

15-213

*"The course that gives CMU its Zip!"*

## Network programming

Nov 27, 2001

### Topics

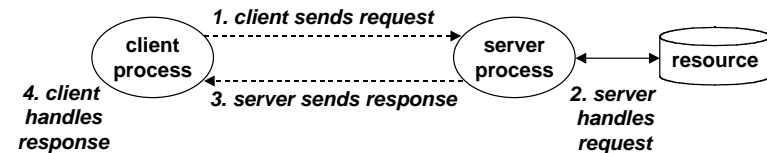
- Client-server model
- Sockets interface
- Echo client and server

class26.ppt

## Client-server model

Every network application is based on the client-server model:

- Application is a *server process* and one or more *client processes*
- Server manages some *resource*, and provides *service* by manipulating resource for *clients*.
- Client makes a request for a service
  - request may involve a conversation according to some server protocol
- Server provides service by manipulating the resource on behalf of client and then returning a response



class26.ppt

- 2 -

CS 213 F'01

## Clients

### Examples of client programs

- Web browsers, ftp, telnet, ssh

### How does the client find the server?

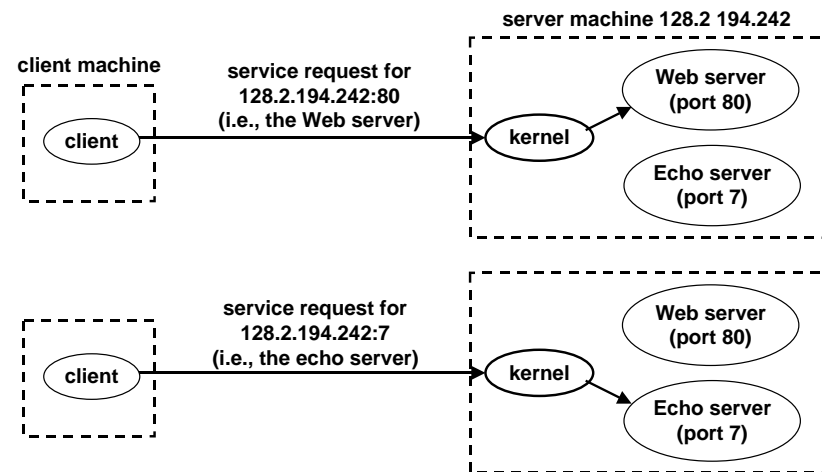
- The address of the server process has two parts: *IP address:port*
  - The *IP address* is a unique 32-bit positive integer that identifies the host (adapter).
    - » dotted decimal form:  $0x8002C2F2 = 128.2.194.242$
  - The *port* is positive integer associated with a service (and thus a server process) on that machine.
    - » port 7: echo server
    - » port 23: telnet server
    - » port 25: mail server
    - » port 80: web server

class26.ppt

- 3 -

CS 213 F'01

## Using ports to identify services



class26.ppt

- 4 -

CS 213 F'01

# 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 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

Telnet server (23)

- resource: terminal
- service: proxies a terminal on the server machine

Mail server (25)

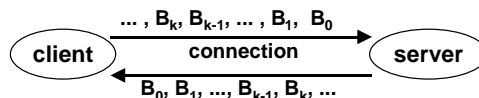
- resource: email “spool” file
- service: stores mail messages in spool file

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

## The two basic ways that clients and servers communicate

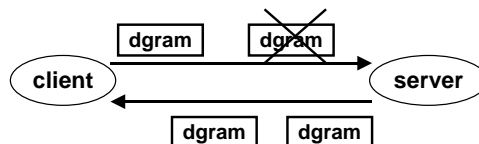
Connections (TCP):

- reliable two-way byte-stream.
- looks like a file.
- akin to placing a phone call.
- slower but more robust.



Datagrams (UDP):

- data transferred in unreliable chunks.
- can be lost or arrive out of order.
- akin to using surface mail.
- faster but less robust.



We will only discuss connections.

## Internet connections (review)

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

- point-to-point, full-duplex, and reliable.

A *socket* is an endpoint of a connection

- *Socket address* is an IPaddress:port pair

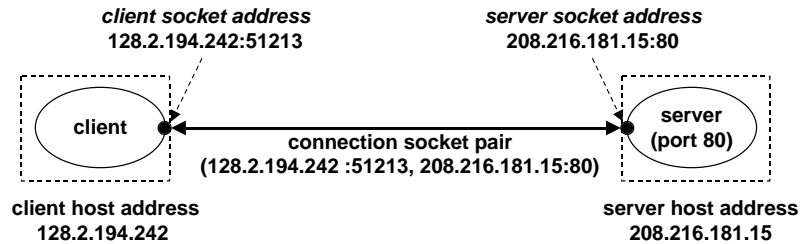
A *port* is a 16-bit integer that identifies a process:

- *ephemeral port*: assigned automatically on client when client makes a connection request
- *well-known port*: associated with some service provided by a server (e.g., port 80 is associated with Web servers)

A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)

- (cliaddr:cliport, servaddr:servport)

## Anatomy of an Internet connection (review)



class26.ppt

- 9 -

CS 213 F'01

## Berkeley 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.

class26.ppt

- 10 -

CS 213 F'01

## What is a socket?

A *socket* is a descriptor that lets an application read/write from/to the network.

- Key idea: Unix uses the same abstraction for both file I/O and network I/O.

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

- Using regular Unix read and write I/O functions.

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

class26.ppt

- 11 -

CS 213 F'01

## Key data structures

Defined in `/usr/include/netinet/in.h`

```
/* Internet address */
struct in_addr {
    unsigned int s_addr; /* 32-bit IP address */
};

/* Internet style socket address */
struct sockaddr_in {
    unsigned short int sin_family; /* Address family (AF_INET) */
    unsigned short int sin_port; /* Port number */
    struct in_addr sin_addr; /* IP address */
    unsigned char sin_zero[...]; /* Pad to sizeof "struct sockaddr" */
};
```

Internet-style sockets are characterized by a 32-bit IP address and a port.

class26.ppt

- 12 -

CS 213 F'01

## Key data structures

Defined in `/usr/include/netdb.h`

```
/* Domain Name Service (DNS) host entry */
struct hostent {
    char    *h_name;          /* official name of host */
    char    **h_aliases;      /* alias list */
    int     h_addrtype;        /* host address type */
    int     h_length;          /* length of address */
    char    **h_addr_list;    /* list of addresses */
}
```

**hostent** is a DNS host entry that associates a *domain name* (e.g., `cmu.edu`) with an IP addr (`128.2.35.186`)

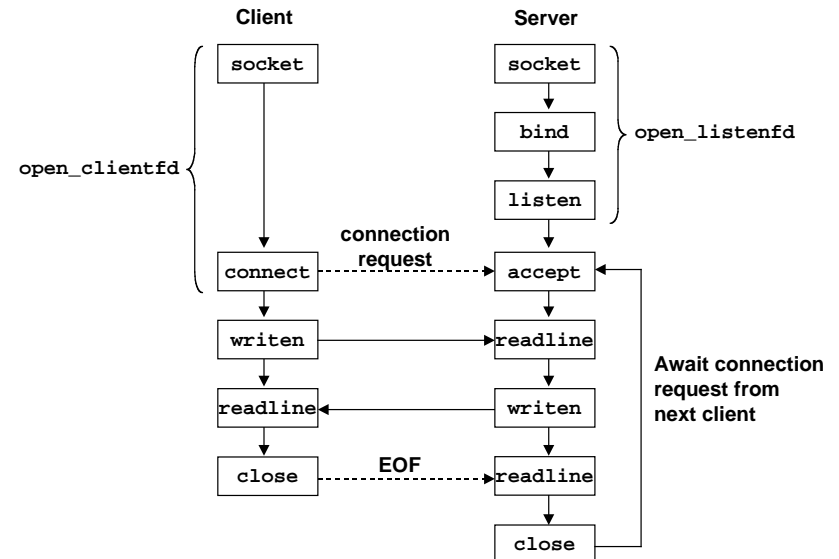
- Can be accessed from user programs
  - `gethostbyname()` [domain name key]
  - `gethostbyaddr()` [IP address key]
- Can also be accessed from the shell using `nslookup` or `dig`.

class26.ppt

– 13 –

CS 213 F'01

## Overview of the Sockets Interface



class26.ppt

– 14 –

CS 213 F'01

## Echo client

```
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];

    if (argc != 3) {
        fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
        exit(0);
    }
    host = argv[1];
    port = atoi(argv[2]);

    clientfd = open_clientfd(host, port);
    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Writen(clientfd, buf, strlen(buf));
        Readline(clientfd, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
}
```

class26.ppt

– 15 –

CS 213 F'01

## Echo client: `open_clientfd()`

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

    clientfd = Socket(AF_INET, SOCK_STREAM, 0);

    /* fill in the server's IP address and port */
    hp = Gethostbyname(hostname);
    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 */
    Connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr));

    return clientfd;
}
```

class26.ppt

– 16 –

CS 213 F'01

## Echo client: open\_clientfd() (socket)

The client creates a socket that will serve as the endpoint of an Internet (AF\_INET) connection (SOCK\_STREAM).

- socket() returns an integer socket descriptor.

```
int clientfd; /* socket descriptor */
clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

class26.ppt

- 17 -

CS 213 F'01

## Echo client: open\_clientfd() (gethostbyname)

The client builds the server's Internet address.

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

typedef struct sockaddr SA; /* generic sockaddr */

...

/* fill in the server's IP address and port */
hp = Gethostbyname(hostname);
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);
```

class26.ppt

- 18 -

CS 213 F'01

## Echo client: open\_clientfd() (connect)

Then the client creates a connection with the server

- The client process suspends (blocks) until the connection is created with the server.
- At this point the client is ready to begin exchanging messages with the server via Unix I/O calls on the descriptor sockfd.

```
int clientfd; /* socket descriptor */
struct sockaddr_in serveraddr; /* server address */

...

/* establish a connection with the server */
Connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr));
```

class26.ppt

- 19 -

CS 213 F'01

## Echo server

```
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), AF_INET);
        haddrp = inet_ntoa(clientaddr.sin_addr);
        printf("server connected to %s (%s)\n", hp->h_name, haddrp);
        echo(connfd);
        Close(connfd);
    }
}
```

class26.ppt

- 20 -

CS 213 F'01

## Echo server: open\_listenfd()

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

    /* create a socket descriptor */
    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    /* eliminates "Address already in use" error from bind. */
    optval = 1;
    Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
               (const void *)&optval , sizeof(int));

    ... (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);
    Bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr));

    /* make it a listening socket ready to accept
       connection requests */
    Listen(listenfd, LISTENQ);

    return listenfd;
}
```

## Echo server: open\_listenfd() (socket)

**socket()** creates a socket descriptor.

- **AF\_INET**: indicates that the socket is associated with Internet protocols.
- **SOCK\_STREAM**: selects a reliable byte stream connection.

```
int listenfd; /* listening socket descriptor */
listenfd = Socket(AF_INET, SOCK_STREAM, 0);
```

## Echo server: open\_listenfd() (setsockopt)

The socket can be given some attributes.

```
/* eliminates "Address already in use" error from bind. */
optval = 1;
Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
           (const void *)&optval , sizeof(int));
```

**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 */
Bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr));
```

## Echo server: open\_listenfd (listen)

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

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

/* make listenfd it a server-side listening socket ready to accept
   connection requests from clients */
Listen(listenfd, LISTENQ);
```

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 line from client */
        /* 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;

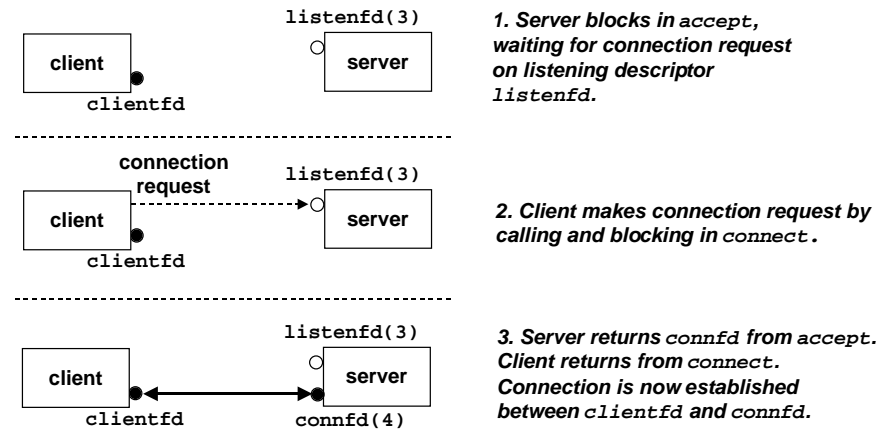
clientlen = sizeof(clientaddr);
connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
```

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

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

`accept ( )` also fills in client's address.

## accept ( ) illustrated



## 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 Unix I/O to read and echo text lines until EOF (end-of-file) is encountered.

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

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

    while((n = Readline(connfd, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", n);
        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 detailed info

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

- This is the network programming bible.

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

- You should compile and run them for yourselves to see how they work.
- Feel free to borrow any of this code.