

# 15-213

*“The course that gives CMU its Zip!”*

## Network Programming

### Nov 11, 2003

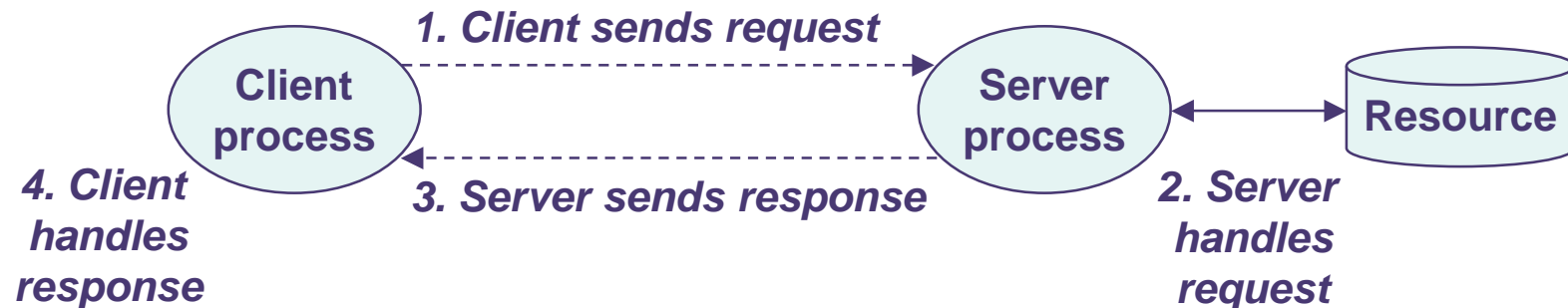
#### Topics

- Programmer’s view of the Internet (review)
- Sockets interface
- Writing clients and servers

# A Client-Server Transaction

Most network applications are 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).*

# A Programmer's View of the Internet

1. Hosts are mapped to a set of 32-bit *IP addresses*.

- 128.2.203.179

2. The set of IP addresses is mapped to a set of identifiers called Internet *domain names*.

- 128.2.203.179 is mapped to [www.cs.cmu.edu](http://www.cs.cmu.edu)

3. A process on one Internet host can communicate with a process on another Internet host over a *connection*.

# 1. IP Addresses

32-bit IP addresses are stored in an *IP address struct*

- IP addresses are always stored in memory in network byte order (big-endian byte order)
- True in general for any integer transferred in a packet header from one machine to another.
  - E.g., the port number used to identify an Internet connection.

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
};
```

Handy network byte-order conversion functions:

htonl: convert long int from host to network byte order.

htons: convert short int from host to network byte order.

ntohl: convert long int from network to host byte order.

ntohs: convert short int from network to host byte order.

## 2. Domain Naming System (DNS)

The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called *DNS*.

- Conceptually, programmers can view the DNS database as a collection of millions of *host entry structures*:

```
/* DNS host entry structure */
struct hostent {
    char    *h_name;          /* official domain name of host */
    char    **h_aliases;     /* null-terminated array of domain names */
    int     h_addrtype;      /* host address type (AF_INET) */
    int     h_length;        /* length of an address, in bytes */
    char    **h_addr_list;   /* null-terminated array of in_addr structs */
};
```

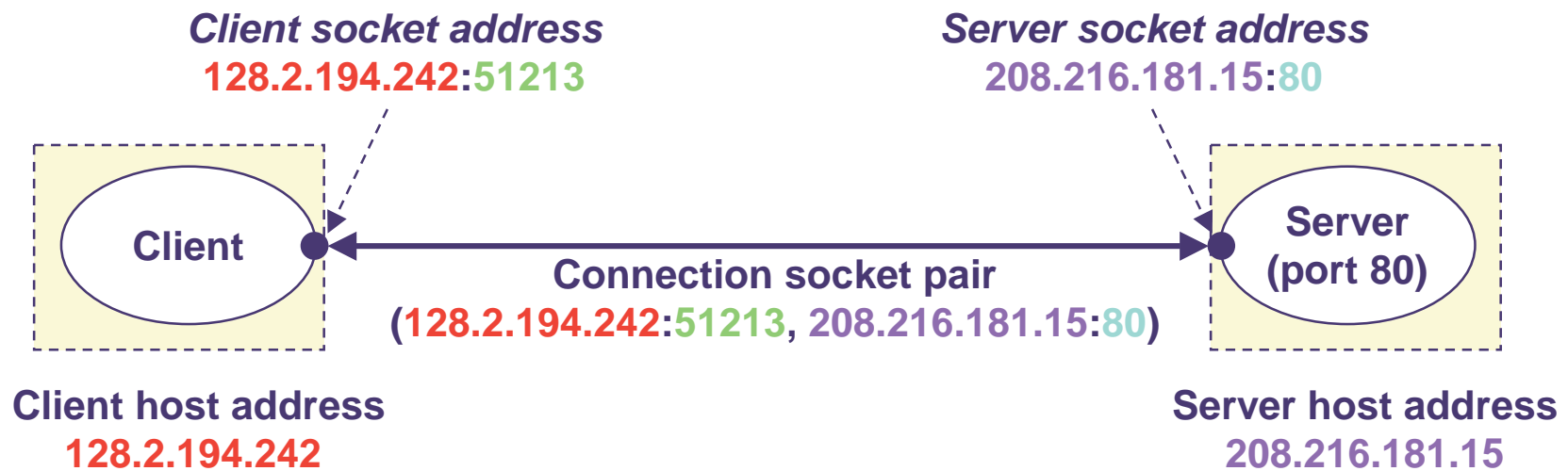
### Functions for retrieving host entries from DNS:

- `gethostbyname`: query key is a DNS domain name.
- `gethostbyaddr`: query key is an IP address.

# 3. Internet Connections

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

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



*Note: 51213 is an ephemeral port allocated by the kernel*

*Note: 80 is a well-known port associated with Web servers*  
15-213, F'03

# Clients

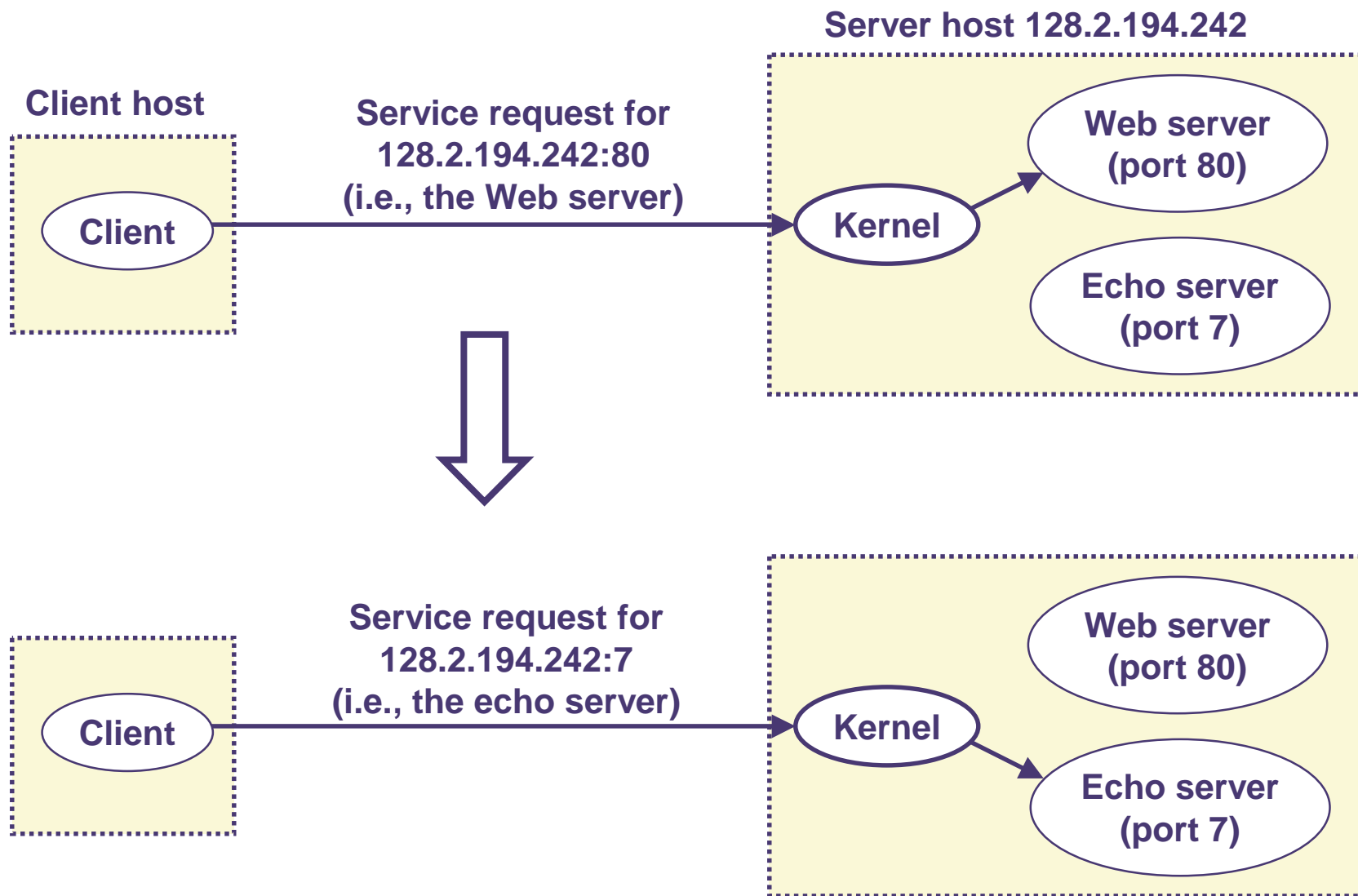
## Examples of client programs

- 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 know ports
  - Port 7: Echo server
  - Port 23: Telnet server
  - Port 25: Mail server
  - Port 80: Web server

# Using Ports to Identify Services





# 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.”**

# Server 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

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

## 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

# Sockets Interface

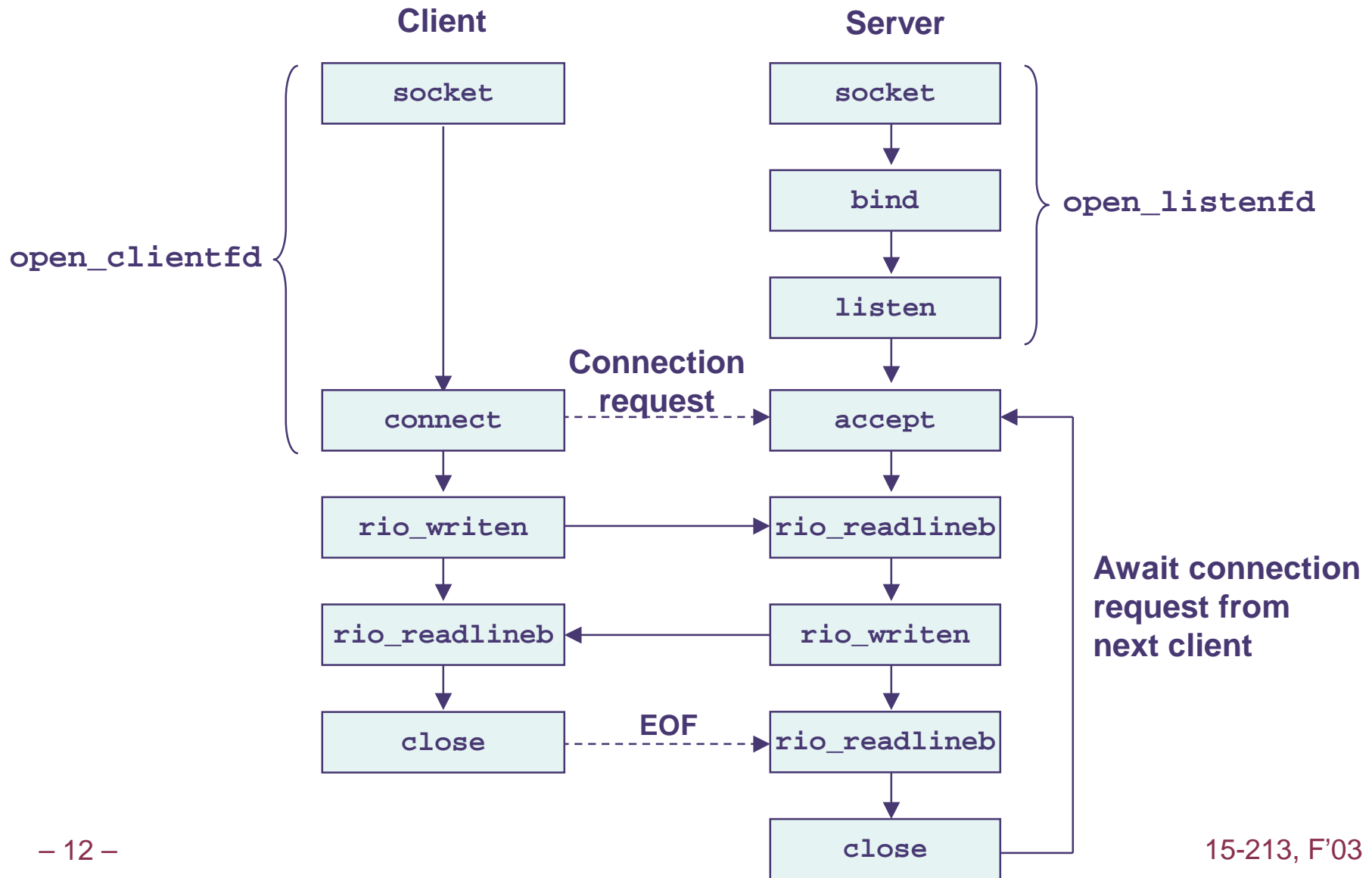
Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.

Provides a user-level interface to the network.

Underlying basis for all Internet applications.

Based on client/server programming model.

# Overview of the Sockets Interface



# Sockets

## What is a socket?

- To the kernel, a socket is an endpoint of communication.
- To an application, a socket is a file descriptor that lets the application read/write from/to the network.
  - Remember: All Unix I/O devices, including networks, are modeled as files.

**Clients and servers communicate with each other by reading from and writing to socket descriptors.**

**The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors.**

# Socket Address Structures

## Generic socket address:

- For address arguments to `connect`, `bind`, and `accept`.
- Necessary only because C did not have generic (`void *`) pointers when the sockets interface was designed.

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

## Internet-specific socket address:

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

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

# Echo Client Main Routine

```
#include "csapp.h"

/* usage: ./echoclient host port */
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];
    rio_t rio;

    host = argv[1];
    port = atoi(argv[2]);

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

# Echo Client: `open_clientfd`

```
int open_clientfd(char *hostname, int port)
{
    int clientfd;
    struct hostent *hp;
    struct sockaddr_in serveraddr;

    if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1; /* check errno for cause of error */

    /* Fill in the server's IP address and port */
    if ((hp = gethostbyname(hostname)) == NULL)
        return -2; /* check h_errno for cause of error */
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    bcopy((char *)hp->h_addr,
          (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
    serveraddr.sin_port = htons(port);

    /* Establish a connection with the server */
    if (connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr)) < 0)
        return -1;
    return clientfd;
}
```

This function opens a connection from the client to the server at `hostname:port`



# Echo Client: `open_clientfd` (`socket`)

`socket` creates a socket descriptor on the client.

- `AF_INET`: indicates that the socket is associated with Internet protocols.
- `SOCK_STREAM`: selects a reliable byte stream connection.

```
int clientfd; /* socket descriptor */  
  
if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)  
    return -1; /* check errno for cause of error */  
  
... (more)
```

# Echo Client: `open_clientfd` (`gethostbyname`)

The client then builds the server's Internet address.

```
int clientfd;                /* socket descriptor */
struct hostent *hp;         /* DNS host entry */
struct sockaddr_in serveraddr; /* server's IP address */

...

/* fill in the server's IP address and port */
if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
bcopy((char *)hp->h_addr,
      (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
serveraddr.sin_port = htons(port);
```

# Echo Client: `open_clientfd` (`connect`)

Finally the client creates a connection with the server.

- Client process suspends (blocks) until the connection is created.
- After resuming, the client is ready to begin exchanging messages with the server via Unix I/O calls on descriptor `clientfd`.

```
int clientfd;           /* socket descriptor */
struct sockaddr_in serveraddr; /* server address */
typedef struct sockaddr SA; /* generic sockaddr */
...
/* Establish a connection with the server */
if (connect(clientfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
    return -1;
return clientfd;
}
```

# Echo Server: Main Routine

```
int main(int argc, char **argv) {
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    struct hostent *hp;
    char *haddrp;

    port = atoi(argv[1]); /* the server listens on a port passed
                           on the command line */
    listenfd = open_listenfd(port);

    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                          sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        haddrp = inet_ntoa(clientaddr.sin_addr);
        printf("server connected to %s (%s)\n", hp->h_name, haddrp);
        echo(connfd);
        Close(connfd);
    }
}
```

# Echo Server: `open_listenfd`

```
int open_listenfd(int port)
{
    int listenfd, optval=1;
    struct sockaddr_in serveraddr;

    /* Create a socket descriptor */
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    /* Eliminates "Address already in use" error from bind. */
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int)) < 0)
        return -1;

    ... (more)
```

# Echo Server: `open_listenfd` (cont)

...

```
/* Listenfd will be an endpoint for all requests to port
   on any IP address for this host */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port = htons((unsigned short)port);
if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
    return -1;

/* Make it a listening socket ready to accept
   connection requests */
if (listen(listenfd, LISTENQ) < 0)
    return -1;

return listenfd;
}
```

# Echo Server: `open_listenfd` (`socket`)

`socket` creates a socket descriptor on the server.

- `AF_INET`: indicates that the socket is associated with Internet protocols.
- `SOCK_STREAM`: selects a reliable byte stream connection.

```
int listenfd; /* listening socket descriptor */

/* Create a socket descriptor */
if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1;
```

# Echo Server: `open_listenfd` (`setsockopt`)

The socket can be given some attributes.

```
...
/* Eliminates "Address already in use" error from bind(). */
if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
               (const void *)&optval , sizeof(int)) < 0)
    return -1;
```

Handy trick that allows us to rerun the server immediately after we kill it.

- Otherwise we would have to wait about 15 secs.
- Eliminates “Address already in use” error from `bind()`.

Strongly suggest you do this for all your servers to simplify debugging.



# Echo Server: `open_listenfd` (initialize socket address)

Next, we initialize the socket with the server's Internet address (IP address and port)

```
struct sockaddr_in serveraddr; /* server's socket addr */
...
/* listenfd will be an endpoint for all requests to port
   on any IP address for this host */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port = htons((unsigned short)port);
```

**IP addr and port stored in network (big-endian) byte order**

- `htonl()` converts longs from host byte order to network byte order.
- `htons()` converts shorts from host byte order to network byte order.

# Echo Server: `open_listenfd` (`bind`)

`bind` associates the socket with the socket address we just created.

```
int listenfd;          /* listening socket */
struct sockaddr_in serveraddr; /* server's socket addr */

...
/* listenfd will be an endpoint for all requests to port
   on any IP address for this host */
if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
    return -1;
```

# Echo Server: `open_listenfd` (`listen`)

`listen` indicates that this socket will accept connection (`connect`) requests from clients.

```
int listenfd; /* listening socket */  
  
...  
/* Make it a listening socket ready to accept connection requests */  
if (listen(listenfd, LISTENQ) < 0)  
    return -1;  
return listenfd;  
}
```

We're finally ready to enter the main server loop that accepts and processes client connection requests.

# Echo Server: Main Loop

The server loops endlessly, waiting for connection requests, then reading input from the client, and echoing the input back to the client.

```
main() {  
  
    /* create and configure the listening socket */  
  
    while(1) {  
        /* Accept(): wait for a connection request */  
        /* echo(): read and echo input lines from client til EOF */  
        /* Close(): close the connection */  
    }  
}
```

# Echo Server: accept

`accept ( )` blocks waiting for a connection request.

```
int listenfd; /* listening descriptor */
int connfd;   /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
```

`accept` returns a **connected descriptor** (`connfd`) with the same properties as the **listening descriptor** (`listenfd`)

- Returns when the connection between client and server is created and ready for I/O transfers.
- All I/O with the client will be done via the connected socket.

`accept` also fills in client's IP address.

# Echo Server: `accept` Illustrated



1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`.



2. Client makes connection request by calling and blocking in `connect`.



3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`.

# Connected vs. Listening Descriptors

## Listening descriptor

- End point for client connection requests.
- Created once and exists for lifetime of the server.

## Connected descriptor

- End point of the connection between client and server.
- A new descriptor is created each time the server accepts a connection request from a client.
- Exists only as long as it takes to service client.

## Why the distinction?

- Allows for concurrent servers that can communicate over many client connections simultaneously.
  - E.g., Each time we receive a new request, we fork a child to handle the request.

# Echo Server: Identifying the Client

The server can determine the domain name and IP address of the client.

```
struct hostent *hp; /* pointer to DNS host entry */
char *haddrp;      /* pointer to dotted decimal string */

hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                  sizeof(clientaddr.sin_addr.s_addr), AF_INET);
haddrp = inet_ntoa(clientaddr.sin_addr);
printf("server connected to %s (%s)\n", hp->h_name, haddrp);
```



# Echo Server: echo

The server uses RIO to read and echo text lines until EOF (end-of-file) is encountered.

- EOF notification caused by client calling `close(clientfd)`.
- **IMPORTANT:** EOF is a condition, not a particular data byte.

```
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", n);
        Rio_writen(connfd, buf, n);
    }
}
```

# Testing Servers Using `telnet`

The `telnet` program is invaluable for testing servers that transmit ASCII strings over Internet connections

- Our simple echo server
- Web servers
- Mail servers

## Usage:

- `unix> telnet <host> <portnumber>`
- Creates a connection with a server running on `<host>` and listening on port `<portnumber>`.

# Testing the Echo Server With telnet

```
bass> echoserver 5000
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 5 bytes: 123
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 8 bytes: 456789

kittyhawk> telnet bass 5000
Trying 128.2.222.85...
Connected to BASS.CMCL.CS.CMU.EDU.
Escape character is '^]'.
123
123
Connection closed by foreign host.
kittyhawk> telnet bass 5000
Trying 128.2.222.85...
Connected to BASS.CMCL.CS.CMU.EDU.
Escape character is '^]'.
456789
456789
Connection closed by foreign host.
kittyhawk>
```

# Running the Echo Client and Server

```
bass> echoserver 5000
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 4 bytes: 123
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 7 bytes: 456789
...

kittyhawk> echoclient bass 5000
Please enter msg: 123
Echo from server: 123

kittyhawk> echoclient bass 5000
Please enter msg: 456789
Echo from server: 456789
kittyhawk>
```

# For More Information

**W. Richard Stevens, “Unix Network Programming: Networking APIs: Sockets and XTI”, Volume 1, Second Edition, Prentice Hall, 1998.**

- **THE network programming bible.**

**Complete versions of the echo client and server are developed in the text.**

- **Available from `csapp.cs.cmu.edu`**
- **You should compile and run them for yourselves to see how they work.**
- **Feel free to borrow any of this code.**