

# Networking and Sockets

CS241 Discussion Section

Week 11

4/16/07 – 4/22/07

# Outline

Brief intro to networking

HTTP Example

Socket Programming

Library Functions

TCP Client and Server Examples

# Networking

Allows computers and other networked devices to talk to each other

How can we tell what the packet we just received means?

Interactions between applications on different machines are governed by *protocols*

Protocols dictate the format and sequence of the information exchange

# Names and Addresses

*A network address* identifies a specific computer on the network

Several kinds of addresses exist: MAC address (for LANs), IP address (for the Internet), etc.

Domain or DNS names are used for convenience, so we don't have to remember numerical addresses

# Ports

*Ports* are numbers that represent an end-point for communication

Ports are logical, not physical, entities

All packets arrive to the same physical interface, and are then differentiated based on the port number (and other contents of the packet header)

Usually, distinct ports are used for communication via different protocols

E.g., port 80 is for HTTP, 22 is for SSH, etc.

See `/etc/services` for a list

# Example: HTTP

How the Web really works

# HTTP

## Hypertext Transfer Protocol

Delivers virtually all files and resources on the World Wide Web

Uses Client-Server Model

## HTTP transaction

Client opens a connection and sends a request message to the server

Server returns a response message

# HTTP (continued)

## Request

GET /path/to/file.html HTTP/1.0

Other request methods possible (POST, HEAD)

## Response

HTTP/1.0 200 OK

### Common Status Codes

200 OK

404 Not Found

500 Server Error

# Sample HTTP exchange

## Scenario

Client wants to retrieve the document at the following URL:  
<http://www.cs.uiuc.edu/>

## What the client does

Client opens a connection to the host `www.cs.uiuc.edu`, port 80

```
telnet www.cs.uiuc.edu 80
```

Client sends the following message, requesting document “/”

```
GET / HTTP/1.0
From: netid@uiuc.edu
User-Agent: MyBrowser
[other parameters...]
[blank line here]
```

# Sample HTTP exchange

## What the server does

### Server responds with:

```
HTTP/1.0 200 OK
```

```
Date: Mon, 16 Apr 2007
```

```
23:59:59 GMT
```

```
Content-Type: text/html
```

```
Content-Length: 1354
```

```
<html>
```

```
<body>
```

# Sockets

Standard API for sending and receiving data  
across computer networks

Can also be used for interprocess communication  
on a single machine

Introduced by BSD operating systems in 1983

POSIX incorporated 4.3BSD sockets in 2001

# Using Sockets in C

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <unistd.h>
```

**On csil-core:**

```
gcc -o test test.c
```

**On some systems, e.g., Solaris:**

```
gcc -o test test.c -lsocket -lnsl
```

# TCP Client/Server Example

Run the provided test-server and test-client executables

test-client sends the string “Hello World!” to IP address 127.0.0.1 port 10000

test-server listens on port 10000 and prints out any text received

Let's try to replicate this behavior

# Generic TCP Client & Server Script

## Client

```
socket()
connect()
while (...) {
    send()/recv()
}
close()
```

## Server

```
socket()
bind()
listen()
while (...) {
    accept()
    send()/recv()
}
close()
```

Let's take it one step at a time,  
starting with the client...

# socket

```
int socket(int domain, int type, int protocol);
```

Creates a communication endpoint

Local action only; no communication takes place

## Parameters

domain: `AF_INET` (IPv4)

type: `SOCK_STREAM` (TCP) or `SOCK_DGRAM` (UDP)

protocol: `0` (socket chooses the correct protocol based on type)

Returns a nonnegative integer corresponding to a socket file descriptor if successful, `-1` with `errno` set if unsuccessful

# connect

```
int connect(int socket, const struct sockaddr  
            *address, socklen_t address_len);
```

Establishes a link to a well-known port of the remote server

Initiates the TCP 3-way handshake (exchange packets with the remote machine to mutually set up a connection)

Returns 0 if successful, -1 with `errno` set if unsuccessful

# struct sockaddr

```
struct sockaddr_in used for address
    sa_family_t sin_family;    /* AF_INET */
    in_port_t sin_port;      /* port number */
    struct in_addr sin_addr;  /* IP address */
```

**For example:**

```
sa.sin_family = AF_INET;
sa.sin_port = htons(80);
sa.sin_addr = inet_addr("127.0.0.1");
```

# send and sendto

```
int send(int socket, const void *msg, int len, int flags);
```

```
int sendto(int socket, const void *msg, int len, int flags,  
           const struct sockaddr *to, socklen_t tolen);
```

sends data pointed by `msg`

`sendto` is used for unconnected datagram sockets. If used in connection-mode, last two parameters are ignored.

Returns the number of bytes actually sent out if successful, -1 with `errno` set if unsuccessful

# close and shutdown

```
int close(int socket);
```

```
int shutdown(int socket, int how);
```

close

Prevents any more reads and writes  
same function as for file systems

shutdown

provides a little more control

how

0 – Further receives are disallowed

1 – Further sends are disallowed

2 – same as close

Returns 0 if successful, -1 with `errno` set if unsuccessful

# What about the server?

First call `socket()` as with the client. Then...

# bind

```
int bind(int socket, const struct  
sockaddr *address, socklen_t  
address_len);
```

Associates the socket with a port on your local machine

Local action; no communication takes place yet

Returns 0 if successful, -1 with `errno` set if unsuccessful

# Caution!

Exiting or crashing after calling `bind()` but before `close()` will cause problems!

Cannot `bind()` the same port for a few minutes

Prevent a different application from accidentally receiving our connections

Unless `SO_REUSEADDR` option is used with `setsockopt()`

Permits the server to be restarted immediately

# setsockopt

```
int sock;
int true = 1;

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    exit(EXIT_FAILURE);

if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char *)&true,
              sizeof(true)) == -1) {
    error = errno;
    while ((close(sock) == -1) && (errno == EINTR));
    errno = error;
    exit(EXIT_FAILURE);
}
```

# listen

```
int listen(int socket, int backlog);
```

Puts the socket into the passive state to accept incoming requests

Internally, it causes the network infrastructure to allocate queues to hold pending requests

`backlog`: number of connections allowed on the incoming queue

`bind()` should have been called beforehand

Returns 0 if successful, -1 with `errno` set if unsuccessful

# accept

```
int accept(int socket, struct sockaddr *restrict  
    address, socklen_t *restrict address_len);
```

Accepts the pending connection requests into the incoming queue

\*`address` is used to return the information about the client making the connection.

`sin_addr.s_addr` holds the Internet address

`listen()` should have been called beforehand

Returns nonnegative file descriptor corresponding to the accepted socket if successful, -1 with `errno` set if unsuccessful

# recv and recvfrom

```
int recv(int socket, void *buf, int len, int flags);
```

```
int recvfrom(int socket, void *buf, int len, int flags,  
             const struct sockaddr *from, socklen_t fromlen);
```

receives data into the buffer `buf`

`recvfrom` is used for unconnected datagram sockets. If used in connection-mode, last two parameters are ignored.

Returns the number of bytes actually read if successful, -1 with `errno` set if unsuccessful

# Testing our code

Compile your client and server implementations

Try testing:

client with server

client with test-server

test-client with server

All should work!

# Recap

## Networking basics

Hosts, addresses, protocols, ports, etc.

## HTTP Example

Client-server model, TCP connection

## Sockets

Client and server actions

See also: Beej's Guide to Network Programming

<http://beej.us/guide/bgnet/>