Network Programming
Announcements

- MP7
  - Extra example: “WikiTalk”
  - Date: (Beginning – Jan. 3, 2008)
    - Every single interaction between users on the “talk” pages of Wikipedia
    - 2,394,385 users
    - 5,021,410 pairs of users “talking”
Network Programming

As an Internet user… you already know a lot about the Internet!
Terminology

- google.com
- facebook.com
- illinois.edu

Domain Names
Uniform Resource Locators (URLs)

- http://google.com/
- http://facebook.com/
- http://illinois.edu/
Terminology

- http://google.com/
- http://facebook.com/
- http://illinois.edu/

Protocol

Hypertext Transfer Protocol (HTTP)
Terminology

- google.com ➔ 74.125.225.70
- facebook.com ➔ 66.220.158.11
- illinois.edu ➔ 128.174.4.87

Internet Protocol (IP) Addresses
Terminology

- google.com ⇒ 74.125.225.70
- facebook.com ⇒ 66.220.158.11
- illinois.edu ⇒ 128.174.4.87

- How are these addresses translated?

  Domain Name System (DNS) via Domain Name Servers
Client-Server Model

- Server: google
- Client: you (and everyone else)
Client-Server Model

- Properties?
  - Client:
  - Server:
Client-Server Model

- **Properties?**
  - **Client:**
    - Initiates contact
    - Waits for server’s response
  - **Server:**
    - Well-known name
    - Waits for contact
    - Processes requests, sends replies
Client-Server Model

- **Properties?**
  - **Client:**
    - Initiates contact
    - Waits for server’s response
  - **Server:**
    - Well-known name
    - Waits for contact
    - Processes requests, sends replies
Network Socket

- All communications across a network happen over a *network socket*.

- Properties:
All communications across a network happen over a **network socket**.

**Properties:**
- A form of Inner-Process Communications
- Bi-directional
- Connection made via a **socket address**
Socket Address

- A *socket address* is:
  - IP Address
  - Port Number

- A socket must also bind to a specific transport-layer protocol.
  - TCP
  - UDP
Port Number?

- **IP Addresses**
  - Get a packet to the destination *computer*

- **Port Numbers**
  - Get a packet to the destination *process*
Port Numbers

- A port number is…
  - An 16-bit unsigned integer
    - 0 - 65535
  - A unique resource shared across the entire system
    - Two processes cannot both utilize port 80.
  - Ports below 1024 are reserved
    - Requires elevated privileges on many OSs
    - Widely used applications have their own port number.
Application Port Numbers

- When we connect to google.com, what port on google.com are we connecting to?

We are connected to an HTTP server.

Public HTTP servers always listen for new connections on port 80.
Initializing a socket...

- Two ways to initialize a socket:
  1. To listen for an incoming connection
     - Often called a “Server Socket”
  2. To connect to a “server socket”
Client-Server Model

Server:
- Creates a socket to listen for incoming connections.
- Must listen on a specific protocol/port.

TCP/80
Client-Server Model

- **Client:**
  - Creates a socket to connect to a remote computer.
Client-Server Model

- Client:
  - Requests a connection to TCP port 80 on 74.125.225.70.
Client-Server Model

- Server:
  - Accepts the connection.

![Client-Server Model Diagram]

TCP/80
Client-Server Model

- **Server:**
  - Spawns a new socket to communicate directly with the newly connected client.
  - Allows other clients to connect.

![Diagram of Client-Server Model]

TCP/80

Two way communications
The *sockaddr* structure

- *Earlier... a socket address* is:
  - IP Address
  - Port Number

- This is represented in a special struct in C called a *sockaddr*.
Address Access/Conversion Functions

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict node,
                 const char *restrict service, const struct addrinfo *restrict hints, struct addrinfo **restrict res);
```

**Parameters**

- **node**: host name or IP address to connect to
- **service**: a port number ("80") or the name of a service (found /etc/services: "http")
- **hints**: a filled out struct addrinfo
Example: Server

```c
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // point to the results

memset(&hints, 0, sizeof hints);  // empty struct
hints.ai_family = AF_UNSPEC;  // IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM;  // TCP stream sockets
hints.ai_flags = AI_PASSIVE;  // fill in my IP for me

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
    exit(1);
}
// servinfo now points to a linked list of 1 or more struct addrinfos
// ... do everything until you don't need servinfo anymore ....

freeaddrinfo(servinfo);  // free the linked-list
```
Example: Client

```c
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // will point to the results

memset(&hints, 0, sizeof hints);  // make sure the struct is empty
hints.ai_family = AF_UNSPEC;  // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM;  // TCP stream sockets

// get ready to connect
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

// servinfo now points to a linked list of 1 or more struct addrinfos

// etc.
```
Creating a “Server Socket”

socket(): Creates a new socket for a specific protocol (eg: TCP)
bind(): Binds the socket to a specific port (eg: 80)
listen(): Moves the socket into a state of listening for incoming connections.
accept(): Accepts an incoming connection.
Creating a “Client Socket”

socket(): Creates a new socket for a specific protocol (eg: TCP)

connect():
    Makes a network connection to a specified IP address and port.
int socket (int family, int type, int protocol);

- Create a socket.
  - Returns file descriptor or -1. Also sets errno on failure.
  - family: address family (namespace)
    - AF_INET for IPv4
    - other possibilities: AF_INET6 (IPv6), AF_UNIX or AF_LOCAL (Unix socket), AF_ROUTE (routing)
  - type: style of communication
    - SOCK_STREAM for TCP (with AF_INET)
    - SOCK_DGRAM for UDP (with AF_INET)
  - protocol: protocol within family
    - typically 0
Example: socket

```c
int sockfd, new_fd; /* listen on sock_fd, new connection on new_fd */
struct sockaddr_in my_addr; /* my address */
struct sockaddr_in their_addr; /* connector addr */
int sin_size;

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
}
```
Function: bind

```c
int bind (int sockfd, struct sockaddr* myaddr, int addrlen);
```

- Bind a socket to a local IP address and port number
  - Returns 0 on success, -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `myaddr`: includes IP address and port number
    - IP address: set by kernel if value passed is `INADDR_ANY`, else set by caller
    - port number: set by kernel if value passed is 0, else set by caller
  - `addrlen`: length of address structure
    - `= sizeof (struct sockaddr_in)`
Example: bind

```c
my_addr.sin_family = AF_INET;  // host byte order
my_addr.sin_port = htons(MYPORT); // short, network
                                // byte order
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);

// automatically fill with my IP
bzero(&(my_addr.sin_zero), 8);  // zero struct

if (bind(sockfd, (struct sockaddr *)&my_addr,
         sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
}
```
## Reserved Ports

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Decimal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcpmux</td>
<td>1/tcp</td>
<td>TCP Port Service</td>
</tr>
<tr>
<td>tcpmux</td>
<td>1/udp</td>
<td>TCP Port Service</td>
</tr>
<tr>
<td>echo</td>
<td>7/tcp</td>
<td>Echo</td>
</tr>
<tr>
<td>echo</td>
<td>7/udp</td>
<td>Echo</td>
</tr>
<tr>
<td>systat</td>
<td>11/tcp</td>
<td>Active Users</td>
</tr>
<tr>
<td>systat</td>
<td>11/udp</td>
<td>Active Users</td>
</tr>
<tr>
<td>daytime</td>
<td>13/tcp</td>
<td>Daytime (RFC 867)</td>
</tr>
<tr>
<td>daytime</td>
<td>13/udp</td>
<td>Daytime (RFC 867)</td>
</tr>
<tr>
<td>qotd</td>
<td>17/tcp</td>
<td>Quote of the Day</td>
</tr>
<tr>
<td>qotd</td>
<td>17/udp</td>
<td>Quote of the Day</td>
</tr>
<tr>
<td>chargen</td>
<td>19/tcp</td>
<td>Character Generator</td>
</tr>
<tr>
<td>chargen</td>
<td>19/udp</td>
<td>Character Generator</td>
</tr>
<tr>
<td>ftp-data</td>
<td>20/tcp</td>
<td>File Transfer Data</td>
</tr>
<tr>
<td>ftp-data</td>
<td>20/udp</td>
<td>File Transfer Data</td>
</tr>
<tr>
<td>ftp</td>
<td>21/tcp</td>
<td>File Transfer Ctl</td>
</tr>
<tr>
<td>ftp</td>
<td>21/udp</td>
<td>File Transfer Ctl</td>
</tr>
<tr>
<td>ssh</td>
<td>22/tcp</td>
<td>SSH Remote Login</td>
</tr>
<tr>
<td>ssh</td>
<td>22/udp</td>
<td>SSH Remote Login</td>
</tr>
<tr>
<td>telnet</td>
<td>23/tcp</td>
<td>Telnet</td>
</tr>
<tr>
<td>telnet</td>
<td>23/udp</td>
<td>Telnet</td>
</tr>
<tr>
<td>smtp</td>
<td>25/tcp</td>
<td>Simple Mail Transfer</td>
</tr>
<tr>
<td>smtp</td>
<td>25/udp</td>
<td>Simple Mail Transfer</td>
</tr>
<tr>
<td>time</td>
<td>37/tcp</td>
<td>Time</td>
</tr>
<tr>
<td>time</td>
<td>37/udp</td>
<td>Time</td>
</tr>
<tr>
<td>nameserver</td>
<td>42/tcp</td>
<td>Host Name Server</td>
</tr>
<tr>
<td>nameserver</td>
<td>42/udp</td>
<td>Host Name Server</td>
</tr>
<tr>
<td>nicname</td>
<td>43/tcp</td>
<td>Who Is</td>
</tr>
<tr>
<td>nicname</td>
<td>43/udp</td>
<td>Who Is</td>
</tr>
<tr>
<td>domain</td>
<td>53/tcp</td>
<td>Domain Name Server</td>
</tr>
<tr>
<td>domain</td>
<td>53/udp</td>
<td>Domain Name Server</td>
</tr>
<tr>
<td>whois++</td>
<td>63/tcp</td>
<td>whois++</td>
</tr>
<tr>
<td>whois++</td>
<td>63/udp</td>
<td>whois++</td>
</tr>
<tr>
<td>gopher</td>
<td>70/tcp</td>
<td>Gopher</td>
</tr>
<tr>
<td>gopher</td>
<td>70/udp</td>
<td>Gopher</td>
</tr>
<tr>
<td>finger</td>
<td>79/tcp</td>
<td>Finger</td>
</tr>
<tr>
<td>finger</td>
<td>79/udp</td>
<td>Finger</td>
</tr>
<tr>
<td>http</td>
<td>80/tcp</td>
<td>World Wide Web HTTP</td>
</tr>
<tr>
<td>http</td>
<td>80/udp</td>
<td>World Wide Web HTTP</td>
</tr>
<tr>
<td>www</td>
<td>80/tcp</td>
<td>World Wide Web HTTP</td>
</tr>
<tr>
<td>www</td>
<td>80/udp</td>
<td>World Wide Web HTTP</td>
</tr>
<tr>
<td>www-http</td>
<td>80/tcp</td>
<td>World Wide Web HTTP</td>
</tr>
<tr>
<td>www-http</td>
<td>80/udp</td>
<td>World Wide Web HTTP</td>
</tr>
<tr>
<td>kerberos</td>
<td>88/tcp</td>
<td>Kerberos</td>
</tr>
<tr>
<td>kerberos</td>
<td>88/udp</td>
<td>Kerberos</td>
</tr>
</tbody>
</table>
int listen (int sockfd, int backlog);

- Put socket into passive state (wait for connections rather than initiate a connection)
  - Returns 0 on success, -1 and sets errno on failure
  - sockfd: socket file descriptor (returned from socket)
  - backlog: bound on length of unaccepted connection queue (connection backlog); kernel will cap, thus better to set high
  - Example:
    ```c
    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }
    ```
Establishing a Connection

- Include file `<sys/socket.h>`

```c
int connect (int sockfd, struct sockaddr* servaddr, int addrlen);
```
- Connect to another socket.

```c
int accept (int sockfd, struct sockaddr* cliaddr, int* addrlen);
```
- Accept a new connection. Returns file descriptor or -1.
int connect (int sockfd, struct sockaddr* servaddr, int addrlen);

- Connect to another socket.
  - Returns 0 on success, -1 and sets errno on failure
  - sockfd: socket file descriptor (returned from socket)
  - servaddr: IP address and port number of server
  - addrlen: length of address structure
    - = sizeof (struct sockaddr_in)

- Can use with UDP to restrict incoming datagrams and to obtain asynchronous errors
Example: connect

```c
their_addr.sin_family = AF_INET; /* interp’d by host */
their_addr.sin_port = htons(PORT);
their_addr.sin_addr = *((struct in_addr*)he->h_addr);

bzero(&(their_addr.sin_zero), 8);
/* zero rest of struct */

if (connect(sockfd, (struct sockaddr*)&their_addr,
            sizeof(struct sockaddr)) == -1) {
    perror("connect");
    exit(1);
}
```
Functions: accept

```c
int accept (int sockfd, struct sockaddr* cliaddr, 
            int* addrlen);
```

- Block waiting for a new connection
  - Returns file descriptor or -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `cliaddr`: IP address and port number of client (returned from call)
  - `addrlen`: length of address structure = pointer to `int` set to `sizeof (struct sockaddr_in)`

- `addrlen` is a **value-result** argument
  - the caller passes the size of the address structure, the kernel returns the size of the client’s address (the number of bytes written)
Example: accept

```c
sin_size = sizeof(struct sockaddr_in);
if ((new_fd = accept(sockfd, (struct sockaddr*)
    &their_addr, &sin_size)) == -1) {
    perror("accept");
    continue;
}
```

- How does the server know which client it is?
  - `their_addr.sin_addr` contains the client’s IP address
  - `their_addr.port` contains the client’s port number

```c
printf("server: got connection from %s\n",
    inet_ntoa(their_addr.sin_addr));
```
Functions: accept

Notes
- After `accept()` returns a new socket descriptor, I/O can be done using `read()` and `write()`
- Why does `accept()` need to return a new descriptor?
Sending and Receiving Data

```c
int send(int sockfd, const void * buf, size_t nbytes, int flags);
```
- Write data to a stream (TCP) or “connected” datagram (UDP) socket.
  - Returns number of bytes written or -1.

```c
int recv(int sockfd, void *buf, size_t nbytes, int flags);
```
- Read data from a stream (TCP) or “connected” datagram (UDP) socket.
  - Returns number of bytes read or -1.
Functions: send

```c
int send(int sockfd, const void * buf, size_t nbytes, int flags);
```

- Send data un a stream (TCP) or “connected” datagram (UDP) socket
  - Returns number of bytes written or -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `buf`: data buffer
  - `nbytes`: number of bytes to try to write
  - `flags`: control flags
    - MSG_PEEK: get data from the beginning of the receive queue without removing that data from the queue
int send(int sockfd, const void * buf, size_t nbytes, int flags);

Example

len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
## Functions: recv

```c
int recv(int sockfd, void *buf, size_t nbytes, int flags);
```

- Read data from a stream (TCP) or “connected” datagram (UDP) socket
  - Returns number of bytes read or -1, sets `errno` on failure
  - Returns 0 if socket closed
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `buf`: data buffer
  - `nbytes`: number of bytes to try to read
  - `flags`: see man page for details; typically use 0
Functions: recv

```c
int recv(int sockfd, char* buf, size_t nbytes);
```

Notes

- **read** blocks waiting for data from the client but does not guarantee that `sizeof(buf)` is read
- Example
  ```c
  if((r = read(newfd, buf, sizeof(buf))) < 0) {
    perror("read"); exit(1);
  }
  ```
Sending and Receiving Data

- Datagram sockets aren't connected to a remote host
  - What piece of information do we need to give before we send a packet?
  - The destination/source address!
Sending and Receiving Data

int sendto (int sockfd, char* buf, size_t nbytes, int flags, struct sockaddr* destaddr, int addrlen);
  o Send a datagram to another UDP socket.
    ■ Returns number of bytes written or -1.

int recvfrom (int sockfd, char* buf, size_t nbytes, int flags, struct sockaddr* srcaddr, int* addrlen);
  o Read a datagram from a UDP socket.
    ■ Returns number of bytes read or -1.
int sendto (int sockfd, char* buf, size_t nbytes,
        int flags, struct sockaddr* destaddr, int
        addrlen);

Send a datagram to another UDP socket

- Returns number of bytes written or -1 and sets \texttt{errno} on failure
- \texttt{sockfd}: socket file descriptor (returned from \texttt{socket})
- \texttt{buf}: data buffer
- \texttt{nbytes}: number of bytes to try to read
- \texttt{flags}: see man page for details; typically use 0
- \texttt{destaddr}: IP address and port number of destination socket
- \texttt{addrlen}: length of address structure
  - \texttt{= sizeof (struct sockaddr\_in)}
int sendto (int sockfd, char* buf, size_t nbytes, int flags, struct sockaddr* destaddr, int addrlen);

Example

n = sendto(sock, buf, sizeof(buf), 0, (struct sockaddr *) &from, fromlen);
if (n < 0)
    perror("sendto");
    exit(1);
}
int recvfrom (int sockfd, char* buf, size_t nbytes, int flags, struct sockaddr* srcaddr, int* addrlen);

- Read a datagram from a UDP socket.
  - Returns number of bytes read (0 is valid) or -1 and sets errno on failure
  - sockfd: socket file descriptor (returned from socket)
  - buf: data buffer
  - nbytes: number of bytes to try to read
  - flags: see man page for details; typically use 0
  - srcaddr: IP address and port number of sending socket (returned from call)
  - addrlen: length of address structure = pointer to int set to sizeof (struct sockaddr_in)
Functions: recvfrom

```c
int recvfrom (int sockfd, char* buf, size_t nbytes, int flags, struct sockaddr* srcaddr, int* addrlen);
```

Example

```c
n = recvfrom(sock, buf, 1024, 0, (struct sockaddr *)&from,&fromlen);
if (n < 0) {
    perror("recvfrom");
    exit(1);
}
```
Tearing Down a Connection

```c
int close (int sockfd);
```
- Close a socket.
  - Returns 0 on success, -1 and sets `errno` on failure.

```c
int shutdown (int sockfd, int howto);
```
- Force termination of communication across a socket in one or both directions.
  - Returns 0 on success, -1 and sets `errno` on failure.
Functions: close

```c
int close (int sockfd);
```

- Close a socket
  - Returns 0 on success, -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)

- Closes communication on socket in both directions
  - All data sent before `close` are delivered to other side (although this aspect can be overridden)

- After `close`, `sockfd` is not valid for reading or writing
Functions: shutdown

```
int shutdown (int sockfd, int howto);
```

- Force termination of communication across a socket in one or both directions
  - Returns 0 on success, -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `howto`:
    - `SHUT_RD` to stop reading
    - `SHUT_WR` to stop writing
    - `SHUT_RDWR` to stop both

- `shutdown` overrides the usual rules regarding duplicated sockets, in which TCP teardown does not occur until all copies have closed the socket
Note on `close` vs. `shutdown`

- **`close()`** : closes the socket but the connection is still open for processes that shares this socket
  - The connection stays opened both for read and write

- **`shutdown()`** : breaks the connection for all processes sharing the socket
  - A read will detect **EOF**, and a write will receive **SIGPIPE**
  - `shutdown()` has a second argument how to close the connection:
    - 0 means to disable further reading
    - 1 to disable writing
    - 2 disables both