

```
if (ListenSocket == INVALID_SOCKET) {
printf("Error at socket(): %ld\n", WSAGetLastError());
freeaddrinfo(result);
WSACleanup();
return 1;
}
```

Binding a Socket

- For a server to accept client connections, it must be bound to a network address within the system. The following code demonstrates how to bind a socket that has already been created to an IP address and port. Client applications use the IP address and port to connect to the host network.

To bind a socket

- The sockaddr structure holds information regarding the address family, IP address, and port number. Call the bind function, passing the created socket and sockaddr structure returned from the getaddrinfo function as parameters. Check for general errors.

```
// Setup the TCP listening socket

iResult = bind( ListenSocket, result->ai_addr, (int)result->ai_addrlen);

if (iResult == SOCKET_ERROR) {

printf("bind failed with error: %d\n", WSAGetLastError());

freeaddrinfo(result);

closesocket(ListenSocket);

WSACleanup();

return 1;

}
```

- Once the bind function is called, the address information returned by the getaddrinfo function is no longer needed. The freeaddrinfo function is called to free the memory allocated by the getaddrinfo function for this address information.

freeaddrinfo(result);

Listening on a Socket

- After the socket is bound to an IP address and port on the system, the server must then listen on that IP address and port for incoming connection requests.

To listen on a socket

- Call the listen function, passing as parameters the created socket and a value for the backlog, maximum length of the queue of pending connections to accept. In this example, the backlog parameter was set to SOMAXCONN. This value is a special constant that instructs the Winsock provider for this socket to allow a maximum reasonable number of pending connections in the queue.

```
if ( listen( ListenSocket, SOMAXCONN ) == SOCKET_ERROR ) {
printf( "Listen failed with error: %ld\n", WSAGetLastError() );
closesocket(ListenSocket);
WSACleanup();
return 1;
}
```

Accepting a Connection

- Once the socket is listening for a connection, the program must handle connection requests on that socket.

To accept a connection on a socket

1. Create a temporary SOCKET object called ClientSocket for accepting connections from clients.

```
SOCKET ClientSocket;
```

2. Normally a server application would be designed to listen for connections from multiple clients. For high-performance servers, multiple threads are commonly used to handle the multiple client connections. One programming technique is to create a continuous loop that checks for connection requests using the listen function. If a connection request occurs, the application calls the accept, AcceptEx, or WSAAccept function and passes the work to another thread to handle the request.

```
ClientSocket = INVALID_SOCKET;

// Accept a client socket

ClientSocket = accept(ListenSocket, NULL, NULL);

if (ClientSocket == INVALID_SOCKET) {

printf("accept failed: %d\n", WSAGetLastError());

closesocket(ListenSocket);

WSACleanup();

return 1;

}
```

3. When the client connection has been accepted, a server application would normally pass the accepted client socket to a worker thread or an I/O completion port and continue accepting additional connections.
   - There are a number of other programming techniques that can be used to listen for and accept multiple connections.
   - These include using the select or WSAPoll functions.

Note: On Unix systems, a common programming technique for servers was for an application to listen for connections. When a connection was accepted, the parent process would call the fork function to create a new child process to handle the client connection, inheriting the socket from the parent. This programming technique is not supported on Windows, since the fork function is not supported. This technique is also not usually suitable for high-performance servers, since the resources needed to create a new process are much greater than those needed for a thread.

RECEIVING AND SENDING DATA ON THE SERVER

To receive and send data on a socket

```
#define DEFAULT_BUFLEN 512
char recvbuf[DEFAULT_BUFLEN];
int iResult, iSendResult;
int recvbuflen = DEFAULT_BUFLEN;
// Receive until the peer shuts down the connection
do {
iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
if (iResult > 0) {
printf("Bytes received: %d\n", iResult);
// Echo the buffer back to the sender
iSendResult = send(ClientSocket, recvbuf, iResult,
0);
if (iSendResult == SOCKET_ERROR) {
printf("send failed: %d\n", WSAGetLastError());
closesocket(ClientSocket);
WSACleanup();
return 1;
}
printf("Bytes sent: %d\n", iSendResult);
} else if (iResult == 0)
printf("Connection closing...\n");
else {
printf("recv failed: %d\n", WSAGetLastError());
closesocket(ClientSocket);
WSACleanup();
return 1;
}
} while (iResult > 0);
```

The send and recv functions both return an integer value of the number of bytes sent or received, respectively, or an error.

DISCONNECTING THE SERVER

- Once the server is completed receiving data from the client and sending data back to the client, the server disconnects from the client and shutdowns the socket.

To disconnect and shutdown a socket

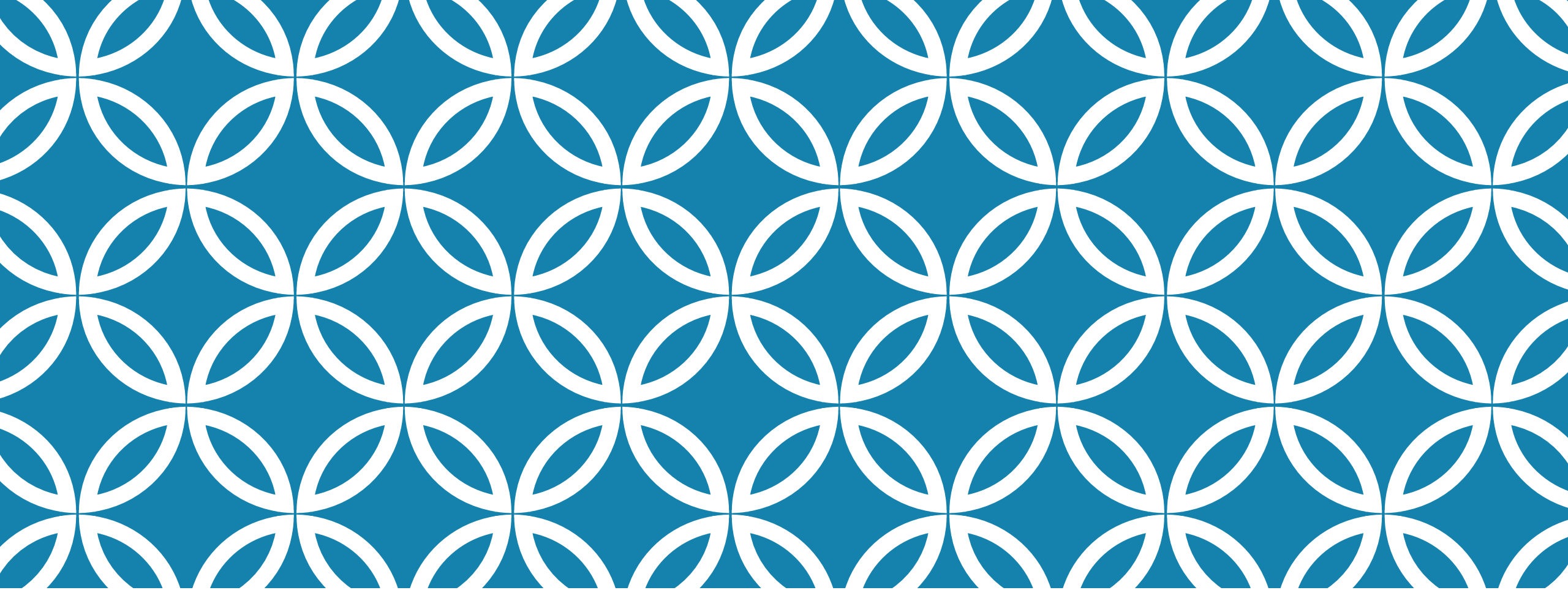
1. When the server is done sending data to the client, the shutdown function can be called specifying SD_SEND to shutdown the sending side of the socket. This allows the client to release some of the resources for this socket. The server application can still receive data on the socket.

```
// shutdown the send half of the connection since no more data will be sent
iResult = shutdown(ClientSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
printf("shutdown failed: %d\n",WSAGetLastError());
closesocket(ClientSocket);
Compiled by Chandra Mohan Jayaswal
WSACleanup();
return 1;
}
```

2. When the client application is done receiving data, the closesocket function is called to close the socket. When the client application is completed using the Windows Sockets DLL, the WSACleanup function is called to release resources.

```
// cleanup

closesocket(ClientSocket);

WSACleanup();

return 0;
```

END OF CHAPTER 3



NETWORK PROGRAMMING

Compiled by:
Er. Madan Kadariya
NCIT

CHAPTER 4:
NETWORK UTILITIES AND APPLICATION

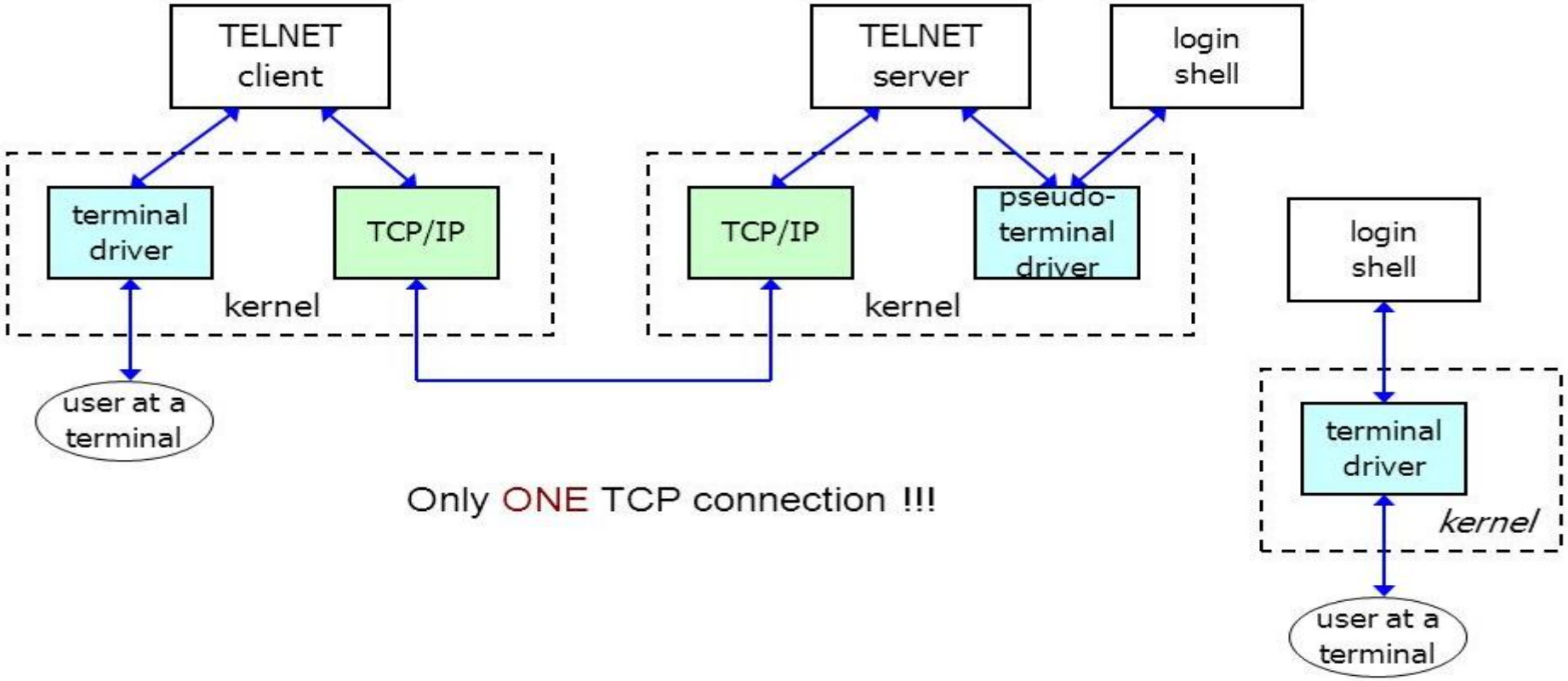
TELNET AND RLOGIN: REMOTE LOGIN

- Remote login is one of the most popular Internet applications. Instead of having a hardwired terminal on each host, we can login to one host and then remote login across the network to any other host (that we have an account on, of course).
- Two popular applications provide remote login across TCP/IP internets.

1. **Telnet** is a standard application that almost every TCP/IP implementation provides. It works between hosts that use different operating systems. Telnet uses option negotiation between the client and server to determine what features each end can provide.
2. **Rlogin** is from Berkeley Unix and was developed to work between Unix systems only, but it has been ported to other operating systems also.

Remote login uses the client-server paradigm.

Fig: Overview of Telnet client-server.



1. The Telnet client interacts with both the user at the terminal and the TCP/IP protocols. Normally everything we type is sent across the TCP connection, and everything received from the connection is output to our terminal.
2. The Telnet server often deals with what's called a *pseudo-terminal* device, at least under Unix systems. This makes it appear to the login shell that's invoked on the server, and to any programs run by the login shell, that they're talking to a terminal device.
3. Only a single TCP connection is used. Since there are times when the Telnet client must talk to the Telnet server (and vice versa) there needs to be some way to delineate commands that are sent across the connection, versus user data.
4. In dashed boxes in Figure to note that the terminal and pseudo terminal drivers, along with the TCP/IP implementation, are normally part of the operating system kernel. The Telnet client and server, however, are often user applications.
5. The login shell on the server host to reiterate that we have to login to the server. We must have an account on that system to login to it, using either Telnet or Rlogin.

❖Remote login is not a high-volume data transfer application. Lots of small packets are normally exchanged between the two end systems. It is found that the ratio of bytes sent by the client (the user typing at the terminal) to the number of bytes sent back by the server is about 1:20. This is because we type short commands that often generate lots of output.

RLOGIN PROTOCOL

Rlogin appeared with 4.2BSD and was intended for remote login only between Unix hosts. This makes it a simpler protocol than Telnet, since option negotiation is not required when the operating system on the client and server are known in advance.

Application Startup:

- Rlogin uses a single TCP connection between the client and server. After the normal TCP connection establishment is complete, the following application protocol takes place between the client and server.

1. The client writes four strings to the server; (a) a byte of 0, (b) the login name of the user on the client host terminated by a byte of 0, (c) the login name of the user on the server host, terminated by a byte of 0, (d) the name of the user's terminal type, followed by a slash, followed by the terminal speed, terminated by a byte of 0. Two login names are required because users aren't required to have the same login name on each system. The terminal type is passed from the client to the server because many full-screen applications need to know it. The terminal speed is passed because some applications operate differently depending on the speed. For example, the vi editor works with a smaller window when operating at slower speeds, so it doesn't take forever to redraw the window.
2. The server responds with a byte of 0.

RLOGIN PROTOCOL...

3. The server has the option of asking the user to enter a password. This is handled as normal data exchange across the Rlogin connection-there is no special protocol. The server sends a string to the client (which the client displays on the terminal), often Password:. If the client does not enter a password within some time limit (often 60 seconds), the server closes the connection. We can create a file in our home directory on the server (named .rhosts) with lines containing a hostname and our username. If we login from the specified host with that username, we are not prompted for a password. If we are prompted by the server for a password, what we type is sent to the server as *cleartext.* Each character of the password that we type is sent as is. Anyone who can read the raw network packets can read the characters of our password. Newer implementations of the Rlogin client, such as 4.4BSD, first try to use Kerberos, which avoids sending cleartext passwords across the network. This requires a compatible server that also supports Kerberos.

4. The server normally sends a request to the client asking for the terminal's window size

RLOGIN...

- **Flow Control**

By default, flow control is done by the Rlogin client. The client recognizes the ASCII STOP and START characters (Control-S and Control-Q) typed by the user, and stops or starts the terminal output.

- **Client Interrupt**

It is rare for the flow of data from the client to the server to be stopped by flow control. This direction contains only characters that we type. Therefore it is not necessary for these special input characters (Control-S or interrupt) to be sent from the client to the server using TCP's urgent mode.

- **Window Size Changes**

With remote login, however, the change in the window size occurs on the client, but the application that needs to be told is running on the server. Some form of notification is required for the Rlogin client to tell the server that the window size has changed, and what the new size is.

Server to Client Commands : the four commands that the Rlogin server can send to the client across the TCPconnection. The problem is that only a single TCP connection is used, so the server needs to mark these command bytes so the client knows to interpret them as commands, and not display the bytes on the terminal.

0x02	Flush output. The client discards all the data received from the server, up through the command byte (the last byte of urgent data). The client also discards any pending terminal output that may be buffered. The server sends this command when it receives the interrupt key from the client.
0x10	The client stops performing flow control.
0x20	The client resumes flow control processing.
0x80	The client responds immediately by sending the current window size to the server, and notifies the server in the future if the window size changes. This command is normally sent by the server immediately after the connection is established.

Client to Server Commands

Only one command from the client to the server is currently defined: sending the current window size to the server. Window size changes from the client are not sent to the server unless the client receives the command 0x80 from the server.

TELNET PROTOCOL

- Telnet was designed to work between any host (i.e., any operating system) and any terminal. Its specification defines the lowest common denominator terminal, called the *network virtual terminal* (NVT). The NVT is an imaginary device from which both ends of the connection, the client and server, map their real terminal to and from. That is, the client operating system must map whatever type of terminal the user is on to the NVT. The server must then map the NVT into whatever terminal type the server supports.
- The NVT is a character device with a keyboard and printer. Data typed by the user on the keyboard is sent to the server, and data received from the server is output to the printer. By default the client echoes what the user types to the printer.
- Telnet uses TCP to transmit data and telnet control information. The default port for telnet is TCP port 23. Telnet, however, predates TCP/IP and was originally run over Network Control Program (NCP) protocols.

TELNET COMMANDS

- Telnet uses in-band signaling in both directions. The byte 0xff (255 decimal) is called IAC, for "interpret as command." The next byte is the command byte. To send the data byte 255, two consecutive bytes of 255 are sent.

Name	Code (decimal)	Description
EOF	236	end-of-file
SUSP	237	suspend current process (job control)
ABORT	238	abort process
EOR	239	end of record
SE	240	suboption end
NOP	241	no operation
DM	242	data mark
BRK	243	break
IP	244	interrupt process
AO	245	abort output
AYT	246	are you there?
EC	247	escape character
EL	248	erase line
GA	249	go ahead
SB	250	suboption begin
WILL	251	option negotiation (Figure 26.9)
WONT	252	option negotiation
IX)	253	option negotiation
DONT	254	option negotiation
IAC	255	data byte 255

Option Negotiation

- Although Telnet starts with both sides assuming an NVT, the first exchange that normally takes place across a Telnet connection is option negotiation. The option negotiation is symmetric - either side can send a request to the other. Either side can send one of four different requests for any given option.

1. **WILL**. The sender wants to enable the option itself.

2. **DO**. The sender wants the receiver to enable the option.

3. **WONT**. The sender wants to disable the option itself.

4. **DONT**. The sender wants the receiver to disable the option.

Since the rules of Telnet allow a side to either accept or reject a request to enable an option (cases 1 and 2 above), but require a side to always honor a request to disable an option (cases 3 and 4 above), these four cases lead to the six scenarios shown as follows.

	Sender		Receiver	Description
1.	WILL	-> <-	DO	**sender** wants to **enable** option receiver says OK
2.	WILL	-> <-	DONT	**sender** wants to **enable** option receiver says NO
3.	DO	-> <-	WILL	sender wants **receiver** to **enable** option receiver says OK
4.	DO	-> <-	WONT	sender wants **receiver** to **enable** option receiver says NO
5.	WONT	-> <-	DONT	**sender** wants to **disable** option receiver must say OK
6.	DONT	-> <-	WONT	sender wants **receiver** to **disable** option receiver must say OK

Table: Six scenarios for Telnet option negotiation.

NETSTAT

- It means network statistics. The netstat is a command-line network utility tool to display the information about network connections. It can

1) display the routing table

netstat –r : shows routing table i.e. destination, gateway, genmask, flags, network interface, etc.

2) display interface statistics

netstat –i : displays statistics for the network interfaces currently configured. If the –a option is also given, it prints all interfaces present in the kernel, not only those that have been configured currently.

3) display network connections

netstat –p tcp: displays the information about TCP connections. The netstat provides statistics for the following: proto, local address, foreign address, TCP states.

THE IFCONFIG/IPCONFIG

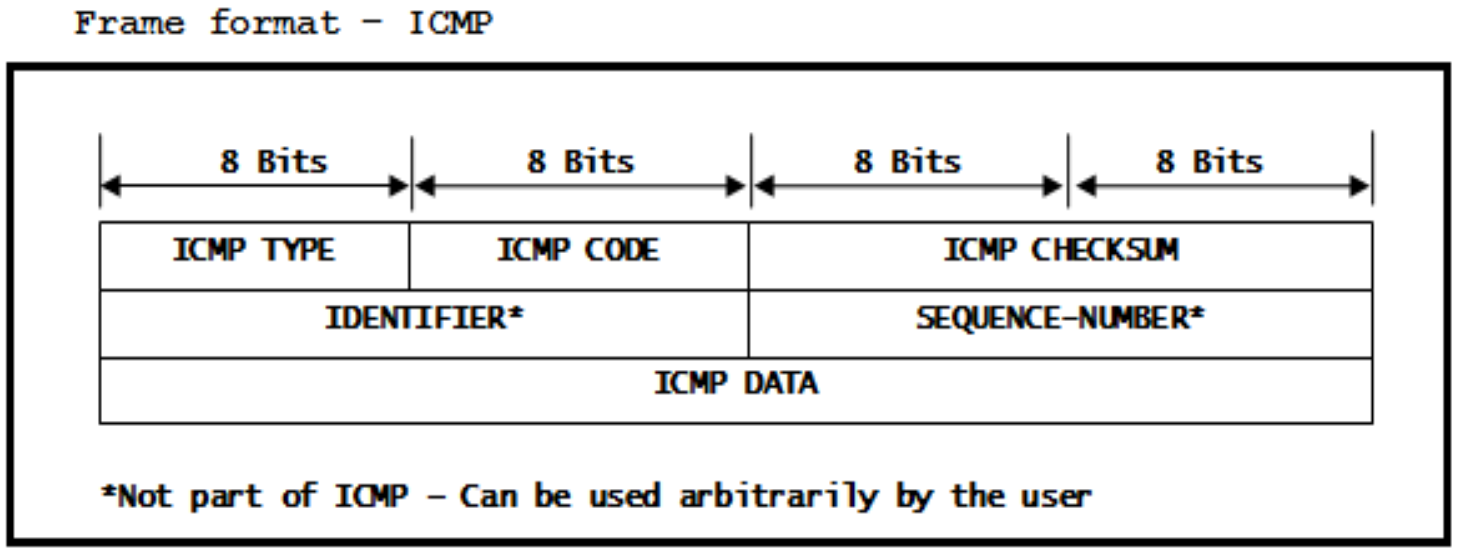
- **ipconfig** – Windows
- **ifconfig** – Unix and Unix-like systems
- The ifconfig stands for “interface configuration”. It is used to assign an address to a network interface and/or configure network interface parameters.

Examples

- **List interfaces (only active)**
  - ifconfig
- **List all interfaces**
  - ifconfig –a
- **Display the configuration of device eth0 only**
  - ifconfig eth0
- **Enable and disable an interface**
  - sudo ifconfig eth1 up
  - sudo ifconfig eth1 down
- **Configure an interface**
  - sudo ifconfig eth0 inet 192.168.0.10 netmask 255.255.255.0

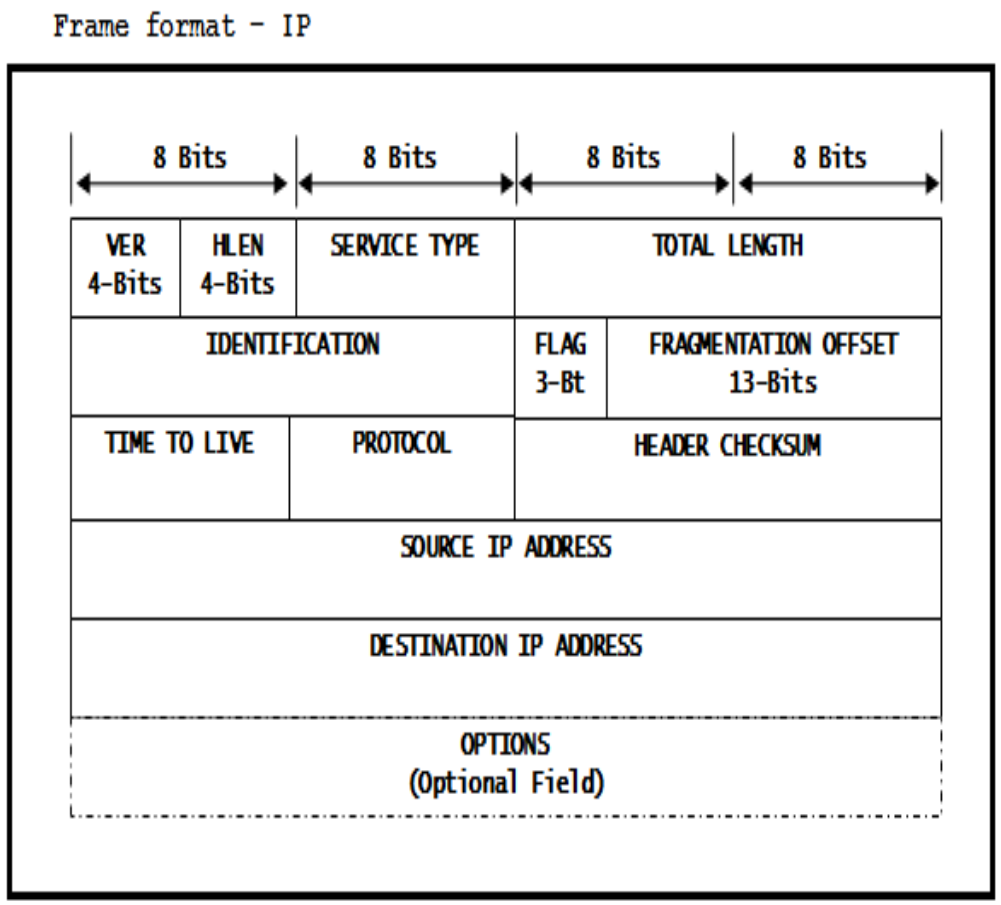
THE PING

- Ping is a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network.
- It measures the round-trip time for messages sent from the originating host to a destination computer and echoed back to the source.
- Ping operates by sending Internet Control Message Protocol (ICMP) Echo Request packets to the target host and waiting for an ICMP Echo Reply.
- The results of the test usually include a statistical summary of the results, including the minimum, maximum, the mean, round-trip time, and usually standard deviation of the mean.
- Example: ping –c 4 www.google.com // c = packet counts



1. **ICMP TYPE** shall be set to **0x08** since this is an 'Echo-Request' message
2. **ICMP CODE** shall always be **0x00** for PING message
3. **ICMP CHECKSUM** is for header and data and is **'0xA5, 0x51'** for our message
4. **ICMP DATA** is "PING data to be sent" defined above

- Before sending the ICMP encapsulated PING command to MAC layer, the messages should be encapsulated into IP datagrams. Below is the frame-format of IPv4 packet:



1. **VER** shall be set to **4** since it's a IPv4 packet
2. **HLEN** is header length in 'lwords' - It shall be set to 5 here since the header length is 20Bytes
3. **SERVICE TYPE** shall be set to **0x00** since ICMP is a normal service
4. **TOTAL LENGTH** shall be set as **'0x00 0x54'** bytes, which includes header and data length
5. **IDENTIFICATION** is the unique identity for all datagrams sent from this source IP – Let's set it as **'0x96, 0xA1'** for our packet
6. **FLAG** and **FRAGMENTATION OFFSET** is set to **0x00, 0x00** since we don't intent to fragment the packet. **Reason**: The packet that we are sending here is smaller than a WLAN frame size
7. **TIME TO LIVE** shall be set to **0x40** since we want to discard the packet after 64 hops
8. **PROTOCOL** is set to **0x01** since it's an ICMP packet
9. Next 2Bytes is **HEADER CHECKSUM** which shall be set to **0x57, 0xFA** in our case
10. **SOURCE IP ADDRESS** is set to **0xc0, 0xa8, 0x01, 0x64** (which is 192.168.1.100)
11. **DESTINATION IP ADDRESS** is set to **0xc0, 0xa8, 0x01, 0x65** (which is 192.168.1.101)
12. **OPTIONS** field is left blank

TFTP

- TFTP stands for Trivial File Transfer Protocol. It is a simple, lockstep protocol for transferring files (FTP), implemented on the top of the UDP/IP protocols using well-known port number 69.
- TFTP was designed to be small and easy to implement and therefore it lacks most of the advance features offered by more robust file transfer protocols.
- TFTP only reads/writes files from/to a remote server. It cannot list, delete, or rename files or directories and it has no provisions for user authentication.
- It uses lockstep acknowledgement. The file is sent in fixed length blocks of 512 bytes by default or the number specified in the block size negotiation.

Steps

- Send RRQ (read request) or WRQ (write request) to port 69 of the server
- Send ACK packet to WRQ or send DATA packet to RRQ from randomly allocated ephemeral port.
- Send ACK to DATA packet or start sending DATA packet.
- DATA packet of size less than 512 bytes indicates end of file transfer. If the file size is multiple of 512, DATA packet of 0 byte (UDP packet with no data) indicates end of file transfer.
- Retransmit in absence of ACK packet.

No login/Access control mechanism is present.

END OF CHAPTER 4
(END OF SYLLABUS OF NETWORK PROGRAMMING)