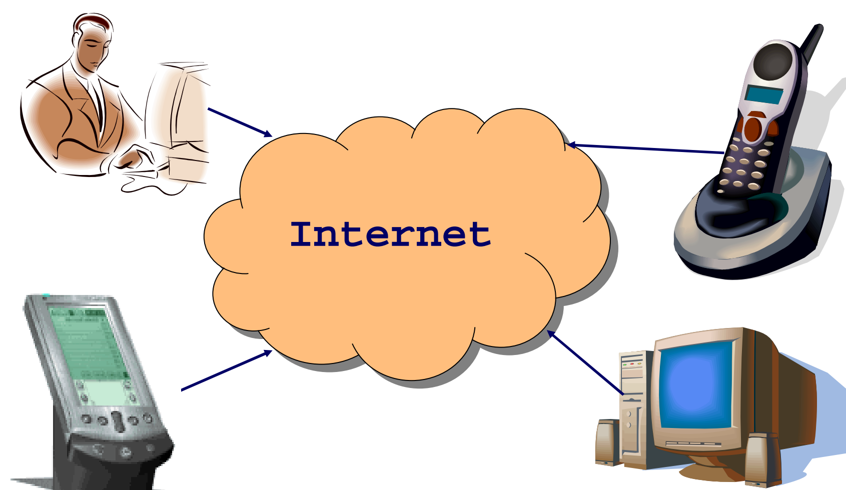
Network Programming

Topics

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

End System: Computer on the 'Net



Also known as a "host"…

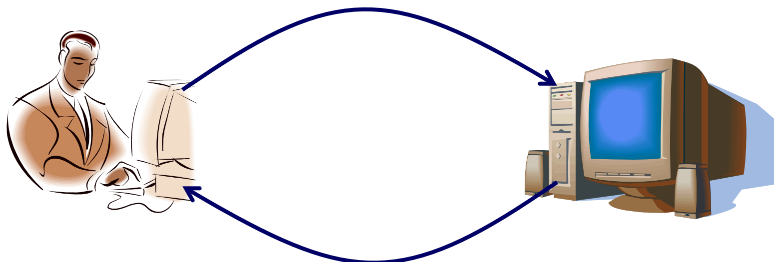
Clients and Servers

Client program

- Running on end host
- Requests service
- E.g., Web browser

Server program

- Running on end host
- Provides service
- E.g., Web server



– 3 –

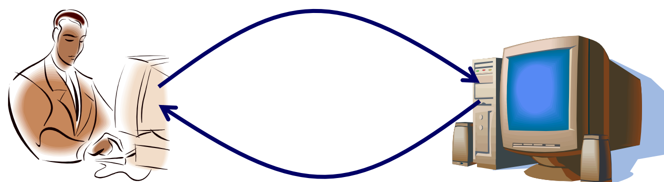
Client-Server Communication

Client "sometimes on"

- Initiates a request to the server when interested
- E.g., Web browser on your laptop or cell phone
- Doesn't communicate directly with other clients
- Needs to know the server's address

Server is "always on"

- Services requests from many client hosts
- E.g., Web server for the www.cnn.com web site
- Doesn't initiate contact with the clients
- Needs a fixed, well-known address



– 4 –

Client and Server Processes

Program vs. process

- Program: collection of code
- Process: a running program on a host

Communication between processes

- Same end host: inter-process communication
  - Governed by the operating system on the end host
- Different end hosts: exchanging messages
  - Governed by the network protocols

Client and server processes

- Client process: process that initiates communication
- Server process: process that waits to be contacted

– 5 –

Delivering the Data: Division of Labor

Network

- Deliver data packet to the destination host
- Based on the destination IP address

Operating system

- Deliver data to the destination socket
- Based on the destination port number

Application

- Read data from and write data to the socket
- Interpret the data (e.g., render a Web page)

– 6 –

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

3. Internet *sockets* are communication endpoints.

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

– 7 –

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.**

– 8 –

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.

– 9 –

3. Internet Sockets

Sending message from one process to another

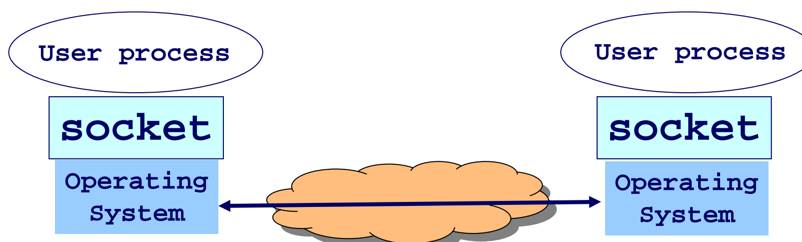
- Message must traverse the underlying network

Process sends and receives through a "socket"

- In essence, the doorway leading in/out of the house

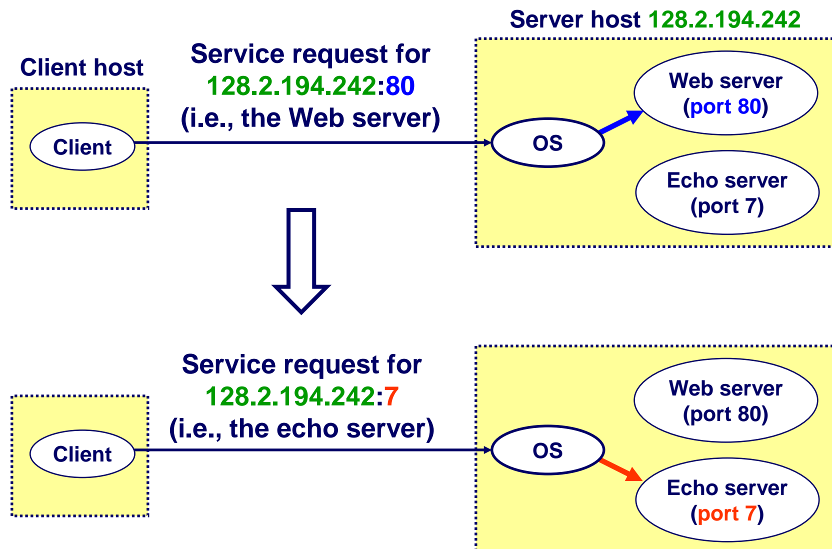
Socket as an Application Programming Interface

- Supports the creation of network applications



– 10 –

Using Ports to Identify Services



– 11 –

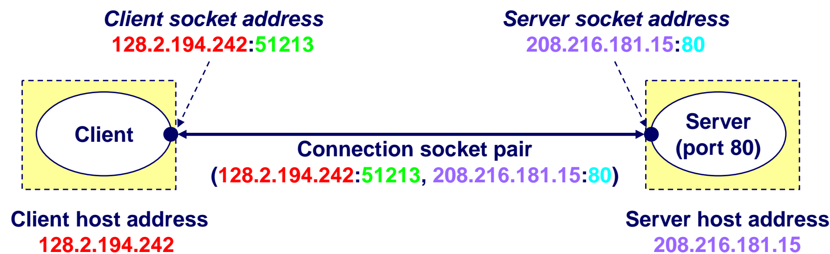
4. Internet Connections

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

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

Client socket address
128.2.194.242:51213

Server socket address
208.216.181.15:80



Client host address
128.2.194.242

Server host address
208.216.181.15

Note: 51213 is an ephemeral port allocated by the kernel

Note: 80 is a well-known port associated with Web servers

– 12 –

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

– 13 –

Well-Known vs. Ephemeral Ports

Server has a well-known port (e.g., port 80)

- Between 0 and 1023

Client picks an unused ephemeral (i.e., temporary) port

- Between 1024 and 65535

See http://www.iana.org/assignments/port-numbers

– 14 –

Servers

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

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."

– 15 –

Port Numbers are Unique on Each Host

Port number uniquely identifies the socket

- Cannot use same port number twice with same address
- Otherwise, the OS can't demultiplex packets correctly

Operating system enforces uniqueness

- OS keeps track of which port numbers are in use
- Doesn't let the second program use the port number

Example: two Web servers running on a machine

- They cannot both use port "80", the standard port #
- So, the second one might use a non-standard port #
- E.g., http://www.cnn.com:8080

– 16 –

Sockets Interface

– 17 –

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.

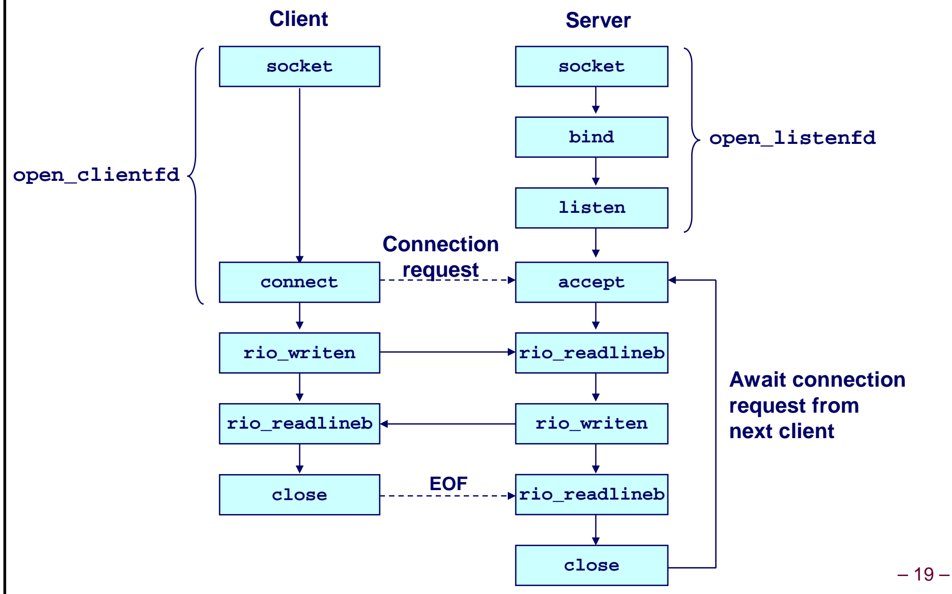
Provides a user-level interface to the network.

Underlying basis for all Internet applications.

Based on client/server programming model.

– 18 –

Overview of the Sockets Interface



Typical Client Program

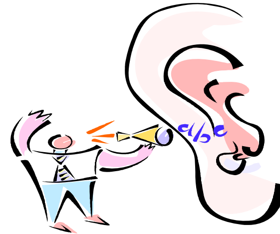
Prepare to communicate
- Create a socket
- Determine server address and port number
- Initiate the connection to the server

Exchange data with the server
- Write data to the socket
- Read data from the socket
- Do stuff with the data (e.g., render a Web page)

Close the socket

Servers Differ From Clients

Passive open
- Prepare to accept connections
- … but don't actually establish
- … until hearing from a client



Hearing from multiple clients
- Allowing a backlog of waiting clients
- ... in case several try to communicate at once

Create a socket for each client
- Upon accepting a new client
- … create a *new* socket for the communication

– 21 –

Typical Server Program

Prepare to communicate
- Create a socket
- Associate local address and port with the socket

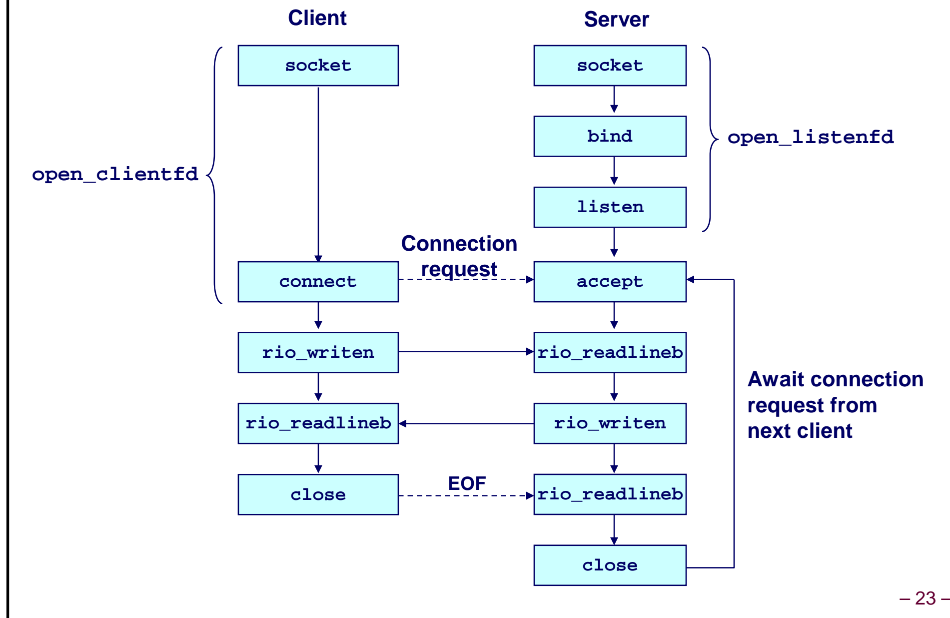
Wait to hear from a client (passive open)
- Indicate how many clients-in-waiting to permit
- Accept an incoming connection from a client

Exchange data with the client over new socket
- Receive data from the socket
- Do stuff to handle the request (e.g., get a file)
- Send data to the socket
- Close the socket

Repeat with the next connection request

– 22 –

Putting It All Together ...



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`

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

```
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;
}
```

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)
```

– 27 –

Echo Client: open_clientfd (gethostbyname)

Server typically known by name and service (“www.cnn.com” and “http”)

Need to translate into IP address and port # (“64.236.16.20” and “80”)

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);
```

– 28 –

Echo Client: `open_clientfd (connect)`

Client contacts the server to establish connection

- Associate the socket with the server address/port
- Acquire a local port number (assigned by the OS)
- Request connection to server, who will hopefully accept

```
  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;
}
```

- `connect` returns 0 on success, and -1 if an error occurs

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)
```

– 31 –

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;
}
```

– 32 –

Echo Server: open_listenfd(socket)

Server creates a **socket,** just like the client does:

- **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, initialize 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()` convers shorts from host byte order to network byte order.

– 35 –

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;
```

– 36 –

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;
}
```

Now all the server can do is wait…

- Waits for connection request to arrive
- Blocking until the request arrives
- And then accepting the new request

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);
```

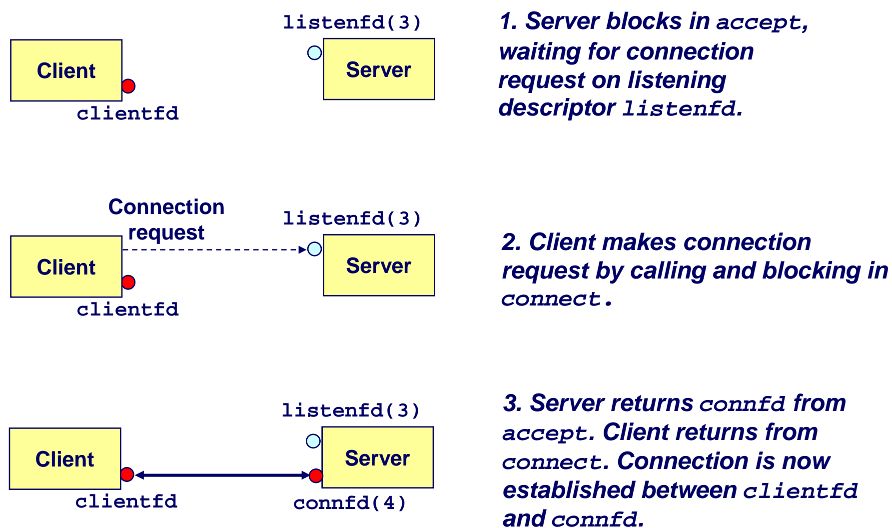
accept returns a *connected descriptor* (**connfd**) with the same properties as the *listening descriptor* (**listenfd**)

- 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.

– 39 –

Echo Server: accept Illustrated



– 40 –

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 create a new thread to handle the request.

– 41 –

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);
```

– 42 –

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:

- **bash$** ***telnet <host> <portnumber>***
- Creates a connection with a server running on <***host***> and listening on port <***portnumber***>.

Testing the Echo Server With `telnet`

```
bash$ echoserver 5000

In a separate terminal window:

bash$ telnet tanner 5000
Trying 128.2.222.85...
Connected to tanner.csc.villanova.edu
Escape character is '^]'.
123
123
Connection closed by foreign host.

bash$
```


Running the Echo Client and Server

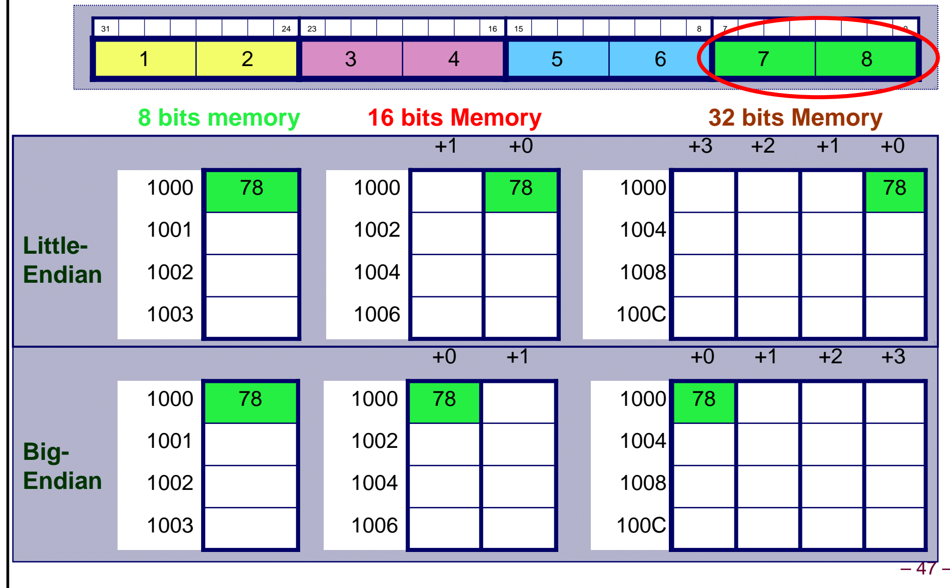
```
bash$ echoserver 5000

In a separate terminal window:

bash$ echoclient tanner 5000
Please enter msg: 123
Echo from server: 123

bash$
```

Endian Example: Where is the Byte?



IP is Big Endian

But, what byte order is used "on the wire"

- That is, what do the network protocol use?

The Internet Protocols picked one convention

- IP is big endian (aka "network byte order")

Writing portable code require conversion

- Use htons() and htonl() to convert to network byte order
- Use ntohs() and ntohl() to convert to host order

Hides details of what kind of machine you're on

- Use the system calls when sending/receiving data structures longer than one byte

Wanna See Real Clients and Servers?

Apache Web server

- Open source server first released in 1995
- Name derives from "a patchy server" ;-)
- Software available online at http://www.apache.org

Mozilla Web browser

- http://www.mozilla.org/developer/

Sendmail

- http://www.sendmail.org/

BIND Domain Name System

- Client resolver and DNS server
- http://www.isc.org/index.pl?/sw/bind/

…

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 textbook.

- 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.

The Web as an Example Client/Server Application

The Web: URL, HTML, and HTTP

Uniform Resource Locator (URL)

- A pointer to a “black box” that accepts request methods
- Formatted string with protocol (e.g., http), server name (e.g., www.cnn.com), and resource name (coolpic.jpg)

HyperText Markup Language (HTML)

- Representation of hyptertext documents in ASCII format
- Format text, reference images, embed hyperlinks
- Interpreted by Web browsers when rendering a page

HyperText Transfer Protocol (HTTP)

- Client-server protocol for transferring resources
- Client sends request and server sends response

Example: HyperText Transfer Protocol

Request

```
GET /~mdamian/csc2405/ HTTP/1.1
Host: www.csc.villanova.edu
<CRLF>
```

Response

```
HTTP/1.1 200 OK
Date: Mon, 16 Feb 2009 08:09:03 GMT
Server: Apache/1.3.27 (Unix)
Last-Modified: Sun, 26 Aug 2007 15:45:05 GMT
Content-Type: text/plain
Content-Length: 259
<CRLF>
...
```

Components: Clients, Proxies, Servers

Clients

- Send requests and receive responses
- Browsers, spiders, and agents

Servers

- Receive requests and send responses
- Store or generate the responses

Proxies (see "HTTP Proxy" assignment!)

- Act as a server for the client, and a client to the server
- Perform extra functions such as anonymization, logging, blocking of access, caching, etc.

Example Client: Web Browser

Generating HTTP requests

- User types URL, clicks a hyperlink, or selects bookmark
- User clicks “reload”, or “submit” on a Web page
- Automatic downloading of embedded images

Layout of response

- Parsing HTML and rendering the Web page
- Invoking helper applications (e.g., Acrobat, PowerPoint)

Maintaining a cache

- Storing recently-viewed objects
- Checking that cached objects are fresh

Client: Typical Web Transaction

User clicks on a hyperlink

- http://www.cnn.com/index.html

Browser learns the IP address

- Invokes gethostbyname(www.cnn.com)
- And gets a return value of 64.236.16.20

Browser creates socket and connects to server

- OS selects an ephemeral port for client side
- Contacts 64.236.16.20 on port 80

Browser writes the HTTP request into the socket

- “GET /index.html HTTP/1.1
  Host: www.cnn.com
  <CRLF>”

In Fact, Try This at a UNIX Prompt…

```
telnet www.cnn.com 80
GET /index.html HTTP/1.1
Host: www.cnn.com
<CRLF>
```

And you'll see the response…

– 57 –

Client: Typical Web Transaction (Cont)

Browser parses the HTTP response message

- Extract the URL for each embedded image
- Create new sockets and send new requests
- Render the Web page, including the images

Opportunities for caching in the browser

- HTML file
- Each embedded image
- IP address of the Web site

– 58 –

Web Server

Web site vs. Web server

- Web site: collections of Web pages associated with a particular host name
- Web server: program that satisfies client requests for Web resources

Handling a client request

- Accept the socket
- Read and parse the HTTP request message
- Translate the URL to a filename
- Determine whether the request is authorized
- Generate and transmit the response

– 59 –

Web Proxy

See assignment.

– 60 –