RV College of Engineering®

Go, change the world

NETWORK PROGRAMMING AND SECURITY (Theory & Lab)

18CS54

TCP versus UDP

What is TCP? What does it provide?

- TCP (Transmission Control Protocol) guarantees that all segments will arrive at the destination and in the right order. (It makes no guarantees about how long it will take.)

What is UDP? Why use it?

- UDP (User Datagram Protocol) makes no such guarantees. So UDP is mostly like IP, but with the extension that it gets data from source process to destination process.
- UDP is used where performance is needed and occasional loss of data is not critical. E.g. audio and video streaming, short protocols such as DNS, name services, etc.
- UPD is a faster protocol. It has less overhead. But the trade off is loss and unordered.

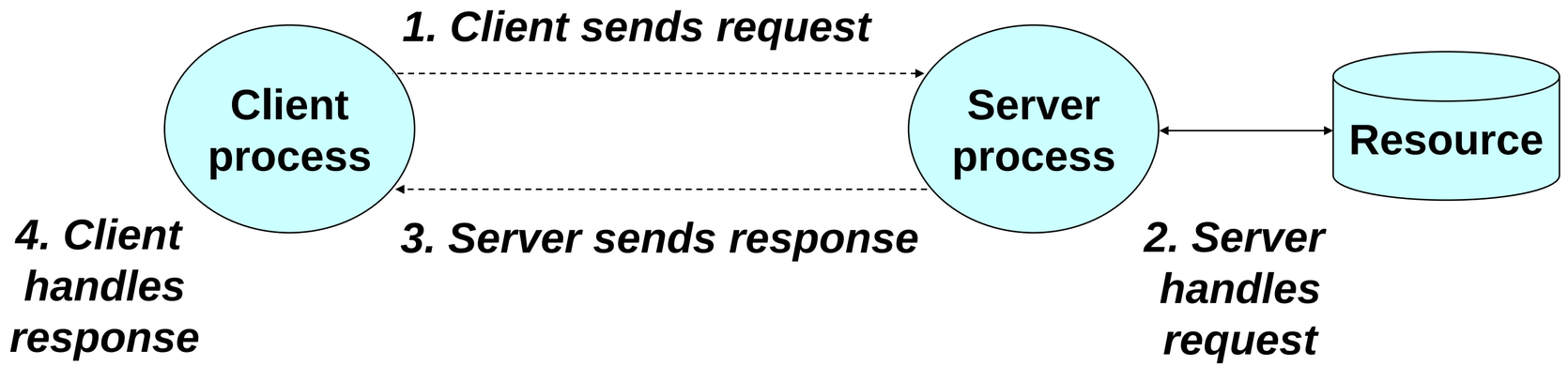
Questions to answer:

1. What is the client-server model?
2. What is the difference between the client and the server?
3. What are example client/server pairs?
4. Are there other models?

TCP

Client-Server Model

- Every network application is based on the client-server model:
  - –A *server* process and one or more *client* processes
  - –Server manages some *resource*.
  - –Server provides *service* by manipulating resource for clients.



Note: clients and servers are processes running on hosts (can be the same or different hosts).

- Examples of client programs
  - –Web browsers, ***ftp, telnet, ssh***

- How does a client find the server?
  - –The IP address in the server socket address identifies the host *(more precisely, an adapter on the host)*
  - –The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.

  - –Examples of well known ports
    - ***Port 7: Echo server***
    - ***Port 23: Telnet server***
    - ***Port 25: Mail server***
    - ***Port 80: Web server.***

Servers

- Servers are long-running processes (daemons).
  - –Created at boot-time (typically) by the init process (process 1)
  - –Run continuously until the machine is turned off.

- Each server waits for requests to arrive on a well-known port associated with a particular service.
  - –***Port 7: echo server***
  - –***Port 23: telnet server***
  - –***Port 25: mail server***
  - –***Port 80: HTTP server***

- A machine that runs a server process is also often referred to as a “server.”

Servers Examples

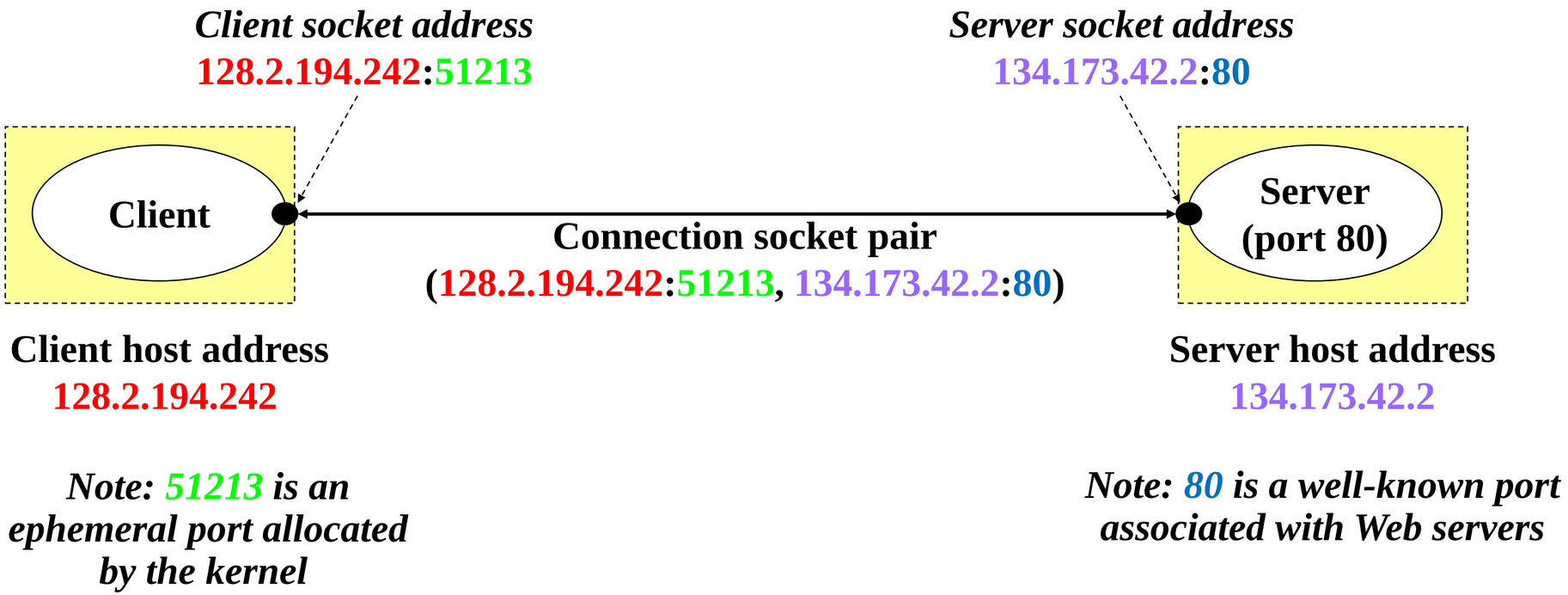
- ***Web server (port 80)***
  - –Resource: files/compute cycles (CGI programs)
  - –Service: retrieves files and runs CGI programs on behalf of the client
- ***FTP server (20, 21)***
  - –Resource: files
  - –Service: stores and retrieve files
- ***Telnet server (23)***
  - –Resource: terminal
  - –Service: proxies a terminal on the server machine
- ***Mail server (25)***
  - –Resource: email “spool” file
  - –Service: stores mail messages in spool file

See /etc/services for a comprehensive list of the services available on a Linux machine.

Client-Server Interaction

- Clients and servers communicate by sending streams of bytes over *connections.*

- Connections are point-to-point, full-duplex (2-way communication), and reliable.



UNIT 1

The Transport Layer and introduction to sockets

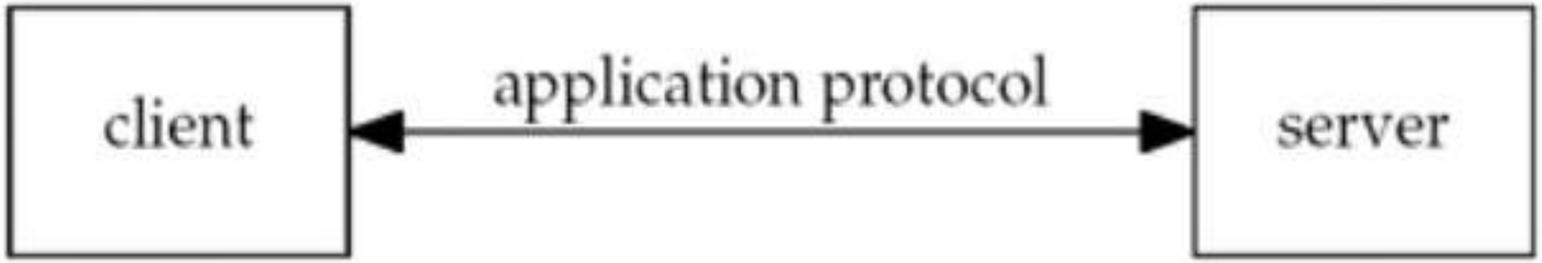
Contents

1. Introduction to all protocols
2. The big Picture
3. Difference between TCP, UDP and SCTP
4. TCP Connection Establishment and Termination and TIME_WAIT state
5. TCP port numbers and concurrent servers, Buffer sizes and limitation

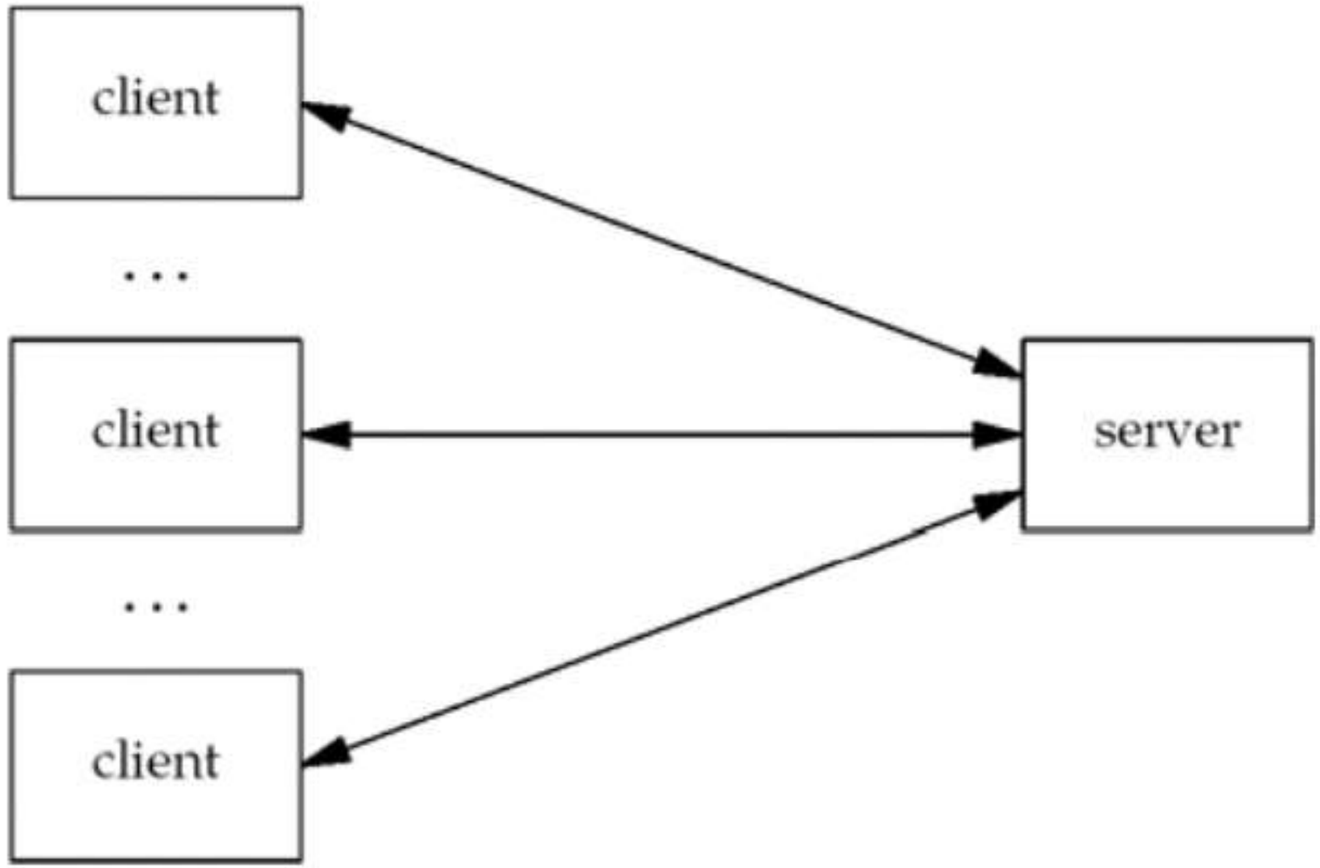
Introduction to all protocols

- **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:



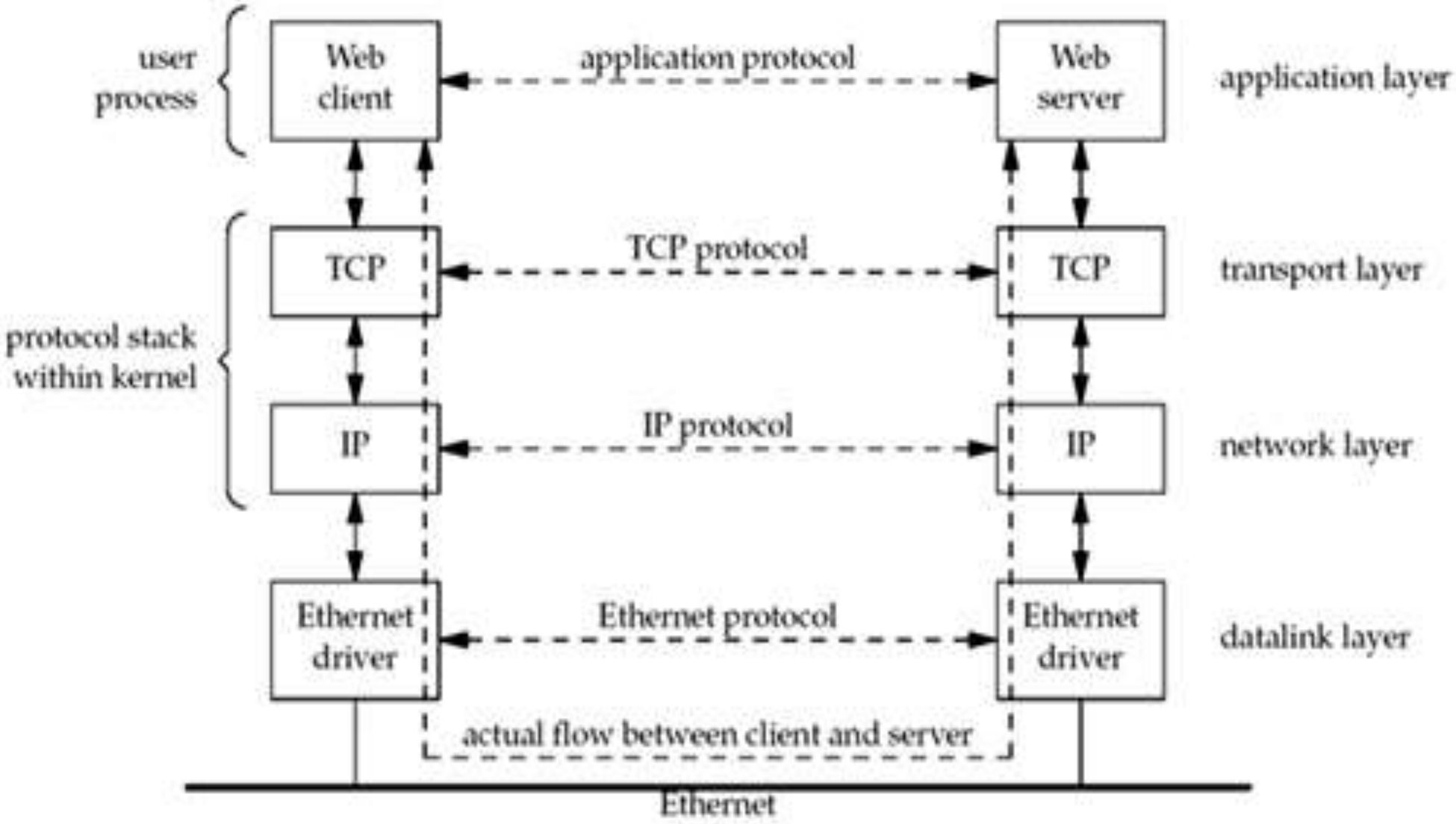
❖:The client application and the server application may be thought of as communicating **via a multiple layers of protocols**

❖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

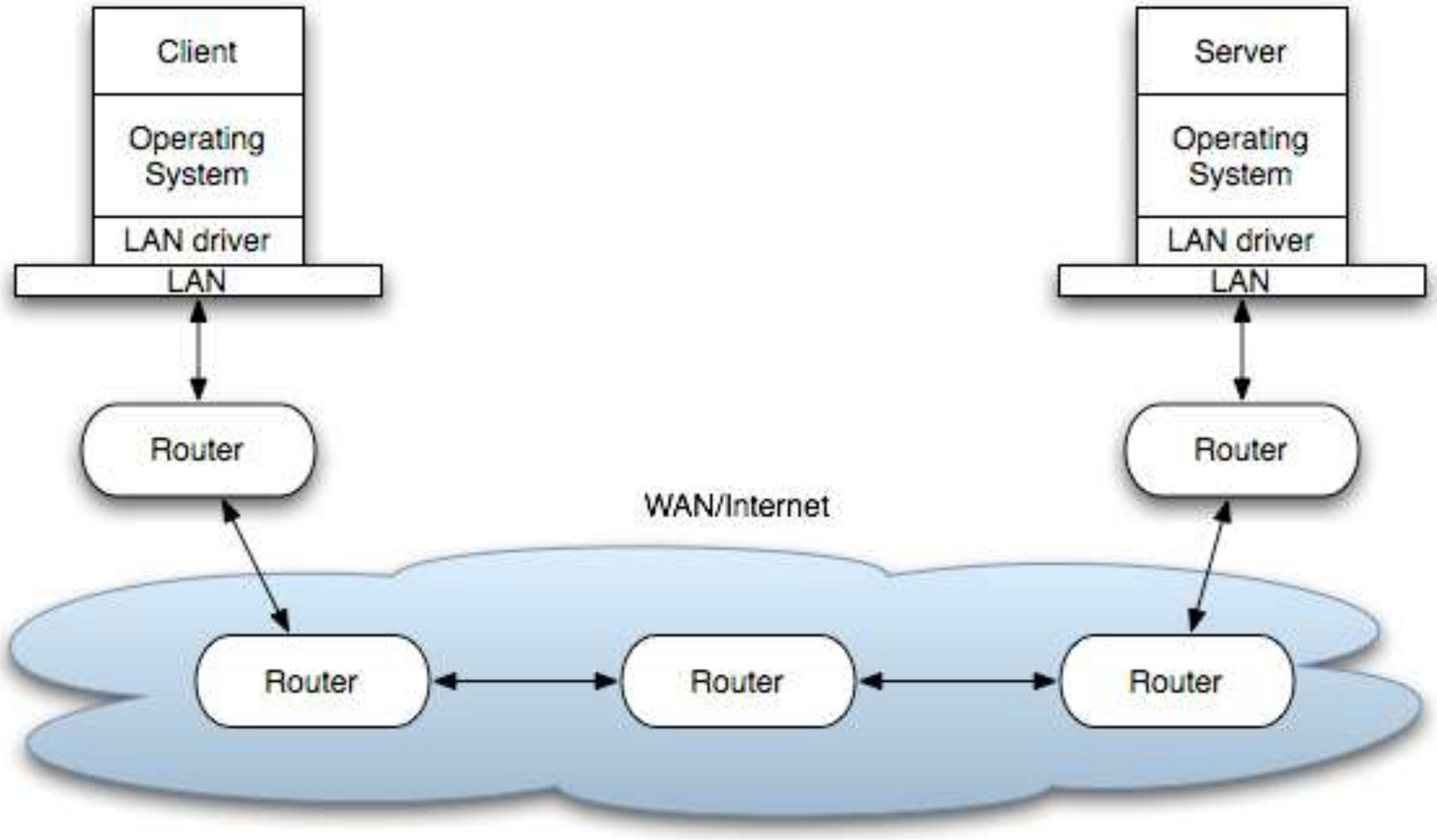
For example, Web clients and servers communicate using the Transmission Control Protocol, or TCP.

TCP, in turn, uses the Internet Protocol, or IP, and IP communicates with a datalink layer of some form.

If the client and server are on the same Ethernet then one can think of the communication in the following way:


user process
Web client
application protocol
Web server
application layer
protocol stack within kernel
TCP
TCP protocol
TCP
transport layer
IP
IP protocol
IP
network layer
Ethernet driver
Ethernet protocol
Ethernet driver
datalink layer
actual flow between client and server
Ethernet

What if client and server may not be on a 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 TCP socket

The socket function creates an Internet (AF_INET) stream (SOCK_STREAM) socket, which is a fancy name for a TCP socket. The function returns a small integer descriptor to identify the socket.

Specify server's IP address and port

The IP address (sin_addr) and port number (sin_port) fields in the Internet socket address structure (sockaddr_in) must be in specific formats:

htons (host to network short): converts the binary port number

inet_pton (presentation to numeric): convert the ASCII command-line argument (such as 206.62.226.35 when we ran this example) into the proper format.

bzero is not an ANSI C function, but is used in this book instead of the ANSI C memset function, because bzero is easier to remember (with only two arguments) than memset (with three arguments).

Establish connection with server

❖Connect

In the unp.h header, SA is defined to be struct sockaddr, a generic socket address structure.

Read and display server's reply

We must be careful when using TCP because it is a **byte-stream** protocol with no record boundaries. Since we cannot assume that the server's reply will be returned by a single read, we always need to code the read in a loop when reading from a TCP socket.

Terminate program

exit terminates the program. Unix always closes all open descriptors when a process terminates.

A Simple Daytime Server


```
#include "unp.h"
#include <time.h>
int
main(int argc, char **argv)
{
int listenfd, connfd;
struct sockaddr_in servaddr;
char buff[MAXLINE];
time_t ticks;
listenfd = Socket(AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(13); /* daytime server */
Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
Listen(listenfd, LISTENQ);
for ( ; ; ) {
connfd = Accept(listenfd, (SA *) NULL, NULL);
ticks = time(NULL);
snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
Write(connfd, buff, strlen(buff));
Close(connfd);
}
}
```

Create a TCP socket

Identical to the client code.

Bind server's well-known port to socket

❖bind: the server's well-known port (13) is bound to the socket by calling bind

❖INADDR_ANY allows the server to accept a client connection on any interface

Convert socket to listening socket

❖listen: converts the socket into a listening socket, on which incoming connections from clients will be accepted by the kernel

❖listenfd in the code is called a **listening descriptor**

Accept client connection, send reply

❖accept

❖connfd in the code is called a **connected descriptor** for communication with the client. A new descriptor is returned by accept for each client that connects to our server.

This book uses this code style for infinite loop:

for (; ;) { // . . . }

snprintf function

❖snprintf instead of sprintf

Similarly:

❖fgets instead of gets

❖strncat or strlcat instead of strcat

❖strncpy or strlcpy instead of a strcpy

Terminate connection

close initiates the normal TCP connection termination sequence: a FIN is sent in each direction and each FIN is acknowledged by the other end.

The server implemented in the above server code is:

❖Protocol-dependent on IPv4

❖Handles only one client at a time. If multiple client connections arrive at about the same time, the kernel queues them, up to some limit, and returns them to accept one at a time.

❖Called an **iterative server**. A **concurrent server** handles multiple clients at the same time.

This chapter focuses on the transport layer:

TCP, UDP, and Stream Control Transmission Protocol (SCTP).

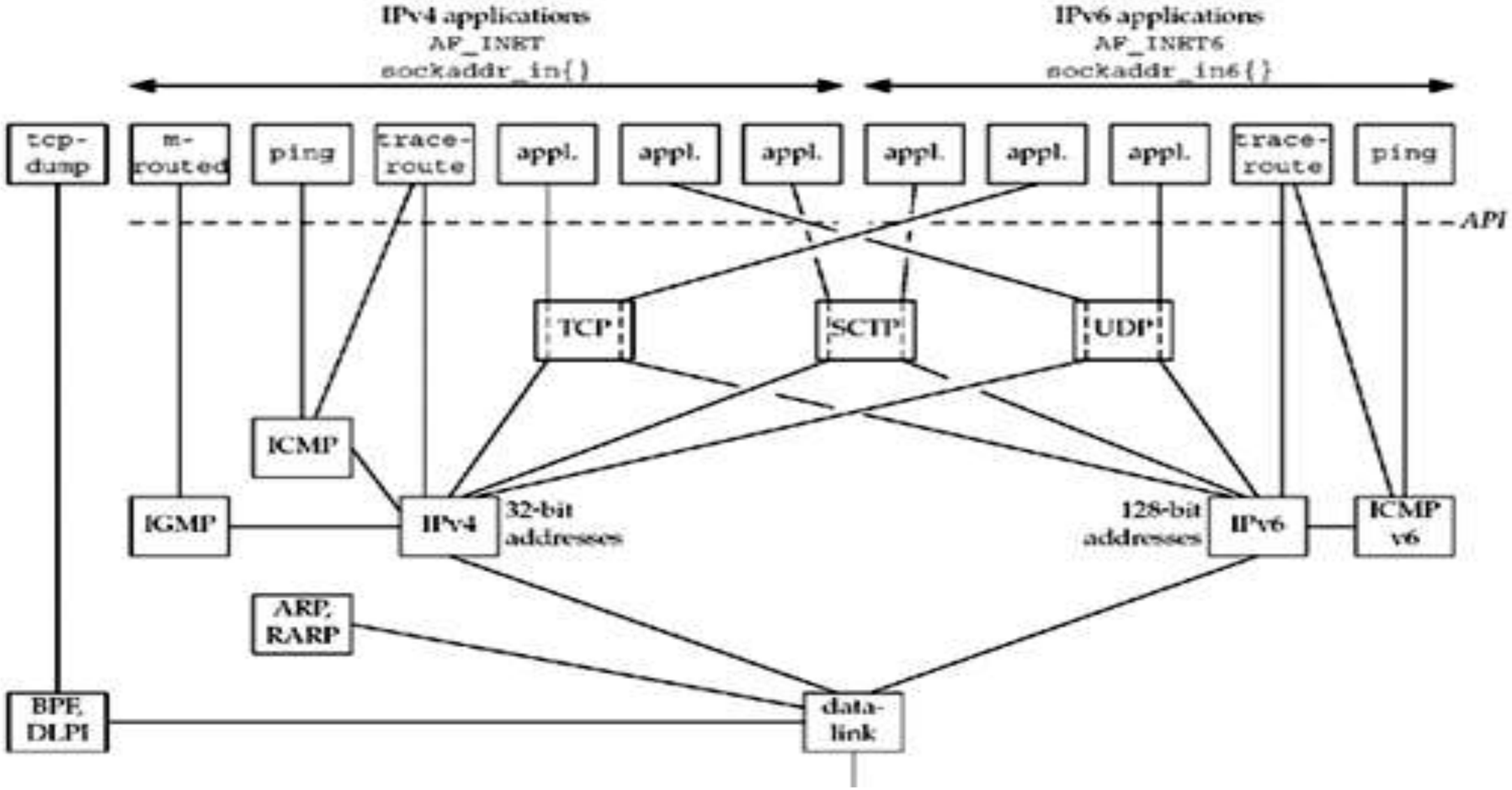
UDP is a simple, unreliable datagram protocol, while TCP is a sophisticated, reliable byte-stream protocol.

SCTP is similar to TCP as a reliable transport protocol, but it also provides message boundaries, transport-level support for multihoming, and a way to minimize head-of-line blocking.

Multihoming *is the practice of connecting a host or a computer network to more than one network. This can be done in order to increase reliability or performance. A typical host or end-user network is connected to just one network.*

The Big Picture

Although the protocol suite is called "TCP/IP," there are more members of this family than just TCP and IP.



Protocol	Description
IPv4	Internet Protocol version 4. IPv4 uses 32-bit addresses and provides packet delivery service for TCP, UDP, SCTP, ICMP, and IGMP.
IPv6	Internet Protocol version 6. IPv6 uses 128-bit addresses.
TCP	Transmission Control Protocol. TCP is a connection-oriented protocol that provides a reliable, full-duplex byte stream to its users
UDP	User Datagram Protocol. UDP is a connectionless protocol, and UDP sockets are an example of datagram sockets.
SCTP	Stream Control Transmission Protocol. SCTP is a connection-oriented protocol that provides a reliable full-duplex association
ICMP	Internet Control Message Protocol. ICMP handles error and control information between routers and hosts.
IGMP	Internet Group Management Protocol. IGMP is used with multicasting.
ARP	Address Resolution Protocol. ARP maps an IPv4 address into a hardware address (such as an Ethernet address). ARP is normally used on broadcast networks such as Ethernet, token ring, and FDDI, and is not needed on point-to-point networks.
RARP	Reverse Address Resolution Protocol. RARP maps a hardware address into an IPv4 address. It is sometimes used when a diskless node is booting.
ICMPv6	Internet Control Message Protocol version 6. ICMPv6 combines the functionality of ICMPv4, IGMP, and ARP.
BPF	BSD packet filter. This interface provides access to the datalink layer. It is normally found on Berkeley-derived kernels.
DLPI	Datalink provider interface.

User Datagram Protocol (UDP)

- Lack of reliability
- Each UDP datagram has a length
- **Connectionless** service

Transmission Control Protocol (TCP)

- **Connection**: TCP provides connections between clients and servers. A TCP client establishes a connection with a server, exchanges data across the connection, and then terminates the connection.
- **Reliability**: TCP requires acknowledgment when sending data. If an acknowledgment is not received, TCP automatically retransmits the data and waits a longer amount of time.
- **Round-trip time** (RTT): TCP estimates RTT between a client and server dynamically so that it knows how long to wait for an acknowledgment.
- **Sequencing**: TCP associates a sequence number with every byte (**segment**, unit of data that TCP passes to IP.) it sends. TCP reorders out-of-order segments and discards duplicate segments.
- **Flow control**
- **Full-duplex**: an application can send and receive data in both directions on a given connection at any
- Time.

Stream Control Protocol (SCTP)

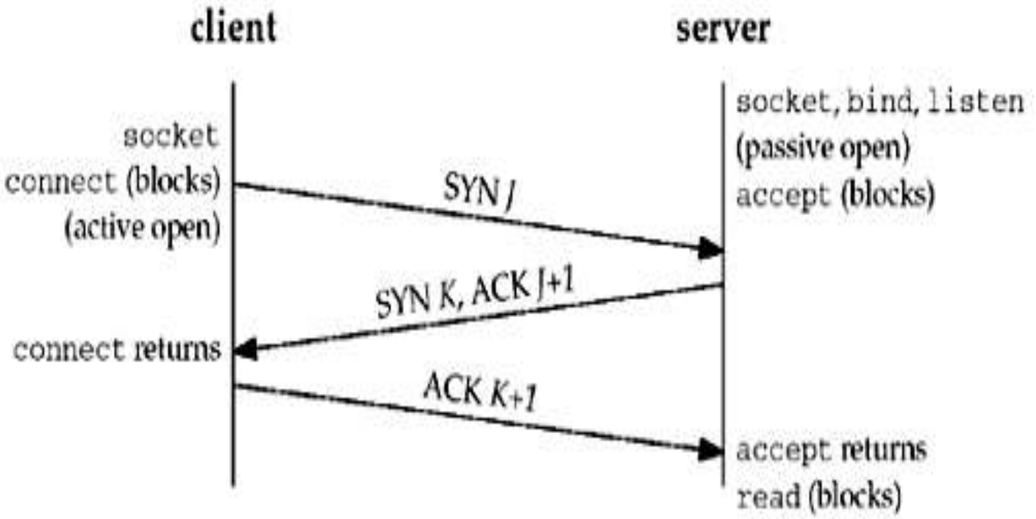
Like TCP, SCTP provides reliability, sequencing, flow control, and full-duplex data transfer.
Unlike TCP, SCTP provides:

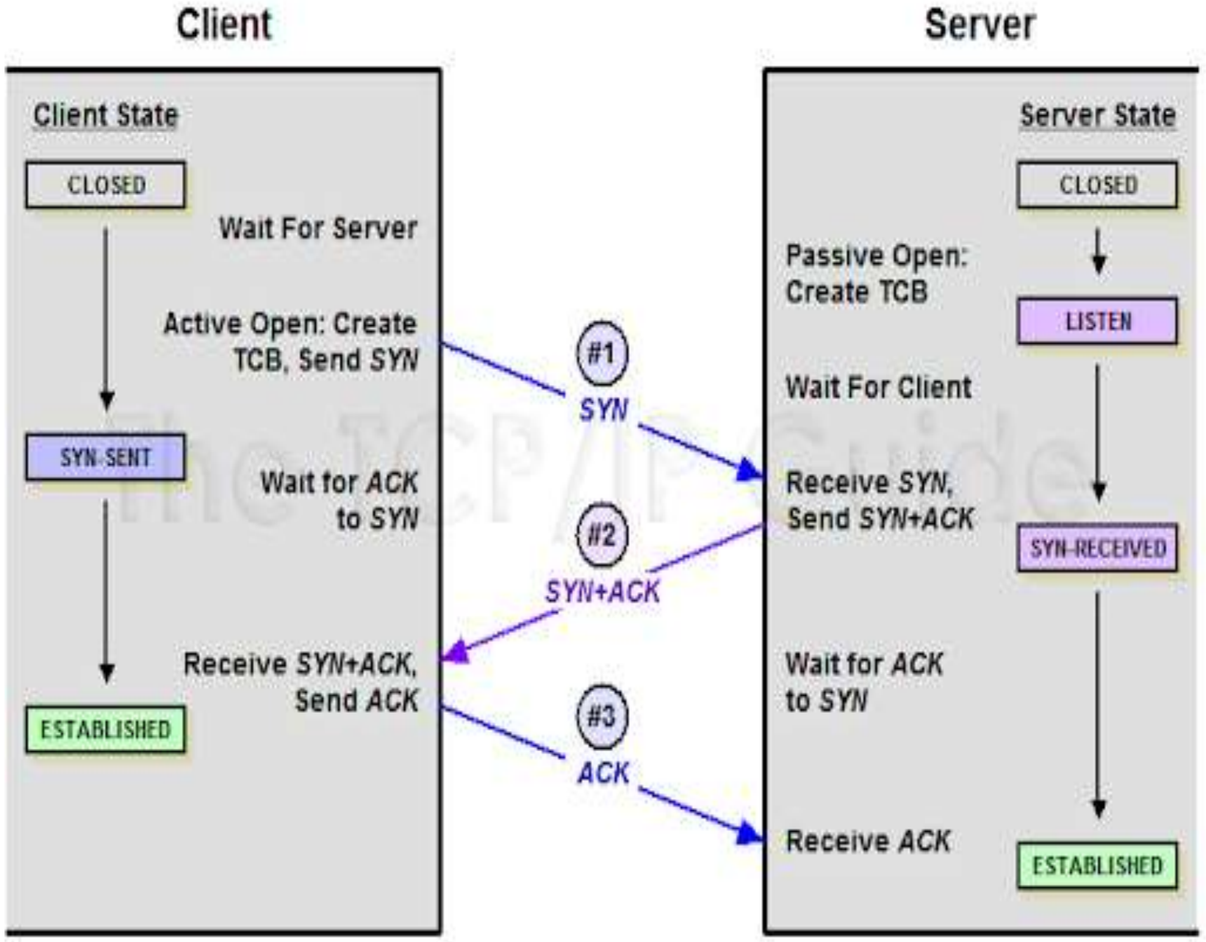
❖**Association** instead of "connection": An association refers to a communication between two systems, which may involve more than two addresses due to multihoming.

❖**Message-oriented**: provides sequenced delivery of individual records. Like UDP, the length of a record written by the sender is passed to the receiving application.

❖**Multihoming**: allows a single SCTP endpoint to support multiple IP addresses. This feature can provide increased robustness against network failure.

TCP Connection Establishment





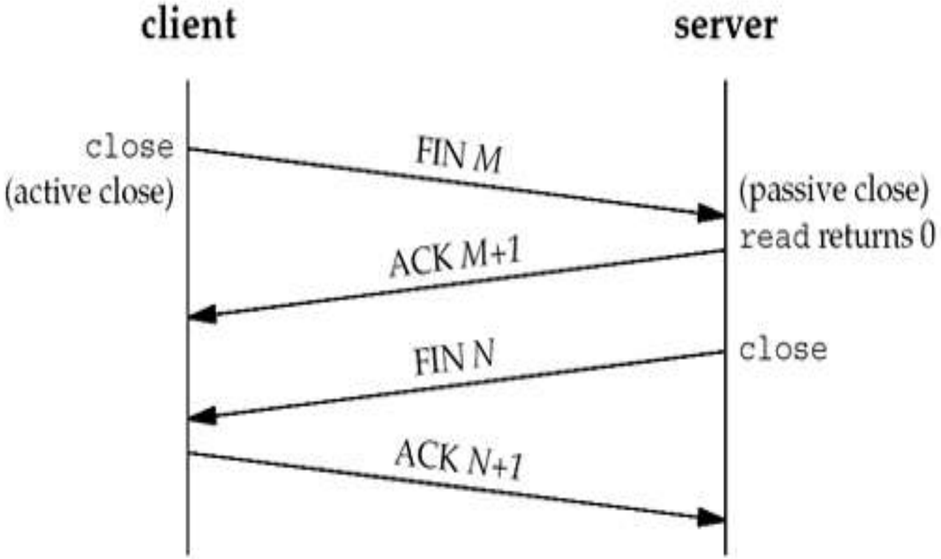
TCP Connection Establishment

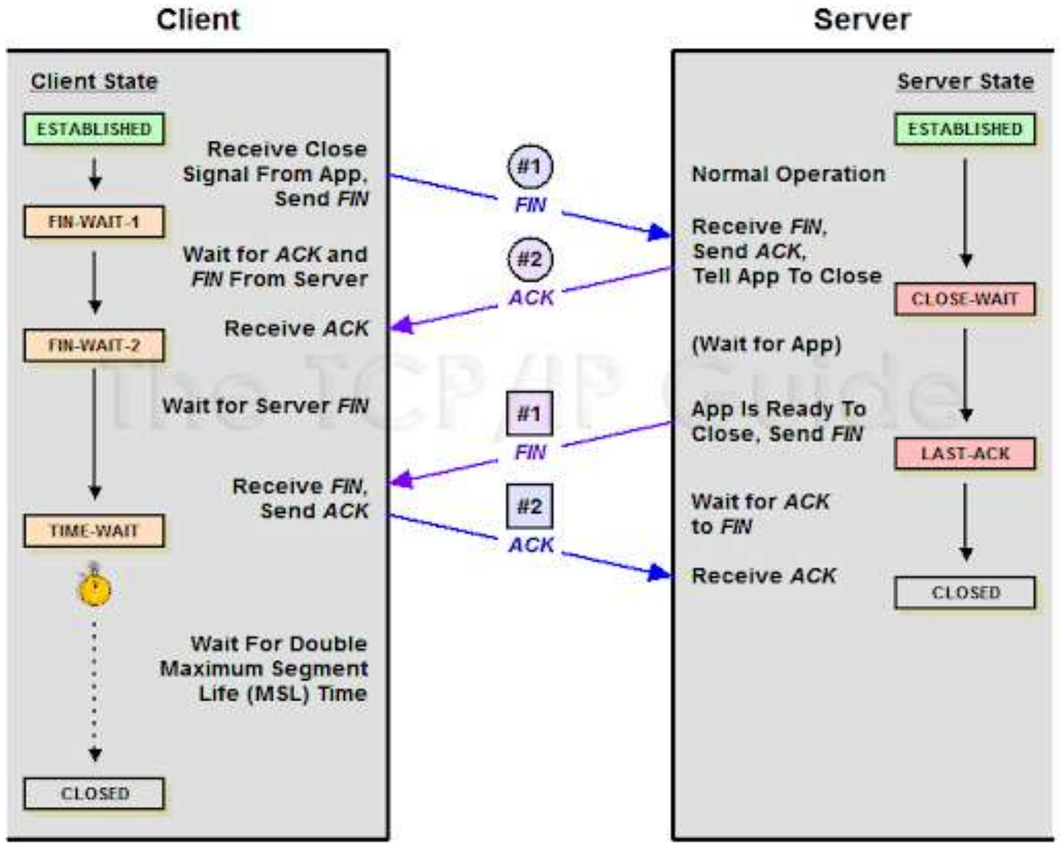
Three-Way Handshake

1. Server: **passive open**, by calling socket, bind, and listen
2. Client: **active open**, by calling connect. The client TCP to send a "synchronize" (SYN) segment with no data but it contains client's initial sequence number for the data to be sent on the connection.
3. Server: acknowledges (ACK) client's SYN. The server sends its SYN and the ACK of the client's SYN in a single segment which also contains its own SYN containing the initial sequence number for the data to be sent on the connection.
4. Client: acknowledges the server's SYN.

The client's initial sequence number as *J* and the server's initial sequence number as *K*. The acknowledgment number in an ACK is the next expected sequence number for the end sending the ACK. Since a SYN occupies one byte of the sequence number space, the acknowledgment number in the ACK of each SYN is the initial sequence number plus one.

TCP Connection Termination





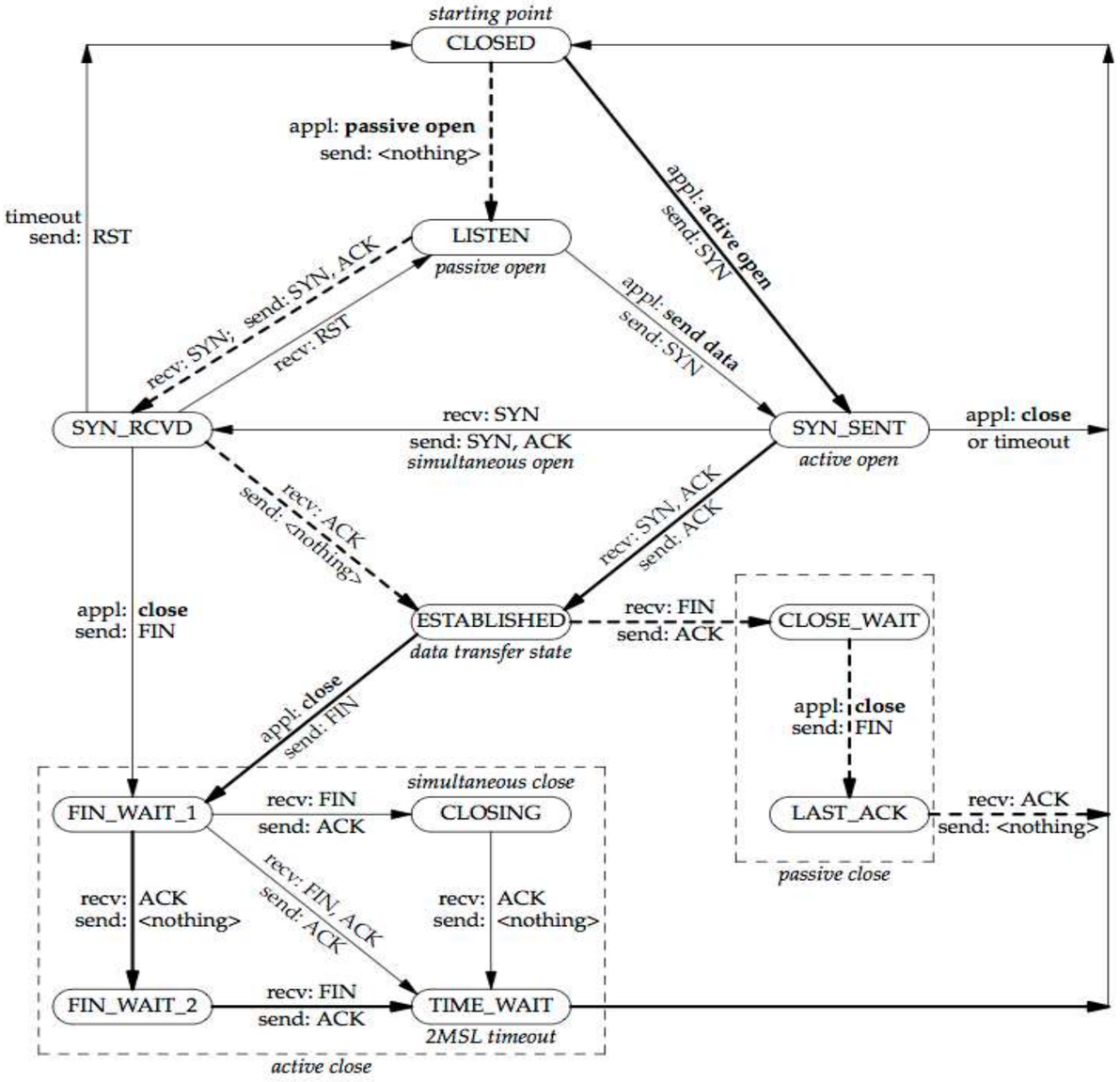
TCP Connection Termination

It takes four segments to terminate a connection:

1. One end calls close first by sending a FIN segment to mean it is finished sending data. This is called **active close**.
2. The other end that receives the FIN performs the **passive close**. The received FIN is acknowledged by TCP (sending an ACK segment). The receipt of the FIN is also passed to the application as an end-of-file.
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.

A FIN occupies one byte of sequence number space just like a SYN. Therefore, the ACK of each FIN is the sequence number of the FIN plus one.

TCP State Transition Diagram



TIME_WAIT State

The end that performs the active close goes through the TIME_WAIT state. The duration that this end point remains in the TIME_WAIT state is twice the **maximum segment lifetime** (MSL), sometimes called 2MSL, which is between 1 and 4 minutes.

The MSL is the maximum amount of time that any given IP datagram can live in a network. The IPv4 TTL field IPv6 hop limit field have a maximum value 255. The assumption is made that a packet with the maximum hop limit of 255 cannot exist in a network for more than MSL seconds.

TCP must handle **lost duplicates** (or **wandering duplicate**).

There are two reasons for the TIME_WAIT state:

❖To implement TCP's full-duplex connection termination reliably. If TCP is performing all the work necessary to terminate both directions of data flow cleanly for a connection (its full-duplex close), then it must correctly handle the loss of any of these four segments.

❖To allow old duplicate segments to expire in the network. When we successfully establish a TCP connection, all old duplicates from previous **incarnations** of the connection have expired in the network.

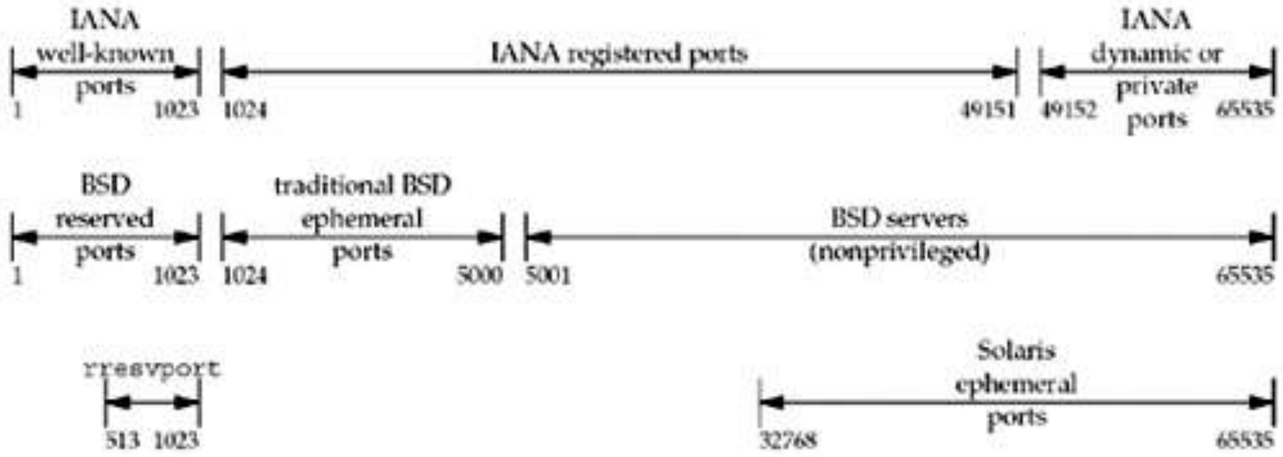
Port Numbers

All three transport layers (UDP, SCTP and TCP) use 16-bit integer port numbers to differentiate between processes.

The **well-known ports**: 0 through 1023.

The **registered ports**: 1024 through 49151

The **dynamic ports** or **private ports**, 49152 through 65535. Also called **ephemeral ports**.



The *Internet Assigned Numbers Authority (IANA) maintains a list of port number assignments.*

http://www.iana.org/

Socket Pair

❖**Socket pair**: the four-tuple that defines the two endpoints of a TCP connection: the local IP address, local port, foreign IP address, and foreign port. A socket pair uniquely identifies every TCP connection on a network.

❖**Socket**: two values (an IP address and a port number) that identify each endpoint.

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).

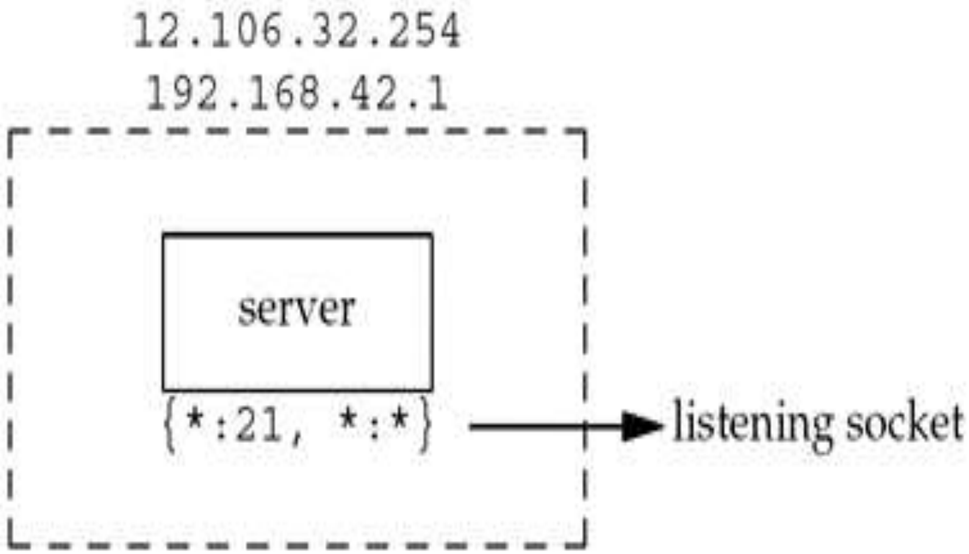


Figure 2.11. TCP server with a passive open on port 21.

A client starts on the host with IP address 206.168.112.219 and executes an active open to the server's IP address of 12.106.32.254. We assume the ephemeral port chosen by the client TCP is 1500 for this example.

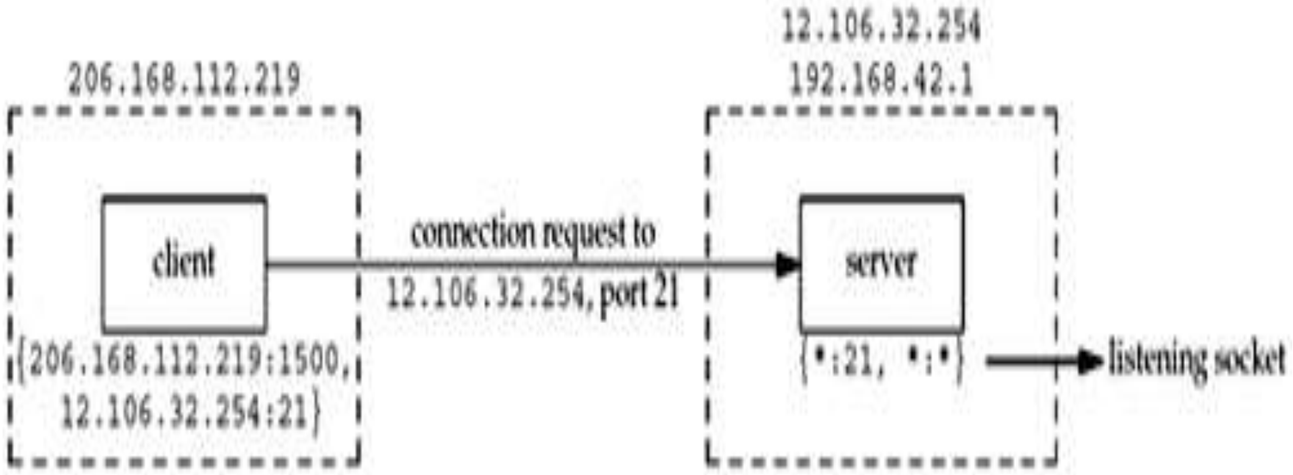


Figure 2.12. Connection request from client to server.

When the server receives and accepts the client's connection, it forks a copy of itself, letting the child handle the client

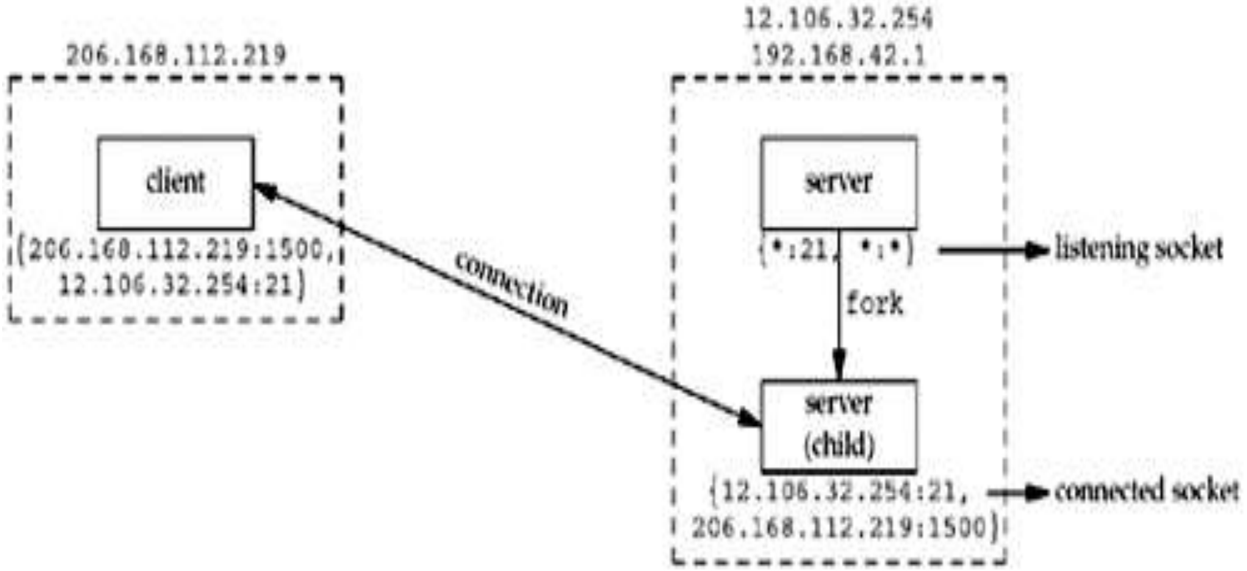


Figure 2.13. Concurrent server has child handle client.

The next step assumes that another client process on the client host requests a connection with the same server. The TCP code on the client host assigns the new client socket an unused ephemeral port number, say 1501.

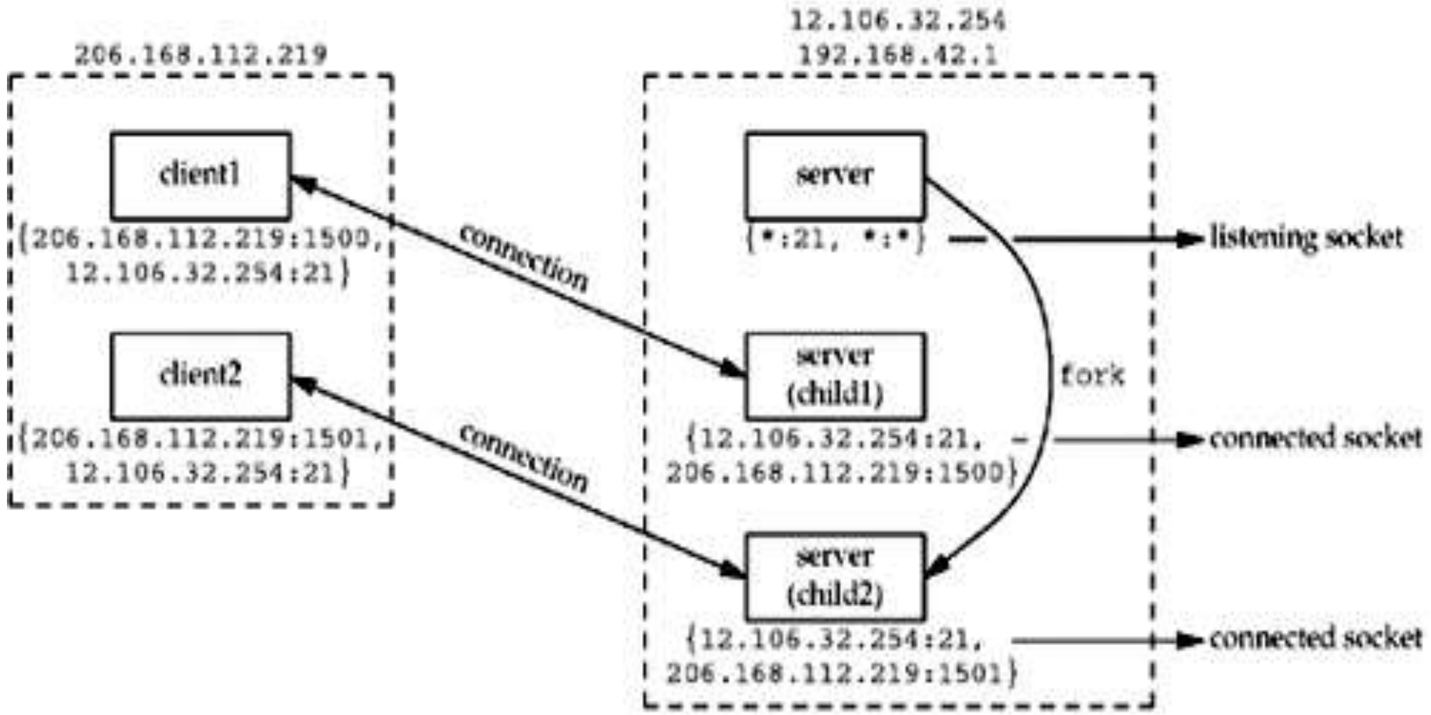
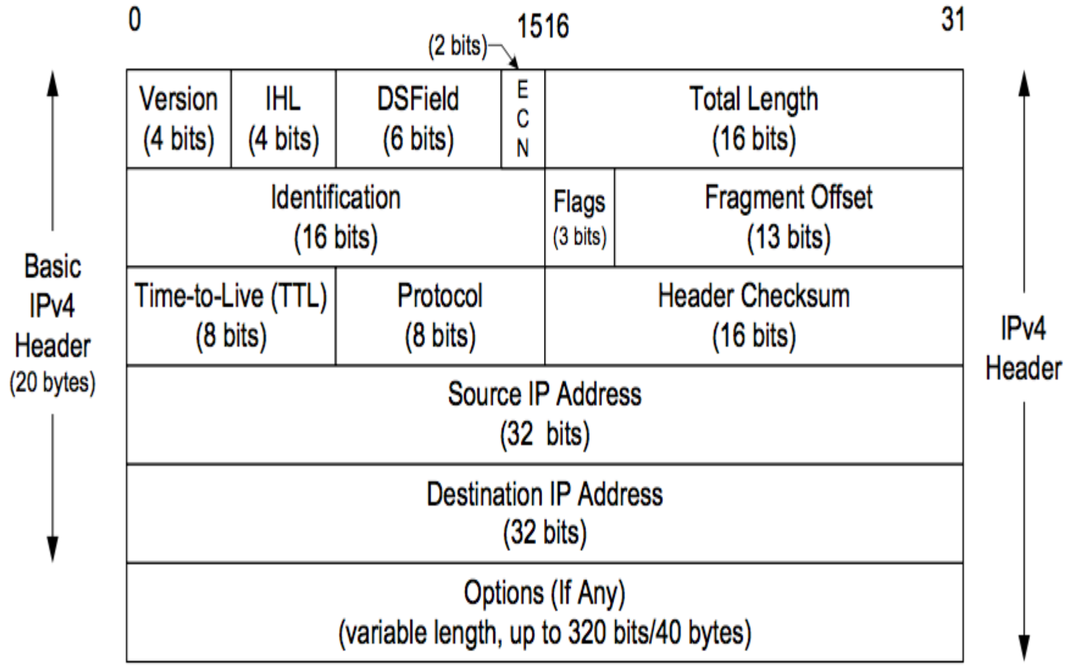


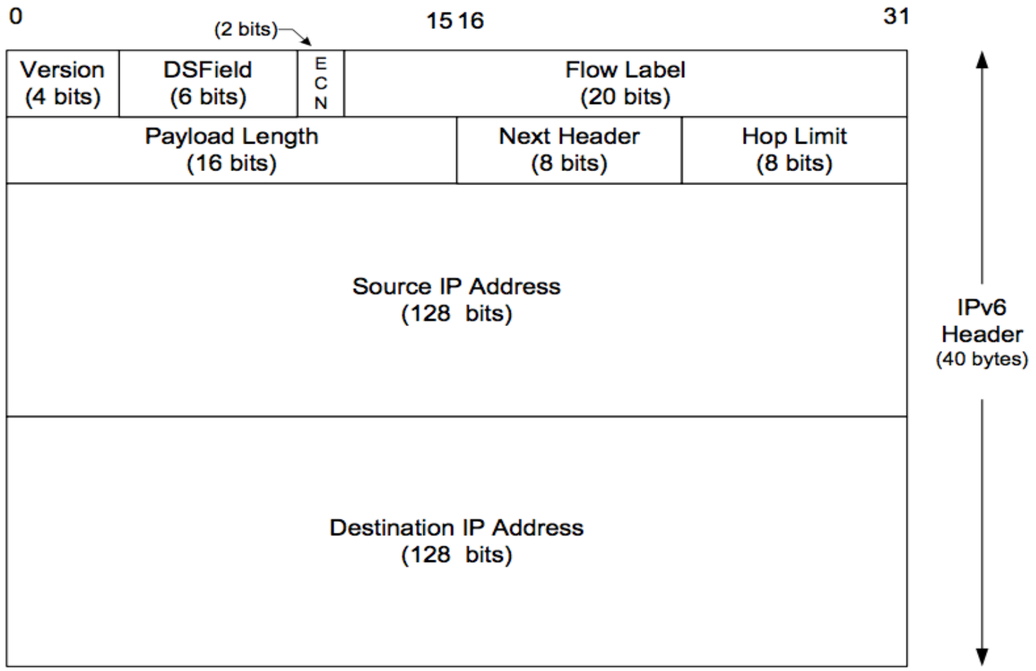
Figure 2.14. Second client connection with same server.

Buffer Sizes and Limitations

❖Maximum size of an IPv4 datagram: 65,535 bytes (including the header), because of the 16-bit total length field.

❖Maximum size of an IPv6 datagram: 65,575 bytes (including the 40-byte IPv6 header), because of the 16-bit payload length field. IPv6 has a jumbo payload option, which extends the payload length field to 32 bits, but this option is supported only on datalinks with a **maximum transmission unit** (MTU) that exceeds 65,535.





Buffer Sizes and Limitations

❖**MTU** (maximum transmission unit): dictated by the hardware. Ethernet MTU is 1,500 bytes; Point-to-point links have a configurable MTU.

- ❑Minimum link MTU for IPv4: 68 bytes. This permits a maximum-sized IPv4 header (20 bytes of fixed header, 40 bytes of options) and minimum-sized fragment (the fragment offset is in units of 8 bytes)

- ❑Minimum link MTU for IPv6: 1,280 bytes.

❖**Path MTU**: smallest MTU in the path between two hosts. Today, the Ethernet MTU of 1,500 bytes is often the path MTU. The path MTU need not be the same in both directions between any two hosts because routing in the Internet is often asymmetric.

Buffer Sizes and Limitations

❖**Fragmentation** is performed by both IPv4 and IPv6 when the size of an IP datagram to be sent out an interface exceeds the link MTU. The fragments are not normally **reassembled** until they reach the final destination.

- ❑IPv4: hosts perform fragmentation on datagrams that they generate and routers perform fragmentation on datagrams that they forward
- ❑IPv6: only hosts perform fragmentation on datagrams that they generate; routers do not fragment datagrams that they are forwarding
- ❑IPv4 header contains fields to handle fragmentation. IPv6 contains an option header with the fragmentation information.

❖"Don't Fragment" (DF) bit in IPv4 header specifies that this datagram must not be fragmented, either by the sending host or by any router. A router that receives an IPv4 datagram with the DF bit set whose size exceeds the outgoing link's MTU generates an ICMPv4 "destination unreachable, fragmentation needed but DF bit set" error message.

- ❑Since IPv6 routers do not perform fragmentation, there is an implied DF bit with every IPv6 datagram. When an IPv6 router receives a datagram whose size exceeds the outgoing link's MTU, it generates an ICMPv6 "packet too big" error message
- ❑**Path MTU discovery** uses IPv4 DF bit and its implied IPv6 counterpart. Path MTU discovery is optional with IPv4, but IPv6 implementations all either support path MTU discovery or always send using the minimum MTU.

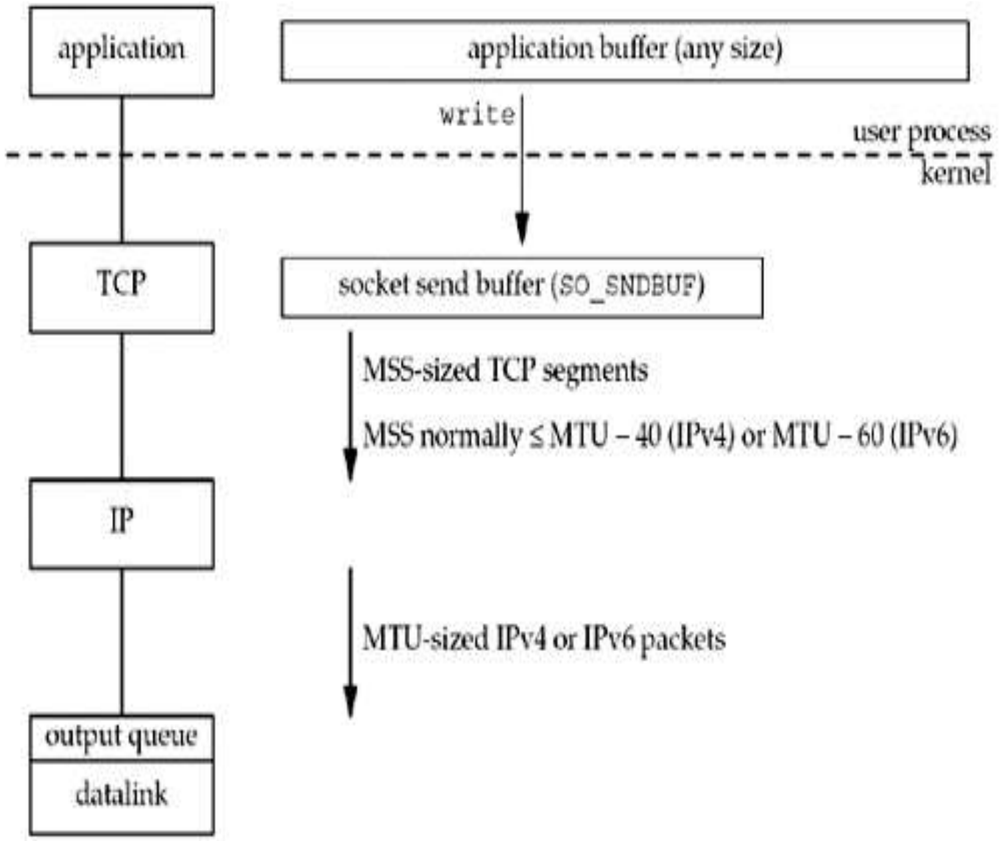
❖**Minimum reassembly buffer size**: the minimum datagram size that we are guaranteed any implementation must support.

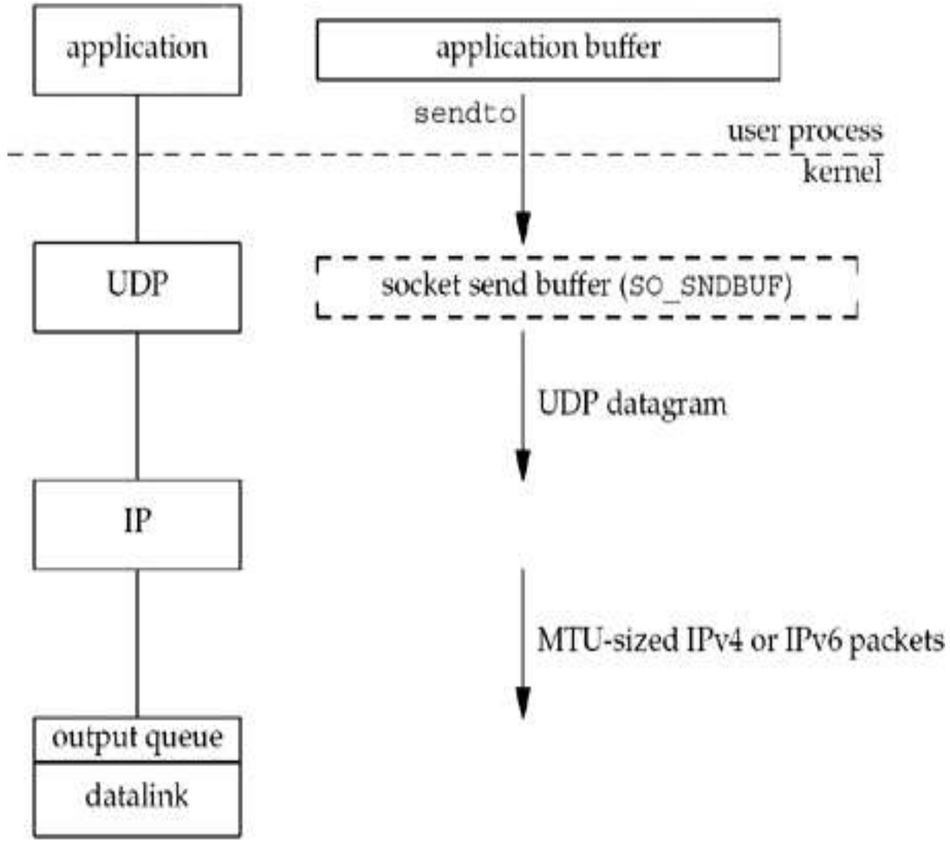
- ❑IPv4: 576 bytes. We have no idea whether a given destination can accept a 577-byte datagram or not. Therefore, many IPv4 applications that use UDP (e.g., DNS, RIP, TFTP, BOOTP, SNMP) prevent applications from generating IP datagrams that exceed this size.
- ❑IPv6: 1,500 bytes

❖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. We saw the MSS option on the SYN segments in Figure 2.5. The goal of the MSS is to tell the peer the actual value of the reassembly buffer size and to try to avoid fragmentation. The MSS is often set to the interface MTU minus the fixed sizes of the IP and TCP headers. On an Ethernet using IPv4, this would be 1,460, and on an Ethernet using IPv6, this would be 1,440. (The TCP header is 20 bytes for both, but the IPv4 header is 20 bytes and the IPv6 header is 40 bytes.)

- ❑IPv4: The MSS value in the TCP MSS option is a 16-bit field, limiting the value to 65,535. The maximum amount of TCP data in an IPv4 datagram is 65,495 (65,535 minus the 20-byte IPv4 header and minus the 20-byte TCP header).
- ❑IPv6: the maximum amount of TCP data in an IPv6 datagram without the jumbo payload option is 65,515 (65,535 minus the 20-byte TCP header). The MSS value of 65,535 is considered a special case that designates "infinity." This value is used only if the jumbo payload option is being used, which requires an MTU that exceeds 65,535.

TCP and UDP Output





RV College of Engineering®

Go, change the world

NETWORK PROGRAMMING AND SECURITY (Theory & Lab)

18CS54

UNIT 1

The Transport Layer and introduction to sockets

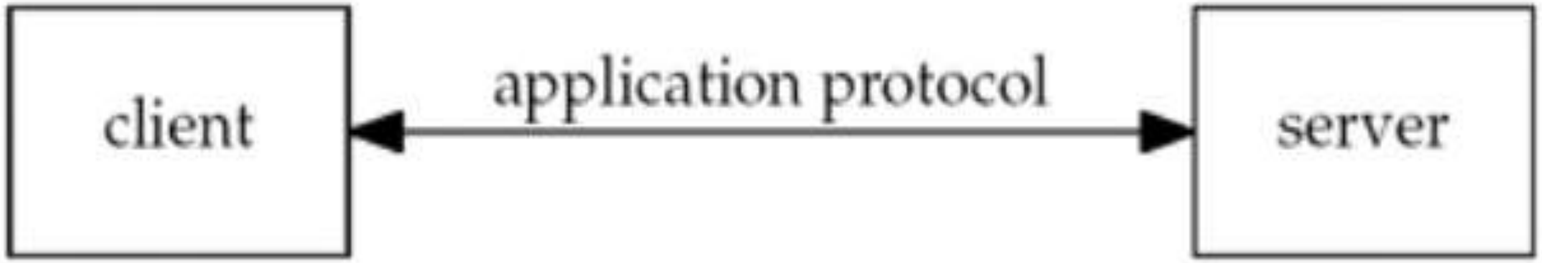
Contents

1. Introduction to all protocols
2. The big Picture
3. Difference between TCP, UDP and SCTP
4. TCP Connection Establishment and Termination and TIME_WAIT state
5. TCP port numbers and concurrent servers, Buffer sizes and limitation

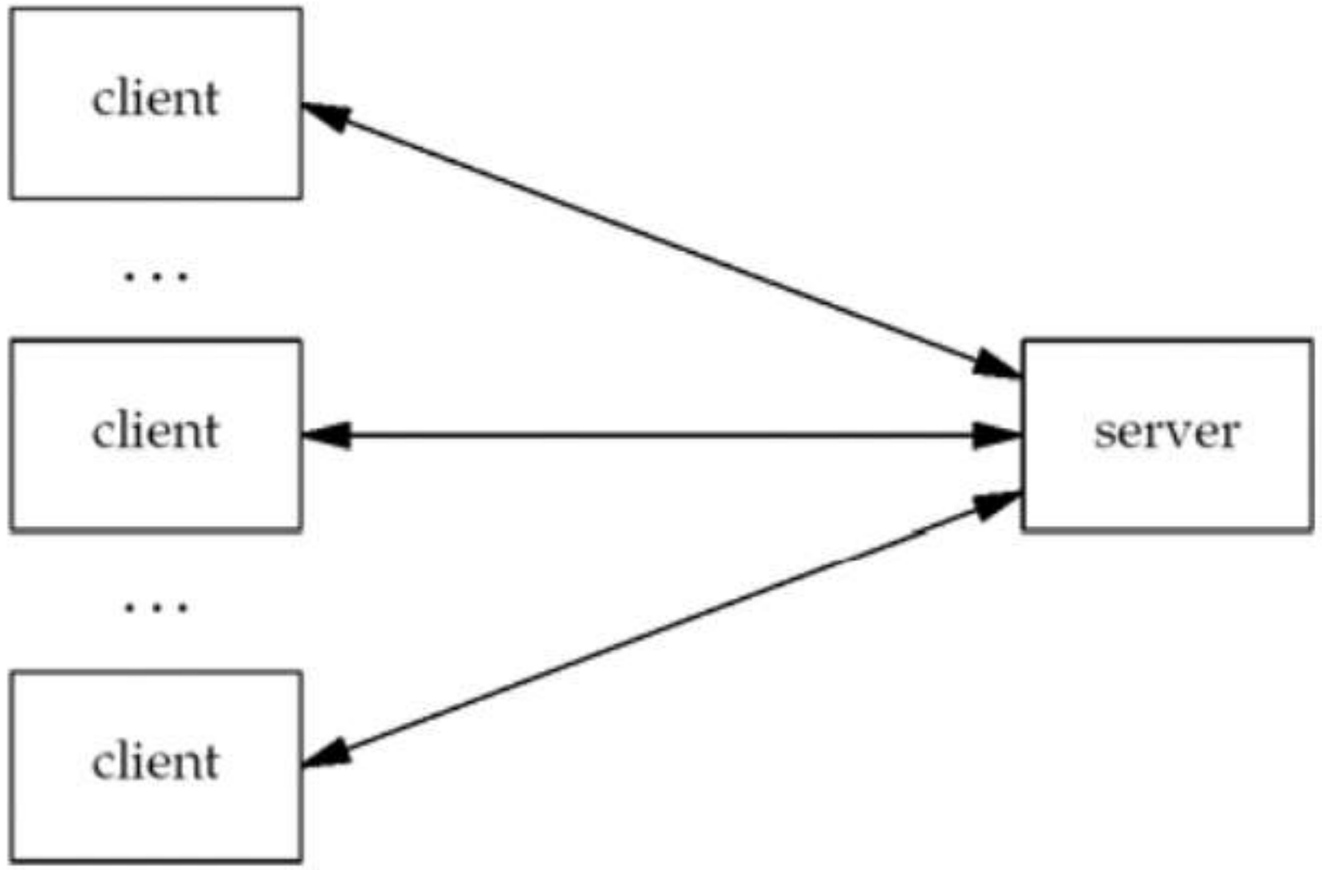
Introduction to all protocols

- **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

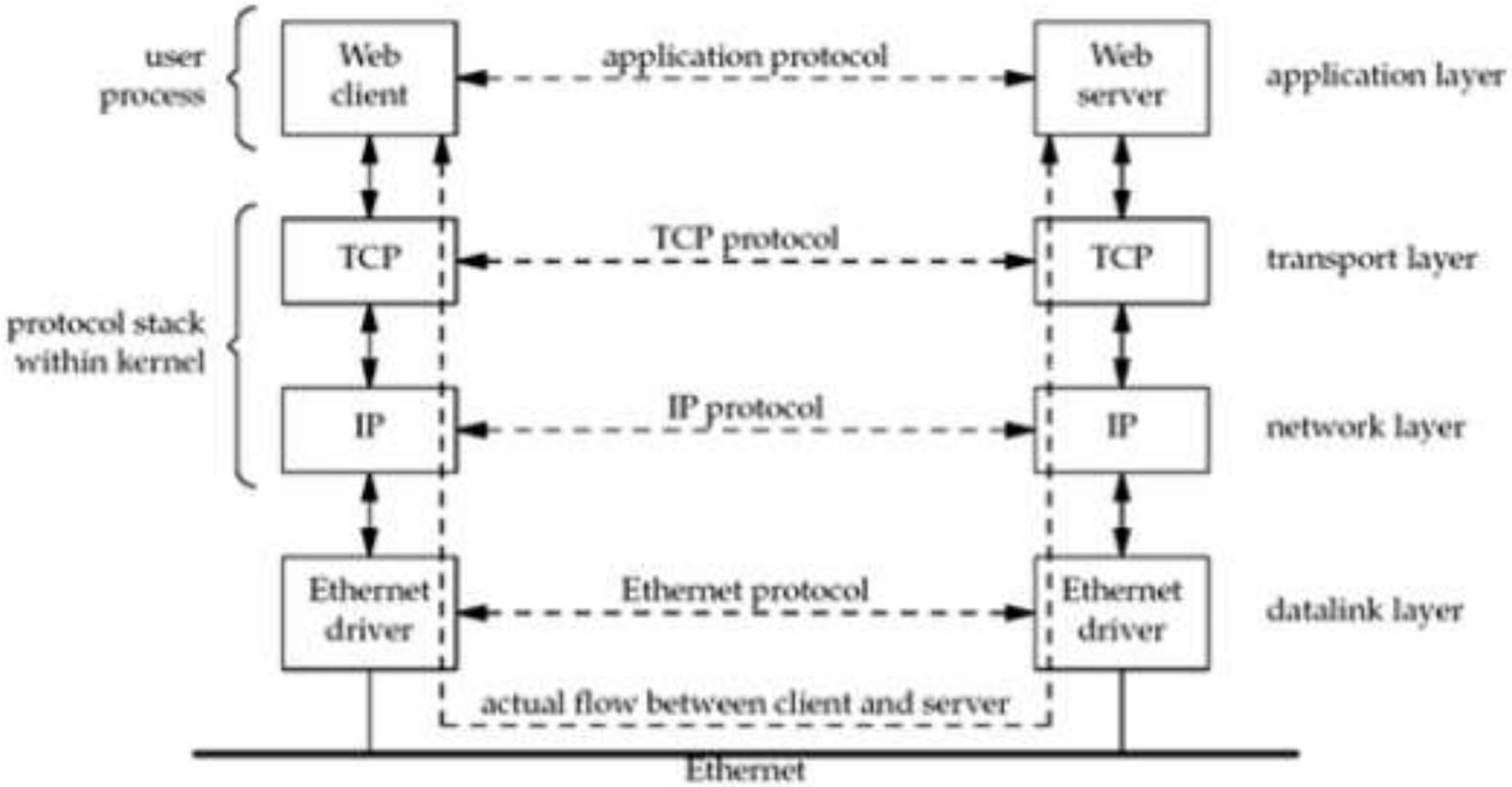
Network application: client and server.



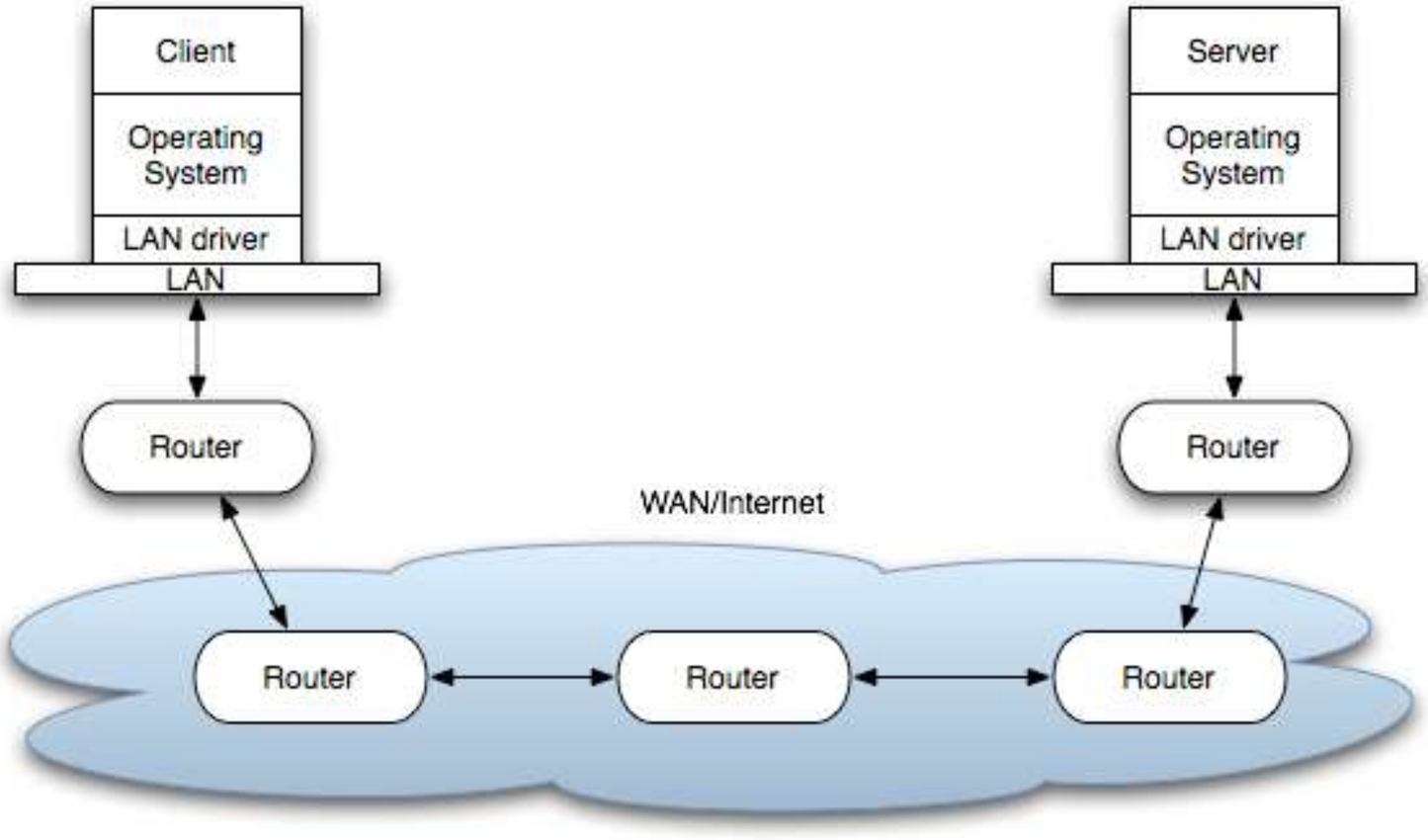
•Client communicates with only one server where as one server communicates with many clients at the same tie as shown in the figure below:



- 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 following



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



Few Questions??

1) When do you use a switch and when do you use a router?

Router and **Switch** are both network connecting devices. **Router** works at network layer and **is** responsibe to find the shortest path for a packet whereas **Switch** connects various devices **in a** network. **Router** connects devices across multiple networks.

2) The largest WAN is ??

the internet

3) What is a socket and why it is required?

A **socket** is one endpoint of a **two way** communication link between two programs running on the network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication take place.

4) What is a port? Define its size and usage.

A port number is a way to identify a specific process to which an Internet or other network message is to be forwarded when it arrives at a server. For the Transmission Control Protocol and the User Datagram Protocol, a port number is a 16-bit integer that is put in the header appended to a message unit. This port number is passed logically between client and server transport layers and physically between the transport layer and the Internet Protocol layer and forwarded on.

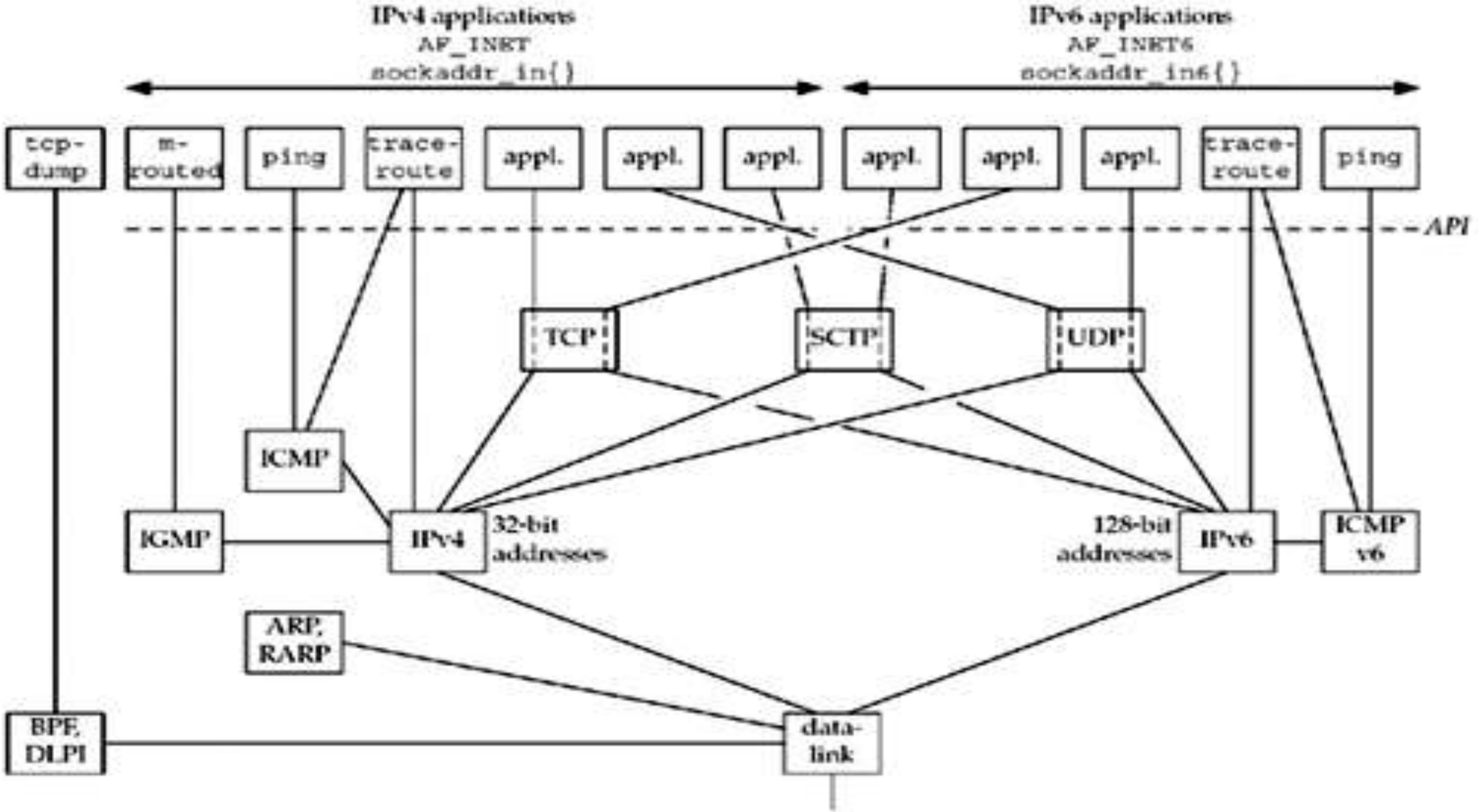
A Simple Daytime Client

```
 1 #include  "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd, n;
 6     char    recvline[MAXLINE + 1];
 7     struct sockaddr_in servaddr;

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

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

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

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

19     while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
20         recvline[n] = 0;        /* null terminate */
21         if (fputs(recvline, stdout) == EOF)
22             err_sys("fputs error");
23     }
24     if (n < 0)
25         err_sys("read error");

26     exit(0);
27 }
```

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.

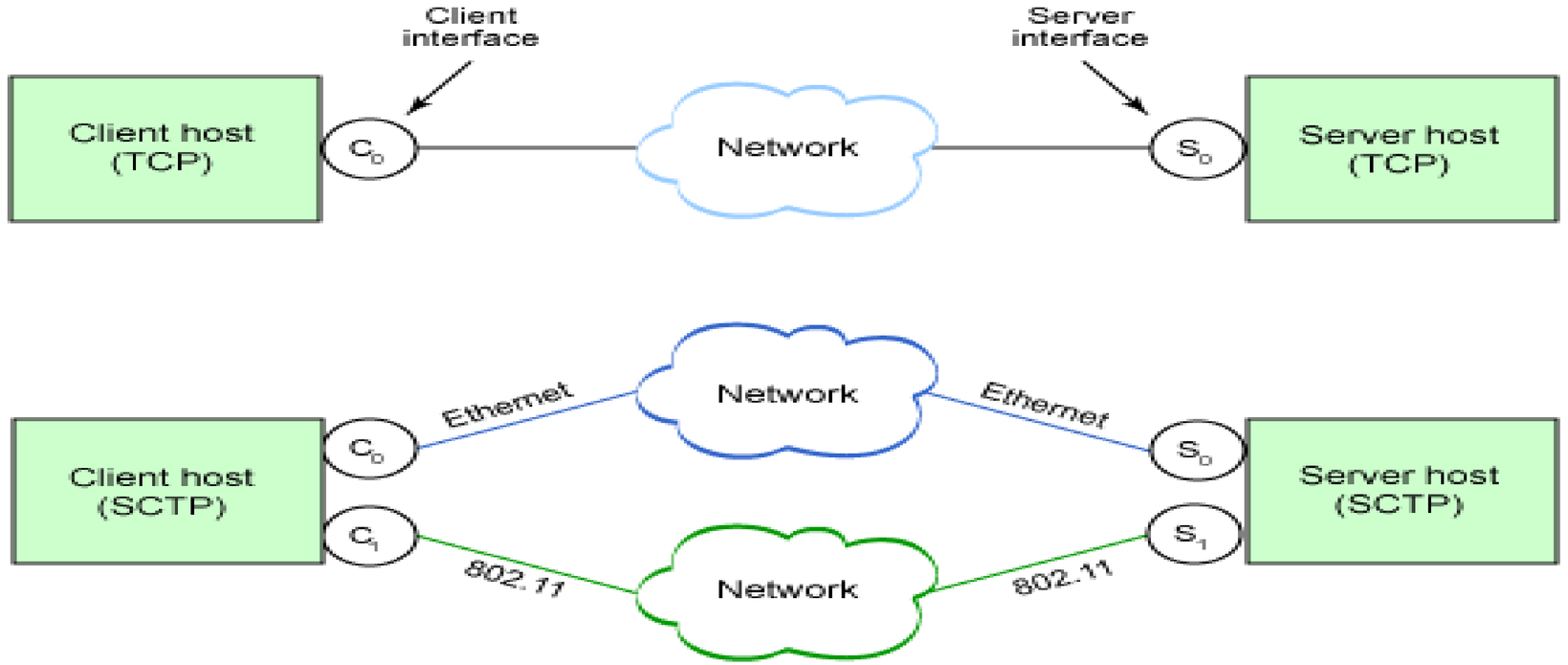
Features of Transmission Control Protocol (TCP):

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

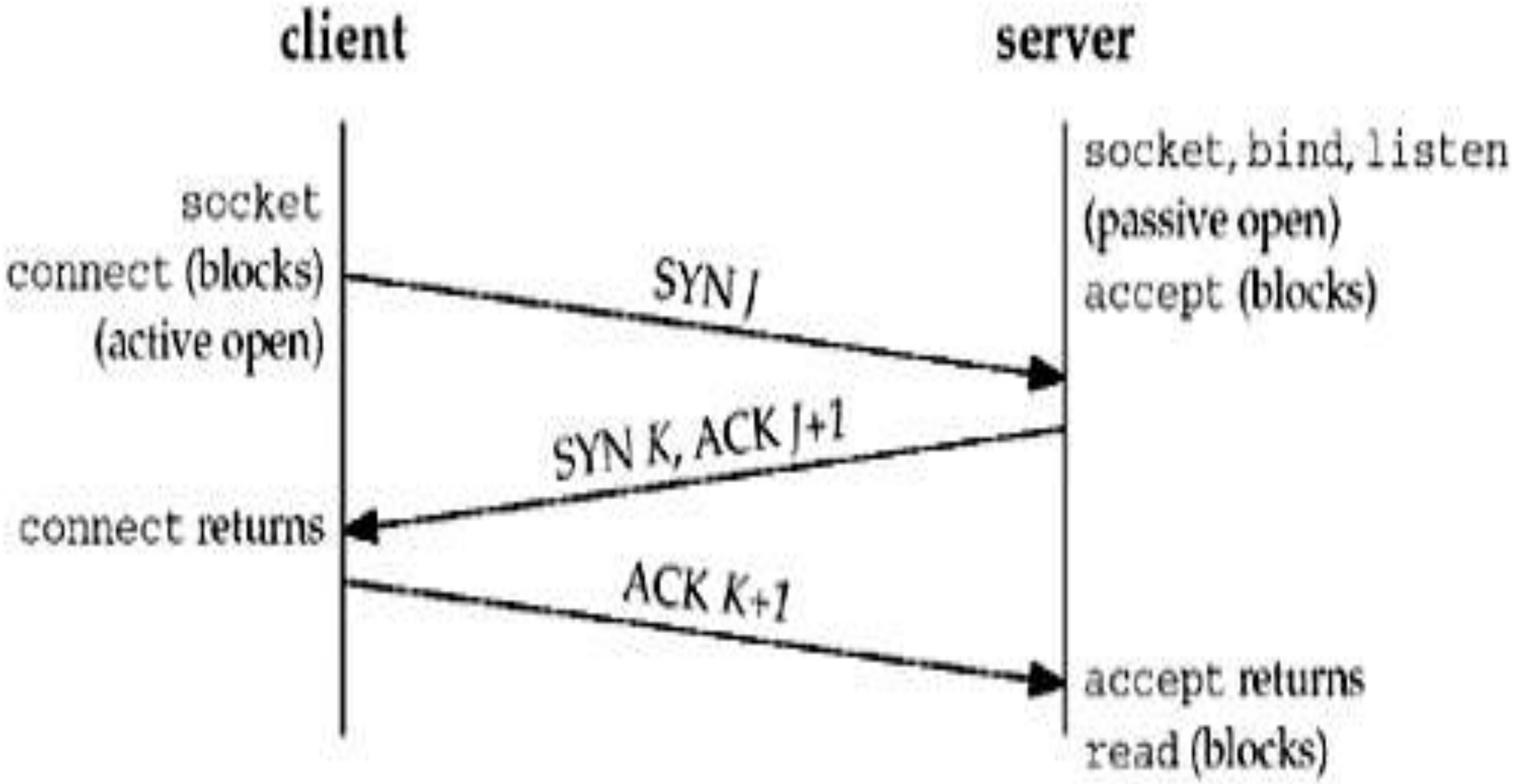
Three-Way Handshake:

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

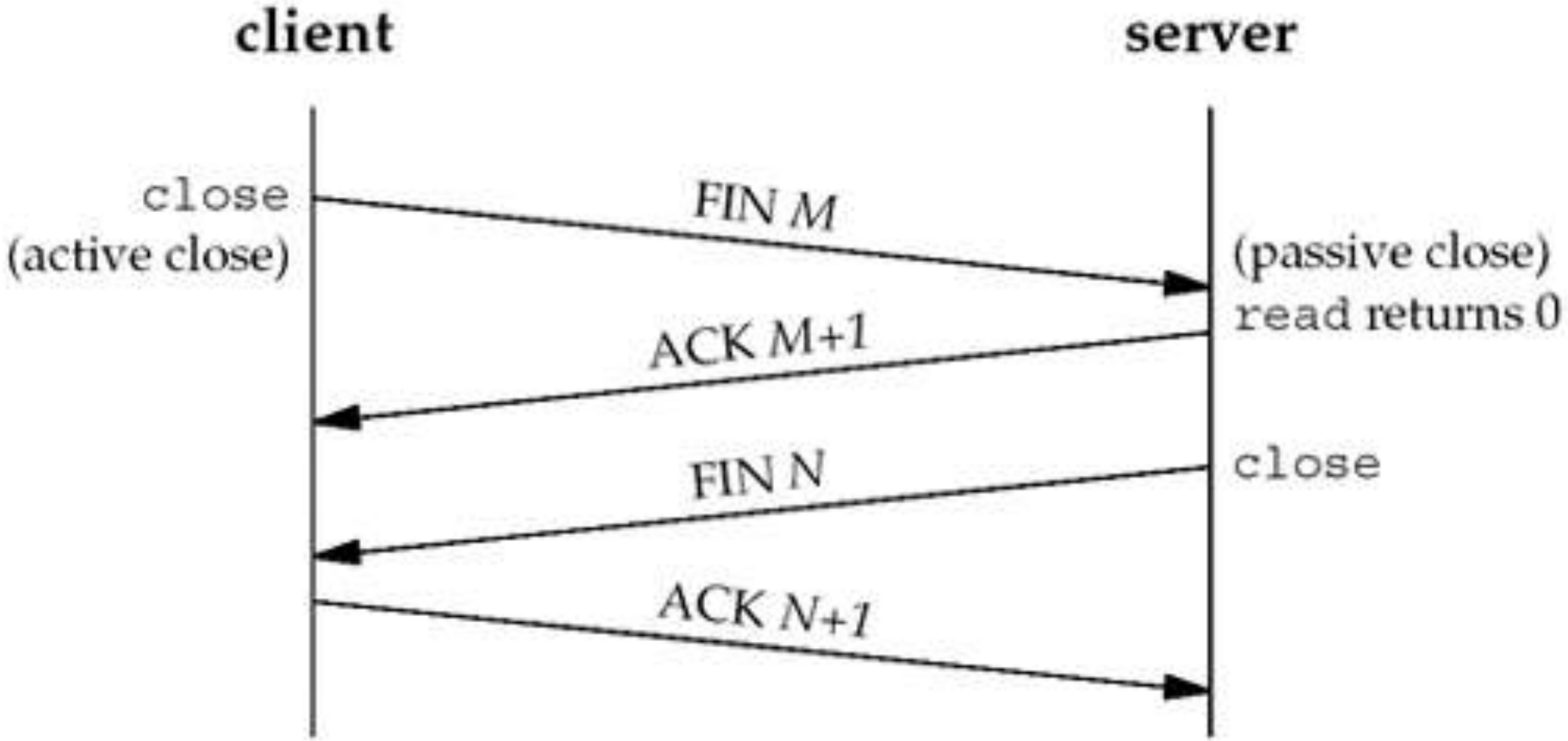
Four-Way Handshake:

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

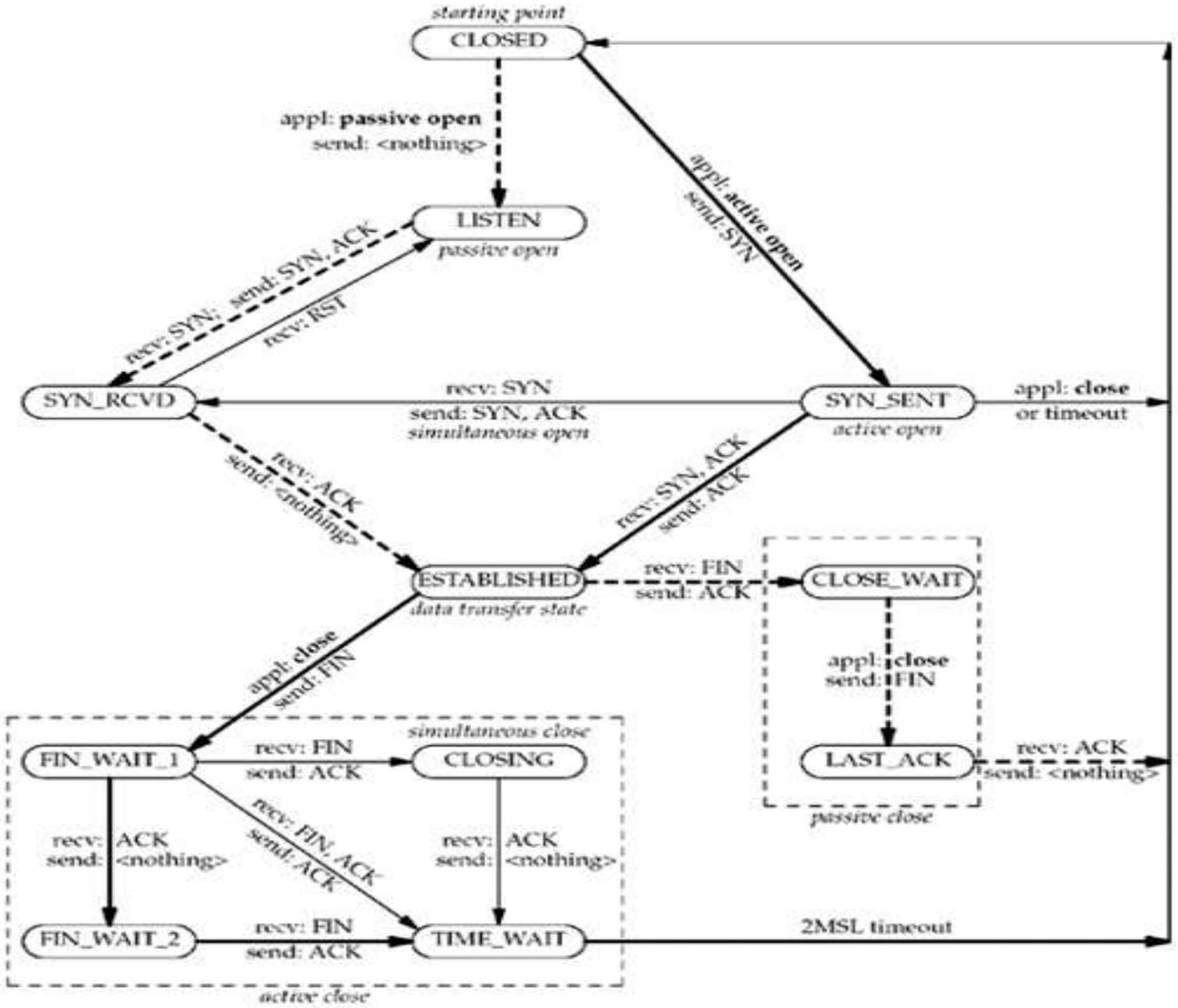
- - → Indicate Normal Transition for Server

Indicate State Transition taken when

appl: application issues operation

recv: Indicate State Transition when segment is received

send: Indicate what is sent for this transition



TCP State:

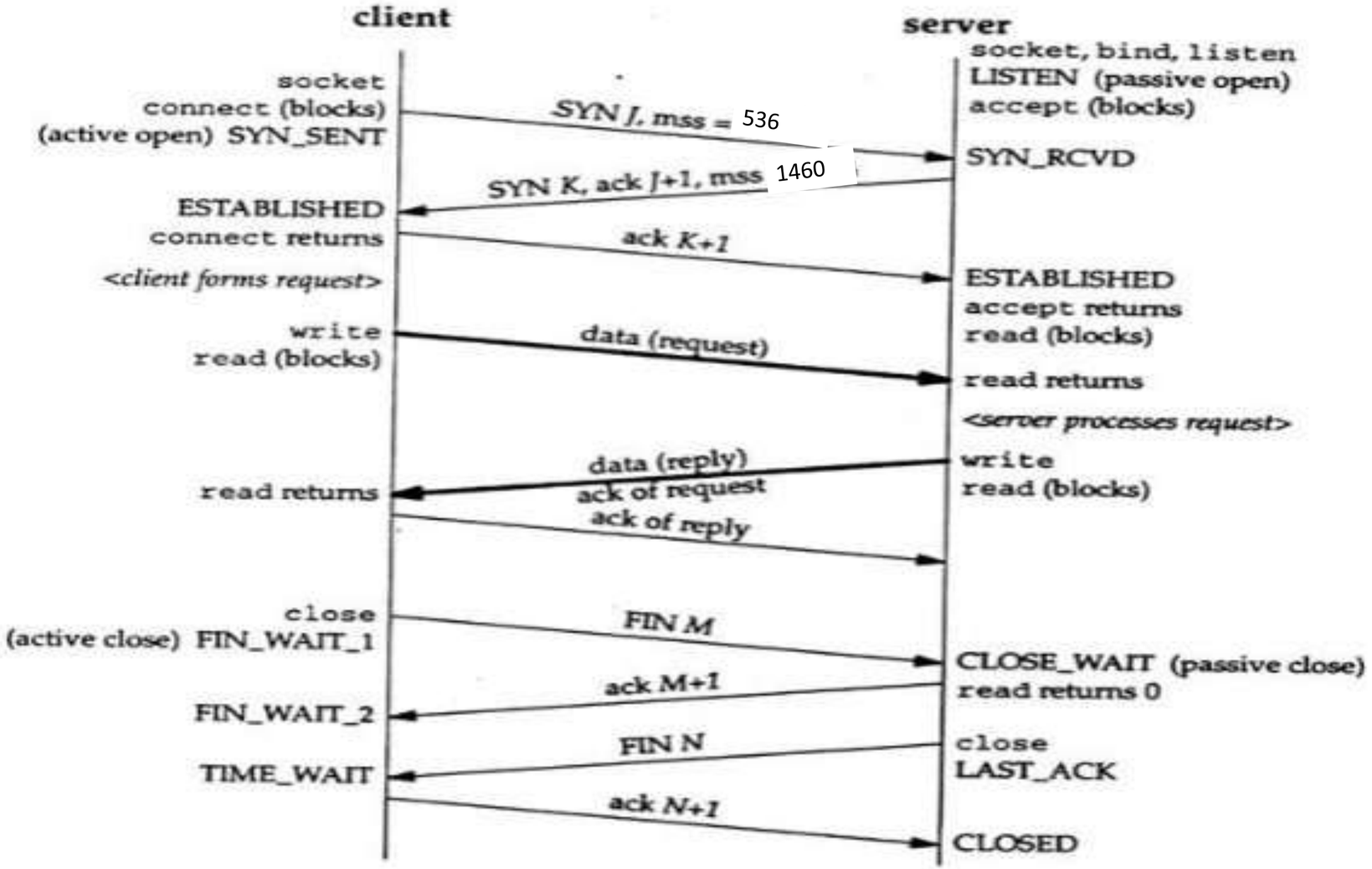
- There are 11 different states defined for a connection and the rules of TCP dictate the transitions from one state to another, based on the current state and the segment received in that state.

- 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

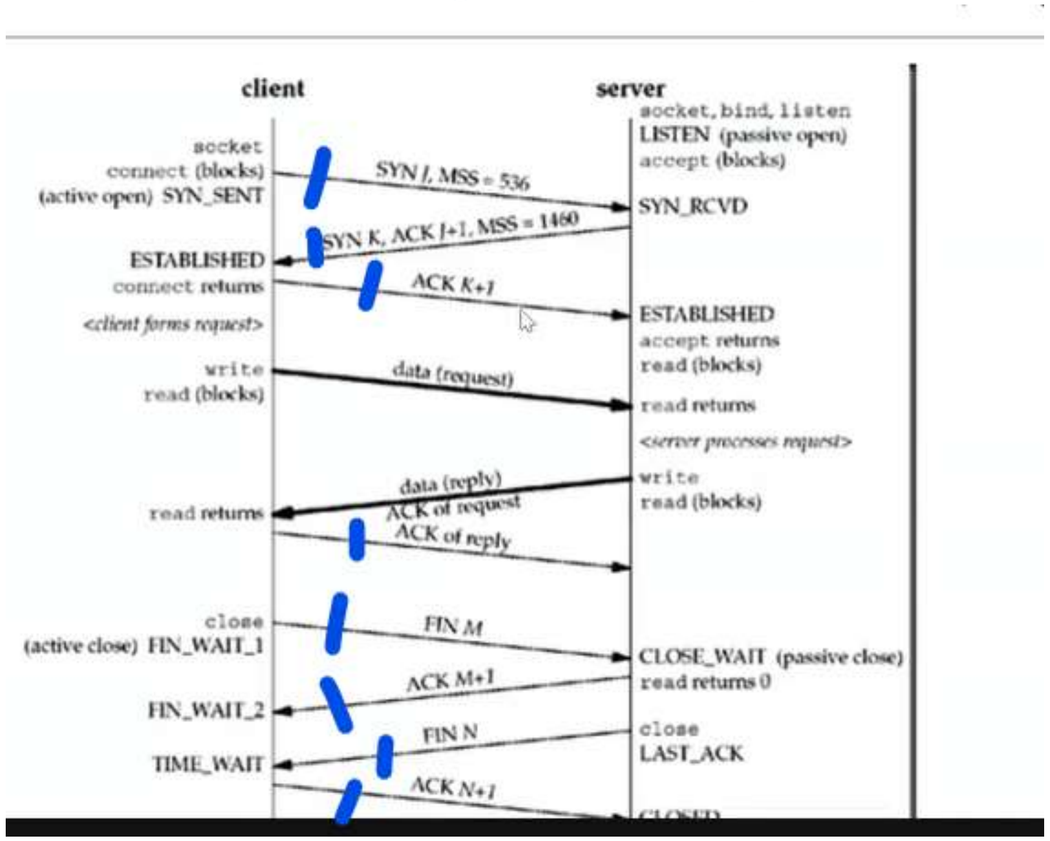


Watching the Packets :

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.

In TCP, for 1 request and reply, it tales 8 segments of overhead as shown :



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.

Netstat -n

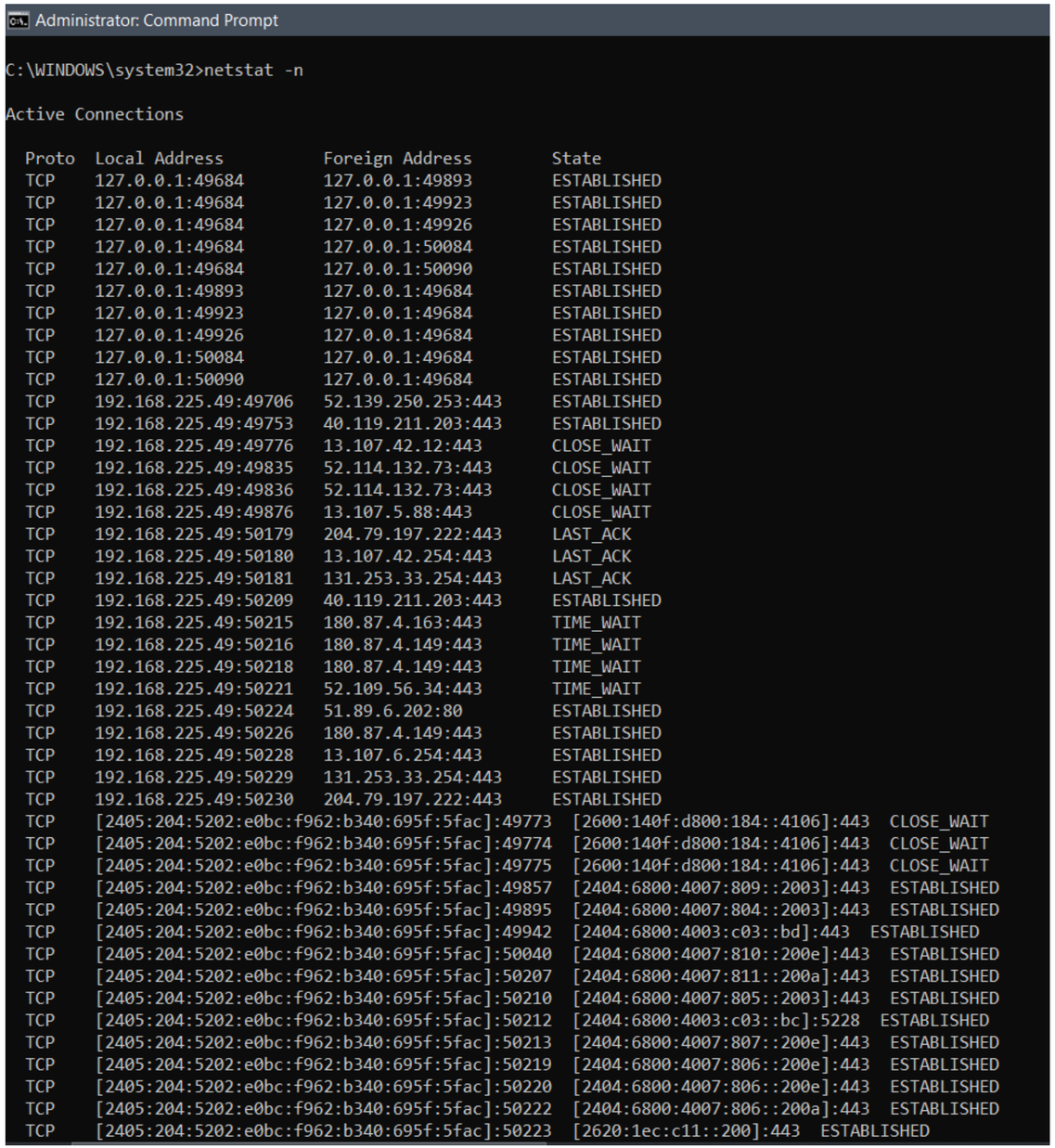
Local address is our computer's address. And port is source port.

Foreign address is the IP address of device you are connecting to.

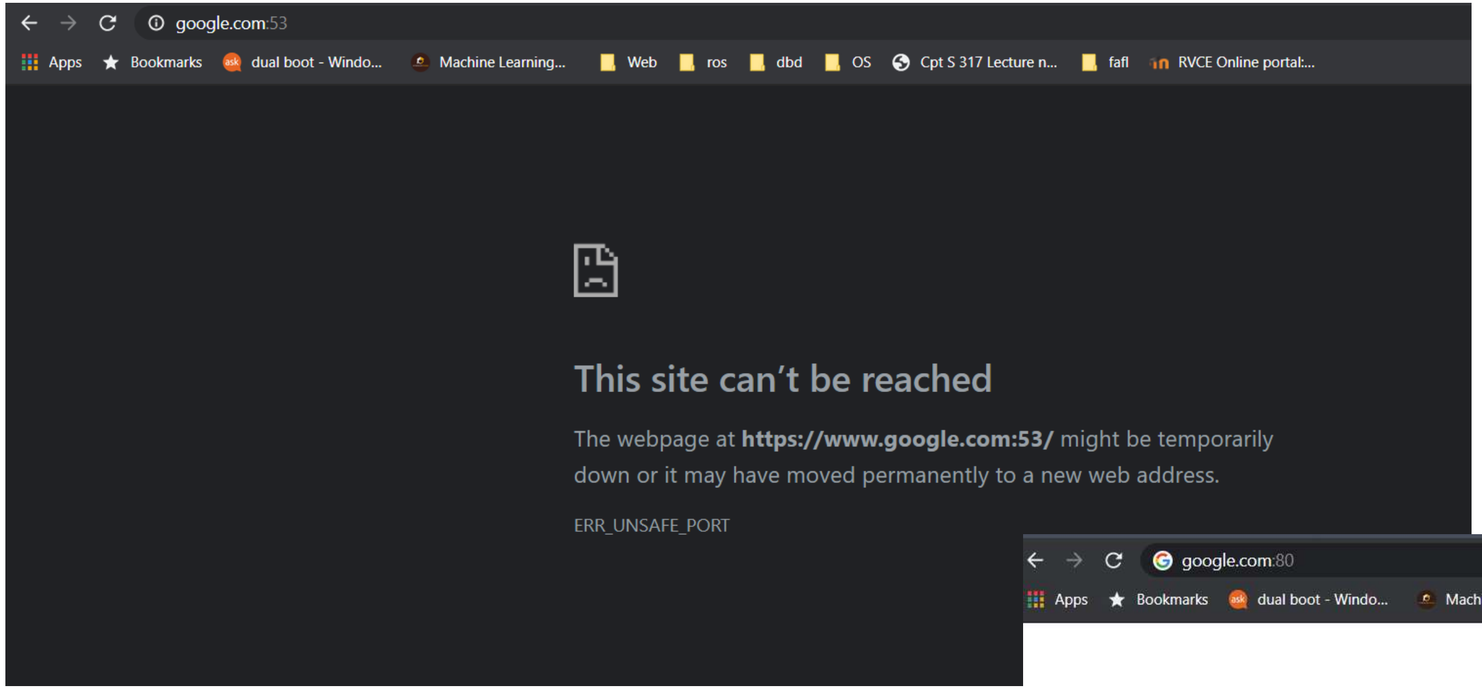
Port numbers:

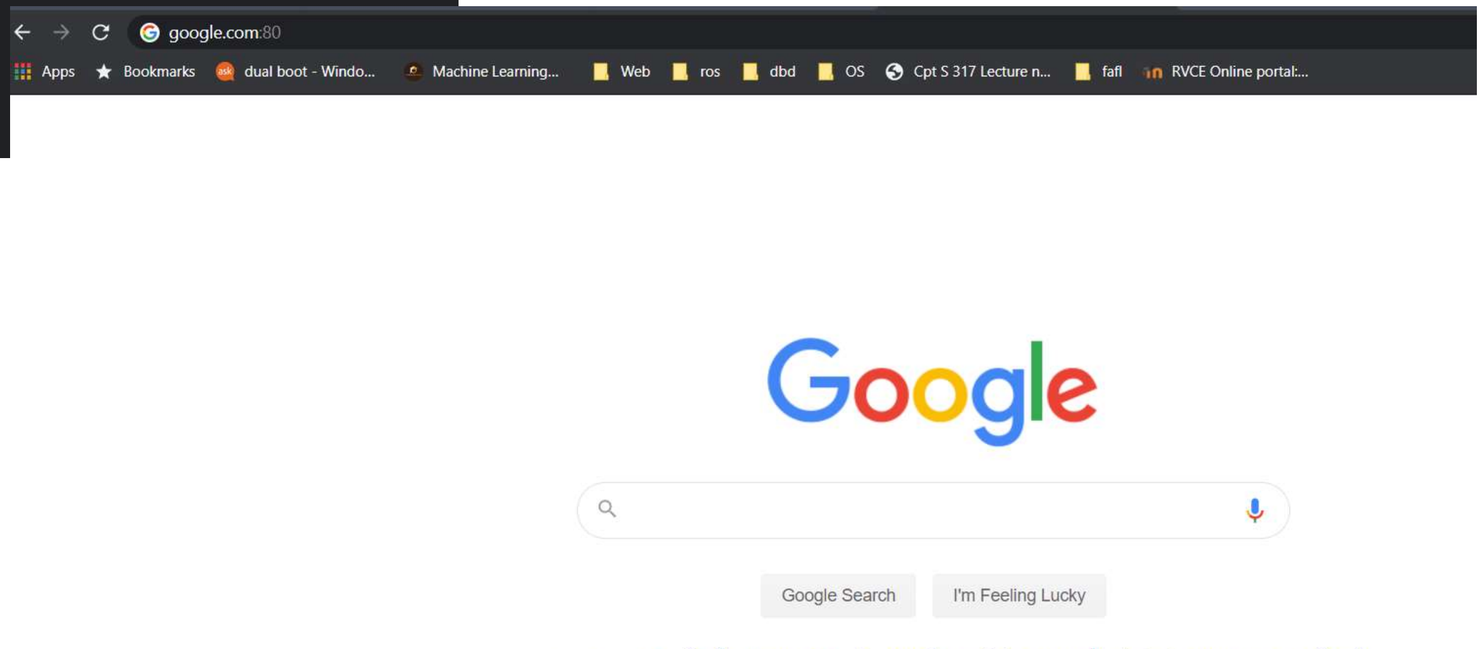
:80 - HTTP

:443 - HTTPS



Access websites using port number



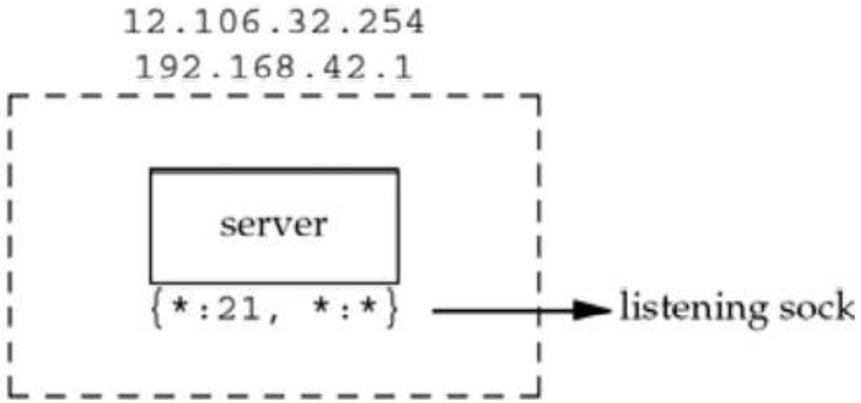


TCP Port Numbers and Concurrent Servers

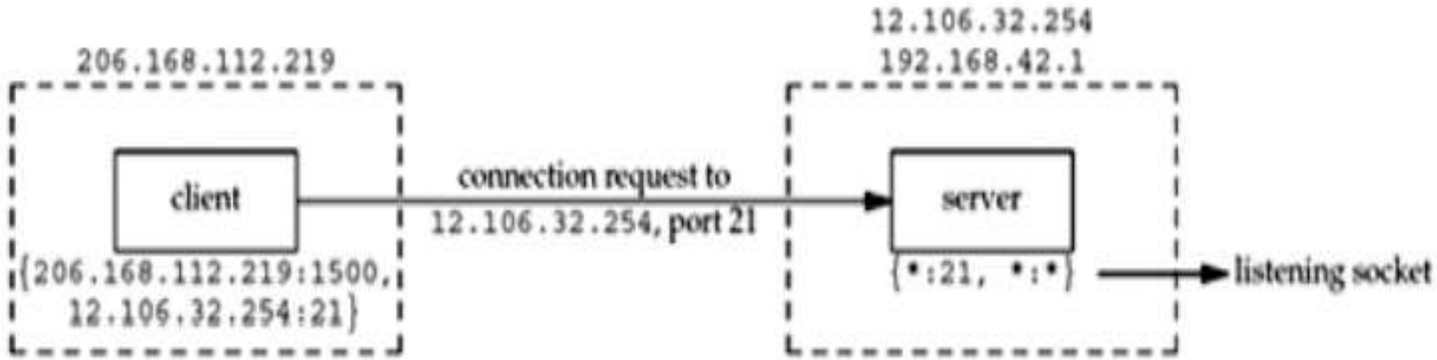
With a concurrent server, where the main server loop spawns a child to handle each new connection, what happens if the child continues to use the well-known port number while servicing a long request?

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

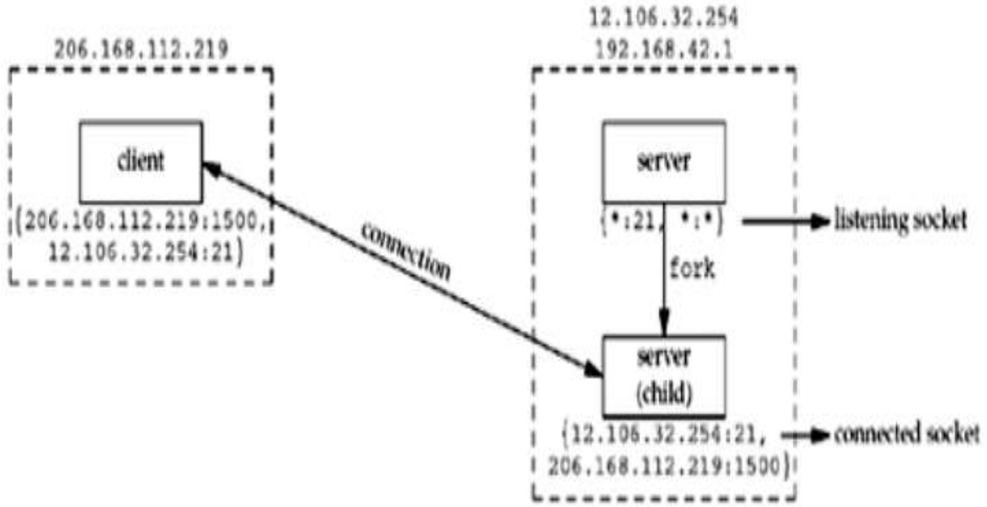


2. Connection request from client to server

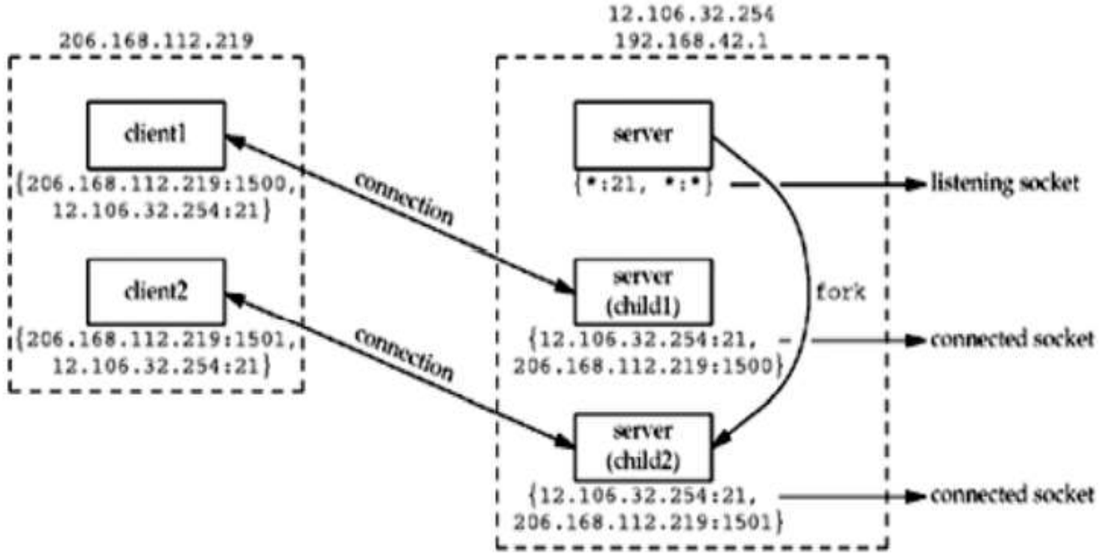


TCP Port Numbers and Concurrent Servers

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

Contents

1. Socket Address structure.
2. Value result arguments, byte ordering functions.
3. Byte manipulation functions, inet_aton, inet_addr and inet_ntoa functions, inet_pton and inet_ntop functions.
4. Socket function, connect function
5. bind, listen
6. accept, fork, exec functions.

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

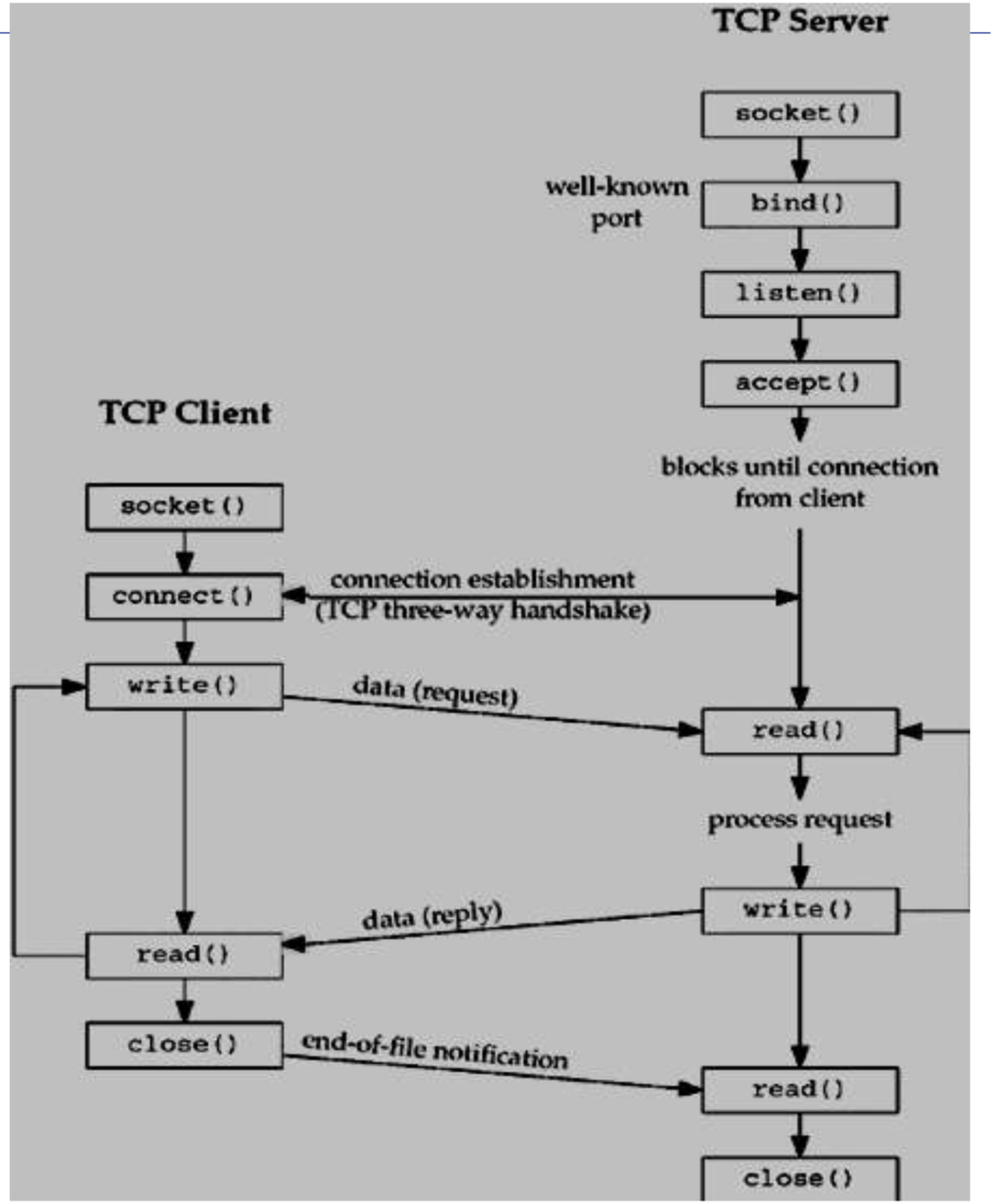
1. The client knows the hostname of the machine on which the server is running and the port number on which the server is listening.
2. To make a connection request, the client tries to rendezvous with the server on the server's machine and port.
3. The client also needs to identify itself to the server 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

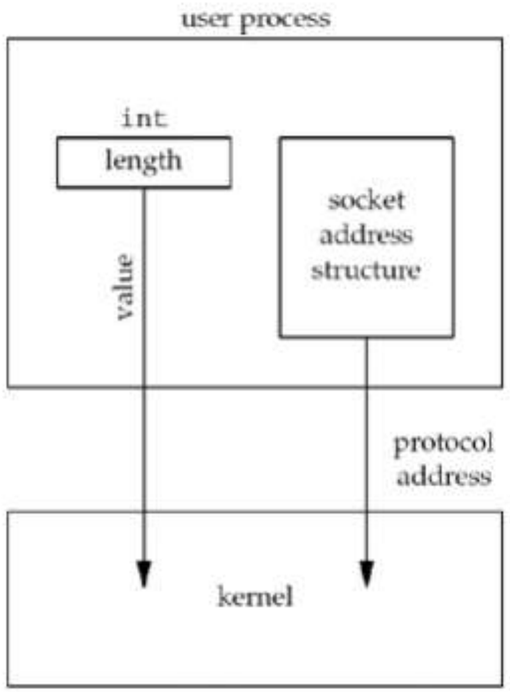
- When a socket address structure is passed to any socket function, it is always passed by reference- A pointer to the structure is passed.
- The length of the structure is also passed as an argument.
- But the way in which the length is passed depends on which direction the structure is being passed: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

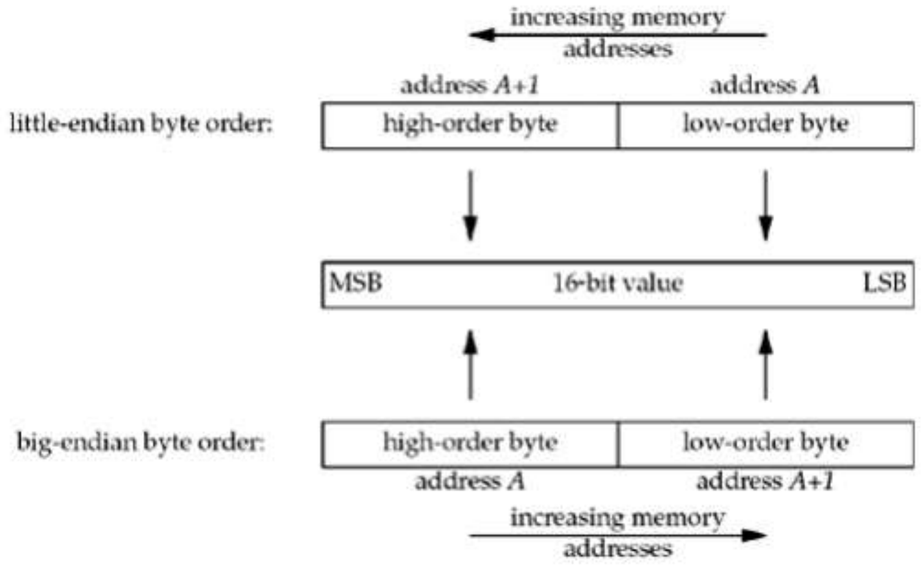
- Four functions, **accept, recvfrom, getsockname, and getpeername**, pass a socket address structure from the kernel to the process, the reverse direction from the previous scenario.
- Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to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

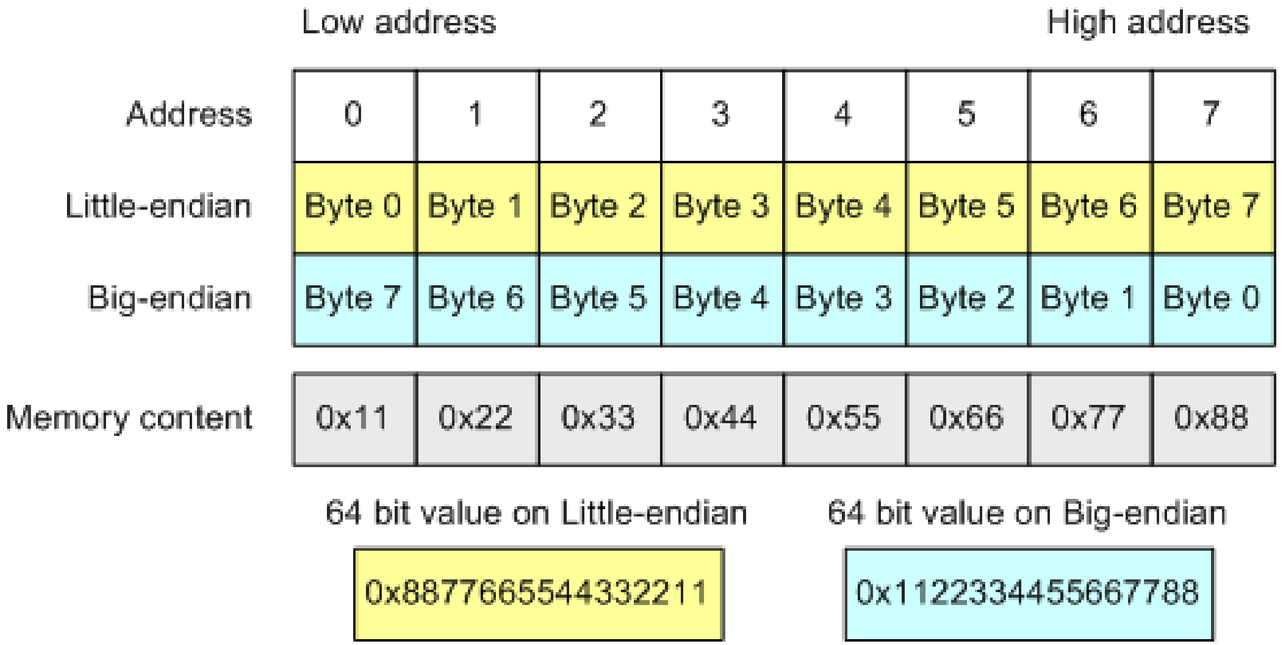
```
struct sockaddr_un  cli;    /* Unix domain */
socklen_t  len;

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).

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 */
```

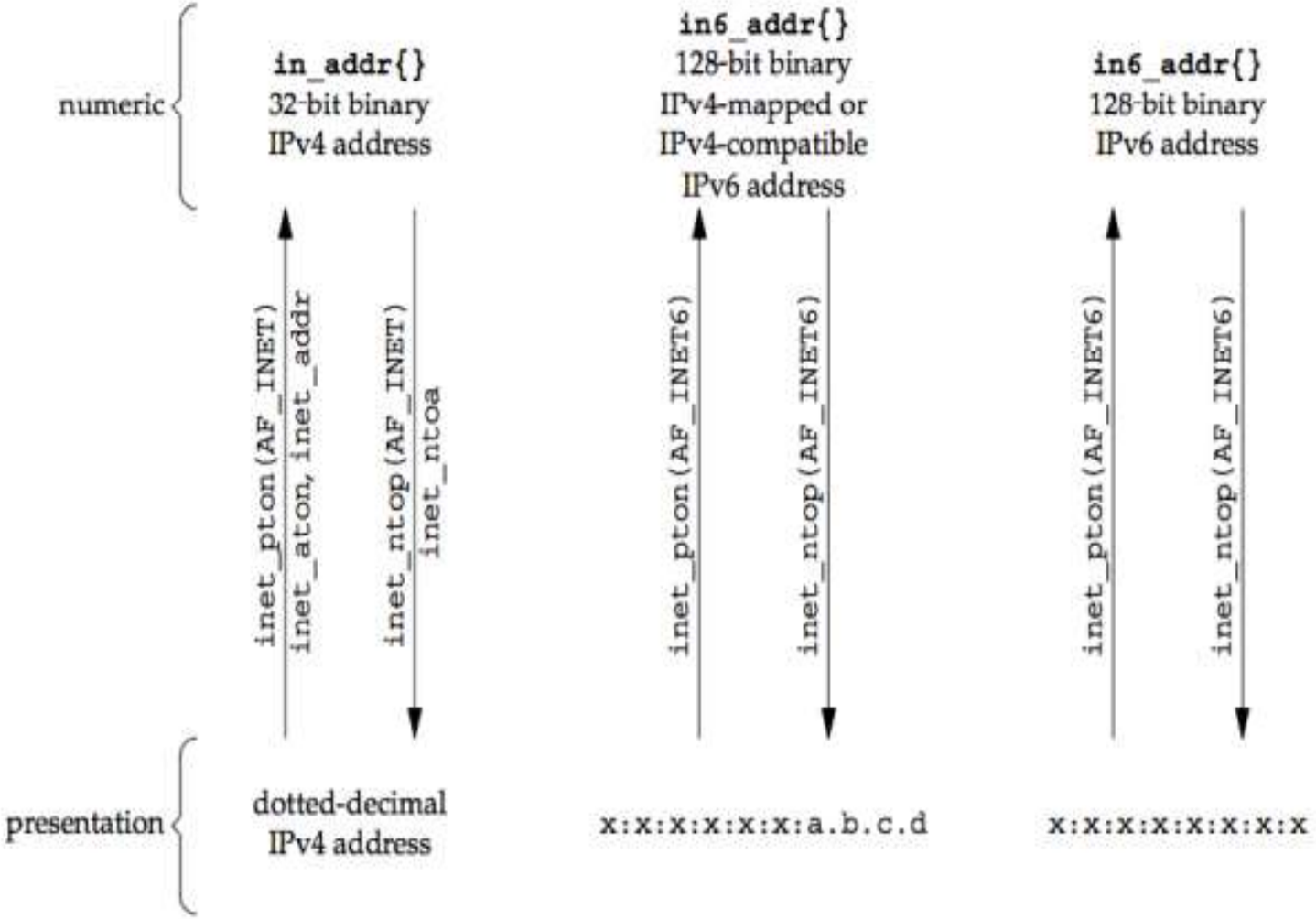
Conversion Functions -3

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 */
```

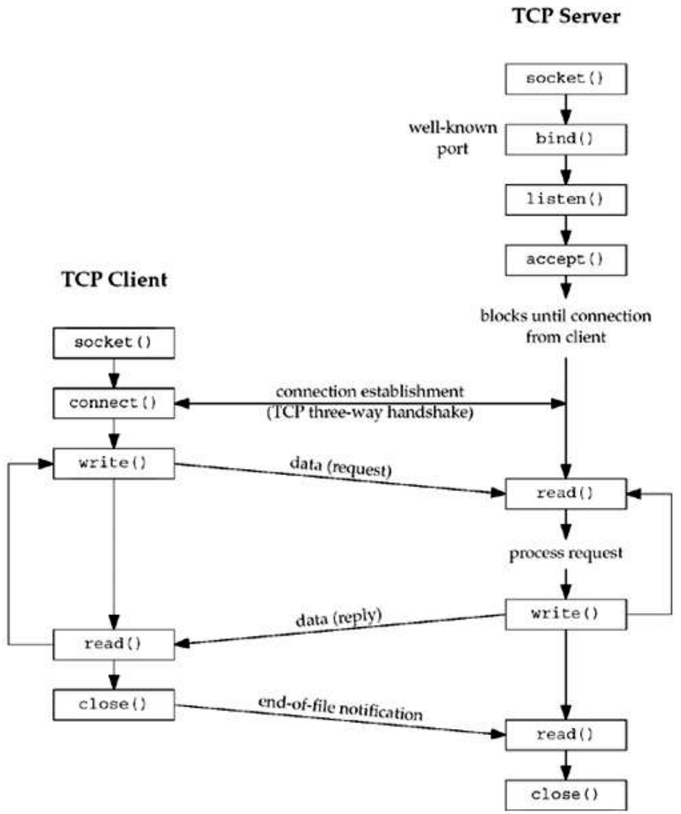
Conversion Functions



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 addres
- The third argument *addrlen* is the size of this address structure.

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

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.

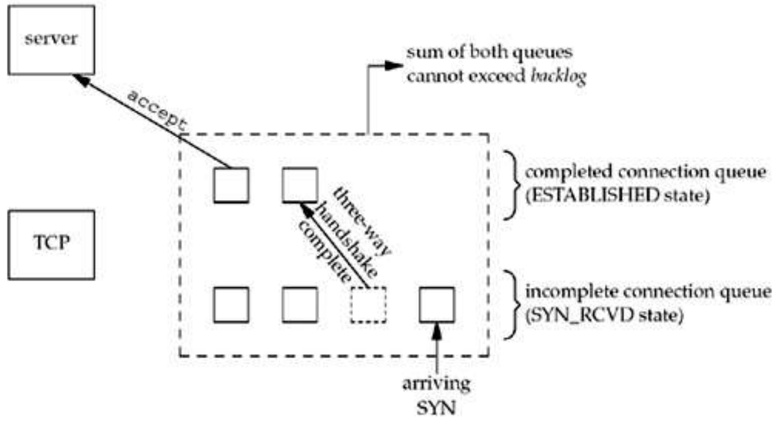
❖ This function is normally called after both the socket and bind functions and must be called before calling the accept function.

listen()

Connection queues *- To understand the backlog argument

- The listening socket, the kernel maintains two queues:

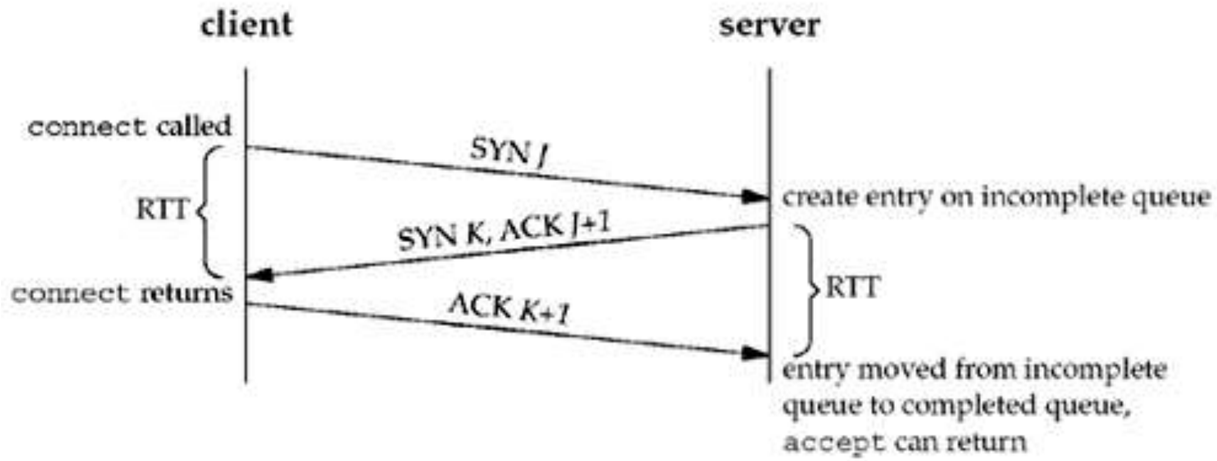
1. An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state.
2. A completed connection queue, which contains an entry for each client with whom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



listen()

Packet exchanges during connection establishment

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



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.

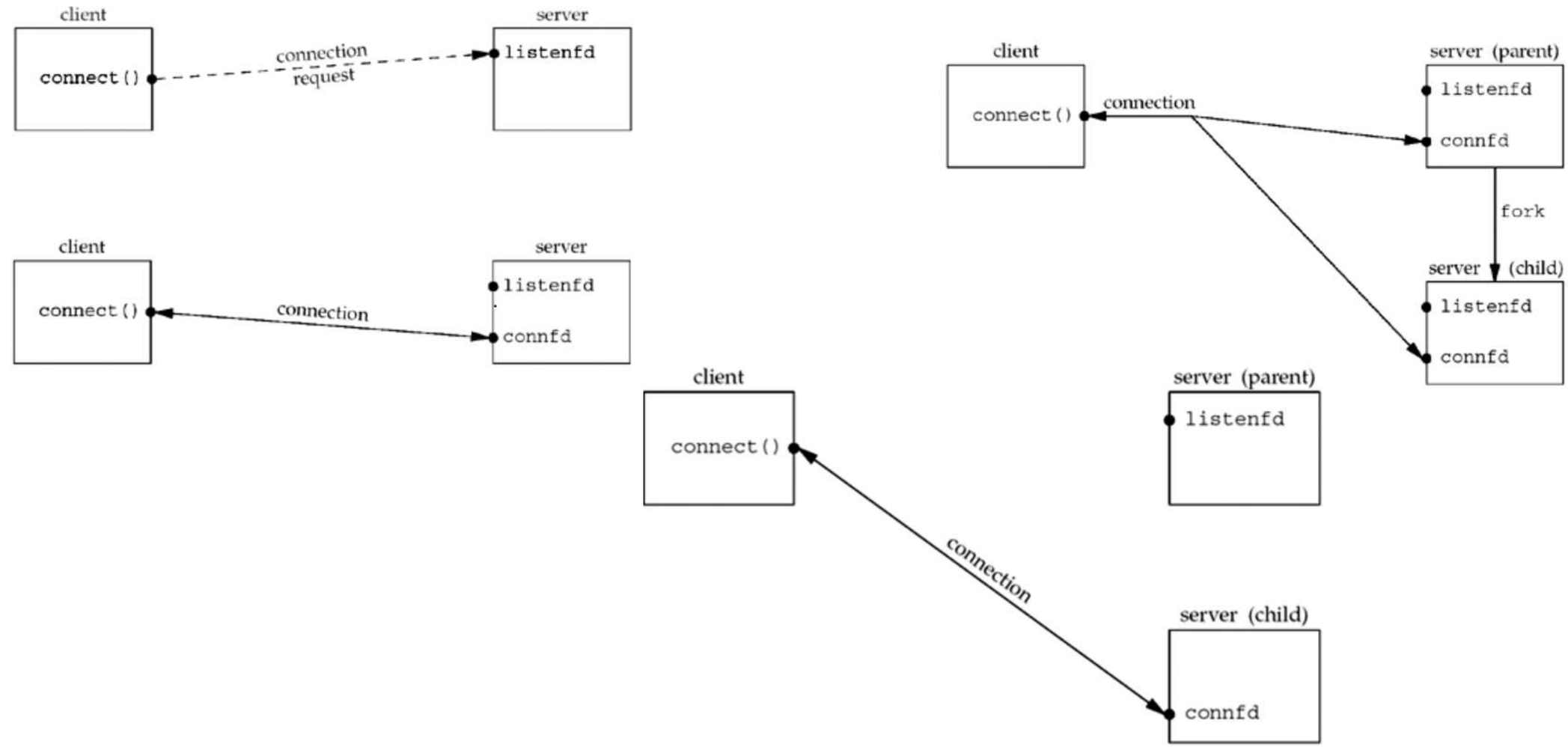
accept()

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 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.

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.

Remaining Topics

- IPv6 Socket Address Structure
- Byte Manipulation functions.
- Buffer size and limitations.

Due to the increased size of the IPv6 address, two new structures are needed. These structures are:

- **in6_addr:** The structure contains an array of sixteen 8-bit elements, that together make up a single 128-bit IPv6 address. The address is usually stored in network byte order.This structure is declared in *netinet/in.h*.
- **sockaddr_in6:** This structure is used to pass address information to the socket function calls that require network address information.

BSD 4.3 sockaddr_in6 address structure

```
struct sockaddr_in6 {
 sa_family_t     sin6_family;
 in_port_t       sin6_port;
 uint32_t        sin6_flowinfo;
 struct in6_addr sin6_addr;
 uint32_t        sin6_scope_id;
};
```

BSD 4.4/ UNIX 98 sockaddr_in6 address structure

```
struct sockaddr_in6 {
 uint8_t         sin6_len;
 sa_family_t     sin6_family;
 in_port_t       sin6_port;
 uint32_t        sin6_flowinfo;
 struct in6_addr sin6_addr;
 uint32_t        sin6_scope_id;
};
```

- sin6_len: This field contains the length of the address for UNIX 98 specifications.

Note

The sin6_len field is provided only for BSD 4.4 compatibility. It is not necessary to use this field even for BSD 4.4/ UNIX 98 compatibility. The field is ignored on input addresses.

- sin6_family :This field specifies the AF_INET6 address family.
- sin6_port:This field contains the transport layer port.
- sin6_flowinfo:This field contains two pieces of information: the traffic class and the flow label.

Note

Currently, this field is not supported and should be set to zero to be compatible with later versions.

- sin6_addr: This field specifies the IPv6 address.
- sin6_scope_id: This field identifies a set of interfaces as appropriate for the scope of the address carried in the sin6_addr field.

New Generic Socket Address Structure

- A new generic socket address structure was defined as part of the IPv6 sockets API, to overcome some of the shortcomings of the existing struct sockaddr.
- Unlike the struct sockaddr, the new struct sockaddr_storage is large enough to hold any socket address type supported by the system.
- The sockaddr_storage structure is defined by including the <netinet/in.h>

New Generic Socket Address Structure

```
struct sockaddr_storage {
  uint8_t      ss_len;       /* length of this struct (implementation dependent) */
  sa_family_t  ss_family;    /* address family: AF_xxx value */
  /* implementation-dependent elements to provide:
   * a) alignment sufficient to fulfill the alignment requirements of
   *    all socket address types that the system supports.
   * b) enough storage to hold any type of socket address that the
   *    system supports.
   */
};
```

New Generic Socket Address Structure

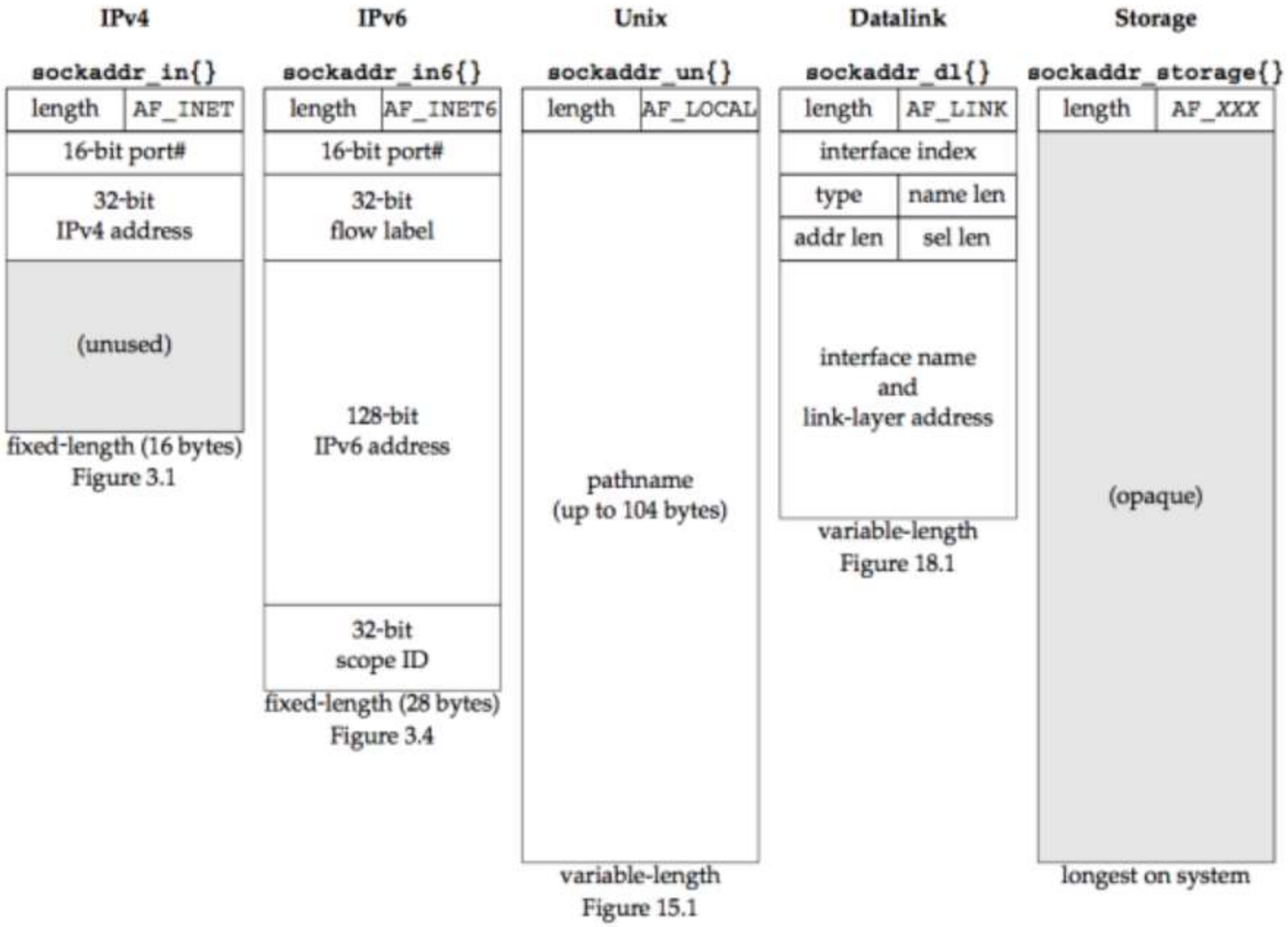
- The sockaddr_storage type provides a generic socket address structure that is different from struct sockaddr in two ways:

1. If any socket address structures that the system supports have alignment requirements, the sockaddr_storage provides the strictest alignment requirement.
2. The sockaddr_storage is large enough to contain any socket address structure that the system supports.

- The fields of the sockaddr_storage structure are opaque to the user, except for ss_family and ss_len (if present).
- The sockaddr_storage must be cast or copied to the appropriate socket address structure for the address given in ss_family to access any other fields.

Comparison of Socket Address structure

- IPv4, IPv6, Unix domain, datalink, and storage.



- Two of the socket address structures are fixed-length, while the Unix domain structure and the datalink structure are variable-length.
- To handle variable-length structures, whenever a pointer is passed to a socket address structure as an argument to one of the socket functions, its length is passed as another argument.

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.
- These functions are useful when dealing with socket address structures to manipulate fields such as IP addresses, which can contain bytes of 0, but are not C character strings.
- Functions that begin with *str*- Deal with null- terminated C string.
- Functions that begin with *b(bzero,bcmp, bcopy)*- Manipulate sockets.
- Functions that begin with *mem(memset,memcmp, memcopy)*- Manipulate sockets.

Byte Manipulation Functions

```
#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

- bzero sets the specified number of bytes to 0 in the destination- Often used 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.

Byte Manipulation Functions

```
#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);
```

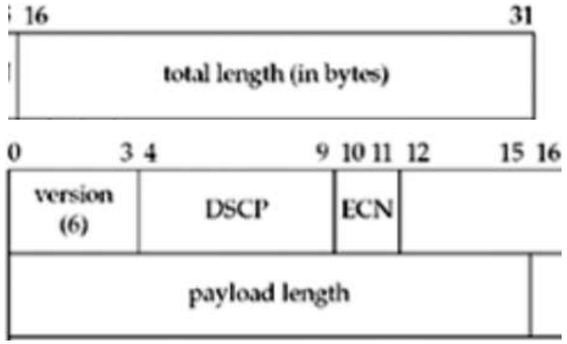
Returns: 0 if equal, <0 or >0 if unequal (see text)

- Memset sets the specified number of bytes to 0 in the destination- Often used to initialize a socket address structure to 0.
- Memcopy moves the specified number of bytes from the source to the destination.
- Memcmp compares two arbitrary byte strings. The return value is zero if the two byte strings are identical; otherwise, it is nonzero.

Buffer Sizes and Limitations

- Certain limits affect the size of IP datagrams.

Data Unit	Maximum size
IPV4	65,535 bytes(including header- 20B)
IPV6	65,575 bytes (including header 40B)
MTU (Hardware- Ethernet)	1500 bytes (including header 20B)
MTU (Hardware- PPP)	Configurable
SLIP (Hardware- PPP)	296 bytes.



- The smallest MTU in the path between two hosts is called the path MTU- Asymmetric.
- 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.
- IPv4 hosts and routers perform fragmentation on datagrams that they generate and that are forwarded.
- In IPv6, only hosts perform fragmentation on datagrams that they generate; IPv6 routers do not fragment datagrams that they are forwarding.

- With DF bit set and the size of the datagram exceeds the outgoing link then router generates ICMPv4 "destination unreachable, fragmentation needed but DF bit set" error message.
- In IPV6 the DF bit is set by default, If IPv6 router receives a datagram whose size exceeds the outgoing link's MTU, it generates an ICMPv6 "packet too big" error message.
- The IPv4 DF bit and its implied IPv6 counterpart can be used for path MTU discovery.
- For example, if TCP uses this technique with IPv4, then it sends all its datagrams with the DF bit set.
- If some intermediate router returns an ICMP "destination unreachable, fragmentation needed but DF bit set" error, TCP decreases the amount of data it sends per datagram and retransmits.

- IPv4 and IPv6 define a minimum reassembly buffer size, the minimum datagram size that we are guaranteed any implementation must support.
- 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.
- The goal of the MSS is to tell the peer the actual value of the reassembly buffer size and to try to avoid fragmentation.
- The MSS is often set to the interface MTU minus the fixed sizes of the IP and TCP headers.
- On an Ethernet using IPv4, this would be 1,460, and on an Ethernet using IPv6, this would be 1,440.

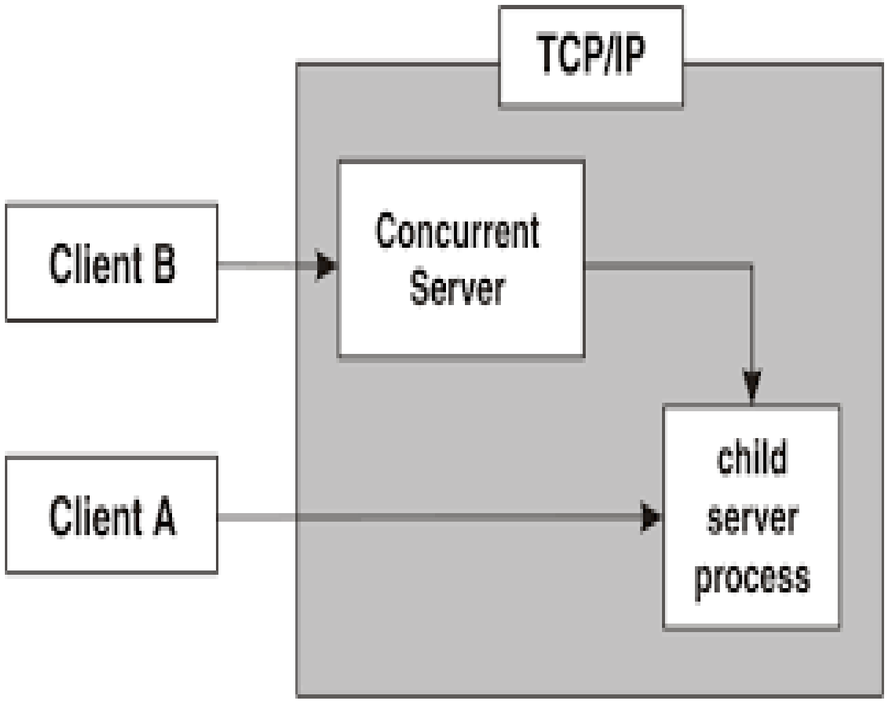
RV College of Engineering®

Go, change the world

Concurrent Servers

Concurrent Server

Concurrent Server: The **server** can be iterative, i.e. it iterates through each client and serves one request at a time. Alternatively, a **server** can handle multiple clients at the same time in parallel, and this type of a **server** is called a **concurrent server**.



How many concurrent connections can a server handle?
65535 simultaneous connections.

On the TCP level the tuple (source ip, source port, destination ip, destination port) must be unique for each simultaneous connection. That means a single client cannot open more than **65535 simultaneous connections** to a server. But a server can (theoretically) serve **65535 Simultaneous connections** per client.

There are several ways we can implement this server: The simplest technique is to call the Unix **fork()** function. Other techniques are to use threads or to pre-fork a fixed number of children when the function starts.

When the server *receives and accepts the client's connection, it forks a copy of itself and lets the* child handle the client as shown in the figure below:

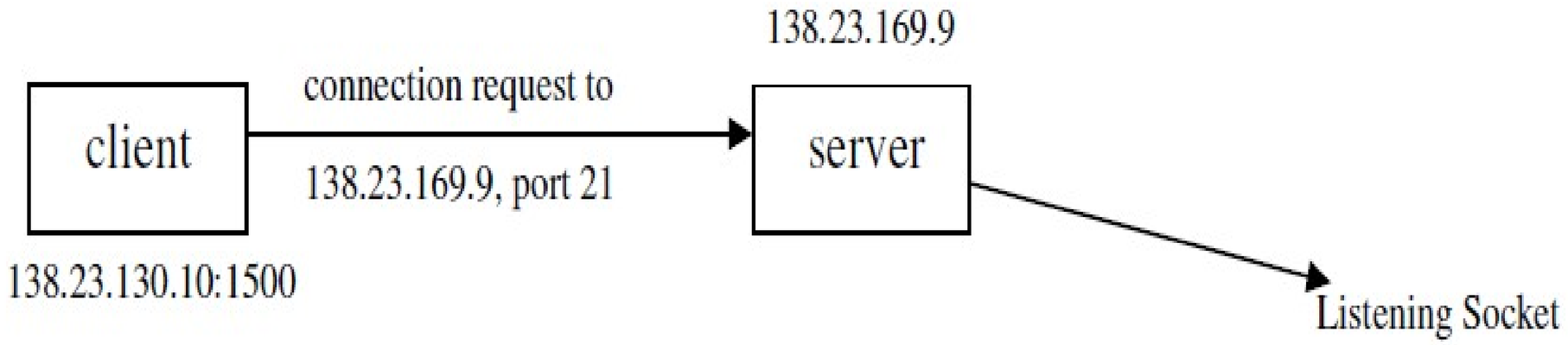


Figure 1: Client with IP address 138.23.130.10 requests to connect to Server (with IP 138.23.169.9 listening from port 21) from its local port number 1500.

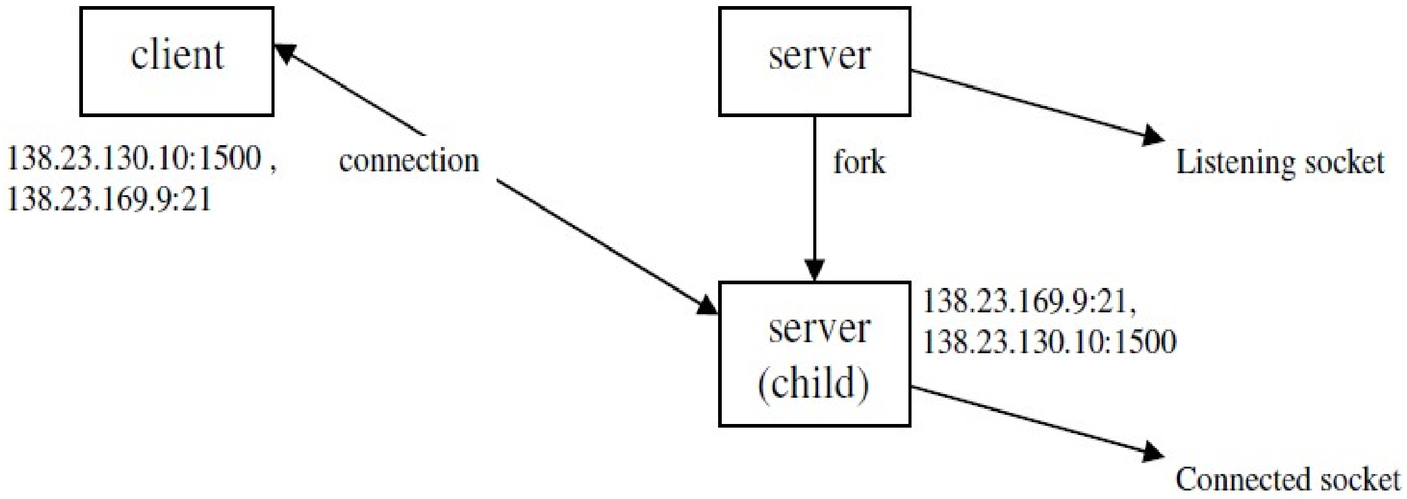


Figure 2: Server fork()s a copy of itself and lets the child handle this client from port 21, but it concurrently keeps listening (by the parent process) from port 21 as well.

Concurrent Server

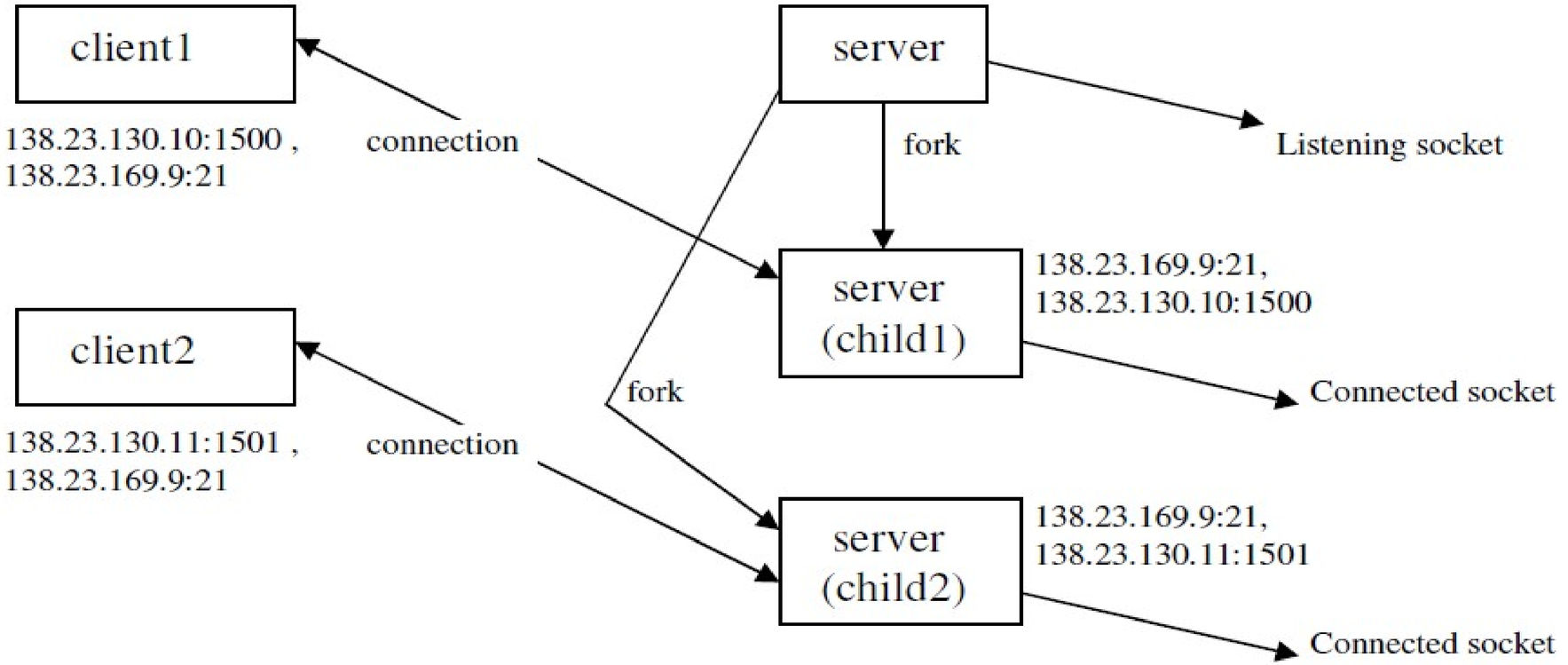


Figure 3: Another client requests to connect to this server. The server fork()s another copy of itself and lets this second child handle this client, whereas it still concurrently keeps listening from port 21 (by the parent process) and keeps serving client1 from port1 (by the first child process).

The listening socket must be distinguished from the connected socket on the server host. Although both sockets use the same local port on the server machine, they are indicated by distinct socket file descriptors, returned server's call of functions socket() and accept() respectively for listening and connected sockets.

Notice from figure 3 above that TCP must look at all four segments in the socket pair to determine which endpoint receives an arriving segment.

- In this figure, there are 3 sockets with the same local port 21 on the server.

- If a segment arrives from 138.23.130.10 port 1500 destined for 138.23.169.9 port 21, it is delivered to the
- first child.

- If a segment arrives from 138.23.130.11 port 1501 destined for 138.23.169.9 port 21, it is delivered to
- the second child.

- All other TCP segments destined for port 21 are delivered to the original listening socket.

The server described in intro/daytimetcpsrv1.c is an **iterative server**.

But 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 simplest way to write a concurrent server under Unix is to fork a child process to handle each client.

The following code shows the outline for a typical concurrent server:

```
pid_t pid;
int listenfd, connfd;
listenfd = Socket( ... );   /* fill in sockaddr_in{} with server's well-known 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);        /* parent closes connected socket */
}
```

- 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.

- We assume that the function doit does whatever is required to service the client.

- When this function returns, we explicitly close the connected socket in the child.

- This is not required since the next statement calls exit, and part of process termination is to close all open descriptors by the kernel.

- Whether to include this explicit call to close or not is a matter of personal programming taste.

Reference count of sockets

Calling close on a TCP socket causes a FIN to be sent, followed by the normal TCP connection termination sequence.

Why doesn't the close of connfd by the parent terminate its connection with the client? To understand what's happening, we must understand that 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.

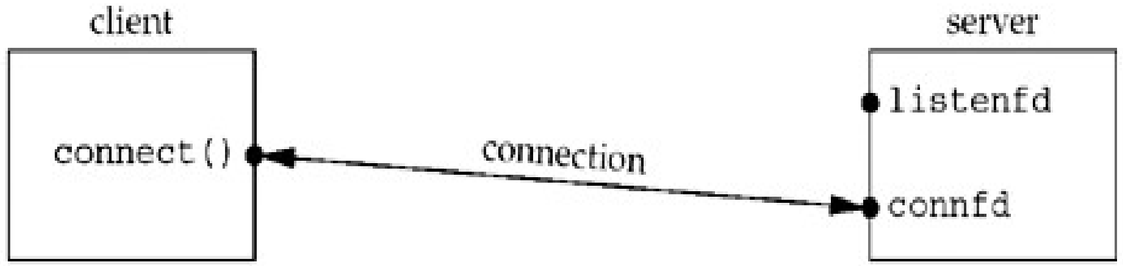
Reference count of sockets

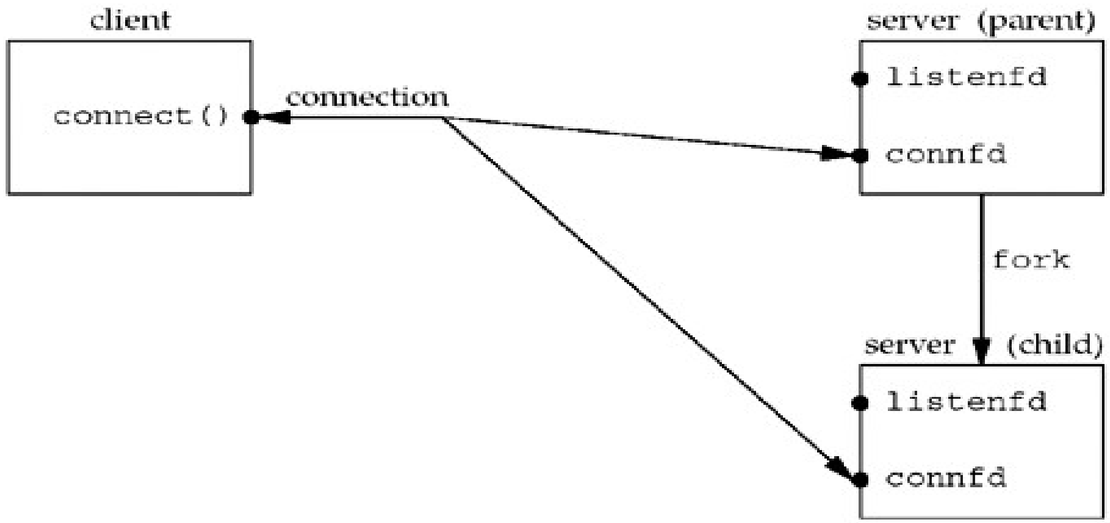
In the above code:

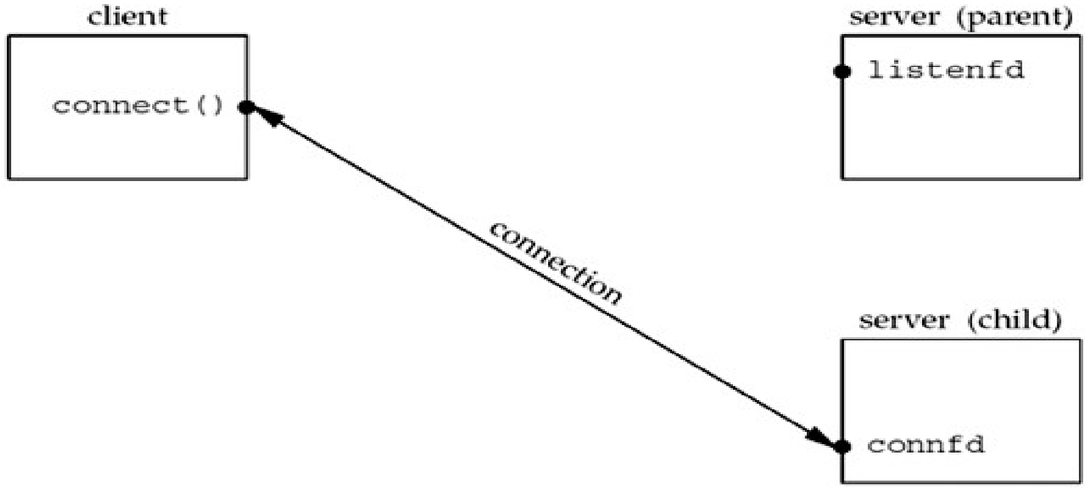
- After socket returns, the file table entry associated with listenfd has a reference count of 1.
- After accept returns, the file table entry associated with connfd has a reference count of 1.
- But, after fork returns, both descriptors are shared (duplicated) between the parent and child, so the file 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.

Visualizing the sockets and connection

The following figures visualize the sockets and connection in the code above: Before call to accept returns, the server is blocked in the call to accept and the connection request arrives from the client:







This is the desired final state of the sockets. The child is handling the connection with the client and the parent can call accept again on the listening socket, to handle the next client connection.

Close Function

The normal Unix close function is also 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 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.

SO_LINGER socket option (detailed in Section 7.5) lets us change this default action with a TCP socket.

Descriptor Reference Counts

As mentioned in end of Section 4.8, when the parent process in our concurrent server closes the connected socket, this just decrements the reference count for the descriptor.

Since the reference count was still greater than 0, this call to close did not initiate TCP's four-packet connection termination sequence. This is the behavior we want with our concurrent server with the connected socket that is shared between the parent and child.

If we really want to send a FIN on a TCP connection, the shutdown function can be used (Section 6.6) instead of close.

Beware if the parent does not call close for each connected socket returned by accept:
1. The parent will eventually run out of descriptors (there is usually a limit to the number of descriptors that any process can have open at any time).
2. None of the client connections will be terminated. When the child closes the connected socket, its reference count will go from 2 to 1 and it will remain at 1 since the parent never closes the connected socket. This will prevent TCP's connection termination sequence from occurring, and the connection will remain open.

getsockname and getpeername Functions

- getsockname returns the local protocol address associated with a socket.
- getpeername returns the foreign protocol address associated with a socket.

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

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

getsockname and getpeername Functions

These two functions are required for the following reasons:

❖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 portnumber), *getsockname* returns the local port number that was assigned. *getsockname* can be called to obtain the address family of a socket.

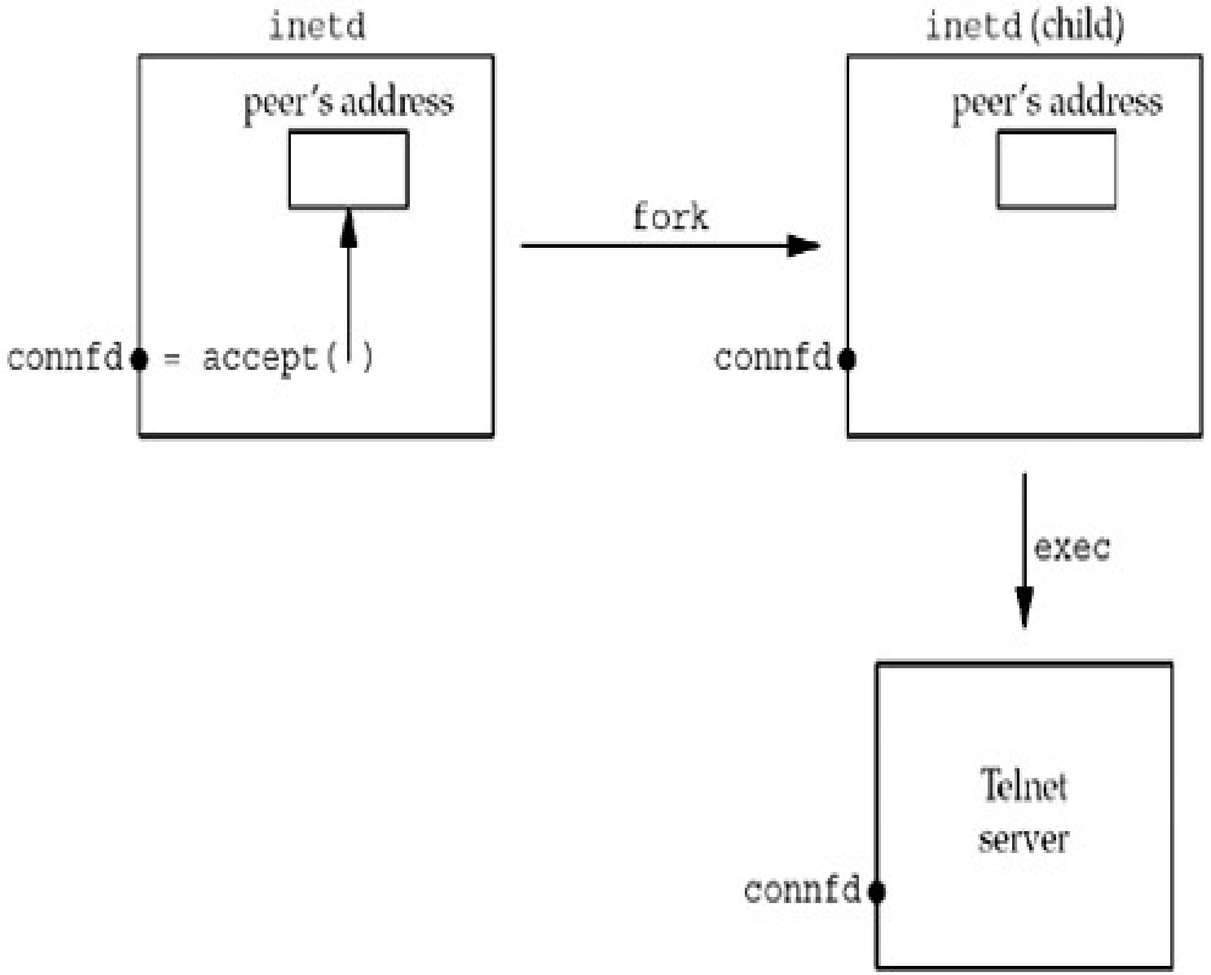
❖In a TCP server that binds the wildcard IP address (intro/daytimetcpsrv.c), once a connection is established with a client (accept returns successfully), the server can call *getsockname* to obtain the local IP address assigned to the connection.

❖The socket descriptor argument to *getsockname* 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.*

getsockname and getpeername Functions

For example, *inetd fork*s and *exec*s a TCP server (following figure):



- *inetd* calls *accept*, which returns two values: the connected socket descriptor (connfd, return value of the function) and the "peer's address" (an Internet socket address structure) that contains the IP address and port number of the client.
- *fork* is called and a child of *inetd* is created, with a copy of the parent's memory image, so the socket address structure is available to the child, as is the connected socket descriptor (since the descriptors are shared between the parent and child).
- When the child *exec*s the real server (e.g. Telnet server that we show), the memory image of the child is replaced with the new program file for the Telnet server (the socket address structure containing the peer's address is lost), and the connected socket descriptor remains open across the exec.
- One of the first function calls performed by the Telnet server is *getpeername* to obtain the IP address and port number of the client.

getsockname and getpeername Functions

In this example, the Telnet server must know the value of connfd when it starts. There are two common ways to do this.

The process calling exec pass it as a command-line argument to the newly execed program.

A convention can be established that a certain descriptor is always set to the connected socket before calling exec.

The second one is what inetd does, always setting descriptors 0, 1, and 2 to be the connected socket.

Example: Obtaining the Address Family of a Socket

The sockfd_to_family function shown in the code below returns the address family of a socket.

https://github.com/shichao-an/unpv13e/blob/master/lib/sockfd_to_family.c

```
int
sockfd_to_family(int sockfd)
{
        struct sockaddr_storage ss;
    socklen_t len;

    len = sizeof(ss);
    if (getsockname(sockfd, (SA *) &ss, &len) < 0)
        return(-1);
    return(ss.ss_family);
}
```

Example: Obtaining the Address Family of a Socket

The sockfd_to_family function shown in the code below returns the address family of a socket.

https://github.com/shichao-an/unpv13e/blob/master/lib/sockfd_to_family.c

This program does the following:

Allocate room for largest socket address structure. Since we do not know what type of socket address structure to allocate, we use a sockaddr_storage value, since it can hold any socket address structure supported by the system.

Call getsockname. We call getsockname and return the address family. The POSIX specification allows a call to getsockname on an unbound socket.