15-213

"The course that gives CMU its Zip!"

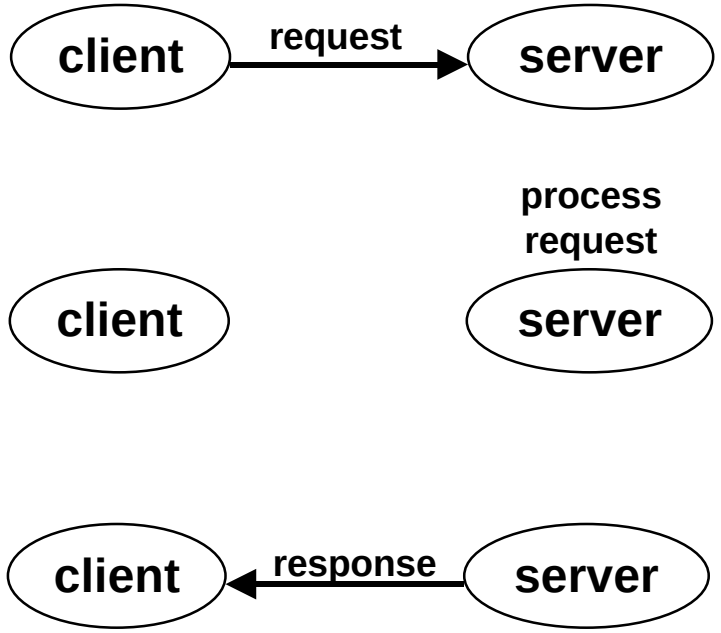
Network programming
Nov 16, 2000

Topics

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

Client-server programming model

- **Client + server = distributed computing**
- **Client & server are both processes**
- **Server manages a resource**
- **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**



Clients

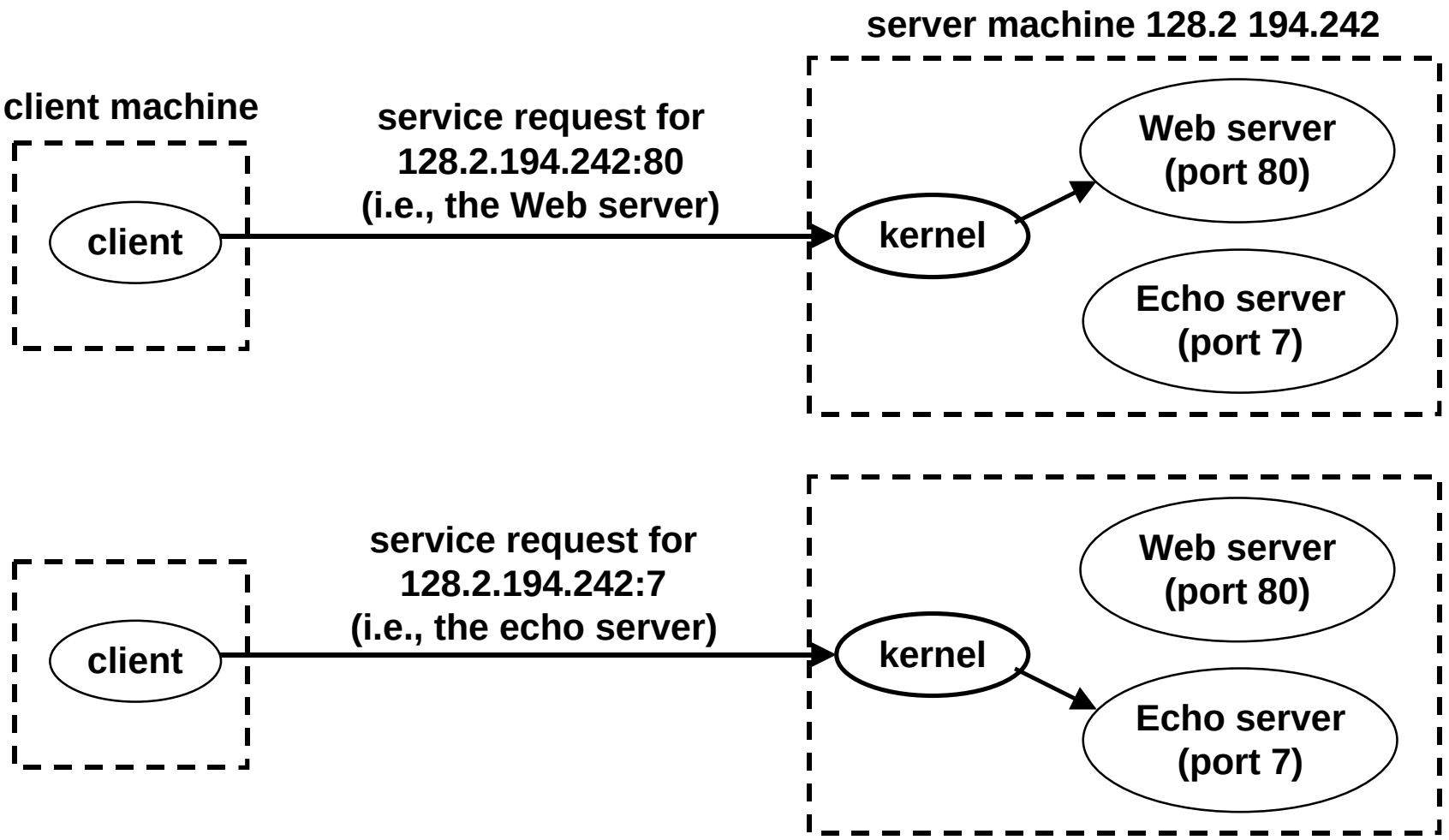
Examples of client programs

- Web browsers, `ftp`, `telnet`, `ssh`

How does the client find the server?

- The address of the server process has two parts: *IPaddress:port*
  - The *IP address* is a unique 32-bit positive integer that identifies the machine.
    - dotted decimal form: `0x8002C2F2 = 128.2.194.242`
  - The *port* is positive integer associated with a service (and thus a server) on that machine.
    - 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 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)

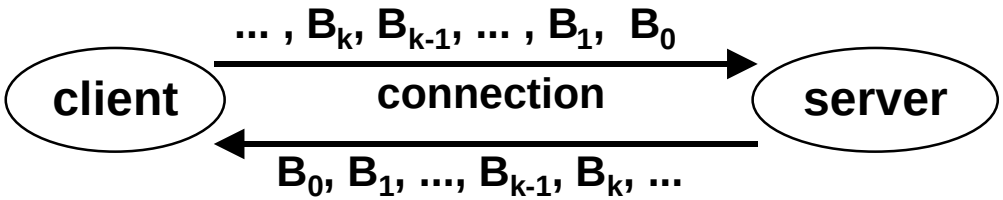
- resource: email “spool” file
- service: stores mail messages in spool file

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

The two basic ways that clients and servers communicate

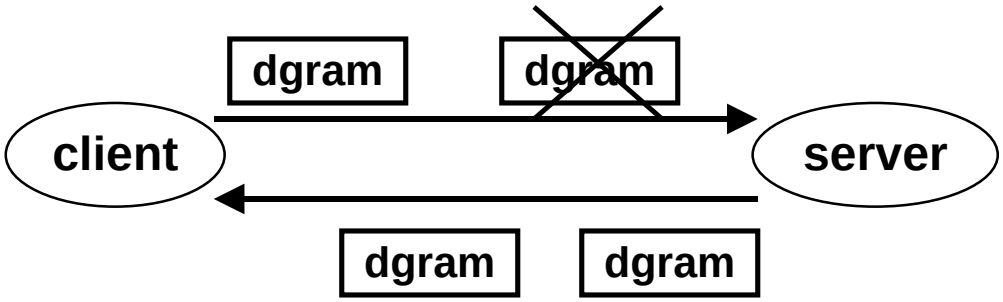
Connections:

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



Datagrams:

- 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.

Linux file I/O: open( )

Must open( ) a file before you can do anything else.

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

open( ) returns a small integer (file descriptor)

- **fd < 0 indicates that an error occurred**

predefined file descriptors:

- **0: stdin**
- **1: stdout**
- **2: stderr**

Linux file I/O: `read()`

`read()` allows a program to access the contents of file.

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* open the file */
/* read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

`read()` returns the number of bytes read from file `fd`.

- **`nbytes` < 0 indicates that an error occurred.**

- **if successful, `read()` places `nbytes` bytes into memory starting at address `buf`**

File I/O: `write()`

`write()` allows a program to modify file contents.

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* open the file */
/* write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

`write()` returns the number of bytes written from `buf` to file `fd`.

- **`nbytes` < 0 indicates that an error occurred.**

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.

What is a socket?

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

- **Key idea: Linux 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 Linux `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.

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.

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)

- DNS (Domain Name Service) is a world-wide distributed database of domain name/IP address mappings.
- Can be accessed from user programs using `gethostbyname()` [domain name to IP address] or `gethostbyaddr()` [IP address to domain name]
- Can also be accessed from the shell using `nslookup` or `dig`.

Echo client: prologue

The client connects to a host and port passed in on the command line.

```
/*
 * error - wrapper for perror
 */
void error(char *msg) {
    perror(msg);
    exit(0);
}

int main(int argc, char **argv) {
    /* local variable definitions */

    /* check command line arguments */
    if (argc != 3) {
       fprintf(stderr,"usage: %s <hostname> <port>\n", argv[0]);
       exit(0);
    }
    hostname = argv[1];
    portno = atoi(argv[2]);
```

Echo client: `socket()`

The client creates a socket that will serve as the endpoint of an Internet (AF_INET) connection (SOCK_STREAM).

```
int sockfd;  /* socket descriptor */

sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    error("ERROR opening socket");
```

`socket()` returns an integer socket descriptor.

- **`sockfd < 0` indicates that an error occurred.**

Echo client: `gethostbyname()`

The client builds the server's Internet address.

```
struct sockaddr_in serveraddr; /* server address */
struct hostent *server;        /* server DNS host entry */
char *hostname;                /* server domain name */

/* gethostbyname: get the server's DNS entry */
server = gethostbyname(hostname);
if (server == NULL) {
    fprintf(stderr,"ERROR, no such host as %s\n", hostname);
    exit(0);
}

/* build the server's Internet address */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
      (char *)&serveraddr.sin_addr.s_addr, server->h_length);
serveraddr.sin_port = htons(portno);
```

Echo client: `connect()`

Then the client creates a connection with the server.

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

if (connect(sockfd, &serveraddr, sizeof(serveraddr)) < 0)
      error("ERROR connecting");
```

At this point the client is ready to begin exchanging messages with the server via `sockfd`.

Echo client: `read()`, `write()`, `close()`

The client reads a message from stdin, sends it to the server, waits for the echo, and terminates.

```
    /* get message line from the user */
    printf("Please enter msg: ");
    bzero(buf, BUFSIZE);
    fgets(buf, BUFSIZE, stdin);

    /* send the message line to the server */
    n = write(sockfd, buf, strlen(buf));
    if (n < 0)
      error("ERROR writing to socket");

    /* print the server's reply */
    bzero(buf, BUFSIZE);
    n = read(sockfd, buf, BUFSIZE);
    if (n < 0)
      error("ERROR reading from socket");
    printf("Echo from server: %s", buf);
    close(sockfd);
    return 0;
```

Echo server: prologue

The server listens on a port passed via the command line.

```
/*
 * error - wrapper for perror
 */
void error(char *msg) {
  perror(msg);
  exit(1);
}

int main(int argc, char **argv) {
  /* local variable definitions */

  /*
   * check command line arguments
   */
  if (argc != 2) {
    fprintf(stderr, "usage: %s <port>\n", argv[0]);
    exit(1);
  }
  portno = atoi(argv[1]);
```

Echo server: `socket()`

`socket()` creates a socket.

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

listenfd = socket(AF_INET, SOCK_STREAM, 0);
  if (listenfd < 0)
    error("ERROR opening socket");
```

`socket()` returns an integer socket descriptor.

- **`listenfd` < 0 indicates that an error occurred.**

AF_INET: indicates that the socket is associated with Internet protocols.

SOCK_STREAM: selects a reliable byte stream connection.

Echo server: `setsockopt()`

The socket can be given some attributes.

```
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 would have to wait about 15 secs.**
- **eliminates “Address already in use” error.**
- **Strongly suggest you do this for all your servers to simplify debugging.**

Echo server: init socket address

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

```
struct sockaddr_in serveraddr; /* server's addr */

/* this is an Internet address */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;

/* a client can connect to any of my IP addresses */
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

/* this is the port to associate the socket with */
serveraddr.sin_port = htons((unsigned short)portno);
```

Binary numbers must be stored in *network byte order* (big-endien)

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

TCP echo server: `bind()`

`bind()` associates the socket with a port.

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


if (bind(listenfd, (struct sockaddr *) &serveraddr,
         sizeof(serveraddr)) < 0)
    error("ERROR on binding");
```

Echo server: `listen()`

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

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

if (listen(listenfd, 5) < 0) /* allow 5 requests to queue up */
    error("ERROR on listen");
```

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 */
      /* read(): read an input line from the client */
      /* write(): echo the line back to the client */
      /* close(): close the connection */
   }
}
```

Echo server: `accept()`

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

```
int listenfd;  /* listening socket */
int connfd;    /* connection socket */
int clientlen; /* byte size of client's address */
struct sockaddr_in clientaddr; /* client addr */

clientlen = sizeof(clientaddr);
connfd = accept(listenfd,
                (struct sockaddr *) &clientaddr, &clientlen);
if (connfd < 0)
   error("ERROR on accept");
```

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

- **all I/O with the client will be done via the connection socket.**
- **useful for concurrent servers where parent creates a new process or thread for each connection request.**

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

Echo server: identifying client

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

```
struct sockaddr_in clientaddr; /* client addr */
struct hostent *hostp;         /* client DNS host entry */
char *hostaddrp;               /* dotted decimal host addr string */


hostp = gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                      sizeof(clientaddr.sin_addr.s_addr), AF_INET);
if (hostp == NULL)
  error("ERROR on gethostbyaddr");
hostaddrp = inet_ntoa(clientaddr.sin_addr);
if (hostaddrp == NULL)
  error("ERROR on inet_ntoa\n");
printf("server established connection with %s (%s)\n",
          hostp->h_name, hostaddrp);
```

Echo server: `read()`

The server reads an ASCII input line from the client.

```
int connfd;         /* child socket */
char buf[BUFSIZE]; /* message buffer */
int n;              /* message byte size */

bzero(buf, BUFSIZE);
n = read(connfd, buf, BUFSIZE);
if (n < 0)
  error("ERROR reading from socket");
printf("server received %d bytes: %s", n, buf);
```

At this point, it looks just like file I/O.

Echo server: `write()`

Finally, the server echoes the input line back to the client, closes the connection, and loops back to wait for the next connection request (from possibly some other client on the network).

```
int connfd;         /* conection socket */
char buf[BUFSIZE]; /* message buffer */
int n;              /* message byte size */

n = write(connfd, buf, strlen(buf));
if (n < 0)
  error("ERROR writing to socket");

close(connfd);
```

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 info

Complete versions of the echo client and server are available from the course web page.

- **follow the “Documents” link.**

You should compile and run them for yourselves to see how they work.