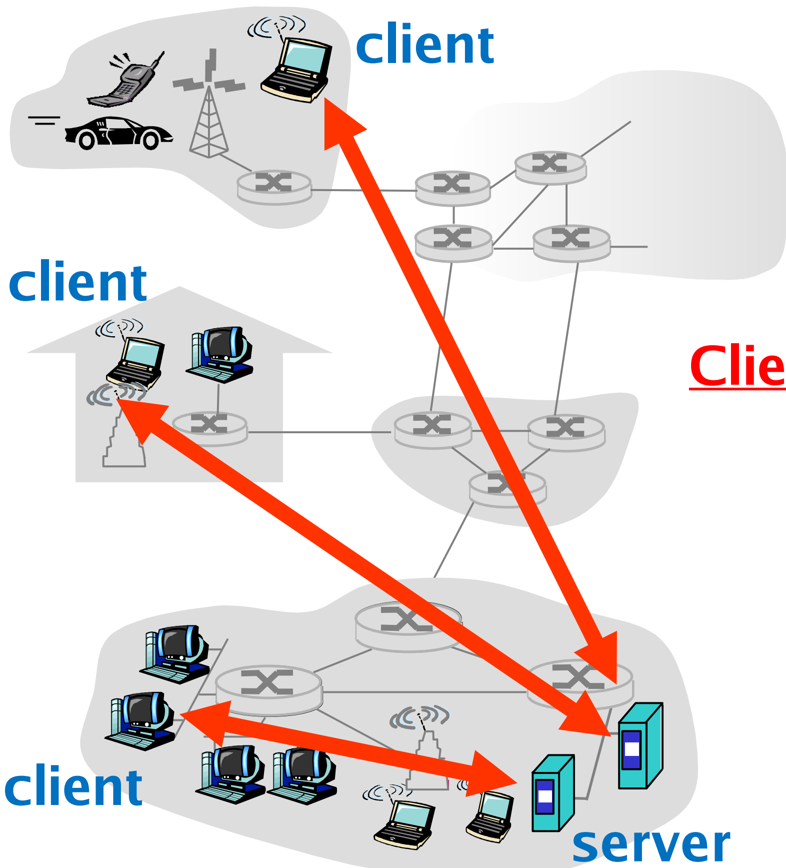
Chapter 2: Application Layer

- Principles of network applications
- Unix processes and IPC
- Web and HTTP
- Electronic Mail - SMTP, POP3, IMAP
- DNS
- Socket programming

Application Architectures

- Client-server
- Peer-to-peer (P2P)
- Hybrid of client-server and P2P

Client-server Architecture



Servers:

- Must **always be online**
- Provide network services to clients
- Assigned a **permanent IP address**
- **Server farms** used for scalability

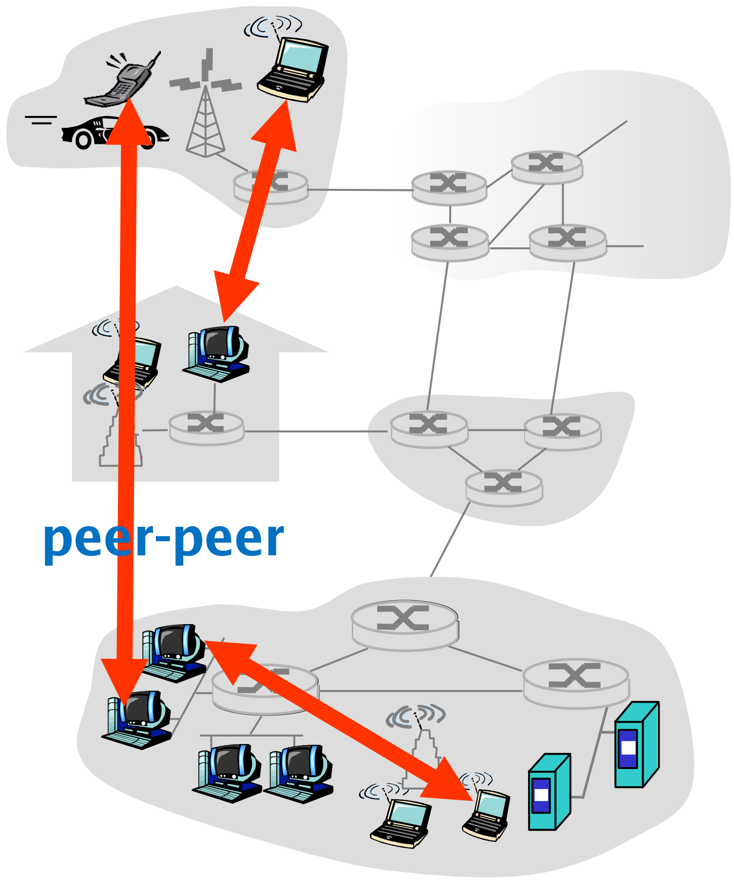
Clients:

- Communicate with servers
- May be intermittently connected
- May have **dynamic IP addresses**
- Do not communicate directly with each other

Pure P2P Architecture

- No "always-on server"
- Arbitrary end systems communicate directly
- Peers are intermittently connected and change IP addresses
- example: Gnutella

Highly scalable but difficult to manage



Hybrid of Client-Server and P2P

Skype

- Voice-over-IP P2P application
- Centralized server: finding address of remote party:
- Client-client connection: direct

Instant messaging

- Chatting between two users is P2P
- Centralized service: client presence detection/location
  - User registers its IP address with central server when it comes online
  - User contacts central server to find IP addresses of buddies

Processes & Network Communications

Process: instance of an executing program.

- Processes within same host communicate using **Inter-Process Communication (IPC)**
- Processes in different hosts communicate using **sockets** (covered in TCP/IP programming)

Client process initiates communication with a remote server

Server process accepts and processes client requests

The UNIX Processes

- **Program** : a collection of instructions
- **Process** (= task): an instance of a program that consists 3 segments:
  - Instruction segment
  - User data segment
  - System data segment

Example

cat file1 file2

ls | wc -l

The UNIX Processes (cont.)

- **init**: a single controlling process at the top of the process tree
- System calls for process creation and manipulation:
  - **fork()**
  - **exec()**
  - **exit()**

Creating Processes

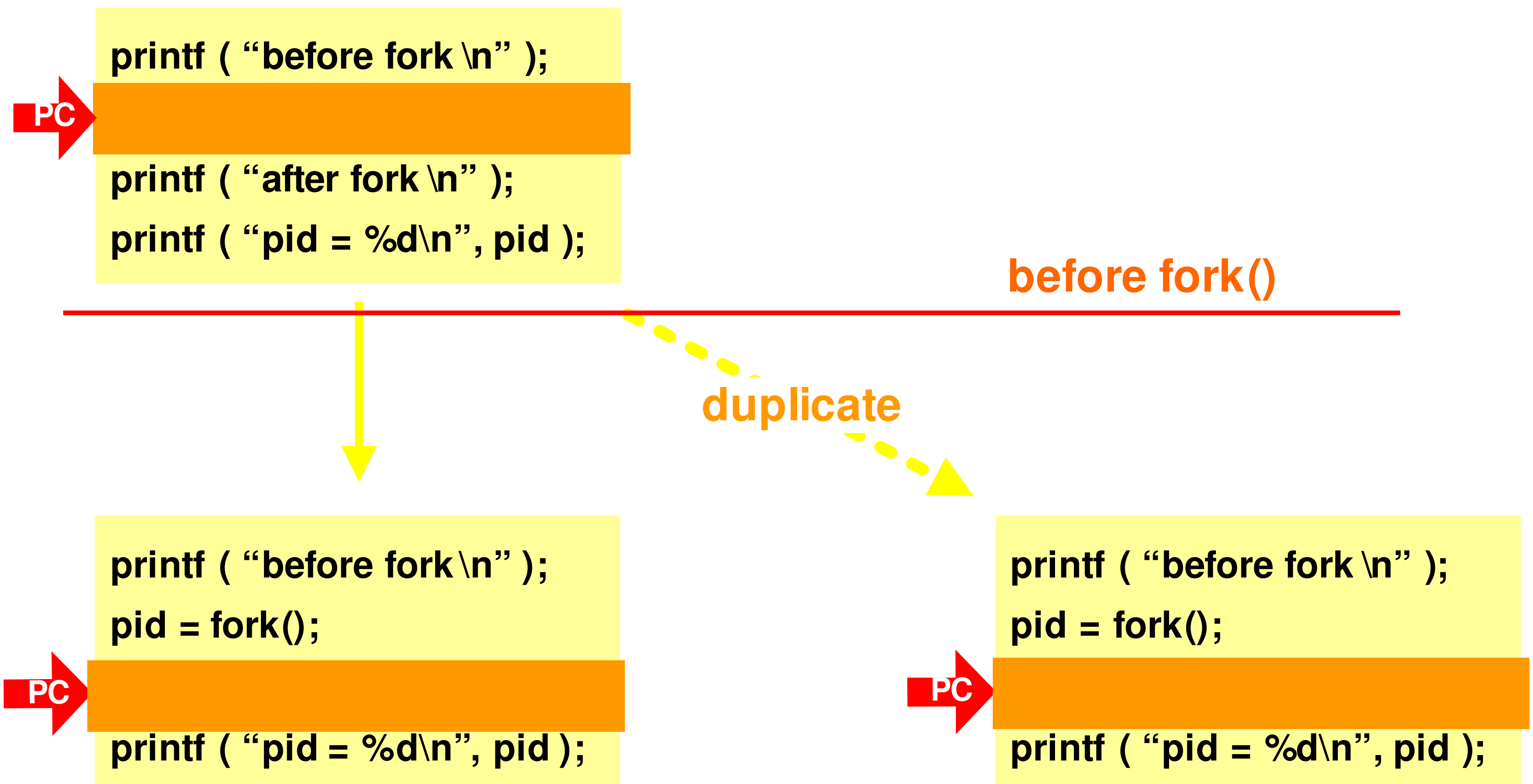
- **fork()** system call creates a new process (child) duplicating the calling process (parent)
- Parent and children run **concurrently** without synchronization

Usage

```
int pid;
pid = fork();
```

- pid in parent = child's pid
- pid in child = 0
- -1 on failure

fork() System Call



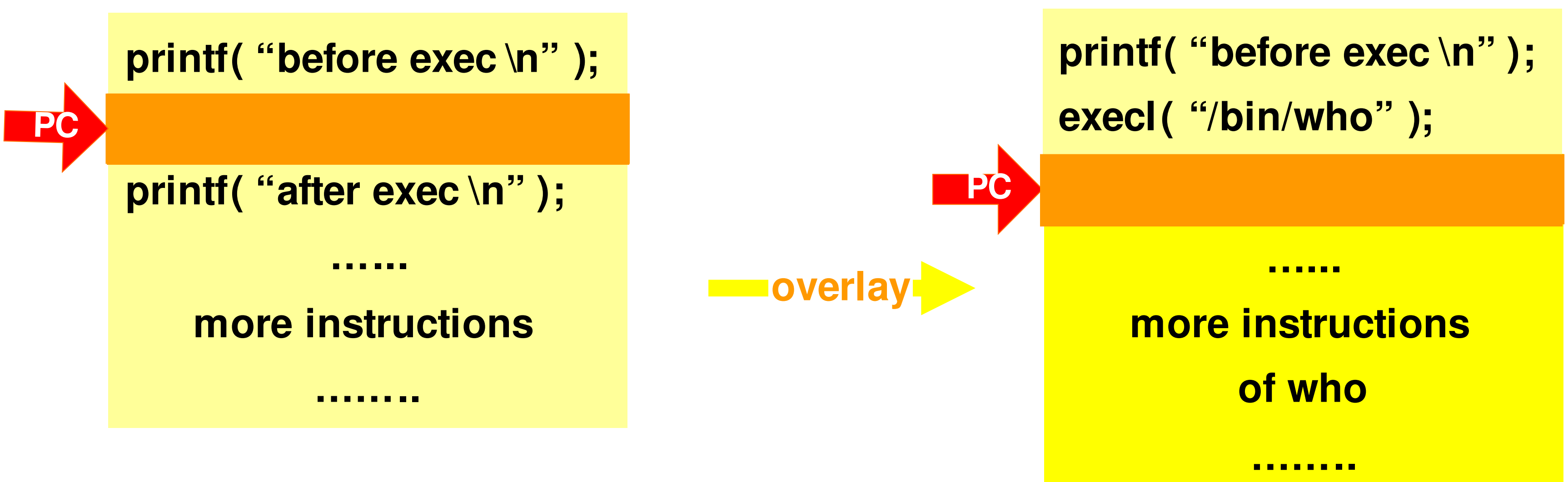
exec() System Call

- Calling process is overlaid by new code and execution begins from the first line
- No return from a successful call to exec()

```
char *path, *arg0, *arg1, … *argn;
execl( path, arg0, arg1, .. argn,
        (char *)0 );
```

- **path:** path to the program executable
- **arg0:** program name
- **arg1 – n:** program parameters
- **(char *)0:** null argument list terminator
- Returns -1 for error

exec() System Call (cont.)



exit() System Call

- Terminates a process
  - Closes all open file descriptors
  - Blocked parent process is started
  - 'Exit status' indicates the success or failure
    - 0 = normal termination, non-zero = error

Usage

```
int status;
void exit( status );
```

IPC using Pipes

- **One-way (half-duplex)** communications channel
- **Couples** one process to another

  connecting the standard output of one process to the standard input of another **(parent and children)**
- Generalization of the UNIX **file concept**

  Command example

  who | wc -l

pipe() System Call

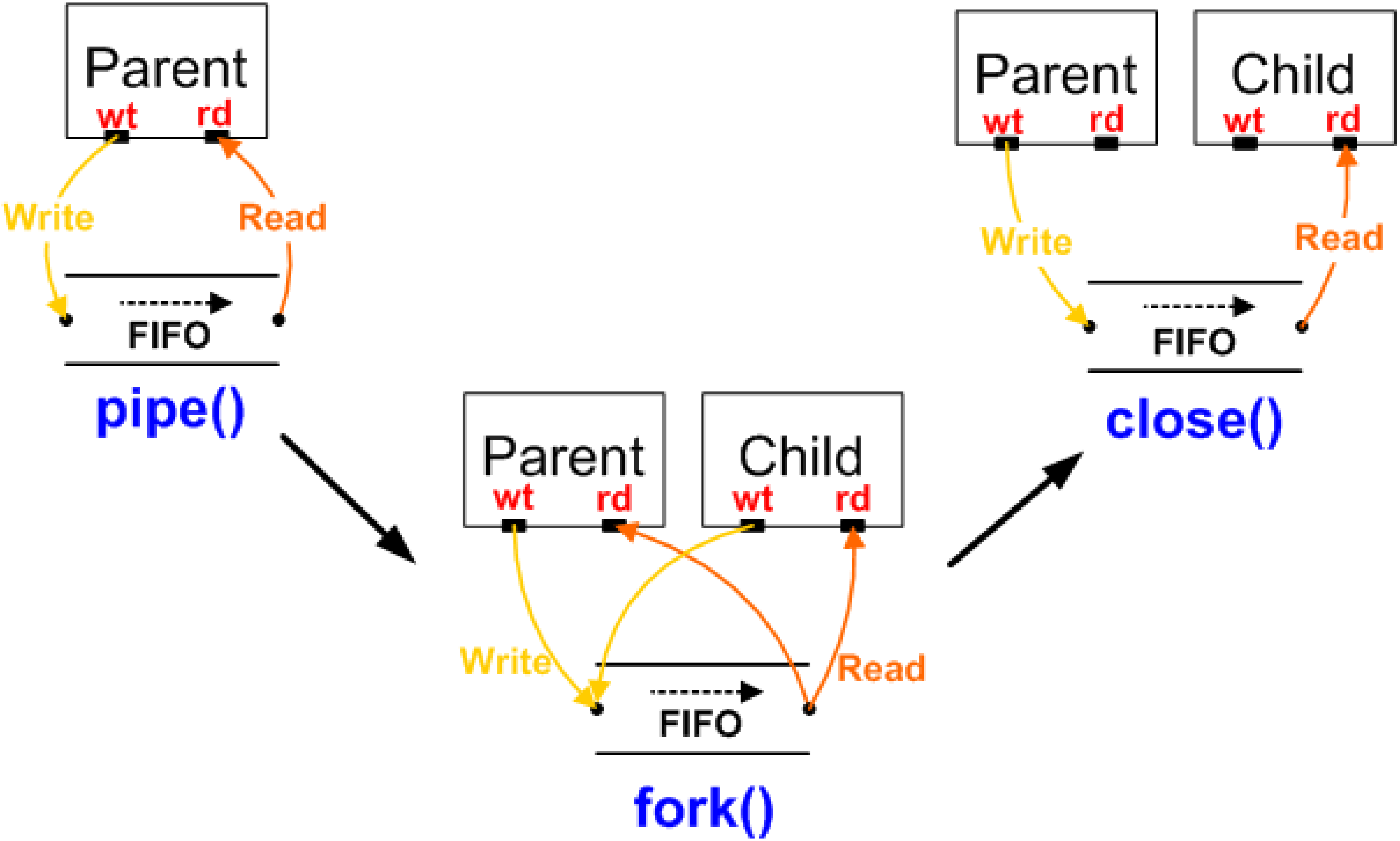
- Creates **2 file descriptors** on success: **write** & **read**
- Manipulated with **read()** and **write()**
- **FIFO** data access
- Used in conjunction with **‘fork’**
  - a child process will inherit any open file descriptors from the parent

pipe() System Call (cont.)

```
int filedes[2];
int retval;
retval = pipe( filedes );
```

- **filedes:** **2-integer array** to hold the file descriptors created
  - array[0] = read
  - array[1] = write
- **Return**:
  - On success = 0.
  - On error = -1, and errno is set

Pipe with Two processes



Programming with Pipes

Size : finite (5120 bytes)

- write on a pipe without enough space

⇒ (default) Suspended until room is made

- read from an empty pipe

⇒ (default) Blocked until data is written

Programming with Pipes (cont.)

- Closing **write** descriptor
  - Other processes still have the pipe open for writing

    ⇒ No affect
  - No other processes have the pipe open for writing

    ⇒ Write descriptor will be closed

    ⇒ Releases blocked reading processes if any

Programming with Pipes (cont.)

- Closing **read** descriptor
  - Other processes still have the pipe open for reading

    ⇒ No affect
  - No other processes have the pipe open for writing

    ⇒ Reads descriptor will be closed

    ⇒ Sends **SIGPIPE** to all writing & waiting to write processes

fork() & pipe() Example - pipe.c

- gcc –o pipe pipe.c
- ./pipe 3

Application Service Requirements

Data loss

- some apps (e.g., audio) can tolerate some loss
- other apps (e.g., file transfer, ssh) require 100% reliable data transfer

Timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

Bandwidth

- some apps (e.g., multimedia) require minimum amount of bandwidth to be "effective"
- other apps ("elastic apps") make use of whatever bandwidth they get

Transport Service Requirements For Common Applications

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100's msec
instant messaging	no loss	elastic	yes and no

Internet Transport Protocols and Services

TCP service:

- Connection-oriented: connection establishment required
- Reliable transport
- Flow control: sender prevented from overwhelming receiver
- Congestion control: throttle sender traffic when network is overloaded
- Does not provide: timing, minimum bandwidth guarantees

UDP service:

- Connectionless
- Unreliable
- Does not provide: connection setup, reliability, flow control, congestion control, timing, or bandwidth guarantee

Internet Applications: Application, Transport Protocols

Application	Application layer protocol	Underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	proprietary (e.g. RealNetworks)	TCP or UDP
Internet telephony	proprietary (e.g., Vonage,Dialpad)	typically UDP

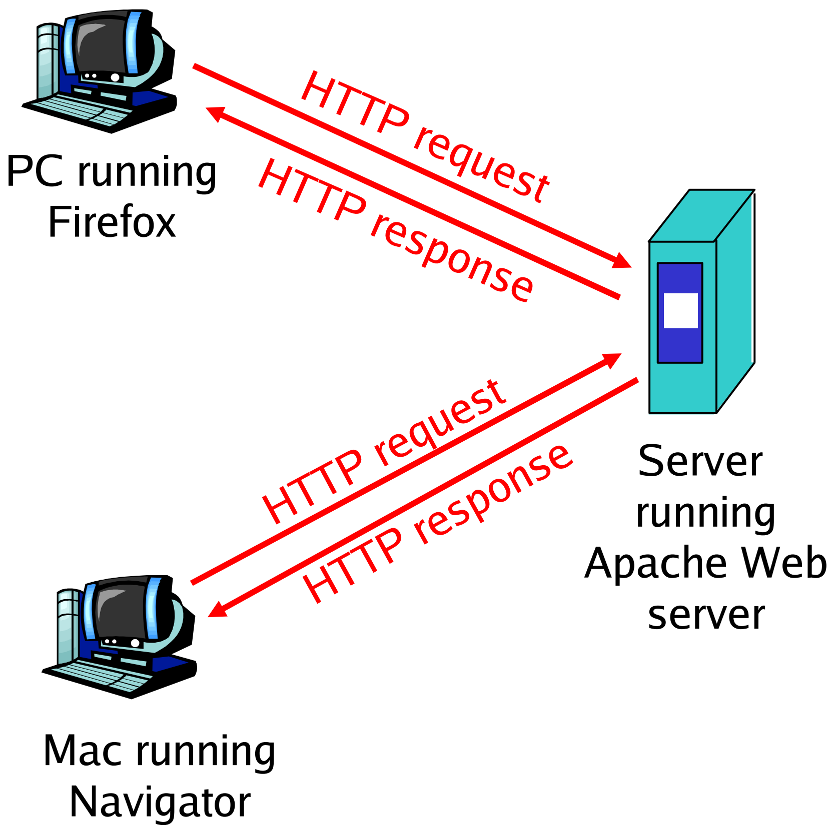
HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- Uses **TCP**:
- **Client** initiates TCP connection to server **port 80**
- **Server** accepts TCP connection from client

HTTP is "stateless"

- Server maintains no information about past client requests



HTTP Connections

Nonpersistent HTTP

- At most one object is sent over a TCP connection.
- HTTP/1.0 uses nonpersistent HTTP

Persistent HTTP

- Multiple objects can be sent over single TCP connection between client and server.
- HTTP/1.1 uses persistent connections in default mode

Nonpersistent HTTP

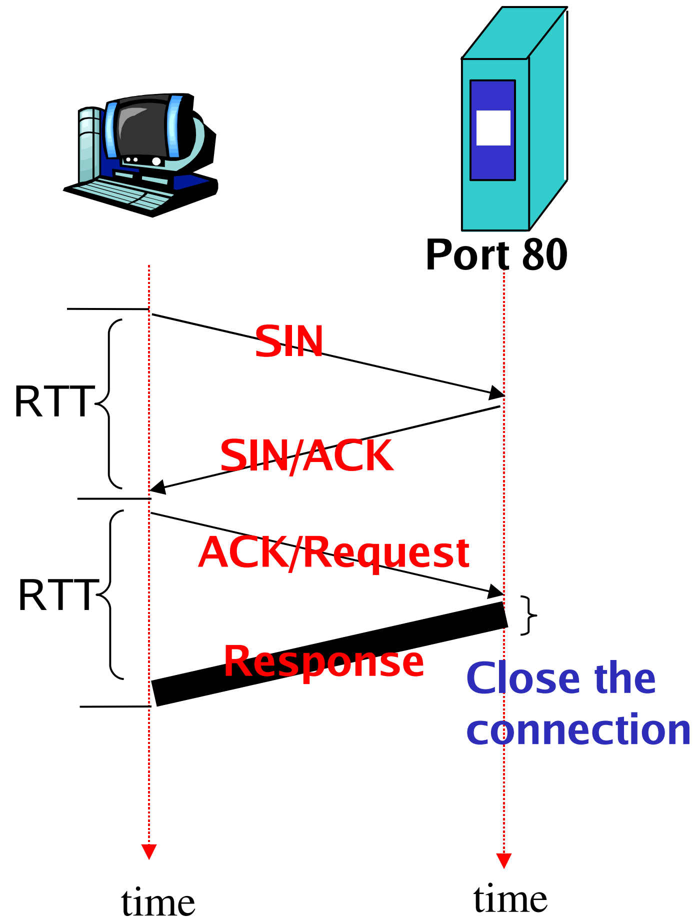
Response time:

- One RTT to initiate TCP connection
- One RTT for HTTP request and first few bytes of HTTP response to return
- file transmission time

total = 2RTT+transmit time

Nonpersistent HTTP issues:

- Requires 2 RTTs per object
- OS overhead for *each* TCP connection



Persistent HTTP

- Server leaves connection open after sending response
- Subsequent HTTP messages between same client/server sent over open connection

Persistent ***without*** pipelining:

- Client issues new request only when previous response has been received
- One RTT for each referenced object

Persistent ***with*** pipelining:

- Default in HTTP/1.1
- Client sends requests as soon as it encounters a referenced object
- As little as one RTT for all the referenced objects

User-server state: cookies

Many major Web sites use cookies

Four components:

1) Cookie header line of **HTTP *response* message**
2) Cookie header line in **HTTP *request* message**
3) **Cookie file** kept on user's host, managed by user's browser
4) **Back-end database** at Web site

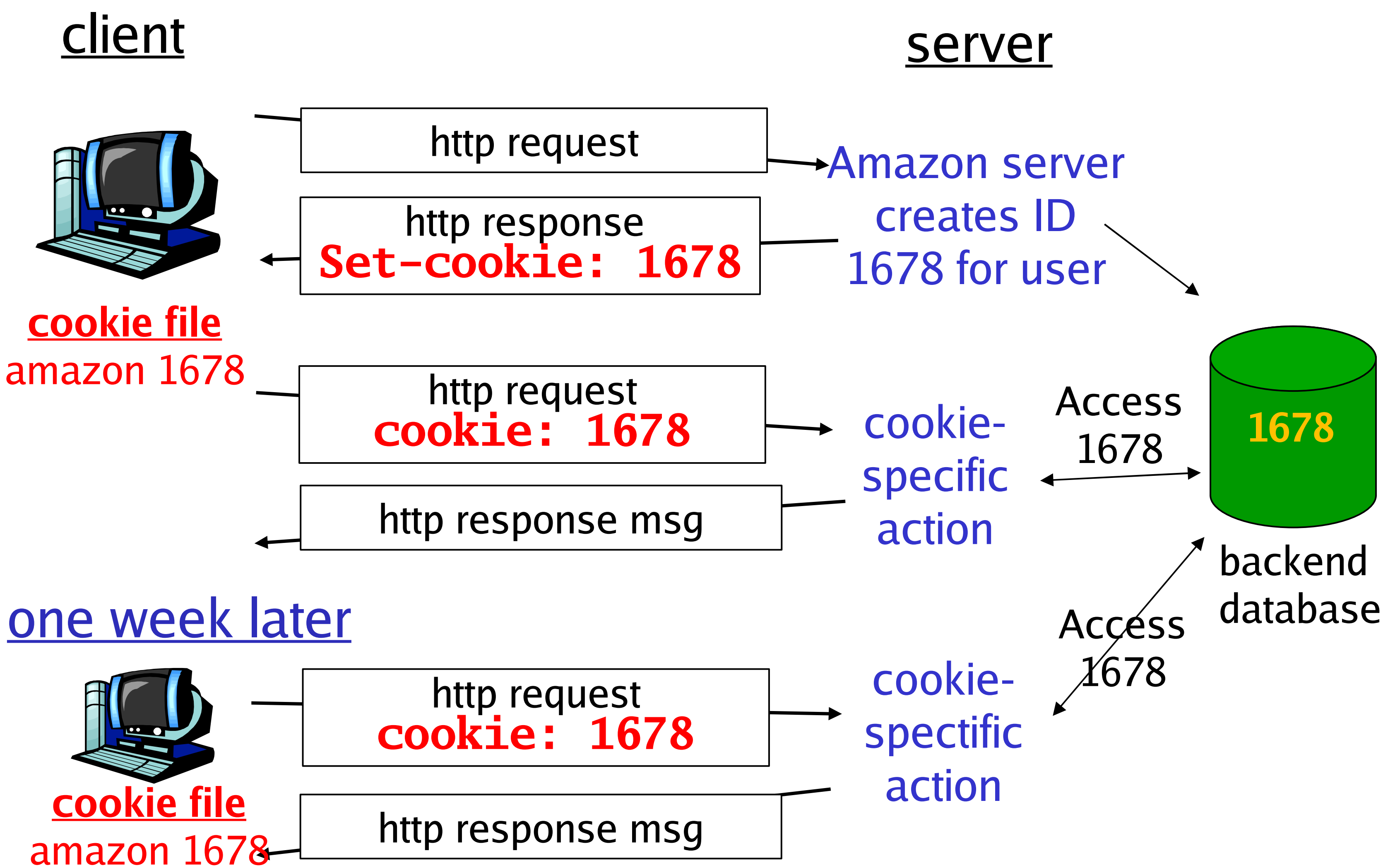
Cookies can provide:

- Authorization
- Shopping carts
- Recommendations
- User session state (Web e-mail)

Cookies and privacy:

- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

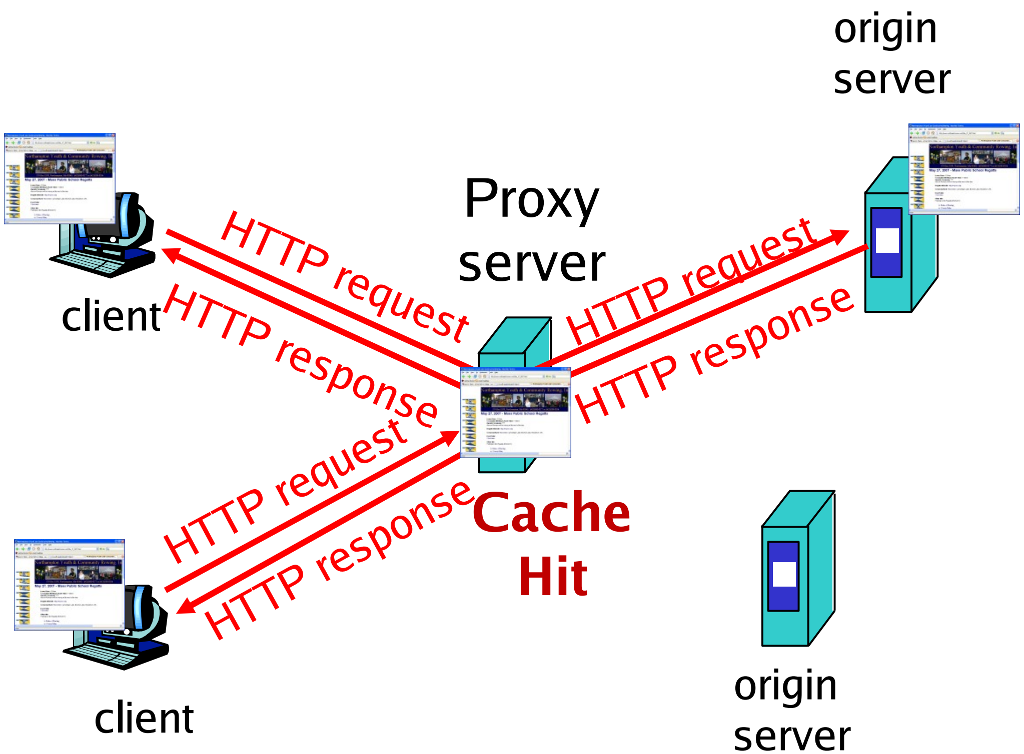
Cookies (cont.)



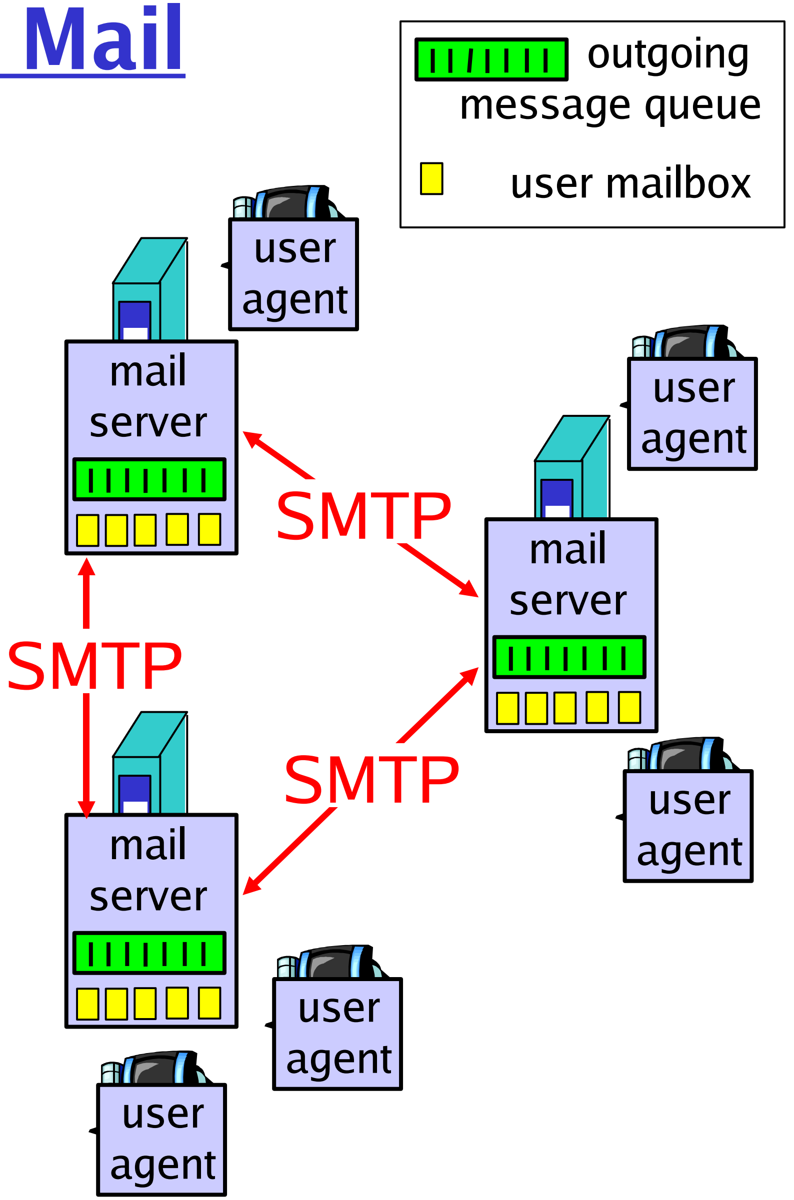
Web caches (proxy server)

Goal: satisfy client request without involving origin server.
=> Reduce response time and traffic.

- User sets browser: Web accesses via cache
- Browser sends all HTTP requests to cache (proxy server)
  - Object in cache: cache returns object (Hit!)
  - Else cache requests object from origin server, then returns object to client



Electronic Mail



Three major components:

- User agents
- Mail servers
- Simple mail transfer protocol: **SMTP**

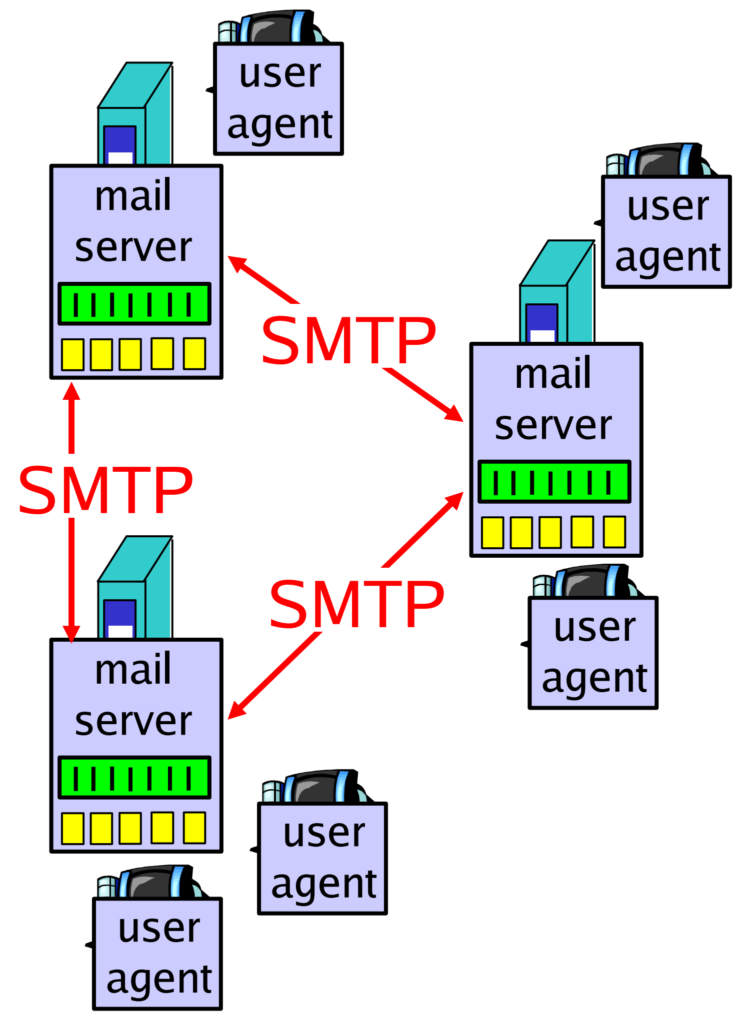
User Agent

- a.k.a. "email clients"
- Composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Mozilla Thunderbird

Electronic Mail (cont.)

Mail Servers

- Mailbox contains incoming messages for user
- Message queue of outgoing (to be sent) mail messages
- SMTP protocol between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server

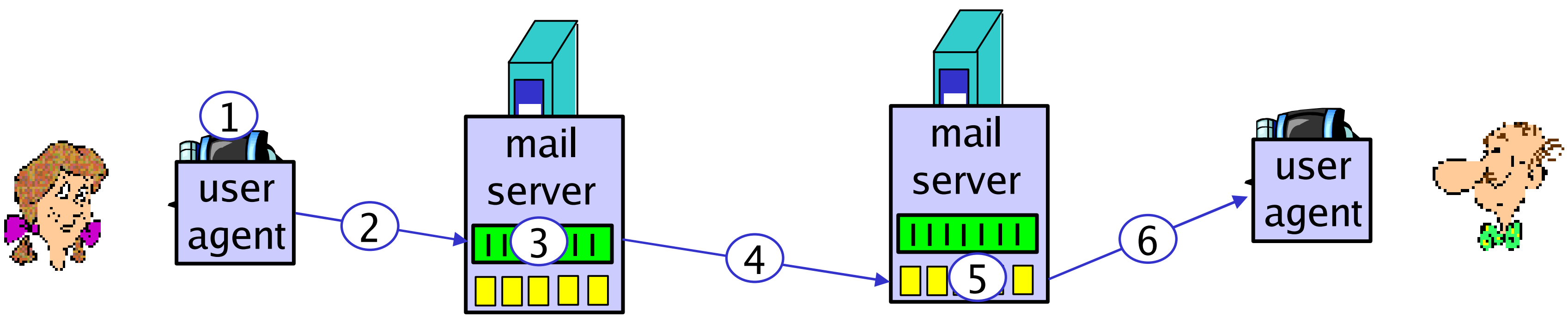


Electronic Mail: SMTP

- Uses TCP to reliably transfer email message from client to server, port 25
- Direct transfer: sending server to receiving server
- 3 phases of transfer
  - Handshaking (greeting)
  - Transfer of messages
  - Closure
- Command/response interaction
  - Commands: ASCII text
  - Response: status code and phrase
- Messages must be in 7-bit ASCII

Scenario: Alice Sends Message to Bob

1) Alice uses UA to compose message to bob@someschool.edu
2) Alice's UA sends message to her mail server; message placed in message queue
3) Client side of SMTP opens TCP connection with Bob's mail server
4) SMTP client sends Alice's message over the TCP connection
5) Bob's mail server places the message in Bob's mailbox
6) Bob invokes his user agent to read message



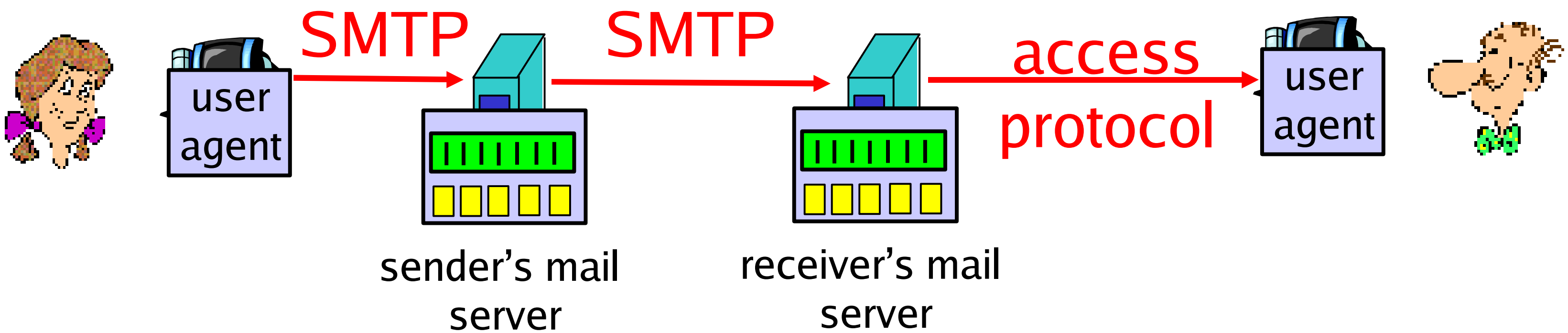
Sample SMTP Interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

Telnet to the Mail Server

telnet <mail server> 25

Mail access protocols



- SMTP: delivery/storage to receiver's server
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]
    - Authorization (agent <-->server) and download
  - IMAP: Internet Mail Access Protocol [RFC 1730]
    - More features (more complex)
    - Manipulation of stored msgs on server
  - HTTP: gmail, Hotmail, Yahoo! Mail, etc.

POP3 (more) and IMAP

POP3

- "Download-and-delete" mode: Bob cannot re-read e-mail if he changes client
- "Download-and-keep" mode: copies of messages are saved on different clients
- **Stateless** across sessions

IMAP

- Keep all messages in one place: the server
- Allows user to organize messages in folders
- **Stateful** across sessions:
  - Names of folders and mappings between message IDs and folder name

DNS: Domain Name System

Map between IP addresses and name

- **IP address**: used for addressing datagrams
- **Name**: used by humans
  - e.g., www.yahoo.com

DNS services

- Hostname to IP address **translation**
- **Aliasing** (host, mail server)
  - Canonical, alias names
- **Load distribution**
  - Set of IP addresses for one canonical name

Accessing hotmail.com

Local DNS server

No.	Time	Source	Destination	Protocol	Info
3	1.134841	192.168.1.75	64.59.144.16	DNS	Standard query AAAA login.passport.net
4	1.145866	64.59.144.16	192.168.1.75	DNS	Standard query response CNAME login.passport.net.nsatc.net
5	1.146336	192.168.1.75	64.59.144.16	DNS	Standard query A login.passport.net
6	1.153746	64.59.144.16	192.168.1.75	DNS	Standard query response CNAME login.passport.net.nsatc.net A 65.54.179.196 A 65.54.183.195
7	1.154236	192.168.1.75	65.54.179.196	TCP	45658 > http [SYN] Seq=0 Ack=0 Win=5840 Len=0 MSS=1460 TSV=886259370 TSER=0 WS=2
8	1.189749	65.54.179.196	192.168.1.75	TCP	http > 45658 [SYN, ACK] Seq=0 Ack=1 Win=16384 Len=0 MSS=1460 WS=0 TSV=0 TSER=0
9	1.189798	192.168.1.75	65.54.179.196	TCP	45658 > http [ACK] Seq=1 Ack=1 Win=5840 Len=0 TSV=886259405 TSER=0
10	1.190142	192.168.1.75	65.54.179.196	HTTP	GET /uilogin.srf?lc=1033&id=2 HTTP/1.1
11	1.229375	65.54.179.196	192.168.1.75	TCP	[TCP Previous segment lost] http > 45658 [FIN, ACK] Seq=474 Ack=438 Win=65098 Len=0

DNS

Handshake

Http

```
▷ Frame 6 (149 bytes on wire, 149 bytes captured)
▷ Ethernet II, Src: 00:16:b6:18:b1:37, Dst: 00:0d:88:36:c0:f1
▷ Internet Protocol, Src Addr: 64.59.144.16 (64.59.144.16), Dst Addr: 192.168.1.75 (192.168.1.75)
▷ User Datagram Protocol, Src Port: domain (53), Dst Port: 32918 (32918)
▽ Domain Name System (response)
    Transaction ID: 0x2640
  ▷ Flags: 0x8180 (Standard query response, No error)
    Questions: 1
    Answer RRs: 3
    Authority RRs: 0
    Additional RRs: 0
  ▷ Queries
  ▽ Answers
    ▽ login.passport.net: type CNAME, class IN, cname login.passport.net.nsatc.net
        Name: login.passport.net
        Type: CNAME (Canonical name for an alias)
        Class: IN (0x0001)
        Time to live: 7 minutes, 24 seconds
        Data length: 27
        Primary name: login.passport.net.nsatc.net
    ▽ login.passport.net.nsatc.net: type A, class IN, addr 65.54.179.196
        Name: login.passport.net.nsatc.net
        Type: A (Host address)
        Class: IN (0x0001)
        Time to live: 7 minutes, 24 seconds
        Data length: 4
        Addr: 65.54.179.196
    ▽ login.passport.net.nsatc.net: type A, class IN, addr 65.54.183.195
        Name: login.passport.net.nsatc.net
```

Canonical hostname

IP address(es)

For hotmail

Returned in different order each time (load distribution)

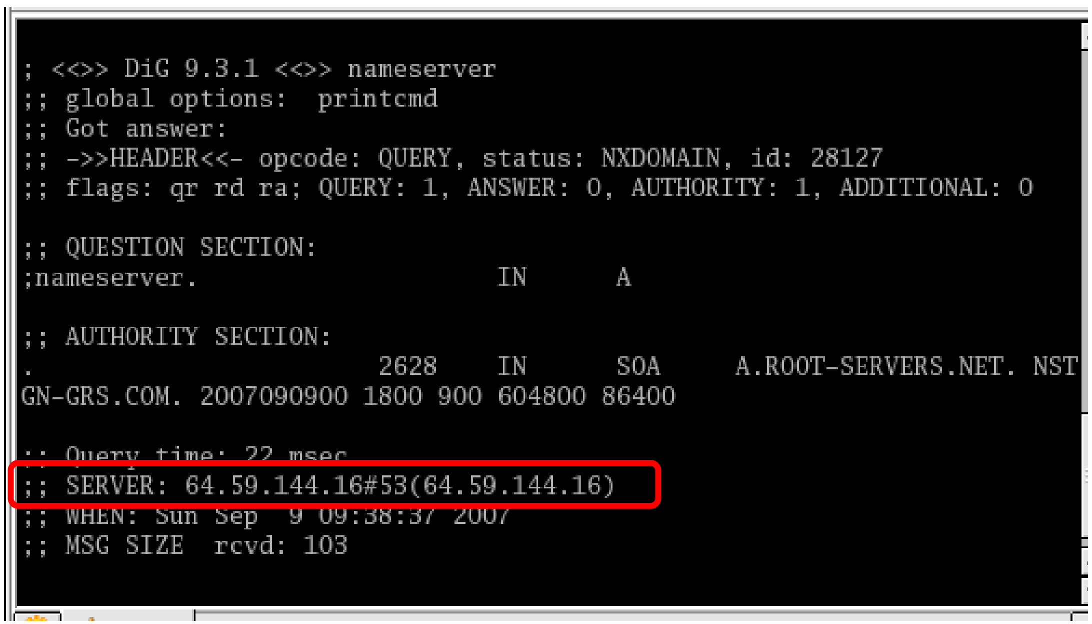
Local DNS (Default Name) Server

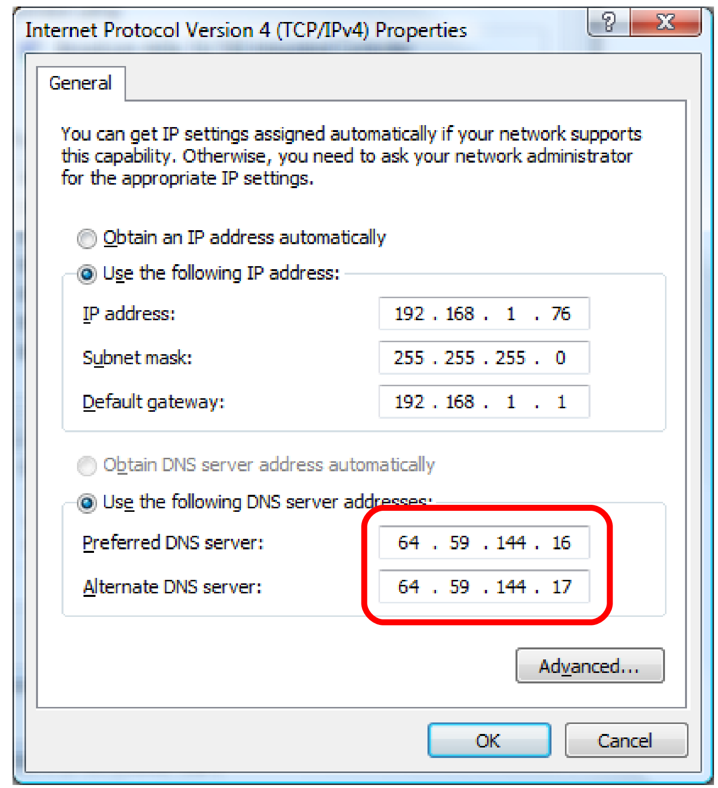
- Host’s query is sent to its local DNS server
- Acts as proxy, forwards query into DNS server hierarchy

Windows

Unix

> dig nameserver

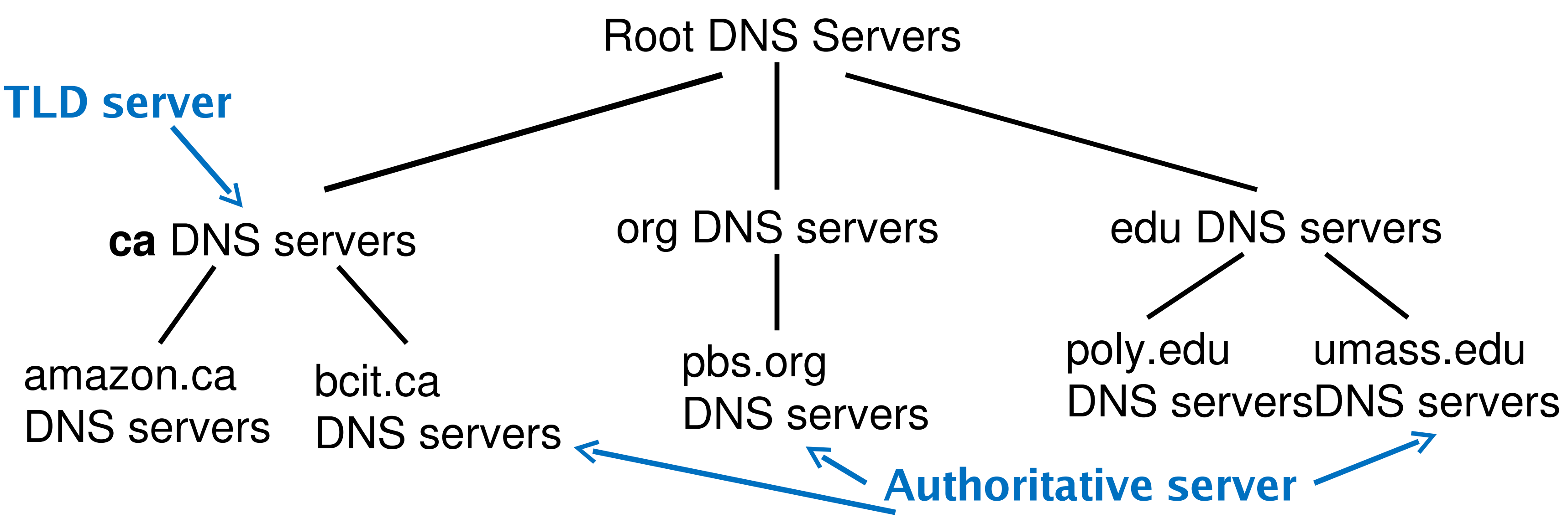




Wireshark Lab

1. Download Wireshark
2. Run Wireshark
   - Capture – Option
     - Interface: eth0 (Linux), (Control Panel - Windows)
     - Capture Filter: src <IP> or dst <IP>
     - “Update list of packets in real time” = yes
     - “Automatic scrolling in live capture” = yes
3. Access to a website

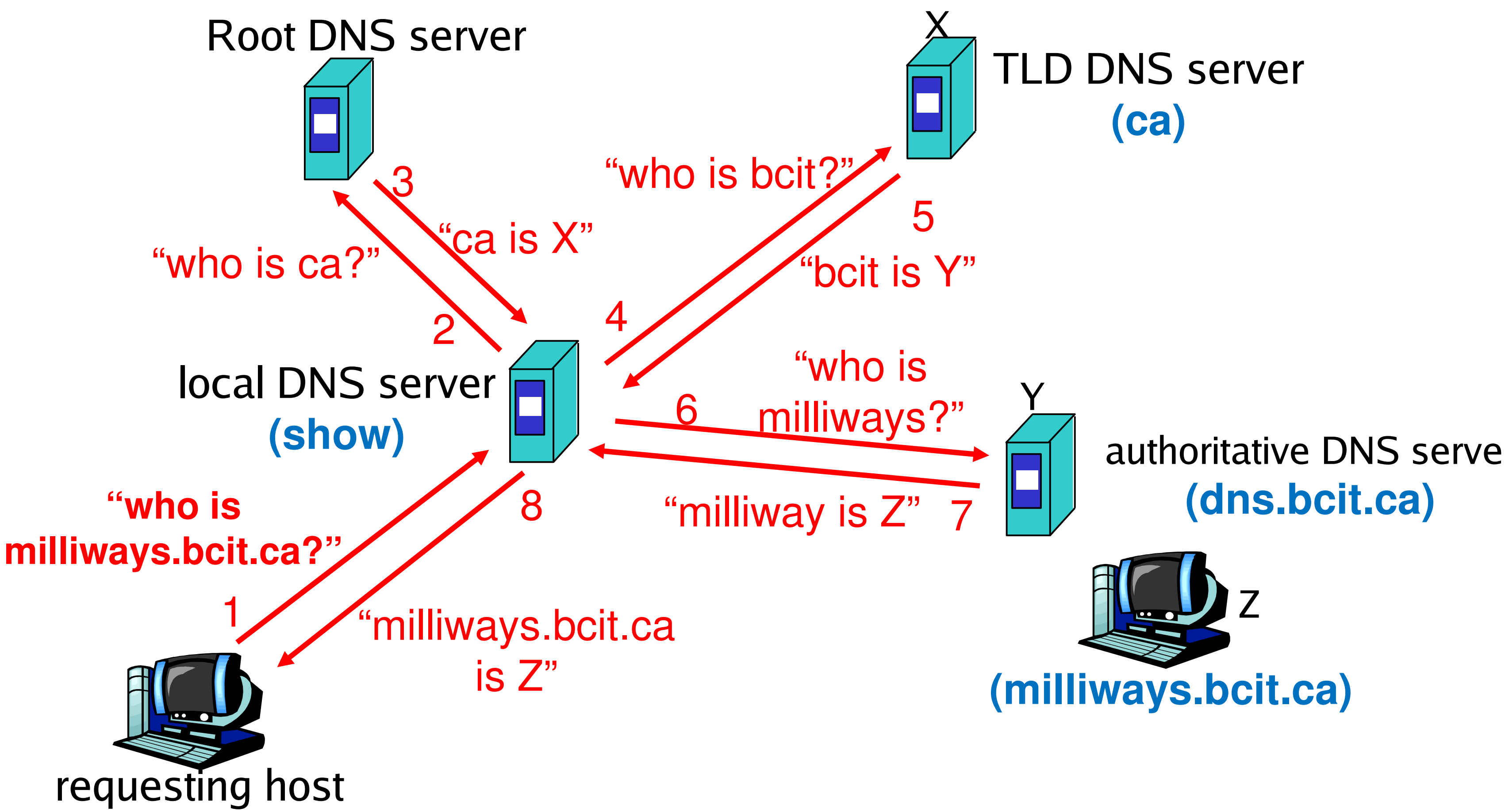
Distributed, Hierarchical Database



- Top-level domain (TLD) servers:
  - Responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp.
  - Network Solutions maintains servers for com TLD
- Authoritative DNS servers:
  - Organization’s DNS servers.
  - Can be maintained by organization or service provider

DNS Name Resolution Example

Host at wants IP address for milliways.bcit.ca

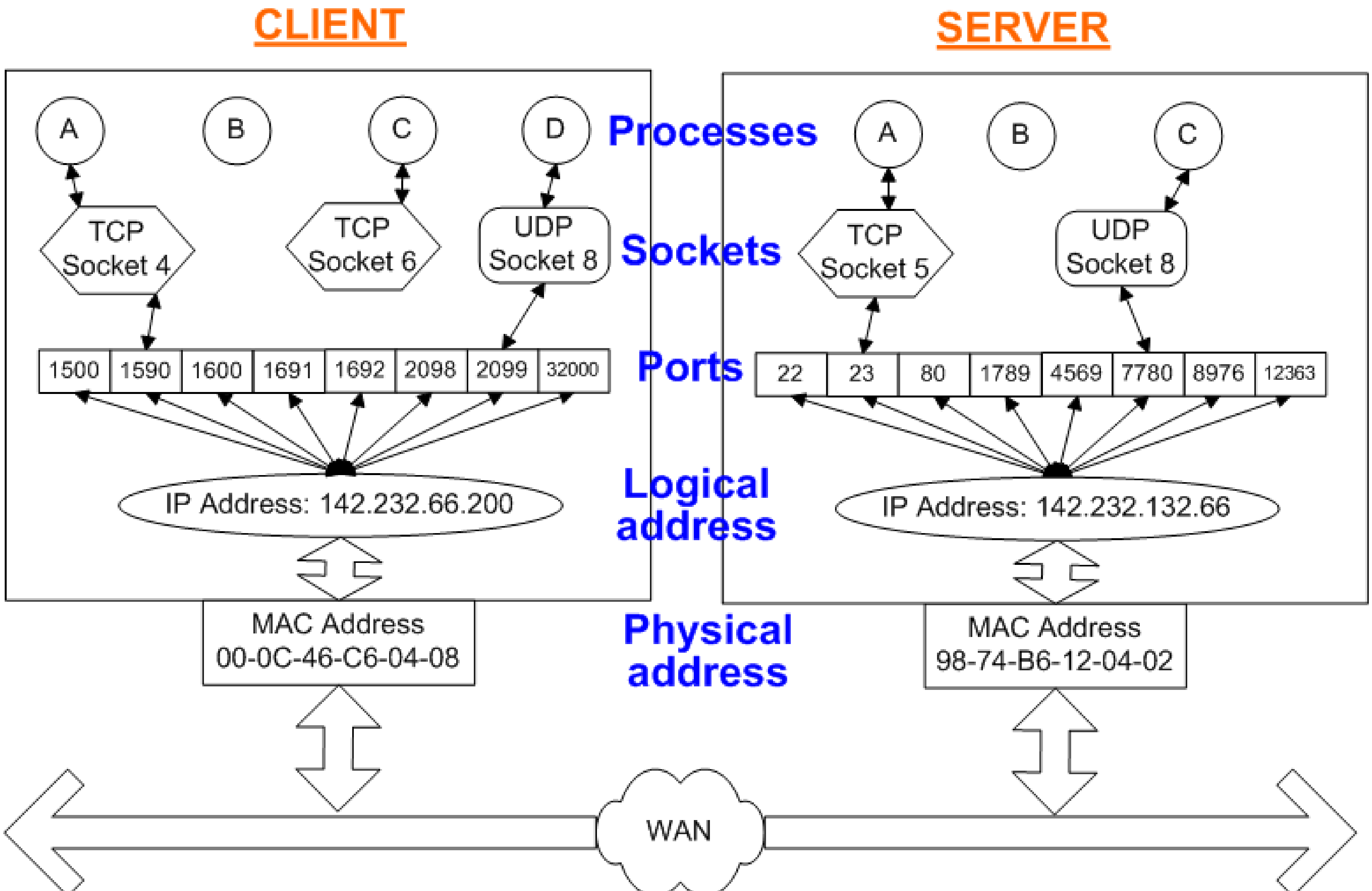


DNS: Caching

- Once (any) name server learns mapping, it *caches* mapping
  - Cache entries timeout (disappear) after some time
  - TLD servers typically cached in local name servers

    => Root name servers not often visited

TCP/IP Client-Server Model



Socket

- Generalization of the **UNIX file access system** designed to incorporate **network protocols**.
- **Communications channel** between a pair of '**sockets**'
- A socket is defined by a group of four integers:
  - **Remote host address**
  - **Remote host port number**
  - **Local host address**
  - **Local host port number**

Create Socket

int socket(family, type, protocol)

- **family (int)**: protocol family

  (e.g. **PF_INET, PF_UNIX)**
- **type (int)**: abstract type of the communication desired

  (e.g. **SOCK_STREAM, SOCK_DGRAM**)
- **protocol (int)**: specific protocol desired

  (e.g. **TCP** or **UDP**)

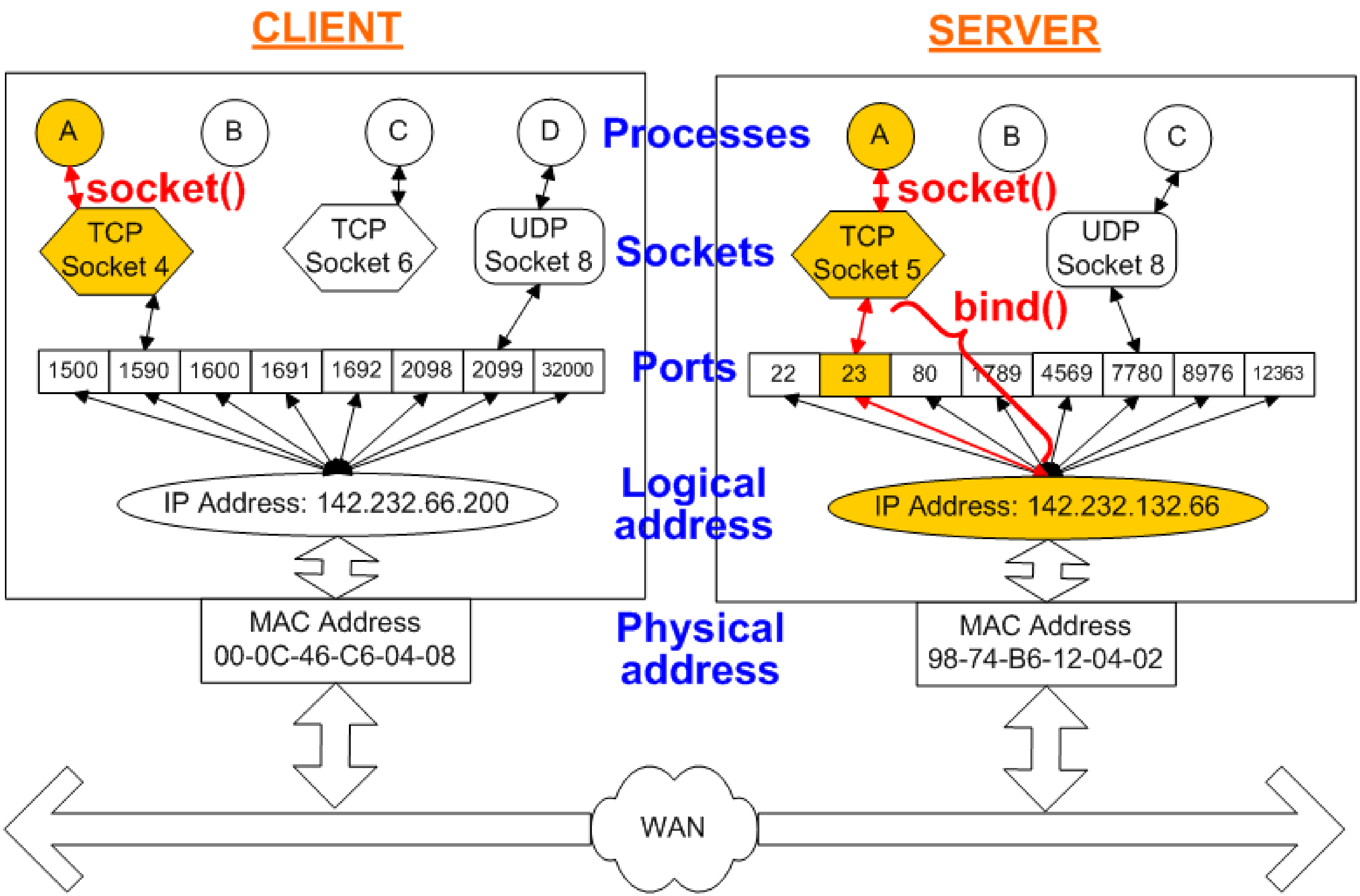
- Returns an **socket file descriptor**

Bind - server

Bind socket to an address

int bind(socket, sockaddr, sockaddrlen)

- **socket (int)**: socket descriptor
- **sockaddr (struct sockaddr *)**: **pointer** to the **address** to which the socket should be bound
- **sockaddrlen (socklen_t)**: the **size** of the **address**
- Returns -1 and sets **errno** for error


CLIENT
SERVER
A
B
C
D
socket()
TCP Socket 4
TCP Socket 6
UDP Socket 8
1500
1590
1600
1691
1692
2098
2099
32000
IP Address: 142.232.66.200
MAC Address 00-0C-46-C6-04-08
Processes
Sockets
Ports
Logical address
Physical address
A
B
C
socket()
TCP Socket 5
UDP Socket 8
bind()
22
23
80
1789
4569
7780
8976
12363
IP Address: 142.232.132.66
MAC Address 98-74-B6-12-04-02
WAN

listen() - server

Marks socket as ‘listening’ and sets the maximum number of listen queue

int listen(sock, backlog)

- **sock (int)**: socket
- **backlog (unsigned int)**: max length of the queue of unprocessed connection requests

Addressing

- **common framework** for all addresses to support multiple protocol families
- In the **Internet family**, transport addresses are **6 bytes** long:
  - **4 bytes** for the **Internet address**
  - **2 bytes** for the **port number**.
- **<netinet/in.h>** defines the appropriate structures to use for the Internet family

```
struct sockaddr_in
{
   short  sin_family;
   unsigned short  sin_port;
   struct in_addr  sin_addr;
   unsigned char sin_zero[8];   /* unused */
}
```

sin_family: address family
sin_port: 16-bit port number
sin_addr: 32-bit Internet address

```
struct in_addr
{
   u_long  s_addr;             /* 32-bit IP Address */
}
```

Get Server Information

gethostbyname(host_name)

- Maps the request into a **DNS** query that it sends to the **resolver** running on the local machine.
- Returns a **pointer** to a **hostent structure** that contains the requested addresses.
- The related function **gethostbyaddr()** takes an **Internet address** as an argument instead.

```
struct hostent
{
  char *h_name;
  char **h_alias;
  int h_addrtype;
  int h_length;
  char **h_addr_list;
};
```

h_name: the official, fully qualified **domain name**

h_alias: a null-terminated list of alternate names **(aliases)**

h_addrtype: **protocol family** the address belongs to

h_length: **length** of the address

h_addr_list: a null-terminated **array of addresses**

Address Manipulation Functions

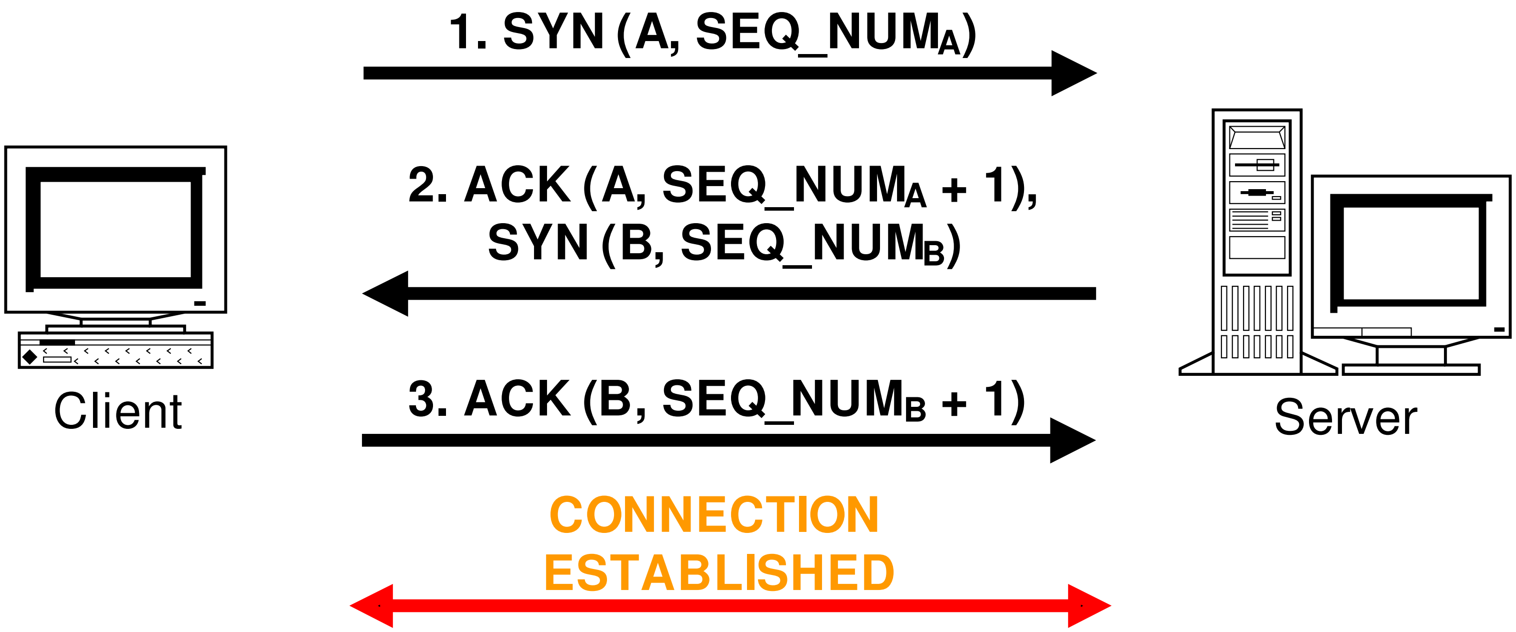
inet_aton() / inet_network()

- Takes strings representing Internet addresses in **dotted notation**
- Returns numbers suitable for use in **sockaddr_in** structures.

inet_ntoa()

- Takes a 32-bit IP address in **network byte order** Internet address
- Returns the corresponding address in dotted-decimal notation.

Establish a Connection to Server



TCP 3-Way Handshake

Send Connect Request (SYN)

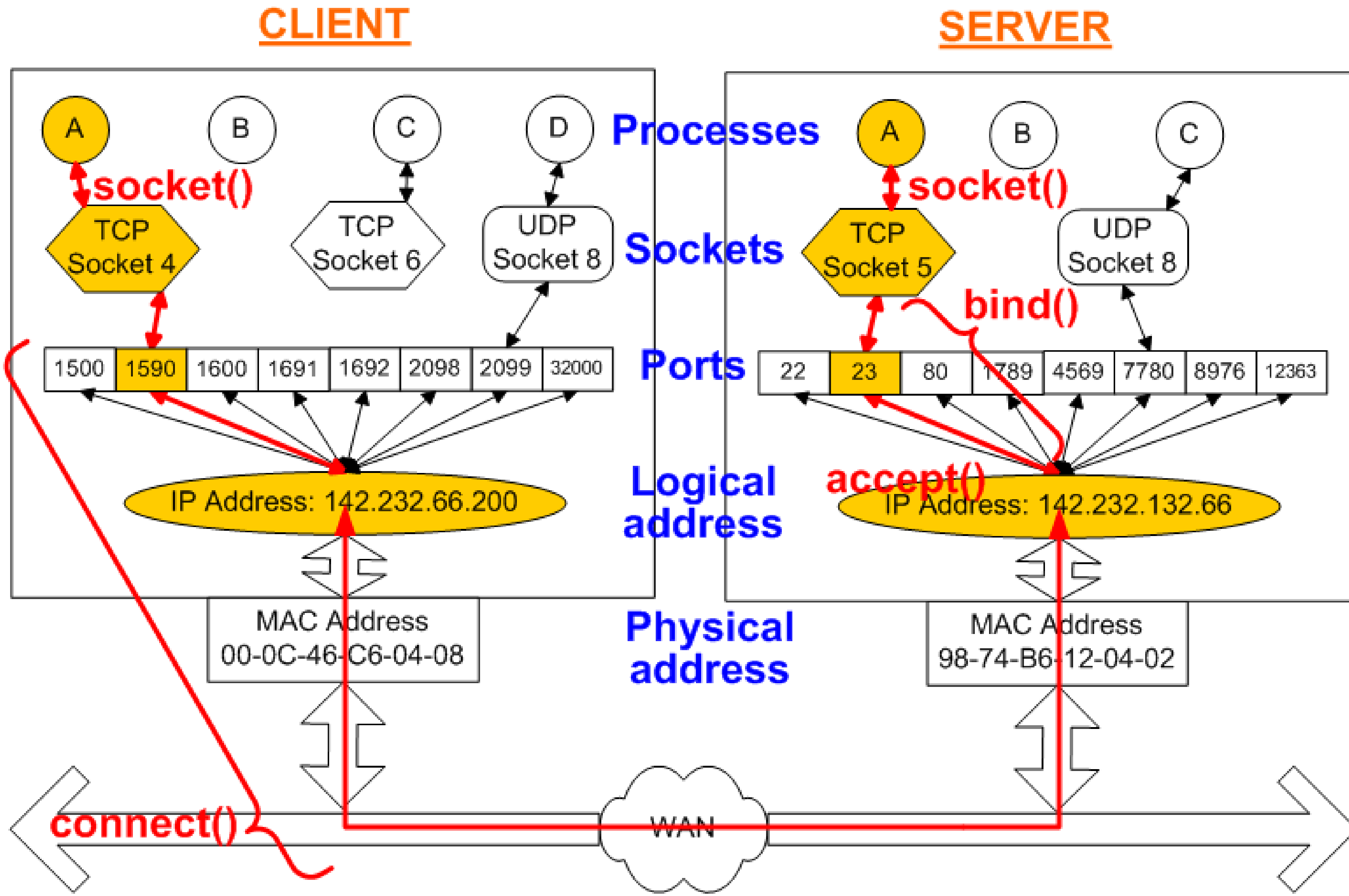
```
connect( int sock,
         struct sockaddr *name,
         int namelen )
```

- Establishes communication with a remote entity
  - initiates a **three-way handshake** for **TCP** connection
  - normally only used for **SOCK_STREAM** sockets.
- **sock**: client socket
- **name**: points to an area of memory containing the address information for the remote entity
- **namelen**: length of ‘name’
- Returns -1 and sets **errno** for error

Accept Connect Request (SYN/ACK)

```
accept ( int sock,
         struct sockaddr *addr,
         int *addrlen)
```

- **sock**: base socket descriptor
- **addr**: points to a struct sockaddr of the remote (client) system.
- **addrlen**: size of addr
- Returns a new socket descriptor (int) that will be used for all subsequent communication with the remote host.


CLIENT
SERVER
A
B
C
D
socket()
TCP Socket 4
TCP Socket 6
UDP Socket 8
1500
1590
1600
1691
1692
2098
2099
32000
IP Address: 142.232.66.200
MAC Address 00-0C-46-C6-04-08
connect()
Processes
Sockets
Ports
Logical address
Physical address
WAN
A
B
C
socket()
TCP Socket 5
UDP Socket 8
bind()
22
23
80
1789
4569
7780
8976
12363
accept()
IP Address: 142.232.132.66
MAC Address 98-74-B6-12-04-02

Send/Receive Data

send(sock, message, length, flags)
recv(sock, message, length, flags)

- **sock**: socket descriptor
- **message**: a pointer to the data to be sent/received
- **length**: data length.
- **flags**: to send **flags** to the **underlying protocol**.
  - flags may be formed by ORing MSG_OOB and MSG_DONTROUTE.
- send() and recv() may only be used with **SOCK_STREAM** type sockets.
- Returns the number of bytes transmitted or -1 for error

Send/Receive Data - UDP

sendto(sock, message, length, flags, dest, destlength)

recvfrom(sock, message, length, flags, from, fromlength)

- The two additional arguments are **pointers** to a **sockaddr** structure and its **length**.

Close Connection

shutdown (socket, how)

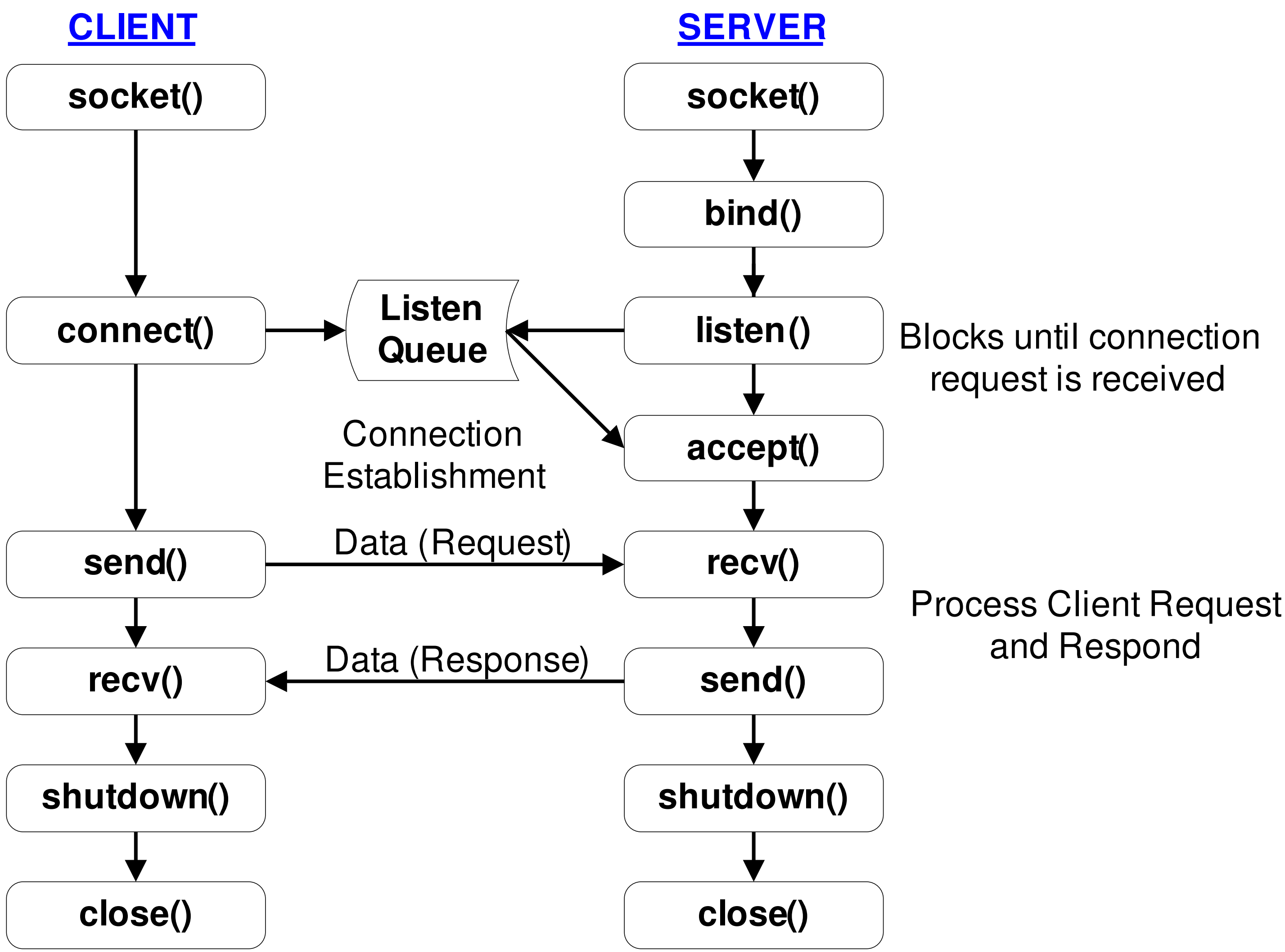
to shut down all or part of a full-duplex connection

- **how**: **direction** of the connection to be shutdown:
  - 0 - additional receives are shutdown.
  - 1 - additional sends are disallowed.
  - 2 - additional sends and receives will be disabled.

close(socket)

to close a socket

TCP/IP Client Server Model



UDP Client Server Model