RV College of Engineering®

Go, change the world

NETWORK PROGRAMMING AND SECURITY (Theory & Lab)

18CS54

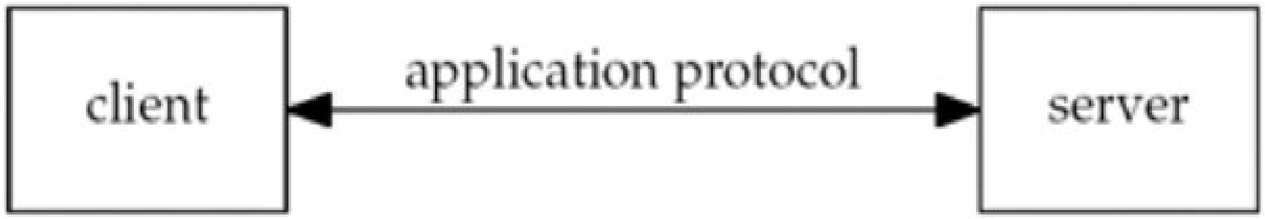
UNIT 1
The Transport Layer and introduction to sockets

Contents

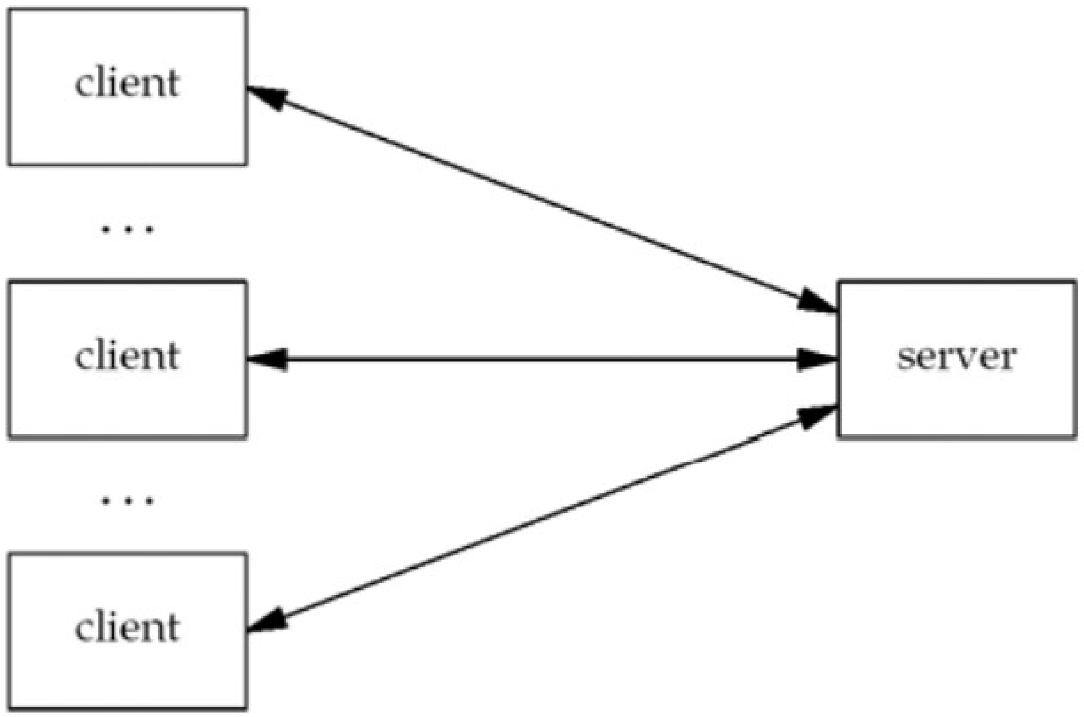
Introduction to all protocols

- **Protocol:** an agreement on how the programs will communicate
- decisions must be made about which program is expected to initiate communication and when responses are expected. Eg. Web Server and web client communication
- Most network-aware applications use client and server communications

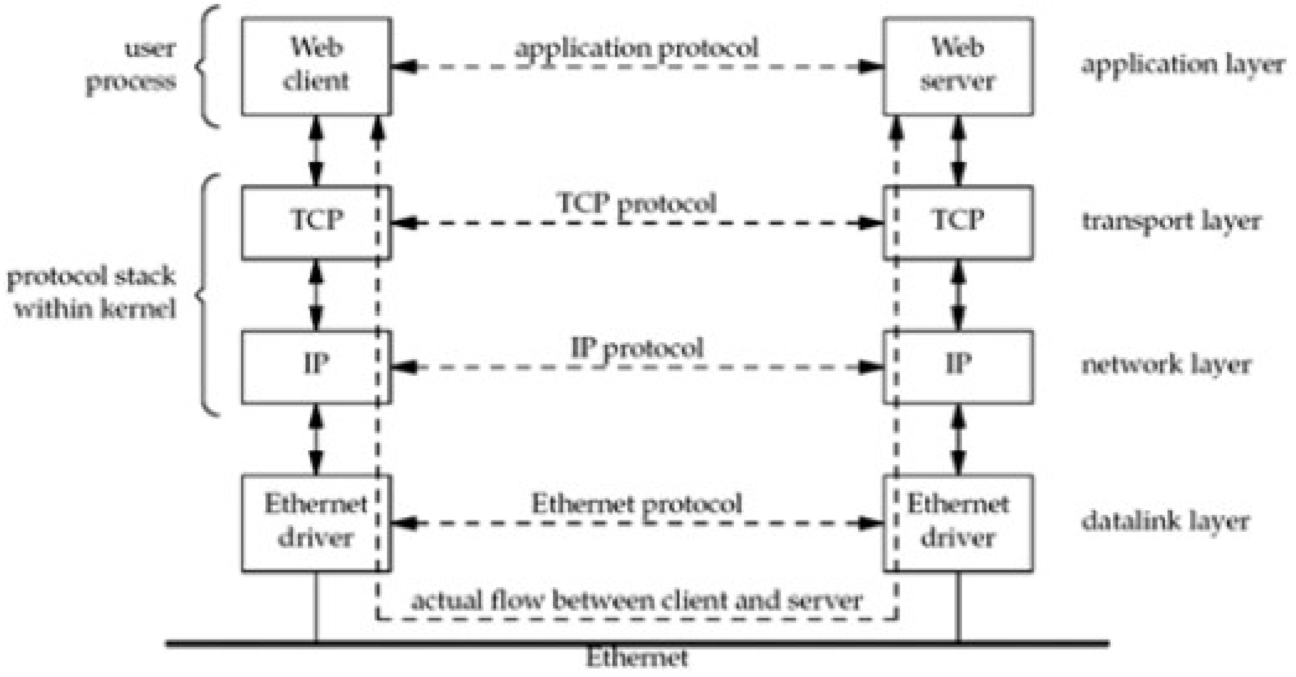
Network application: client and server.



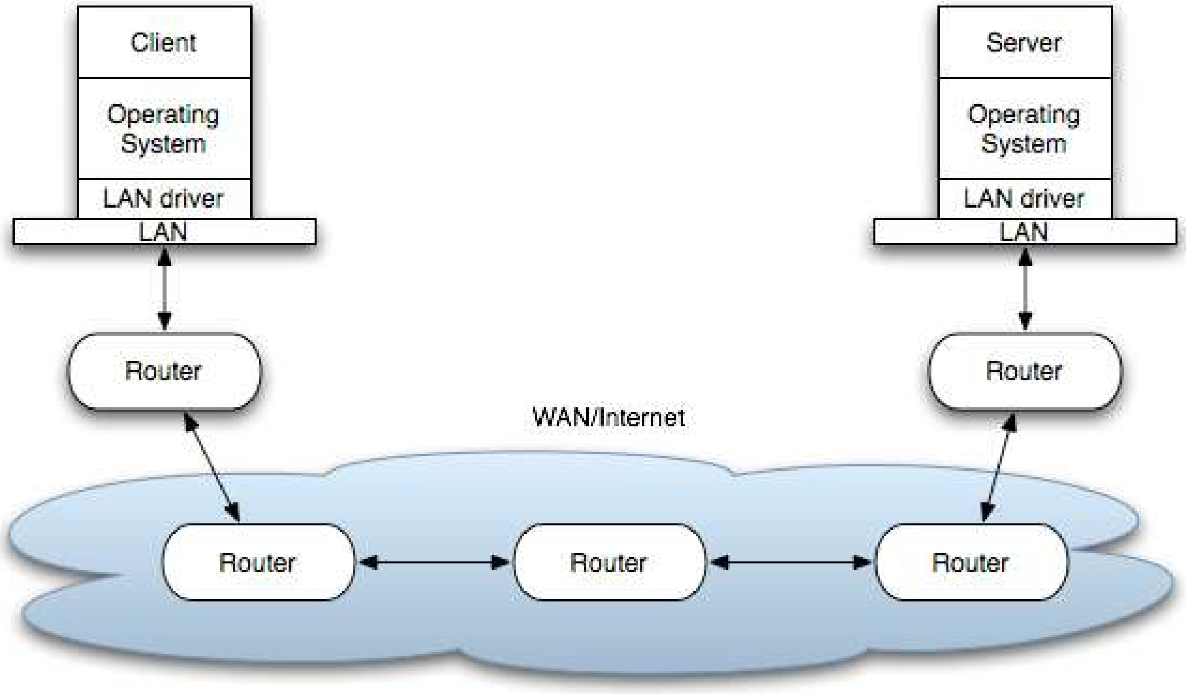
•Client communicates with only one server where as one server communicates with many clients at the same tie as shown in the figure below:



- The client application and the server application may be thought of as communicating **via a multiple layers of protocols**
- Note that the client and server are typically user processes, while the TCP and IP protocols are normally part of the protocol stack within the kernel
- If both are in the same Ethernet then one can think of the communication in the followinꞡ



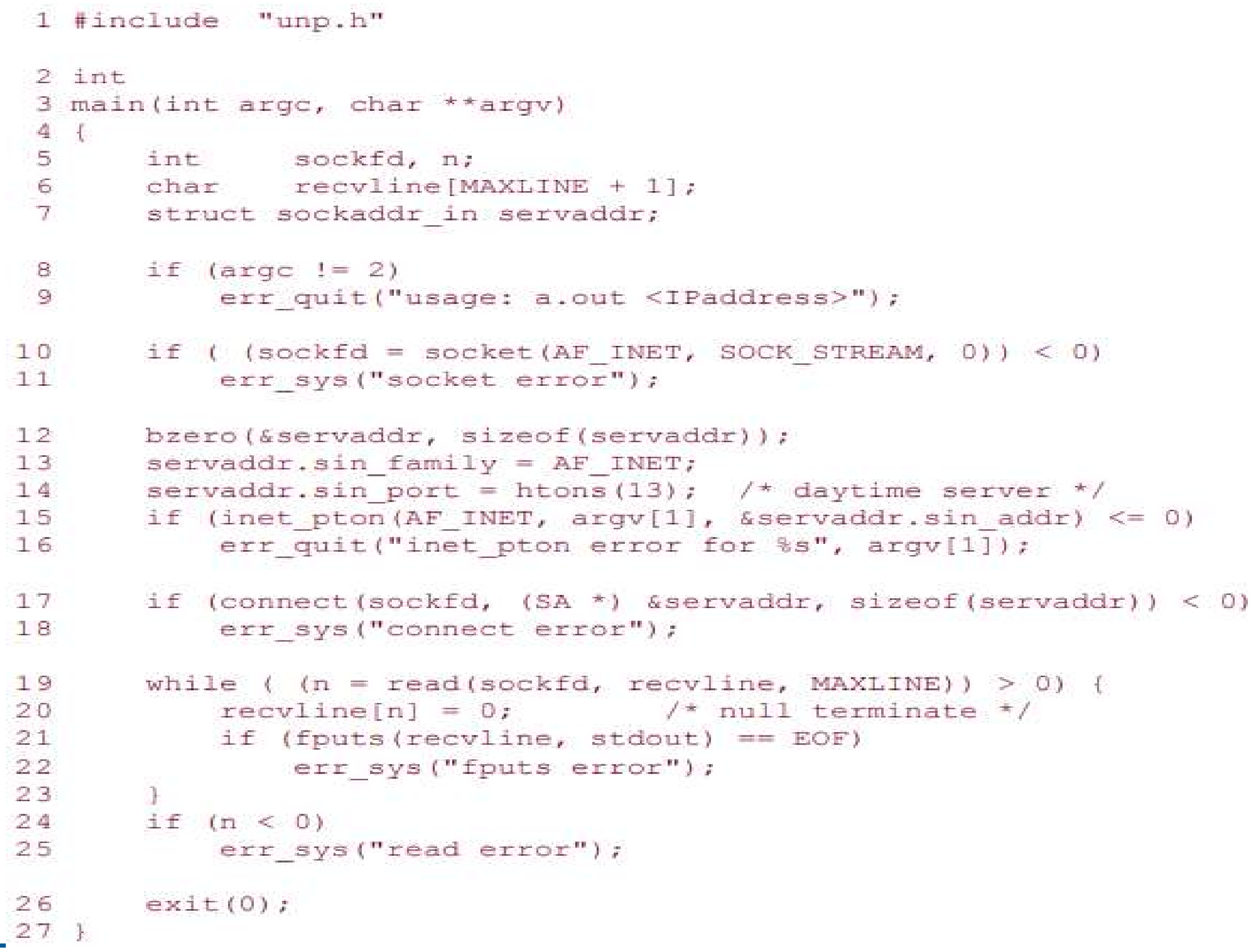
What if client and server are not on the same LAN???



Few Questions??

- When do you use a switch and when do you use a router?
- The largest WAN is ??
- What is a socket and why it is required?
- What is a port? Define its size and usage.

A Simple Daytime Client

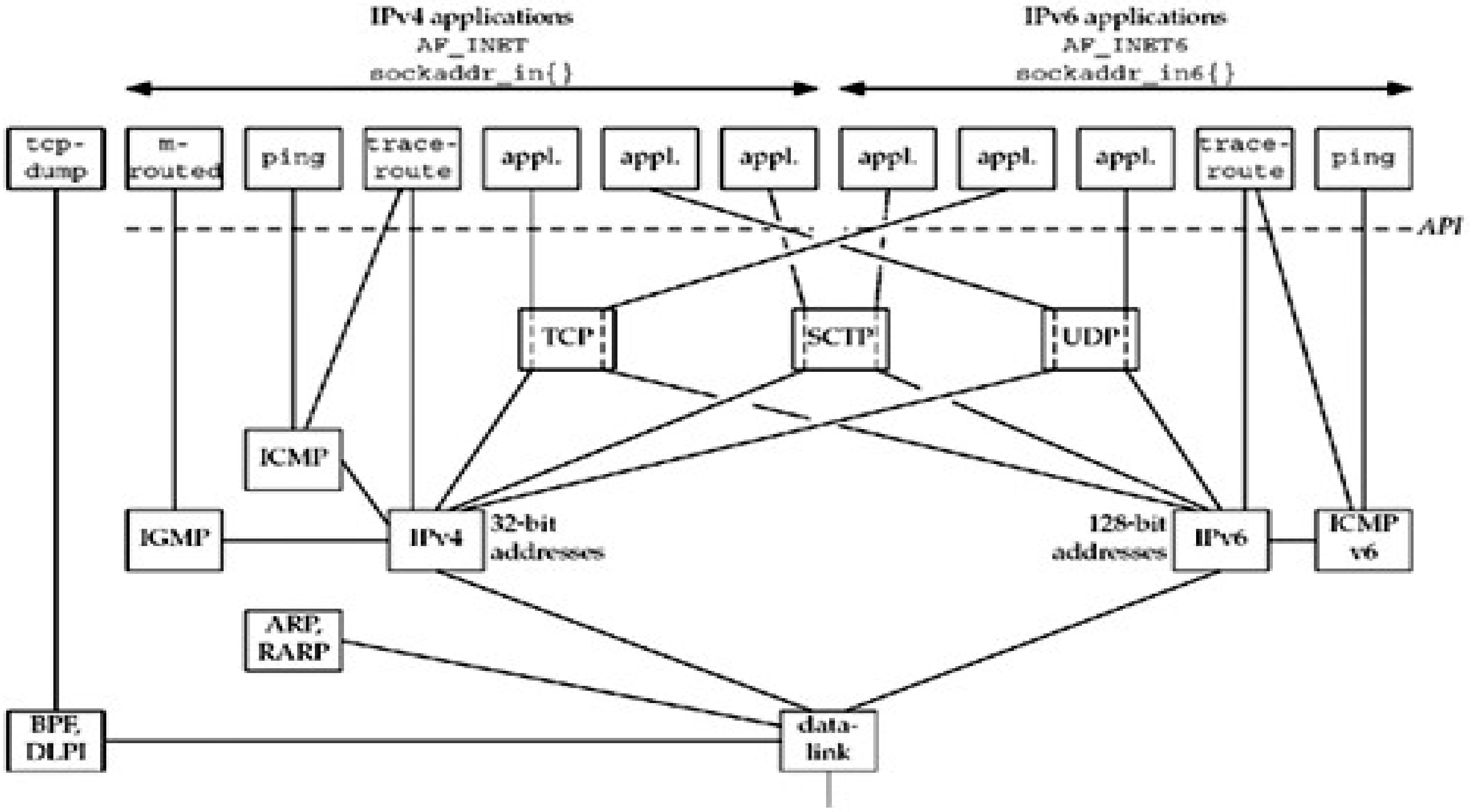


```
#include  "unp.h"

int
main(int argc, char **argv)
{
    int     sockfd, n;
    char    recvline[MAXLINE + 1];
    struct sockaddr_in servaddr;

    if (argc != 2)
        err_quit("usage: a.out <IPaddress>");

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        err_sys("socket error");

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(13);  /* daytime server */
    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
        err_quit("inet_pton error for %s", argv[1]);

    if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
        err_sys("connect error");

    while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
        recvline[n] = 0;        /* null terminate */
        if (fputs(recvline, stdout) == EOF)
            err_sys("fputs error");
    }
    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Create the socket

Use the htons (host to network short) to convert binary port number pton(presentation to numeric) is used to convert from ASCII ipaddress to proper format

The Big Picture



The protocols:

1. IPv4
2. IPv6
3. TCP
4. SCTP
5. UDP
6. ICMP
7. IGMP: Internet Group Management Protocol
8. ARP : IP to Mac address
9. RARP: Mac to IP address
10. BPF (BSD Packet Filter)
11. DLPI (Data Link Provider Interface)

Features of User Datagram Protocol (UDP):

1. UDP is a simple transport-layer protocol, described in RFC 768
2. The application writes a message to a UDP socket, which is then encapsulated in a UDP datagram, which is then further encapsulated as an IP datagram, which is then sent to its destination.
3. There is no guarantee that a UDP datagram will ever reach its final destination, that order will be preserved across the network, or that datagrams arrive only once.
4. Lack of reliability, no automatic retransmission incase of errors
5. The length of a datagram is passed to the receiving application along with the data.
6. UDP provides a connectionless service

Examples include Voice over IP (VoIP), online games, and media streaming.

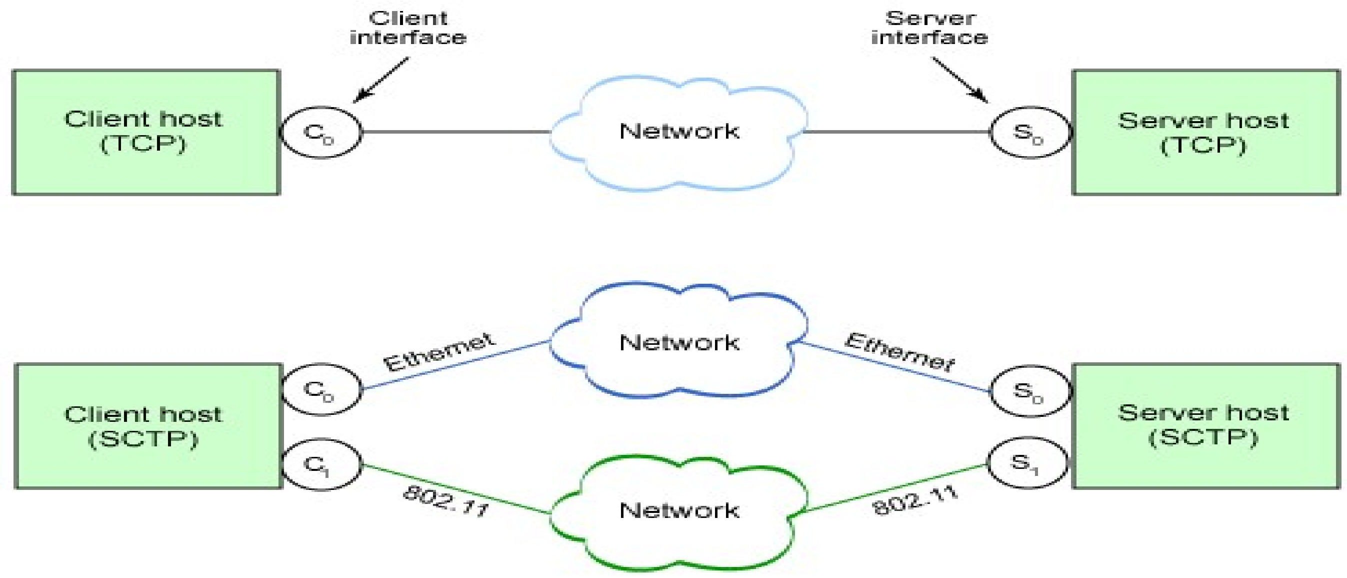
Features of Transmission Control Protocol (TCP):

1. TCP provides connections between clients and servers, described in RFC 793
2. TCP also provides reliability. When TCP sends data to the other end, it requires an acknowledgment in return.
3. Automatic retransmission incase of errors
4. Therefore, TCP cannot be described as a 100% reliable protocol; it provides reliable delivery of data or reliable notification of failure.
5. TCP contains algorithms to estimate the round-trip time (RTT) between a client and server dynamically so that it knows how long to wait for an acknowledgment.
6. TCP also sequences the data by associating a sequence number with every byte that it sends.
7. This helps in reordering, finding duplicates at the receivers end
8. TCP provides flow control
9. TCP connection is full-duplex

Examples WWW, FTP,Email etc.

Features of Stream Control Transmission Protocol (SCTP):

1. SCTP provides services similar to those offered by UDP and TCP, described in RFC 2960
2. SCTP provides “associations” between clients and servers and is SCTP is message-oriented
3. An association refers to a communication between two systems, which may involve more than two addresses due to multihoming.



Difference between TCP and UDP

1. Reliability
2. Data delivery in order
3. Connection between client and server
4. Flow control
5. Type of connection: Full duplex

TCP Connection Establishment

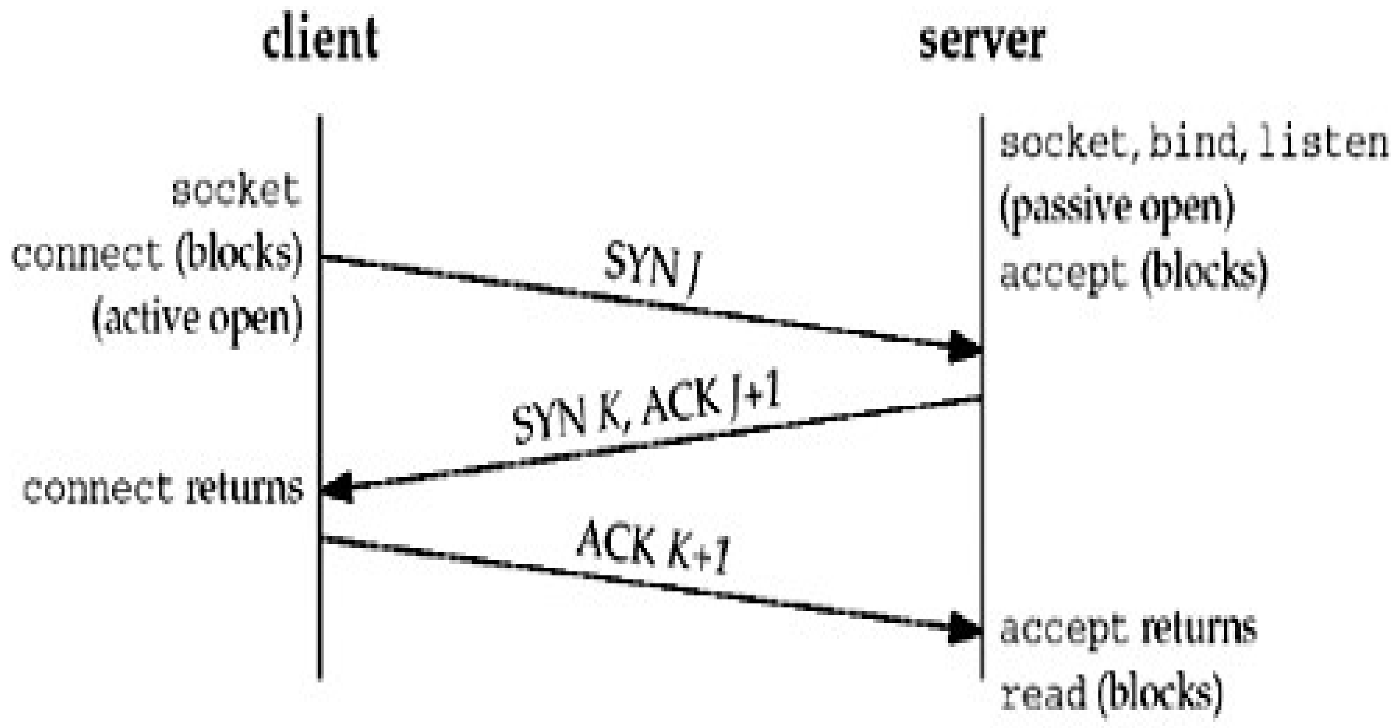
Three-Way Handshake:

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 (which we will talk about shortly).
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 in a single segment.
4. The client must acknowledge the server's SYN.

TCP Connection Establishment and Termination

Three-Way Handshake:



TCP Connection Termination

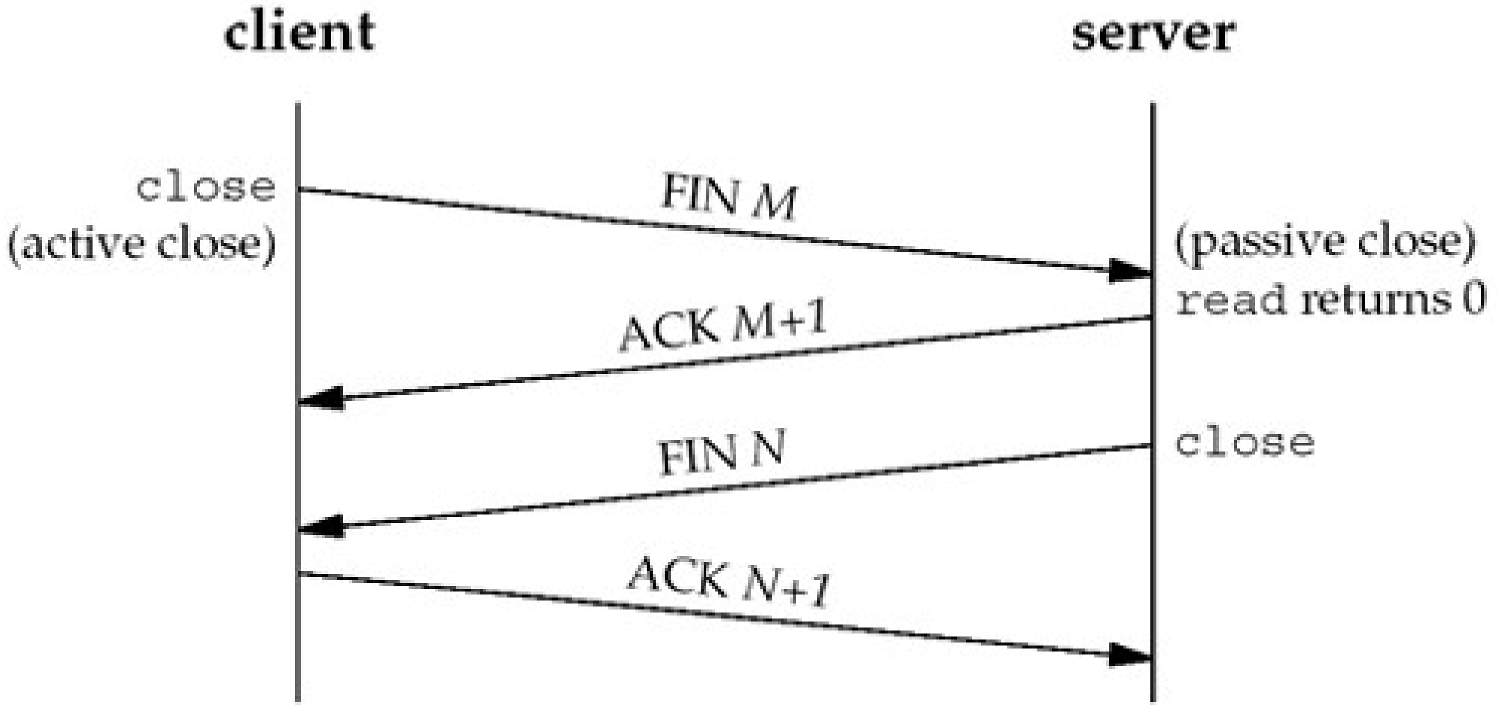
Four-Way Handshake:

The following scenario occurs when a TCP connection is established:

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 (after any data that may have already been queued for the application to receive), since the receipt of the FIN means the application will not receive any Page 68 ABC Amber CHM Converter Trial version, http://www.processtext.com/abcchm.html 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 Establishment and Termination

Four-Way Handshake:



TCP State Transition Diagram:

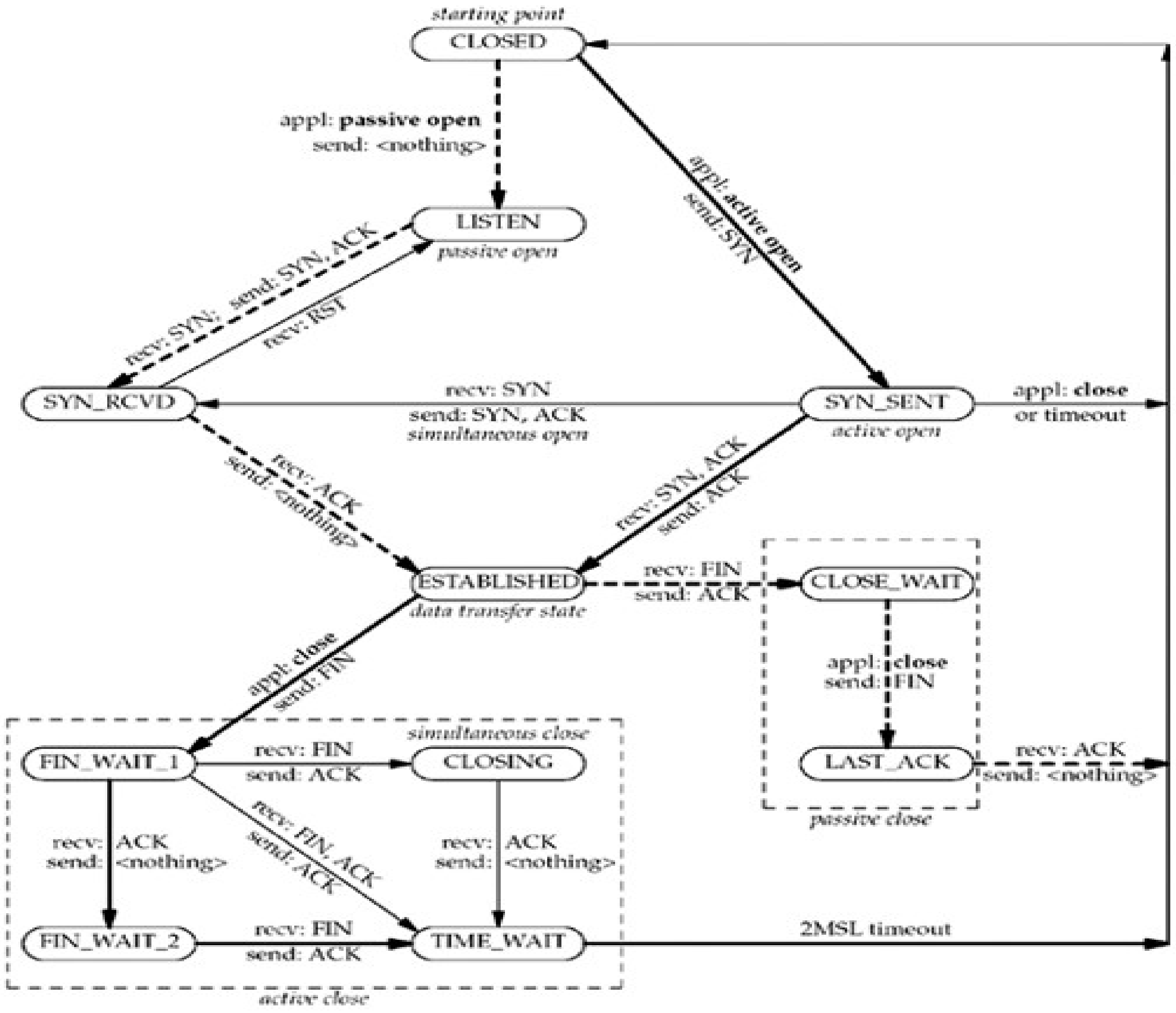
→ Indicate Normal Transition for client

- - → Indicate Normal Transition for Server

appl: Indicate State Transition taken when application issues operation

recv: Indicate State Transition when segment is received

send: Indicate what is sent for this transition



TCP State:

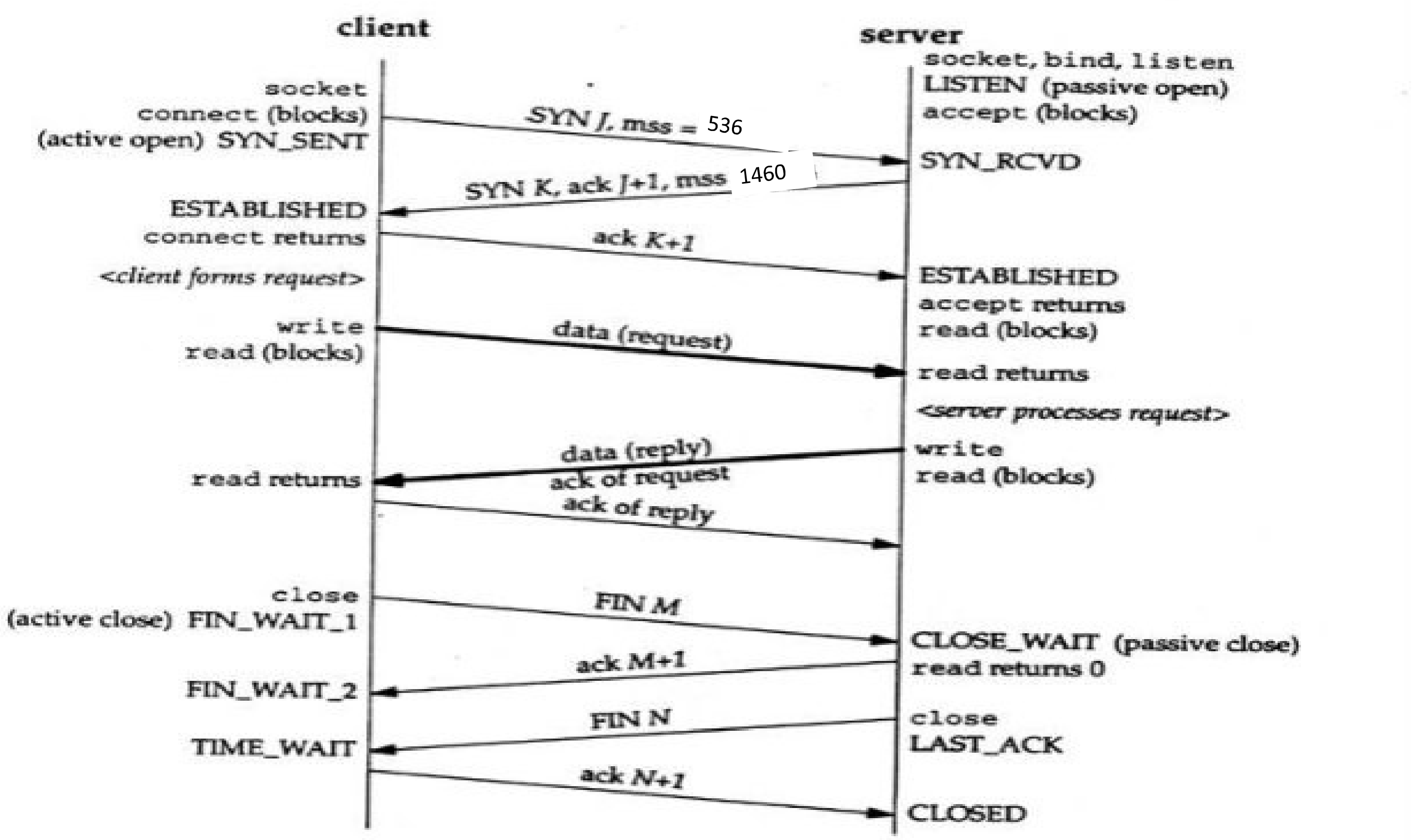
- There are 11 different states defined for a connection and the rules of TCP dictate the transitions from one state to another, based on the current state and the segment received in that state.

- if an application performs an active open in the CLOSED state, TCP sends a SYN and the new state is SYN_SENT. If TCP next receives a SYN with an ACK, it sends an ACK and the new state is ESTABLISHED. This final state is where most data transfer occurs.

- *netstat*, is a useful tool for debugging client/server applications

Watching the Packets :

The actual packet exchange that takes place for a complete TCP connection: the conn



Watching the Packets :

Note:

If the entire purpose of this connection was to send a one-segment request and receive a one-segment reply, there would be eight segments of overhead involved when using TCP. If UDP was used instead, only two packets would be exchanged: the request and the reply.

TIME_WAIT State:

- Undoubtedly, one of the most misunderstood aspects of TCP with regard to network programming is its TIME_WAIT state.

- The end that performs the active close goes through this state. The duration that this endpoint remains in this state is twice the maximum segment lifetime (MSL), sometimes called 2MSL.(between1 and 4min according to the RFC)
- The MSL is the maximum amount of time that any given IP datagram can live in a network.
- This time is bounded because every datagram contains an 8-bit hop limit with a maximum value of 255.
- a packet with the maximum hop limit of 255 cannot exist in a network for more than MSL seconds.

TIME_WAIT State:

There are two reasons for the TIME_WAIT state:

1. To implement TCP's full-duplex connection termination reliably
2. To allow old duplicate segments to expire in the network

Scenario for explaining why TIME_WAIT State is very important:

- Assume we have a TCP connection between 12.106.32.254 port 1500 and 206.168.112.219 port 21.
- This connection is closed and then sometime later, we establish another connection between the same IP addresses and ports: 12.106.32.254 port 1500 and 206.168.112.219 port 21.
- This latter connection is called an incarnation of the previous connection since the IP addresses and ports are the same.
- TCP must prevent old duplicates from a connection from reappearing at some later time and being misinterpreted as belonging to a new **incarnation** of the same connection. To do this, TCP will not initiate a new incarnation of a connection that is currently in the TIME_WAIT state.
- Since the duration of the TIME_WAIT state is twice the MSL, this allows MSL seconds for a packet in one direction to be lost, and another MSL seconds for the reply to be lost.
- By enforcing this rule, we are guaranteed that when we successfully establish a TCP connection, all old duplicates from previous incarnations of the connection have expired in the network.

Port Numbers:

- All three transport layers use 16-bit integer port numbers to differentiate between UDP,TCP and SCTP processes.
- When a client wants to contact a server, the client must identify the server with which it wants to communicate. TCP, UDP, and SCTP define a group of well-known ports to identify well-known services.
  - For example, every TCP/IP implementation that supports FTP assigns the well-known port of 21 (decimal) to the FTP server. Trivial File Transfer Protocol (TFTP) servers are assigned the UDP port of 69.
- Clients, on the other hand, normally use ephemeral ports, that is, short-lived ports. These port numbers are normally assigned automatically by the transport protocol to the client.

Port Numbers:

The port numbers are divided into three ranges:

1. **The well-known ports:** 0 through 1023. These port numbers are controlled and assigned by the IANA. When possible, the same port is assigned to a given service for TCP, UDP, and SCTP. For example, port 80 is assigned for a Web server, for both TCP and UDP, even though all implementations currently use only TCP.

2. **The registered ports:** 1024 through 49151. These are not controlled by the IANA, but the IANA registers and lists the uses of these ports as a convenience to the community. When possible, the same port is assigned to a given service for both TCP and UDP. For example, ports 6000 through 6063 are assigned for an X Window server for both protocols, even though all implementations currently use only TCP.

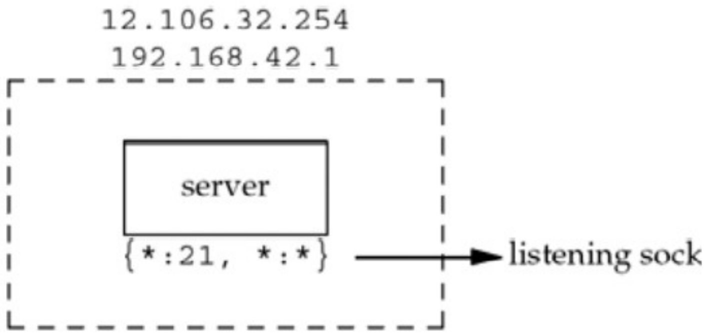
3. **The dynamic or private ports:** 49152 through 65535. The IANA says nothing about these ports. These are what we call ephemeral ports.

TCP Port Numbers and Concurrent Servers

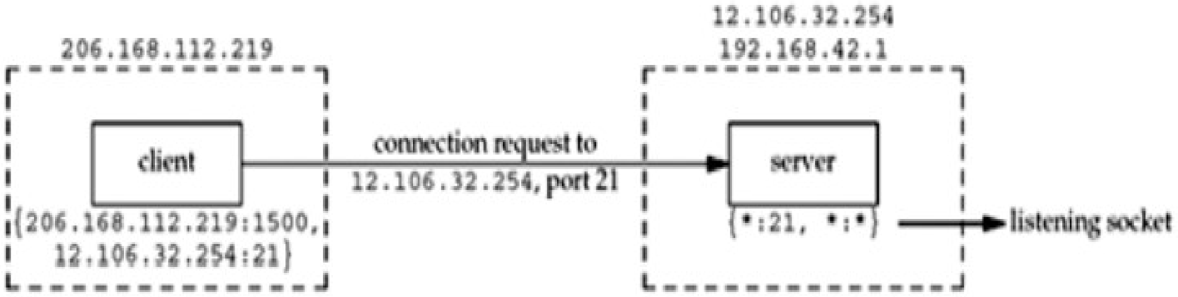
With a concurrent server, where the main server loop spawns a child to handle each new connection, what happens if the child continues to use the well-known port number while servicing a long request?

Let's examine a typical sequence. First, the server is started on the host freebsd, which is multihomed with IP addresses 12.106.32.254 and 192.168.42.1, and the server does a passive open using its well-known port number (21, for this example). It is now waiting for a client request,

1. TCP Server with a passive open on port 21

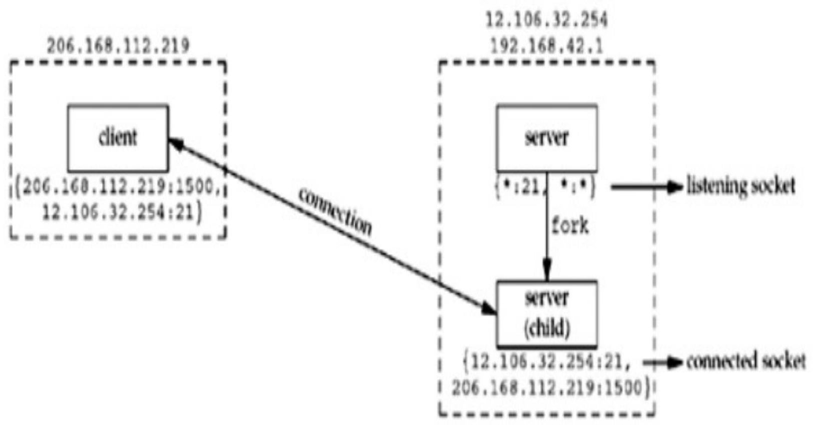


2. Connection request from client to server

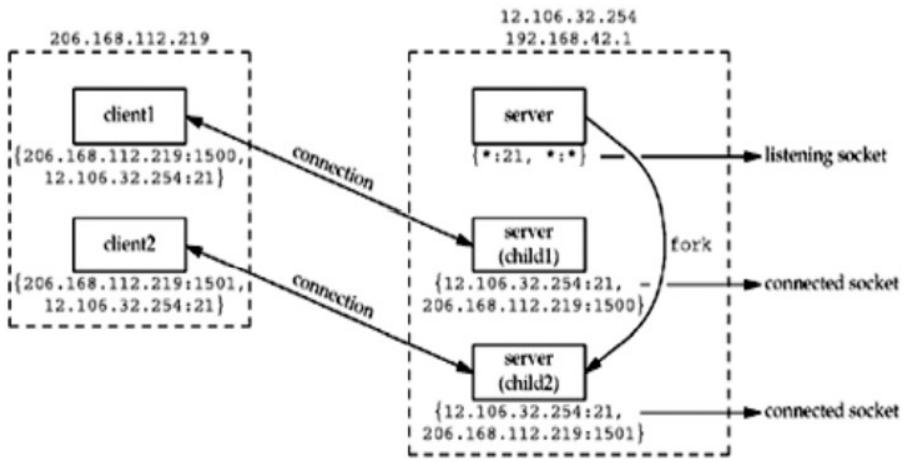


TCP Port Numbers and Concurrent Servers

3. Concurrent server has child handle client



4. Second client connection with same server



Buffer Sizes and Limitations:

Certain limits affect the size of IP datagrams:

1. The maximum size of an IPv4 datagram is 65,535 bytes, including the IPv4 header. This is because of the 16-bit total length field where for an IPv6 datagram it is 65,575 bytes, including the 40-byte IPv6 header
2. Many networks have an MTU which can be dictated by the hardware. For example, the Ethernet MTU is 1,500 bytes.
3. The minimum link MTU for IPv4 is 68 bytes.
4. The smallest MTU in the path between two hosts is called the path MTU
5. When an IP datagram is to be sent out an interface, if the size of the datagram exceeds the link MTU, fragmentation is performed by both IPv4 and IPv6. The fragments are not normally reassembled until they reach the final destination.
6. If the "don't fragment" (DF) bit is set in the IPv4 header (Figure A.1), it specifies that this datagram must not be fragmented, either by the sending host or by any router.
7. IPv4 and IPv6 define a minimum reassembly buffer size, the minimum datagram size that we are guaranteed any implementation must support.
8. TCP has a maximum segment size (MSS) that announces to the peer TCP the maximum amount of TCP data that the peer can send per segment.
9. SCTP keeps a fragmentation point based on the smallest path MTU found to all the peer's addresses.

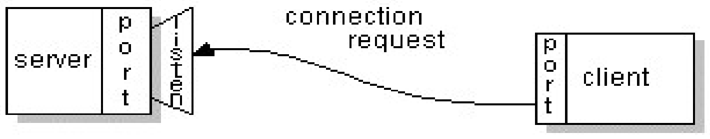
Socket Address Structure

Definition:

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

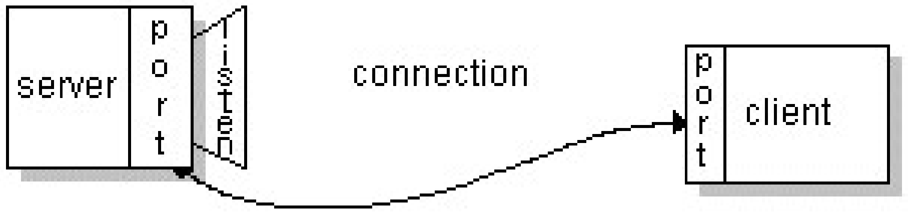
Client Side

1. The client knows the hostname of the machine on which the server is running and the port number on which the server is listening.
2. To make a connection request, the client tries to rendezvous with the server on the server's machine and port.
3. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



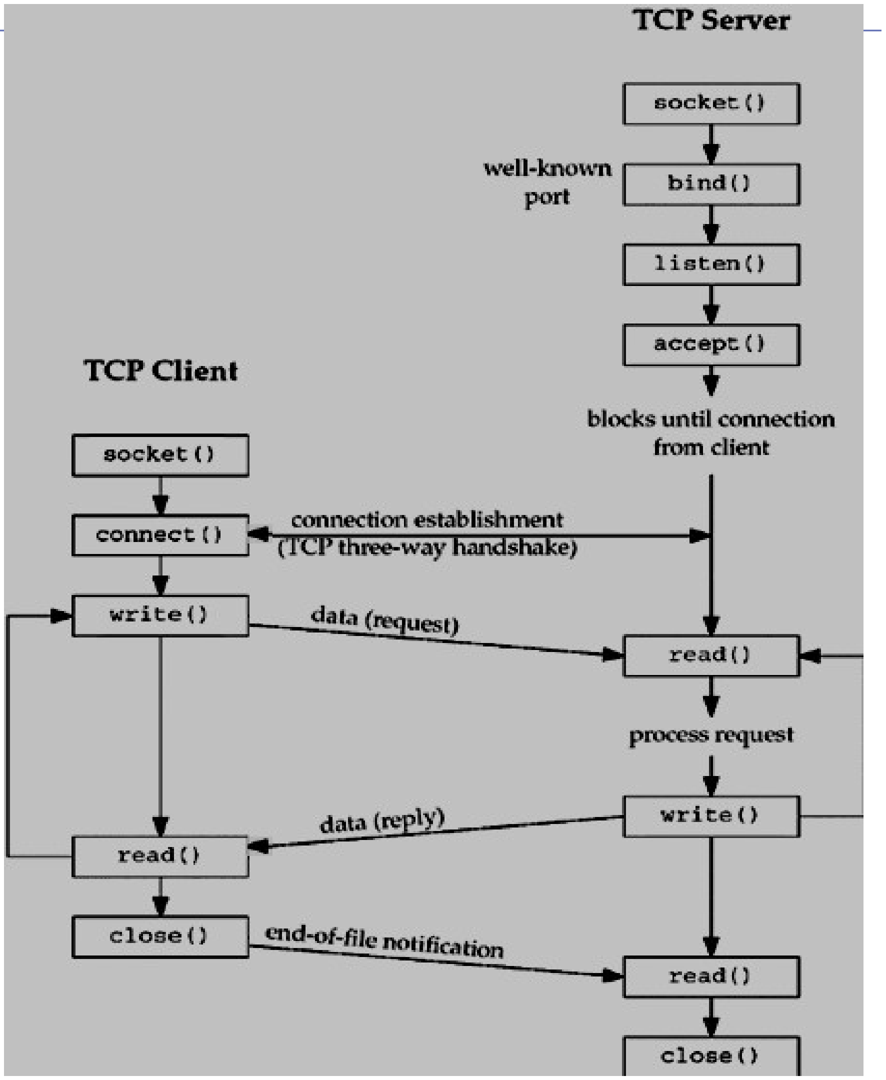
Server Side

1. The server has the name and port number defined.
2. It listens to the request and accepts connection from the client.
3. On acceptance, the server creates new socket assigns the client to it.
4. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



Socket Address Structure

Socket Functions- There are different set of functions at client and server side.



Socket Address Structure

1. Most socket functions require a pointer to a socket address structure as an argument.
2. Each supported protocol suite defines its own socket address structure.
3. The names of these structures begin with sockaddr_ and end with a unique suffix for each protocol suite.

IPv4 Socket Address Structure

An IPv4 socket address structure, commonly called an "Internet socket address structure," is named sockaddr_in and is defined by including the header.

```
struct in_addr {
 in_addr_t s_addr;      /* 32-bit IPv4 address */
                        /* network byte ordered */
};
Struct  sockaddr_in {
uint8_t  sin_len;               /* length of structure (16) */
sa_family_t  sin_family;   /* AF_INET */
 in_port_t     sin_port;          /* 16-bit TCP or UDP port number
          network byte ordered */
struct in_addr     sin_addr;   /* 32-bit IPv4 address  network
     byte ordered */
 char sin_zero[8];                /* unused */
};
```

Generic Socket Address Structure

- A socket address structures is always passed by reference when passed as an argument to any socket functions.
- But any socket function that takes one of these pointers as an argument must deal with socket address structures from any of the supported protocol families.

```
struct sockaddr {
uint8_t  sa_len;
sa_family_t sa_family;      /* address family: AF_xxx value */
char sa_data[14];           /* protocol-specific address */

};
```

- The socket functions are then defined as taking a pointer to the generic socket address structure.
- C function prototype for the bind function:

```
int bind(int, struct sockaddr *, socklen_t);
```

- This requires that any calls to these functions must cast the pointer to the protocol-specific socket address structure to be a pointer to a generic socket address structure. For example,

```
struct sockaddr_in serv; /* IPv4 socket address structure */  /* fill in serv{} */
bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

Value- Result Arguments

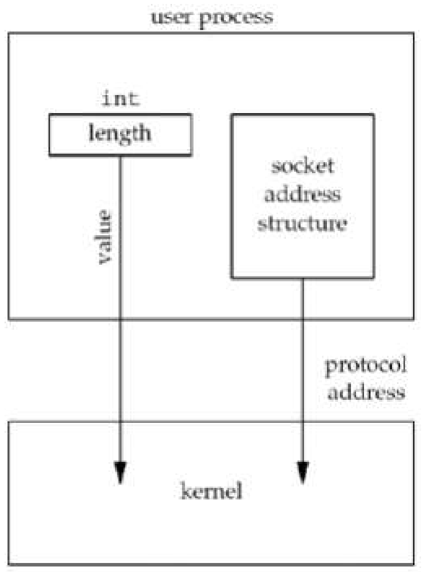
- When a socket address structure is passed to any socket function, it is always passed by reference- A pointer to the structure is passed.
- The length of the structure is also passed as an argument.
- But the way in which the length is passed depends on which direction the structure is being passed: from the process to the kernel, or vice versa.

Value- Result Arguments

- Three functions, **bind, connect, and sendto,** pass a socket address structure from the process to the kernel.
- One argument to these three functions is the pointer to the socket address structure and another argument is the integer size of the structure, as in

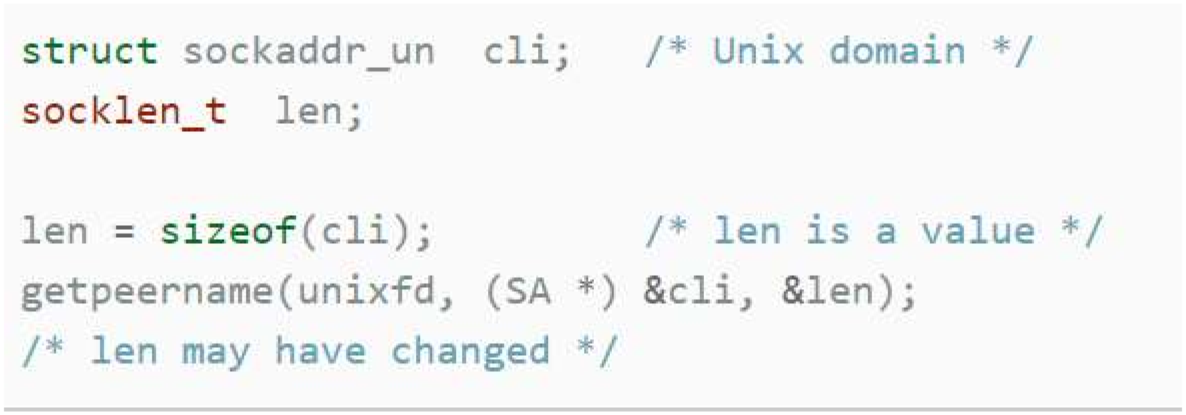
```
struct sockaddr_in serv;
/* fill in serv{} */
 connect (sockfd, (SA *) &serv, sizeof(serv));
```

Figure 3.7. Socket address structure passed from process to kernel.



Since the kernel is passed both the pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernel.

- Four functions, **accept, recvfrom, getsockname, and getpeername**, pass a socket address structure from the kernel to the process, the reverse direction from the previous scenario.
- Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to an integer containing the size of the structure, as in

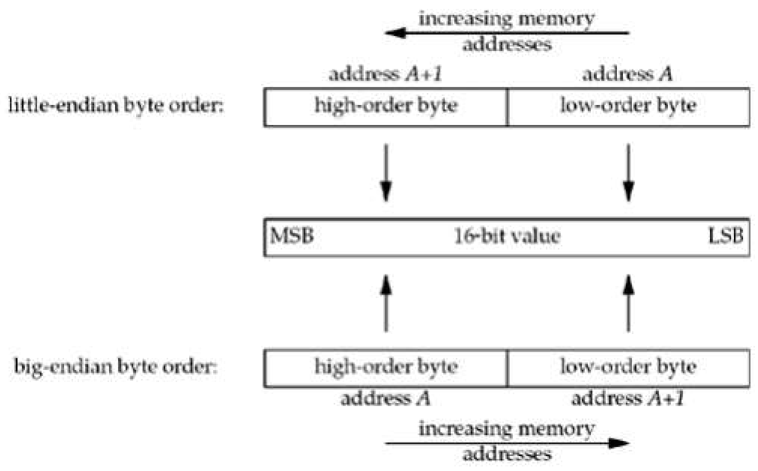


```
struct sockaddr_un  cli;   /* Unix domain */
socklen_t  len;

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

Byte Ordering Functions

- Consider a 16-bit integer that is made up of 2 bytes.
- There are two ways to store the two bytes in memory: with the low-order byte at the starting address, known as little-endian byteorder, or with the high-order byte at the starting address, known as big-endian byte order.

Figure 3.9. Little-endian byte order and big-endian byte order for a 16-bit integer.

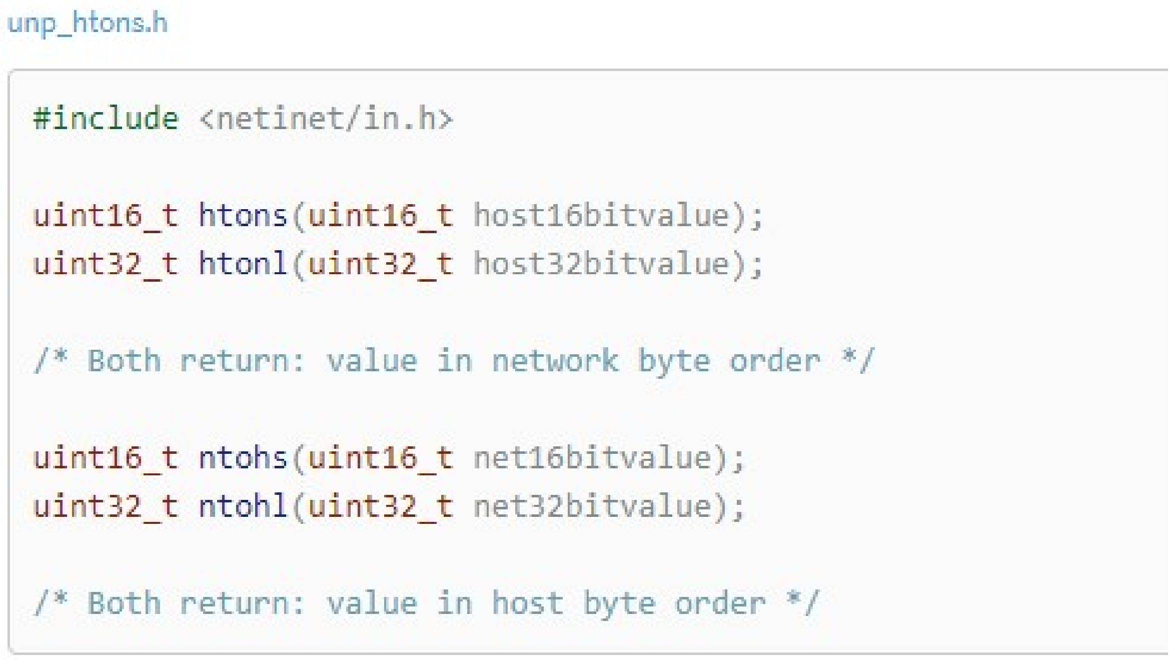


Byte Ordering Functions

- The terms "little-endian" and "big-endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.
- Host byte order refer to the byte ordering used by a given system.
- Networking protocols must specify a network byte order.
- The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multibyte fields will be transmitted.
- The Internet protocols use big-endian byte ordering for these multibyte integers.

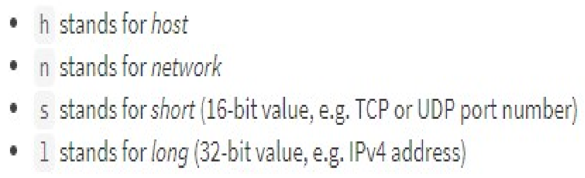
Conversion Functions -1

We use the following four functions to convert between these two byte orders:



unp_htons.h

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

uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);

/* Both return: value in network byte order */

uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);

/* Both return: value in host byte order */
```

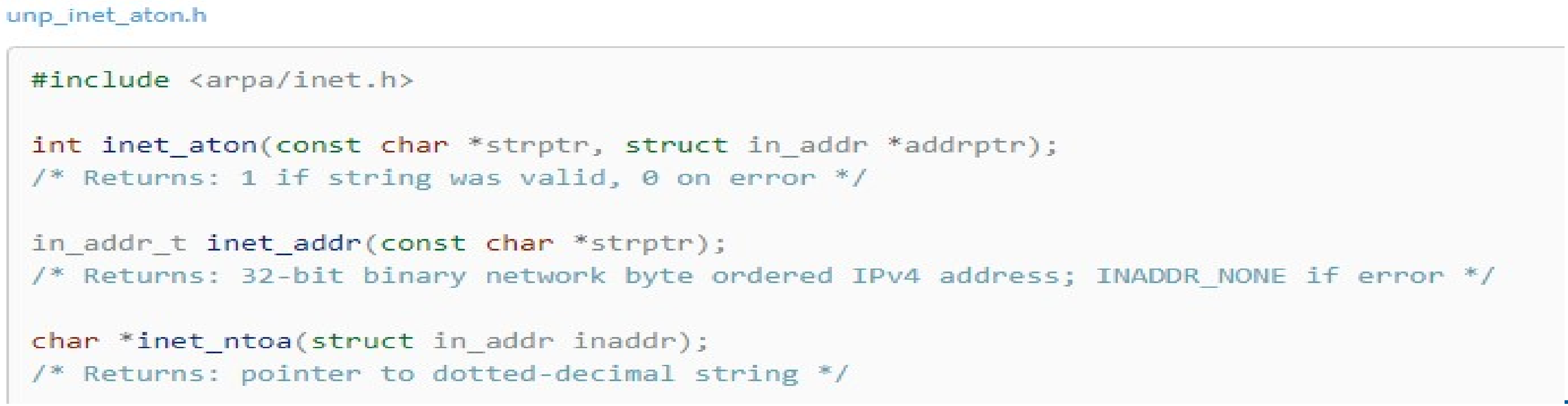


- `h` stands for *host*
- `n` stands for *network*
- `s` stands for *short* (16-bit value, e.g. TCP or UDP port number)
- `l` stands for *long* (32-bit value, e.g. IPv4 address)

Conversion Functions -2

inet_aton, inet_addr, and inet_ntoa Functions

- These functions convert Internet addresses between ASCII strings (what humans prefer to use) and network byte ordered binary values (values that are stored in socket address structures).
- inet_addr does the same conversion, returning the 32-bit binary network byte ordered value as the return value. The problem with this function is that all 232 possible binary values are valid IP addresses (0.0.0.0 through 255.255.255.255), but the function returns the constant INADDR_NONE (typically 32 one-bits) on an error.
- Today, inet_addr is deprecated and any new code should use inet_aton instead.



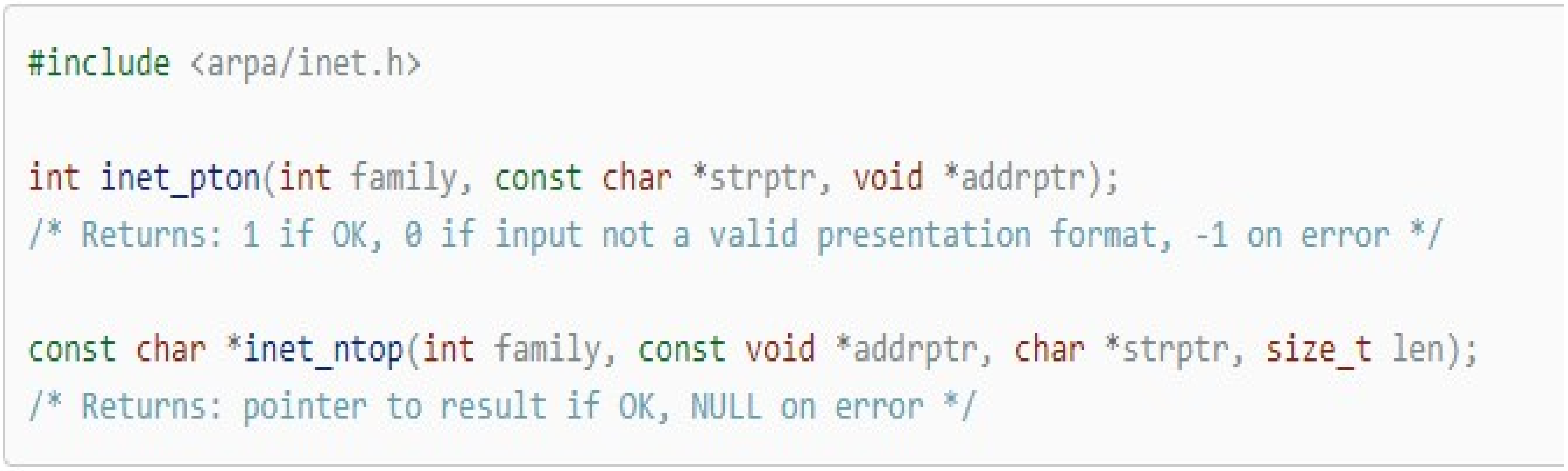
unp_inet_aton.h

```
#include <arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr);
/* Returns: 1 if string was valid, 0 on error */

in_addr_t inet_addr(const char *strptr);
/* Returns: 32-bit binary network byte ordered IPv4 address; INADDR_NONE if error */

char *inet_ntoa(struct in_addr inaddr);
/* Returns: pointer to dotted-decimal string */
```

Conversion Functions -3

inet_pton and inet_ntop Functions

- These two functions are new with IPv6 and work with both IPv4 and IPv6 addresses. The letters "p" and "n" stand for presentation and numeric.

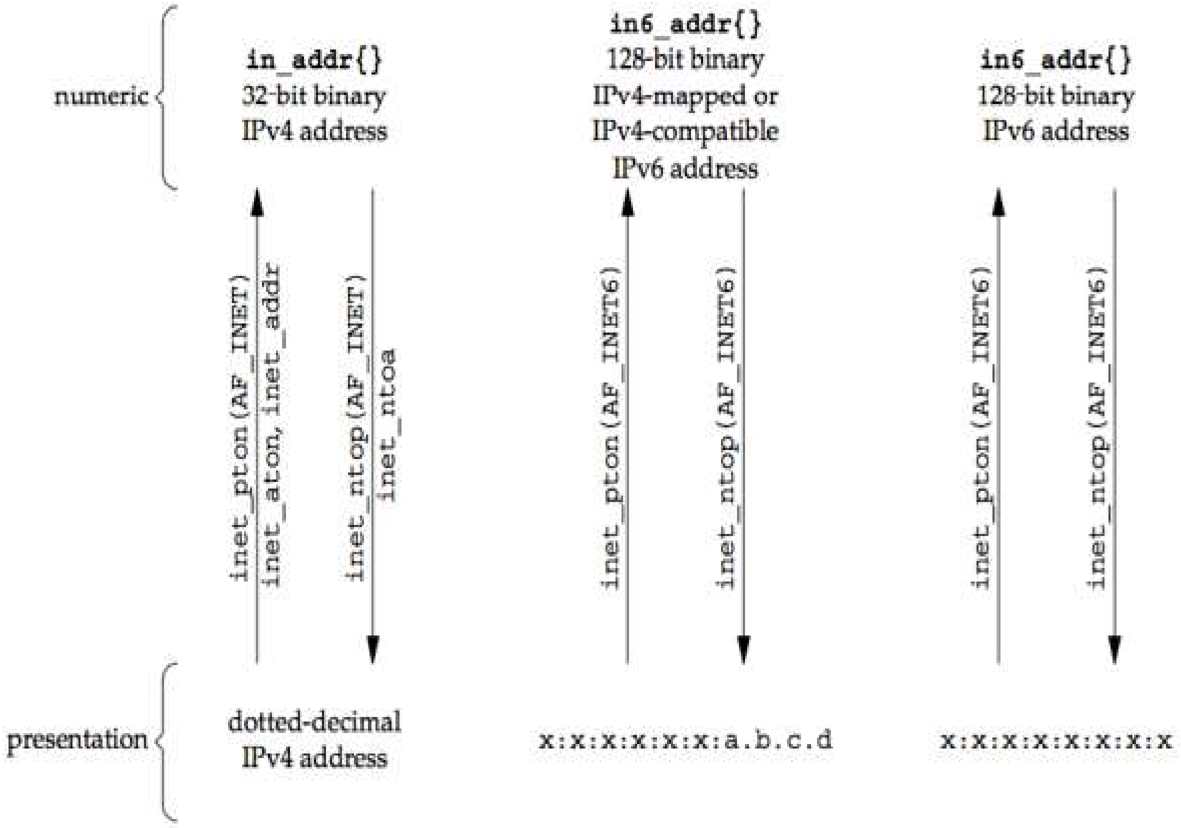


```
#include <arpa/inet.h>

int inet_pton(int family, const char *strptr, void *addrptr);
/* Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error */

const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);
/* Returns: pointer to result if OK, NULL on error */
```

Conversion Functions -3

- inet_pton: converts the string pointed to by strptr, storing the binary result through the pointer addrptr. If successful, the return value is 1. If the input string is not a valid presentation format for the specified family, 0 is returned.
- inet_ntop does the reverse conversion, from numeric (addrptr) to presentation (strptr).
- len argument is the size of the destination.

Conversion Functions

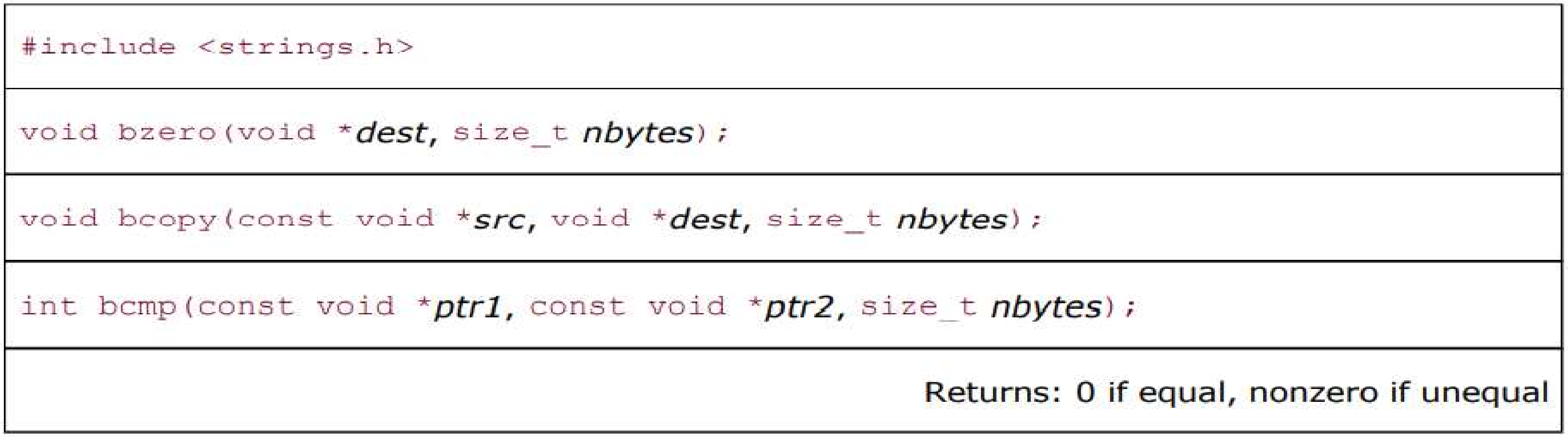


Byte Manipulation Functions

- There are two groups of functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string.
- We need these types of functions when dealing with socket address structures because we need to manipulate fields such as IP addresses, which can contain bytes of 0, but are not C character strings.
- The first group of functions, whose names begin with b (for byte), are from 4.2BSD and are still provided by almost any system that supports the socket functions.
- The second group of functions, whose names begin with mem (for memory), are from the ANSI C standard and are provided with any system that supports an ANSI C library.

Berkeley-derived functions

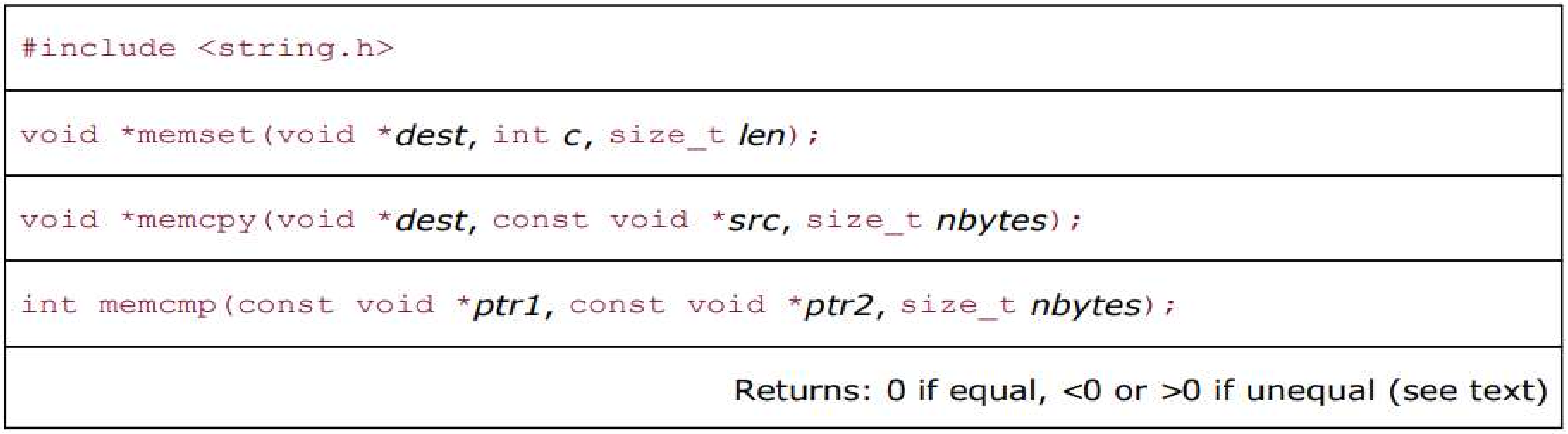
- *bzero* sets the specified number of bytes to 0 in the destination. We often use this function to initialize a socket address structure to 0.
- *bcopy* moves the specified number of bytes from the source to the destination.
- *bcmp* compares two arbitrary byte strings. The return value is zero if the two byte strings are identical; otherwise, it is nonzero.



```
#include <strings.h>

void bzero(void *dest, size_t nbytes);

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

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

                                        Returns: 0 if equal, nonzero if unequal
```

The ANSI C functions:

- *memset* sets the specified number of bytes to the value c in the destination.
- *memcpy* is similar to *bcopy*, but the order of the two pointer arguments is swapped.
- *bcopy* correctly handles overlapping fields, while the behaviour of *memcpy* is undefined if the source and destination overlap. The ANSI C *memmove* function must be used when the fields overlap.

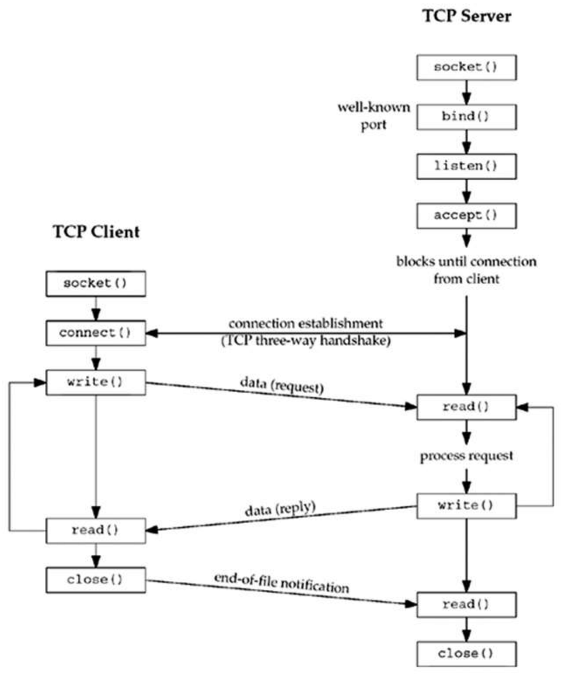


```
#include <string.h>

void *memset(void *dest, int c, size_t len);

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

int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);

                              Returns: 0 if equal, <0 or >0 if unequal (see text)
```

Socket Functions

- Socket functions required to write a complete TCP client and server, along with concurrent servers.
- Each client connection causes the server to fork a new process just for that client.
- A timeline of the typical scenario that takes place between a TCP client and server.

First, the server is started, then sometime later, a client is started that connects to the server.

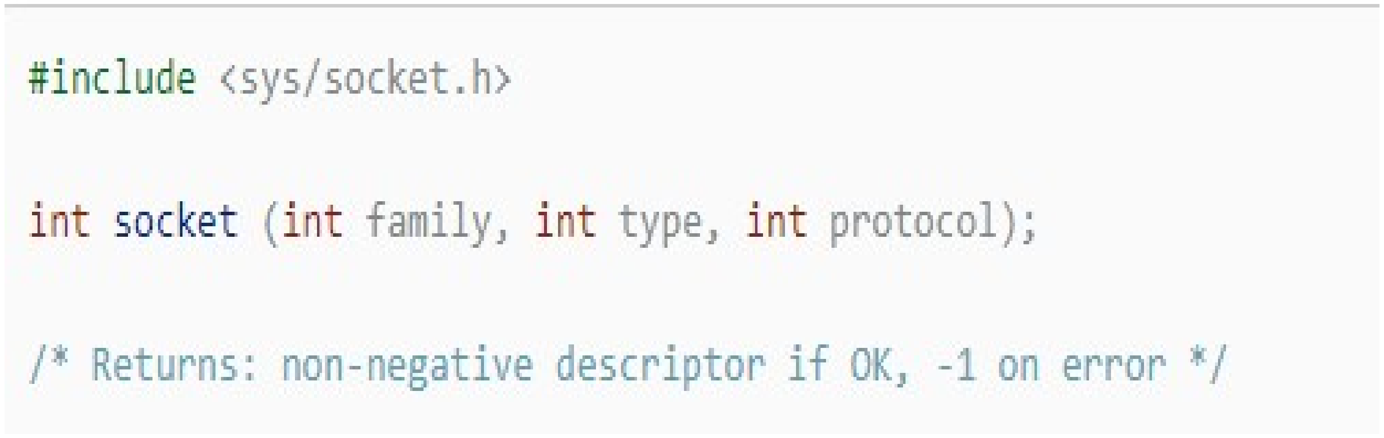


Socket Functions

- The client sends a request to the server, the server processes the request, and the server sends a reply back to the client.
- This continues until the client closes its end of the connection, which sends an end-of-file notification to the server.
- The server then closes its end of the connection and either terminates or waits for a new client 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.).

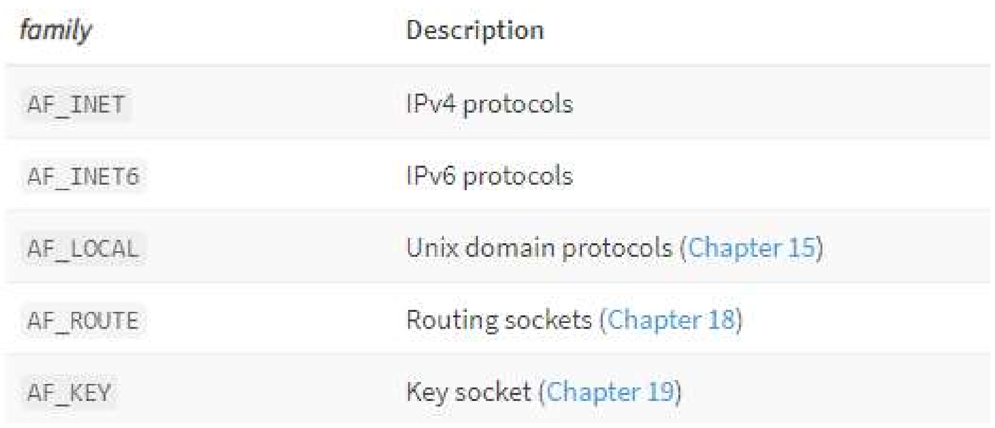


```
#include <sys/socket.h>

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

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

socket()

Arguments:

family specifies the protocol family and is one of the constants in the table below.



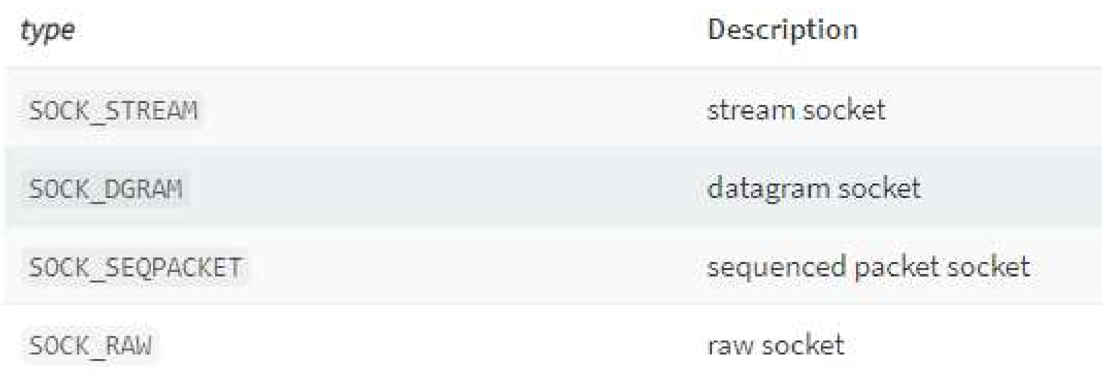
family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key socket (Chapter 19)

This argument is often referred to as domain instead of family.

socket()

Arguments:

The socket *type* is one of the constants shown in table below:



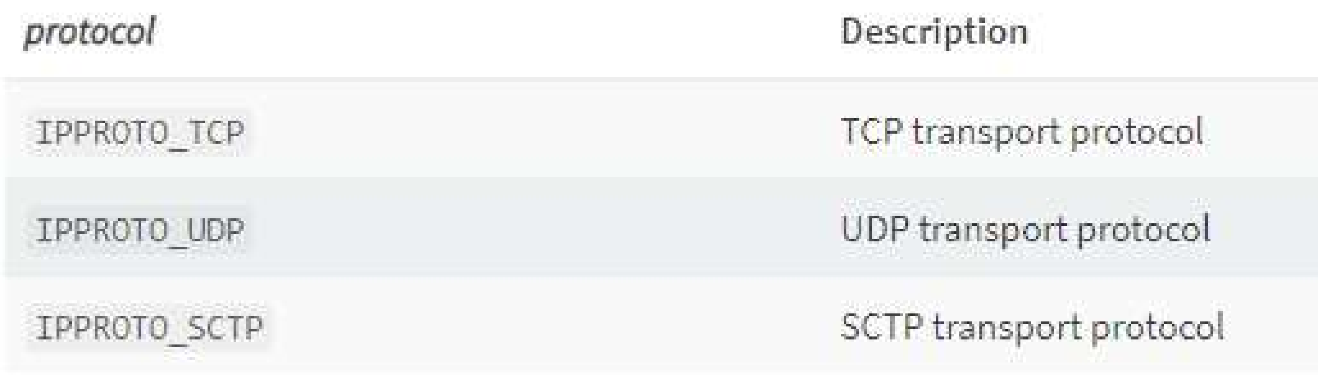
type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

This argument is often referred to as domain instead of family.

socket()

Arguments:

The *protocol* argument to the socket function should be set to the specific protocol type found in the table below, or 0 to select the system's default for the given combination of *family* and *type*.

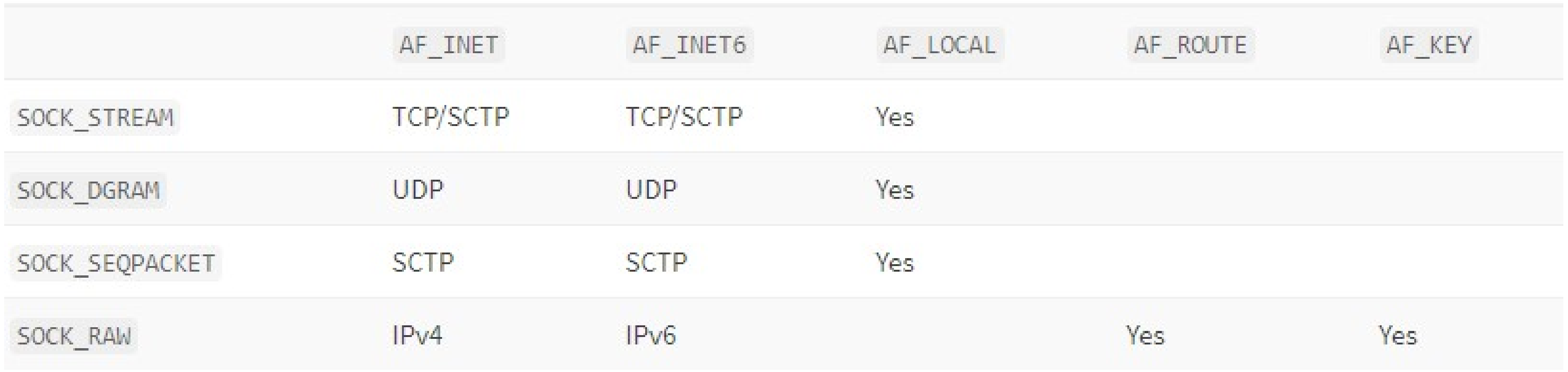


protocol	Description
`IPPROTO_TCP`	TCP transport protocol
`IPPROTO_UDP`	UDP transport protocol
`IPPROTO_SCTP`	SCTP transport protocol

socket()

Arguments:

Not all combinations of socket *family* and *type* are valid.

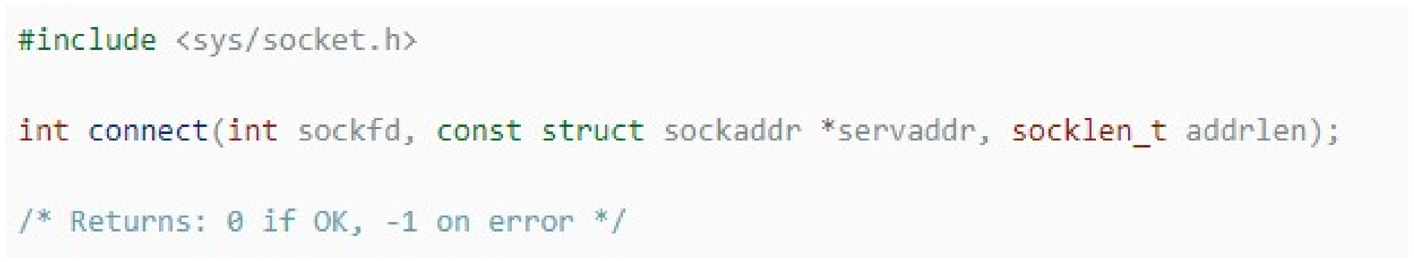


	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP/SCTP	TCP/SCTP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_SEQPACKET	SCTP	SCTP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

❖ On success, the socket function returns a small non-negative integer value, similar to a file descriptor. This is called a **socket descriptor**, or a *sockfd*.

connect()

The connect function is used by a TCP client to establish a connection with a TCP server.



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

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

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

- *sockfd* is a socket descriptor returned by the socket function.
- The *servaddr* and *addrlen* arguments are a pointer to a socket address structure (which contains the IP address and port number of the server) and its size.

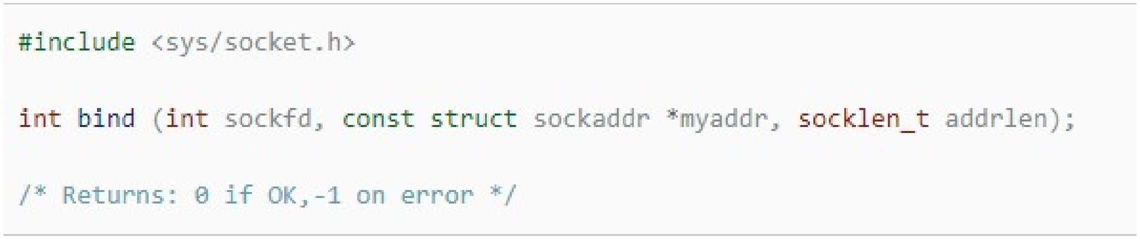
Note: The client does not have to call bind before calling connect: the kernel will choose both an ephemeral port and the source IP address if necessary.

connect()

- 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. There are several different error returns possible:
  - ➢If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned.
  - ➢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.
  - ➢If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered a **soft error**.

bind()

- The bind function assigns a local protocol address to a socket. 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.



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

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

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

- The second argument *myaddr* is a pointer to a protocol-specific address
- The third argument *addrlen* is the size of this address structure.

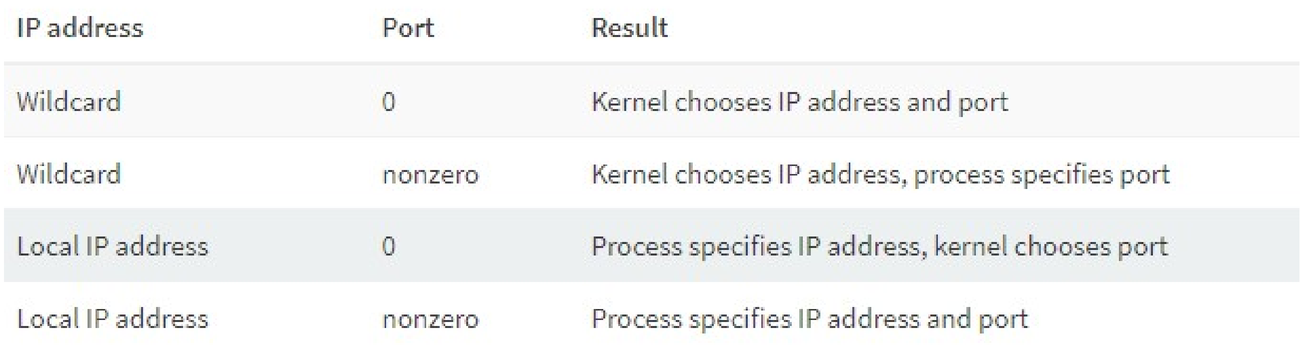
<u>With TCP, calling bind lets us specify a port number, an IP address, both, or neither.</u>

bind()

- **Servers bind their well-known port when they start.**
- If a TCP client or server does not do this, the kernel chooses an ephemeral port for the socket when either connect or listen is called.
  - It is normal for a TCP client to let the kernel choose an ephemeral port, unless the application requires a reserved port
  - However, it is rare for a TCP server to let the kernel choose an ephemeral port, since servers are known by their well-known port.

bind()

- As mentioned, 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.



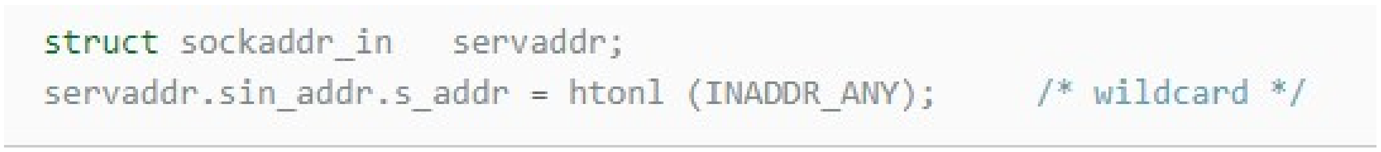
IP address	Port	Result
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

- 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).

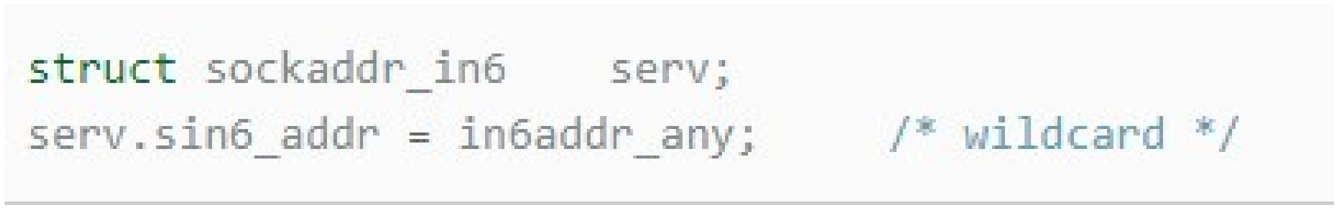
bind()

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 */
```



```
struct sockaddr_in6    serv;
serv.sin6_addr = in6addr_any;     /* wildcard */
```

bind()

Binding a non-wildcard IP address

- A common example of a process binding a non-wildcard IP address to a socket is a host that provides Web servers to multiple organizations:

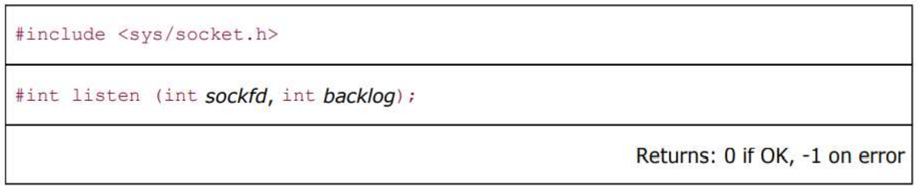
1. First, each organization has its own domain name, such as www.organization.com.
2. Next, each organization's domain name maps into a different IP address, but typically on the same subnet.

listen()

- The listen function is called only by a TCP server and it performs two actions:

1. The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
2. The second argument backlog to this function specifies the maximum number of connections the kernel should queue for this socket.

❖ This function is normally called after both the socket and bind functions and must be called before calling the accept function.



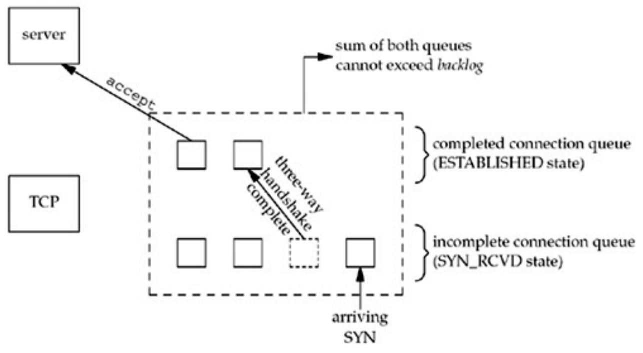
```
#include <sys/socket.h>

#int listen (int sockfd, int backlog);

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

listen()

Connection queues *- To understand the backlog argument

- The listening socket, the kernel maintains two queues:

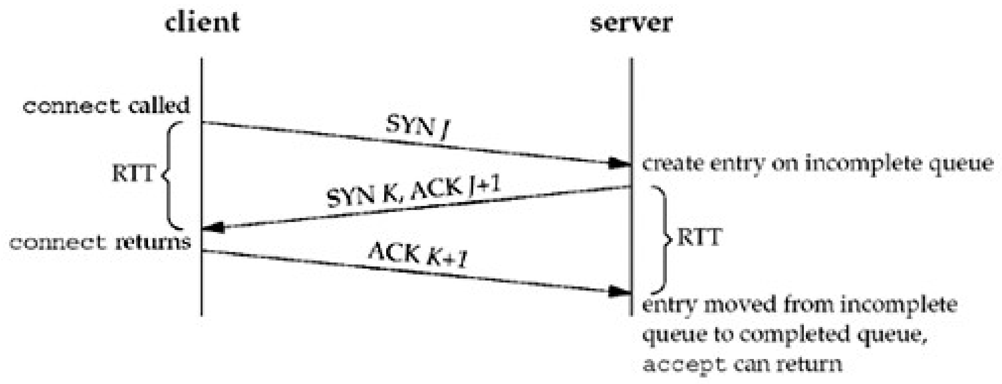
1. An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state.
2. A completed connection queue, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state


server
accept
sum of both queues cannot exceed backlog
completed connection queue (ESTABLISHED state)
three-way handshake complete
TCP
incomplete connection queue (SYN_RCVD state)
arriving SYN

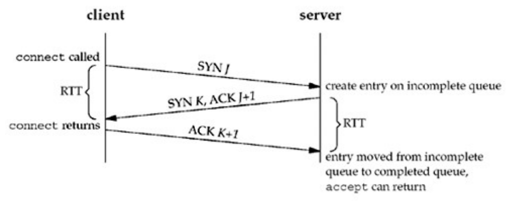
listen()

Packet exchanges during connection establishment

- The following figure depicts the packets exchanged during the connection establishment with these two queues:


client
server
connect called
RTT
connect returns
SYN J
SYN K, ACK J+1
ACK K+1
create entry on incomplete queue
RTT
entry moved from incomplete queue to completed queue, accept can return

listen()


client
server
connect called
SYN J
create entry on incomplete queue
RTT
SYN K, ACK J+1
RTT
connect returns
ACK K+1
entry moved from incomplete queue to completed queue, accept can return

Packet exchanges during connection establishment *

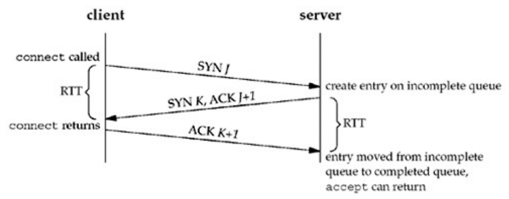
- When a SYN arrives from a client, TCP creates a new entry on

the incomplete queue and then responds with the second segment of the three-way handshake: the server's SYN with an ACK of the client's SYN

This entry will remain on the incomplete queue, until:

- The third segment of the three-way handshake arrives (the client's ACK of the server's SYN), or
- The entry times out.
- If the three-way handshake completes normally, the entry moves from the incomplete queue to the end of the completed queue.
- When the process calls accept:
  - The first entry on the completed queue is returned to the process, or
  - If the queue is empty, the process is put to sleep until an entry is placed onto the completed queue.

listen()


client
server
connect called
SYN J
create entry on incomplete queue
RTT
SYN K, ACK J+1
RTT
connect returns
ACK K+1
entry moved from incomplete queue to completed queue, accept can return

The backlog argument *

Several points to consider when handling the two queues:

- Sum of both queues.
- Multiplied by 1.5.
- Do not specify value of 0 for backlog, as different implementations interpret this differently If you do not want any clients connecting to your listening socket, close the listening socket.
- One RTT.
- Configurable maximum value.

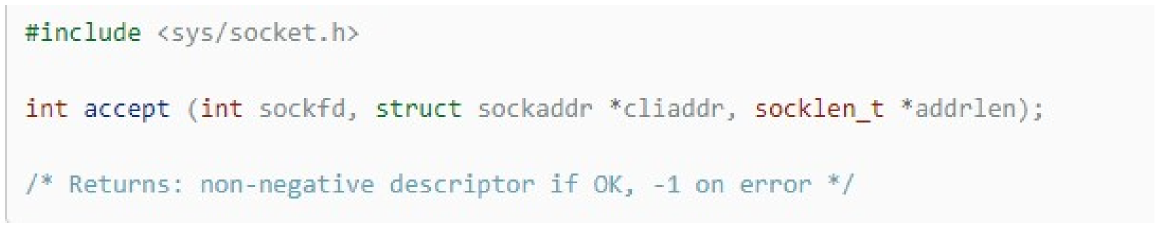
listen()

SYN Flooding *

- SYN flooding is a type of attack (the attacker writes a program to send SYNs at a high rate to the victim) that attempts to fill the incomplete connection queue for one or more TCP ports.
- Additionally, the source IP address of each SYN is set to a random number (called IP spoofing) so that the server's SYN/ACK goes nowhere.
- This also prevents the server from knowing the real IP address of the attacker.
- By filling the incomplete queue with bogus SYNs, legitimate SYNs are not queued, providing a denial of service to legitimate clients.

accept()

- accept is called by a TCP server to return the next completed connection from the front of the completed connection queue.
- If the completed connection queue is empty, the process is put to sleep (assuming the default of a blocking socket).



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

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

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

- The *cliaddr* and *addrlen* arguments are used to return the protocol address of the connected peer process (the client).
- *addrlen* is a value-result argument
  - Before the call, we set the integer value referenced by **addrlen* to the size of the socket address structure pointed to by *cliaddr*;
- On return, this integer value contains the actual number of bytes stored by the kernel in the socket address structure.

accept()

- If successful, accept returns a new descriptor automatically 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.

It is important to differentiate between these two sockets:

- 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 accepted (for which the TCP three-way handshake completes).
- When the server is finished serving a given client, the connected socket is closed.

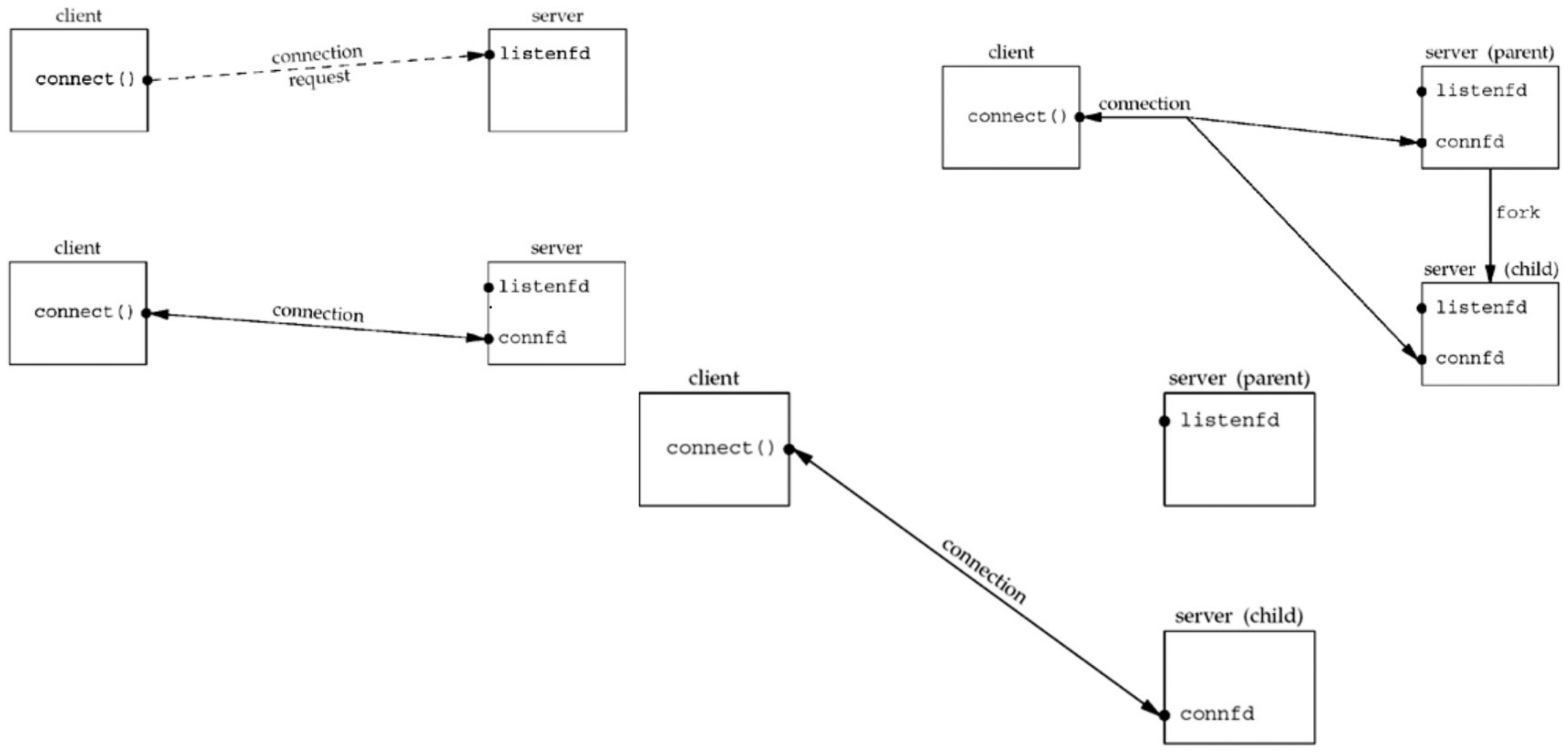
accept()

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).

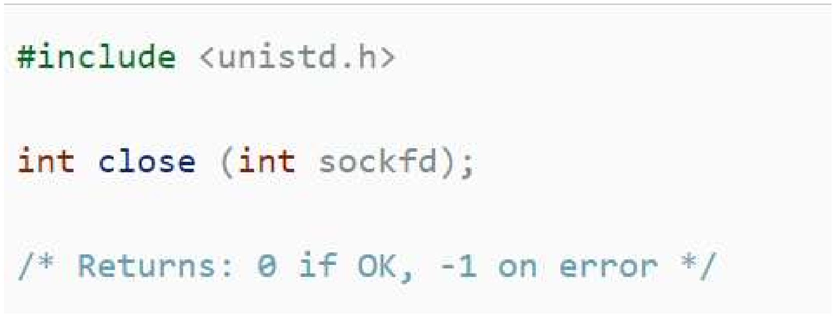
Visualizing the sockets and connection

Before call to accept, the server is blocked to accept the connection



Close()

- Used to close a socket and terminate a TCP connection.

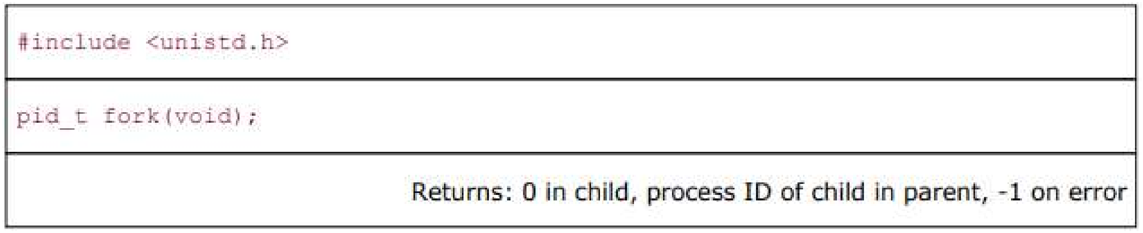


```
#include <unistd.h>

int close (int sockfd);

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

- The default action is mark the socket as closed and return to the process immediately.

fork and exec Functions

- fork() and exec() are used in creation of concurrent servers.
- The fork function returns twice.



```
#include <unistd.h>

pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, -1 on error

- It returns once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child).
- It also returns once in the child, with a return value of 0.
- The return value tells the process whether it is the parent or the child.
- All descriptors open in the parent before the call to fork are shared with the child after fork returns.

fork and exec Functions

- In network servers, the parent calls accept and then calls fork.
- The connected socket is then shared between the parent and child.
- Normally, the child then reads and writes the connected socket and the parent closes the connected socket.
- There are two typical uses of fork:

1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task.
2. A process wants to execute another program. Since the only way to create a new process is by calling fork, the process first calls fork to make a copy of itself, and then one of the copies (typically the child process) calls exec (described next) to replace itself with the new program.

fork and exec Functions

- The only way in which an executable program file on disk can be executed by Unix is for an existing process to call one of the six exec functions.
- exec replaces the current process image with the new program file, and this new program normally starts at the main function.
- The process ID does not change.
- The process that calls exec as the calling process and the newly executed program as the new program.

fork and exec Functions

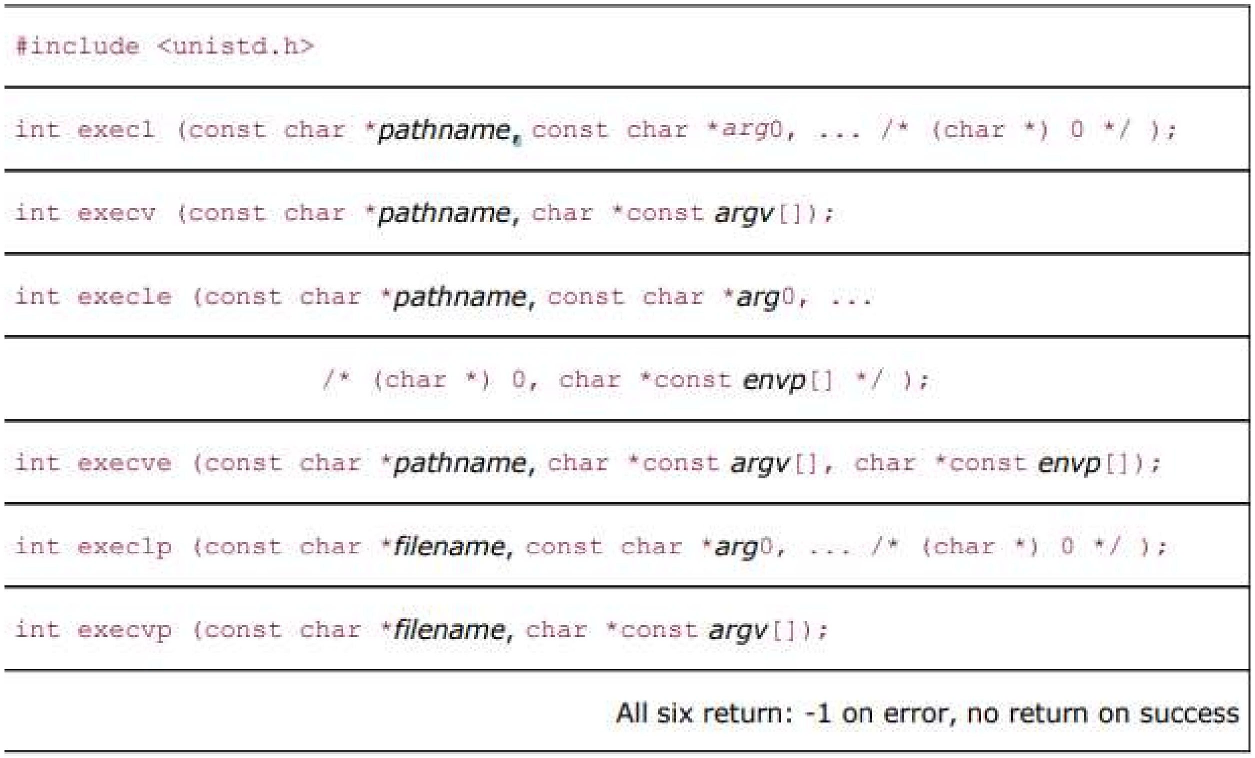
- The differences in the six exec functions are:

  (a) whether the program file to execute is specified by a filename or a pathname

  (b) whether the arguments to the new program are listed one by one or referenced through an array of pointers.

  (c) whether the environment of the calling process is passed to the new program or whether a new environment is specified.

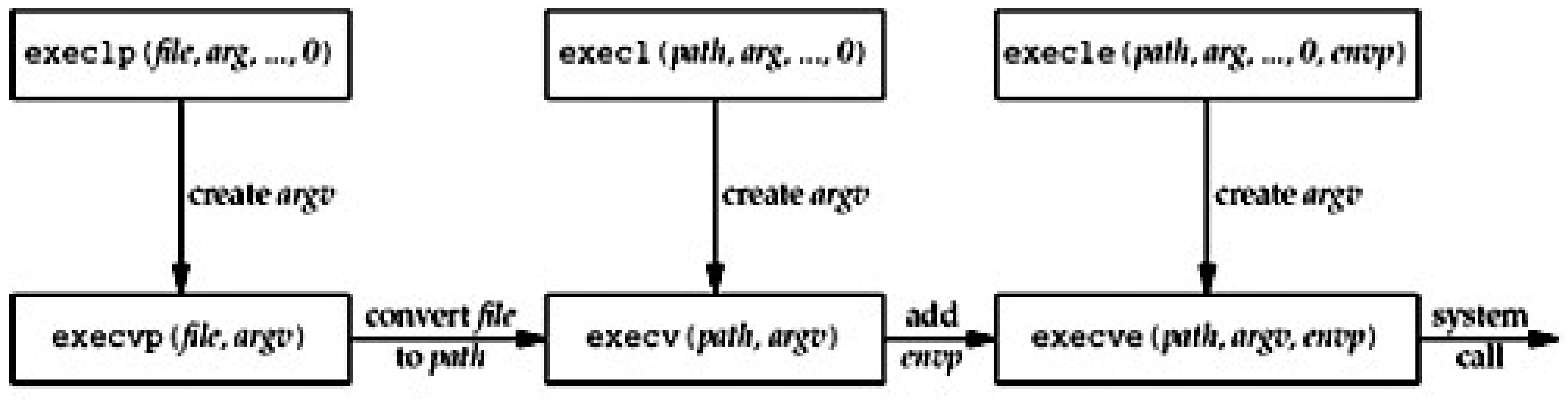
exec Functions



```
#include <unistd.h>

int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */ );

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

int execle (const char *pathname, const char *arg0, ...
                   /* (char *) 0, char *const envp[] */ );

int execve (const char *pathname, char *const argv[], char *const envp[]);

int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */ );

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

                         All six return: -1 on error, no return on success
```

Descriptors open in the process before calling exec normally remain open across the exec.

exec Functions



Only execve is a system call within the kernel and the other five are library functions that call execve.

exec Functions

Differences among these six functions:

•The three functions in the top row specify each argument string as a separate argument to the exec function, with a null pointer terminating the variable number of arguments. The three functions in the second row have an argv array, containing pointers to the argument strings. This argv array must contain a null pointer to specify its end, since a count is not specified.

•

•The two functions in the left column specify a filename argument. This is converted into a pathname using the current PATH environment variable. If the filename argument to execlp or execvp contains a slash (/) anywhere in the string, the PATH variable is not used. The four functions in the right two columns specify a fully qualified pathname argument.

•The four functions in the left two columns do not specify an explicit environment pointer. Instead, the current value of the external variable environ is used for building an environment list that is passed to the new program. The two functions in the right column specify an explicit environment list. The envp array of pointers must be terminated by a null pointer.