CS 241 Section Week #11
(4/08/10)

# Outline

- MP6 Overview
- Socket Programming
- Hypertext Transfer Protocol

# MP6

# MP6

**Goal: Build a simple HTTP web server**

- Setting up the TCP sockets
- Handle multiple requests at the same time by using threads
- Return pages stored locally
- Act as a proxy to retrieve a webpage from another website and send it to the client

## MP6 Tasks

- Create a socket to listen for incoming TCP connections on a specific port.
- Upon accepting a connection, launch a thread for the incoming TCP connection
- In the handler for each connection, you need to recv() data from the socket.

## MP6 Tasks

- We give you in aux_functions.h
  - a struct called HTTPResponse
  - getResponseString():
  - getFileNotFoundResponseString()
    - Notify browser that the file is not found.
  - getNotImplementedResponseString()
    - Notify browser that you are unable to handle its request
  - getFileNameFromHTTPRequest()
    - Return the requested file contained in the request

## MP6 Tasks

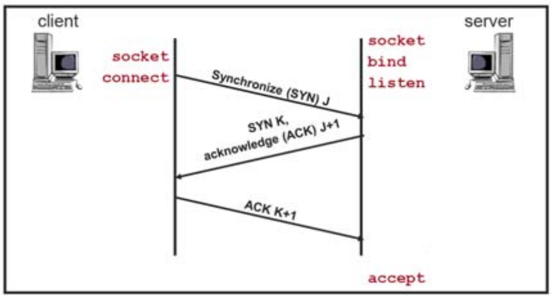
- Using HTTPResponse you must now use send() to send the contents of HTTPResponse.vptrResponse back to the web browser
- You will need to continue to recv() requests on this socket until the web browser closes its TCP connection with your web server
- Upon receiving proxy request, you should create a client, open the requested url and send the content back to the browser.
- You should modify the links inside the external pages so that the later transfer of data are done through the proxy

## Socket Programming

# Socket

- Standard APIs for sending and receiving data across computer networks
- Introduced by BSD operating systems in 1983
- POSIX incorporated 4.3BSD sockets and XTI in 2001
- `#include <sys/socket.h>`

# Typical TCP Server-Client


client

server

socket
connect

socket
bind
listen

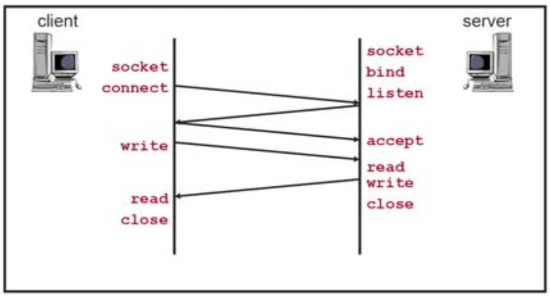
Synchronize (SYN) J

SYN K,
acknowledge (ACK) J+1

ACK K+1

accept

# Typical TCP Server-Client


client

server

socket
connect

write

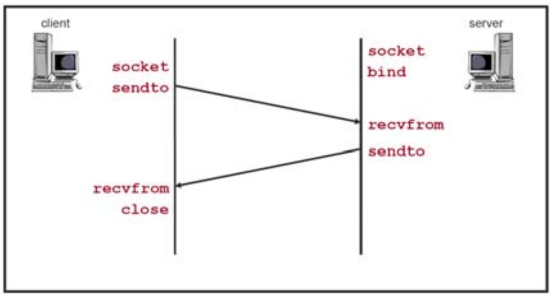
read
close

socket
bind
listen

accept

read
write
close

# Typical UDP Server-Client


client

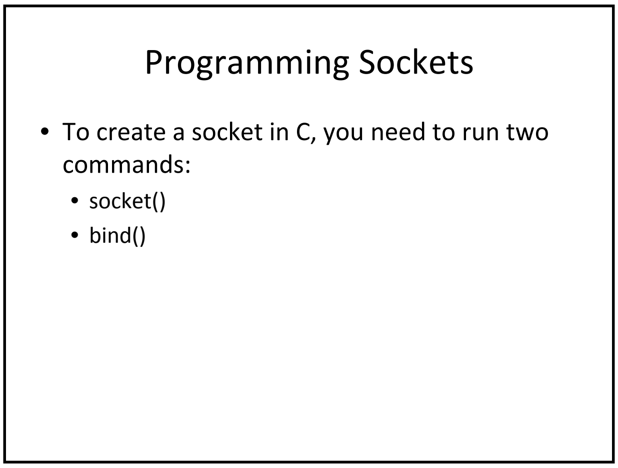
server

socket
sendto

recvfrom
close

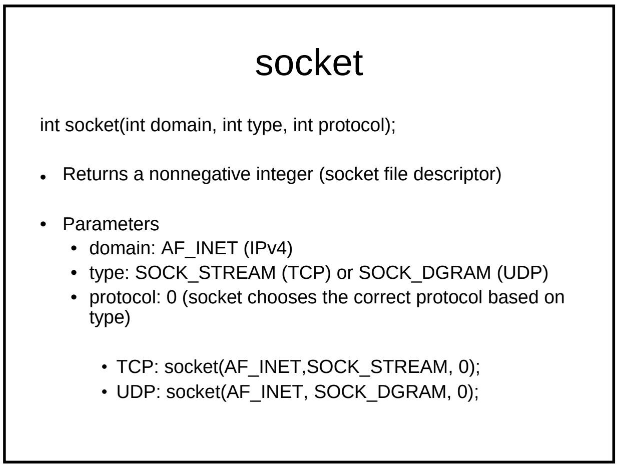
socket
bind

recvfrom
sendto



## Programming Sockets

- To create a socket in C, you need to run two commands:
  - socket()
  - bind()



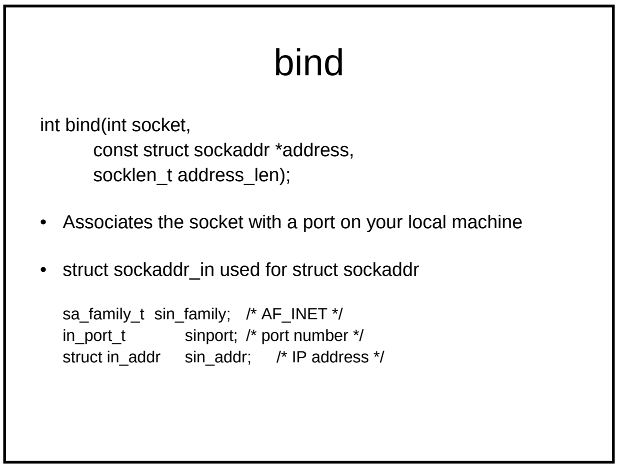
## socket

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

- Returns a nonnegative integer (socket file descriptor)

- Parameters
  - domain: AF_INET (IPv4)
  - type: SOCK_STREAM (TCP) or SOCK_DGRAM (UDP)
  - protocol: 0 (socket chooses the correct protocol based on type)

    - TCP: socket(AF_INET,SOCK_STREAM, 0);
    - UDP: socket(AF_INET, SOCK_DGRAM, 0);



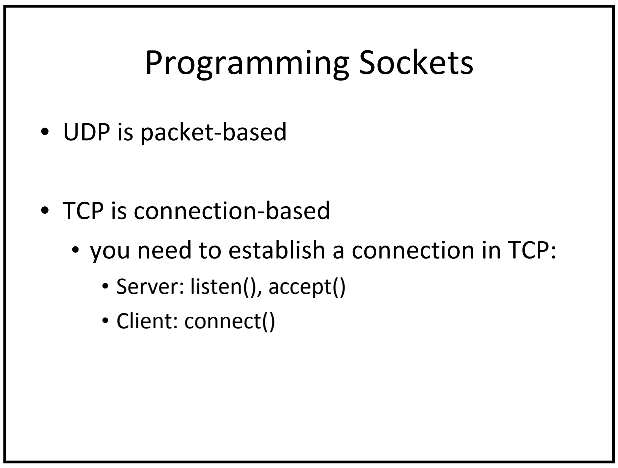
## bind

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

- Associates the socket with a port on your local machine

- struct sockaddr_in used for struct sockaddr

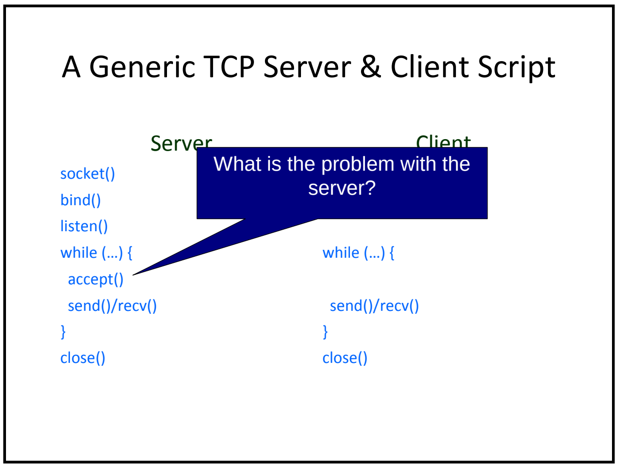
```
sa_family_t  sin_family;   /* AF_INET */
in_port_t         sinport;  /* port number */
struct in_addr    sin_addr;     /* IP address */
```



## Programming Sockets

- UDP is packet-based

- TCP is connection-based
  - you need to establish a connection in TCP:
    - Server: listen(), accept()
    - Client: connect()

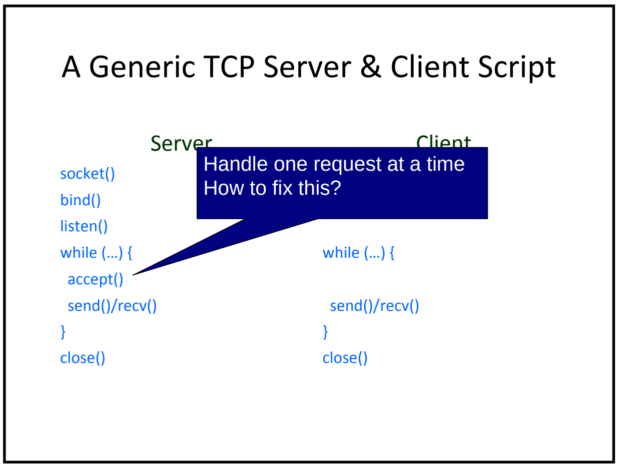


## A Generic TCP Server & Client Script

Server | Client

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

What is the problem with the server?

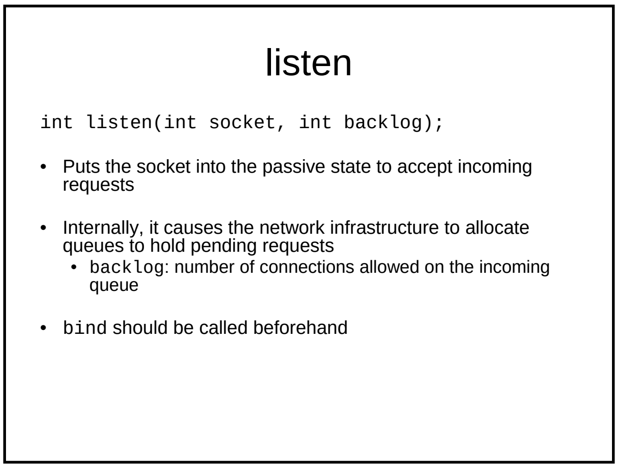


## A Generic TCP Server & Client Script

Server | Client

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

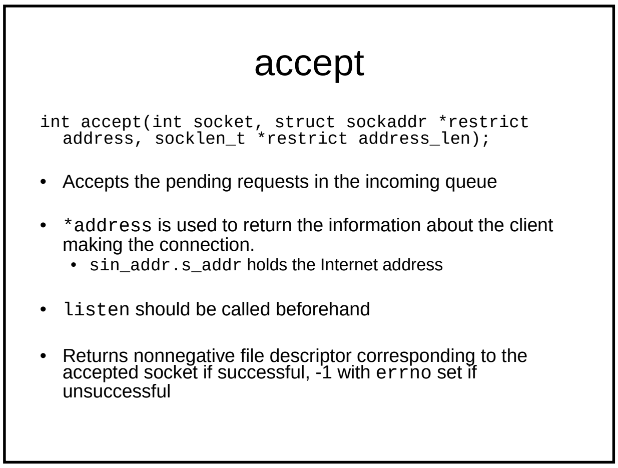
Handle one request at a time
How to fix this?



## listen

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

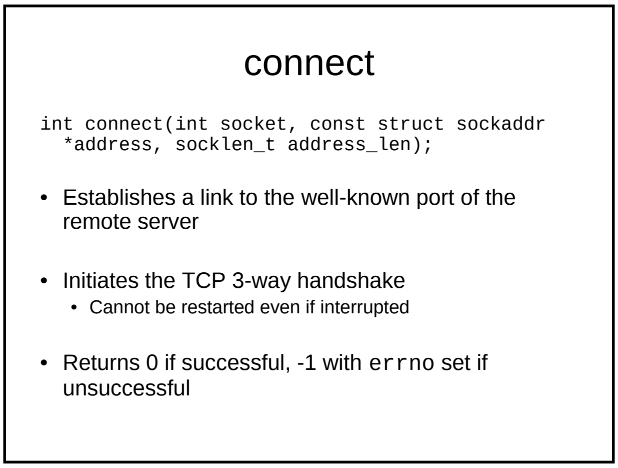
- Puts the socket into the passive state to accept incoming requests
- Internally, it causes the network infrastructure to allocate queues to hold pending requests
  - `backlog`: number of connections allowed on the incoming queue
- `bind` should be called beforehand



## accept

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

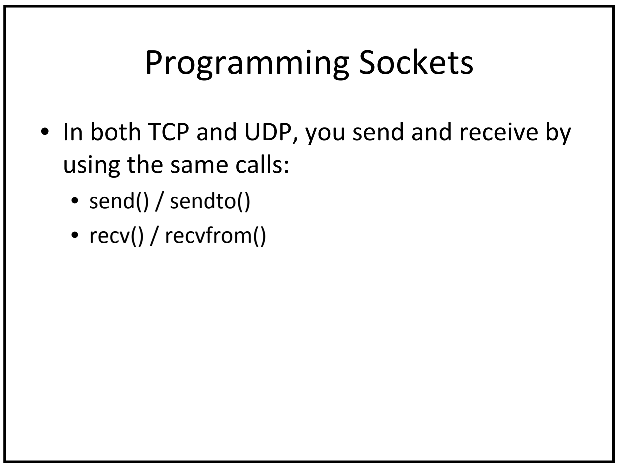
- Accepts the pending requests in the incoming queue
- `*address` is used to return the information about the client making the connection.
  - `sin_addr.s_addr` holds the Internet address
- `listen` should be called beforehand
- Returns nonnegative file descriptor corresponding to the accepted socket if successful, -1 with `errno` set if unsuccessful



## connect

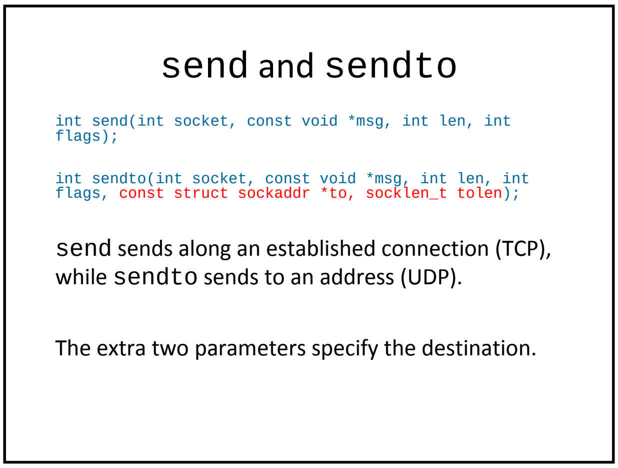
```
int connect(int socket, const struct sockaddr
  *address, socklen_t address_len);
```

- Establishes a link to the well-known port of the remote server
- Initiates the TCP 3-way handshake
  - Cannot be restarted even if interrupted
- Returns 0 if successful, -1 with `errno` set if unsuccessful



## Programming Sockets

- In both TCP and UDP, you send and receive by using the same calls:
  - send() / sendto()
  - recv() / recvfrom()

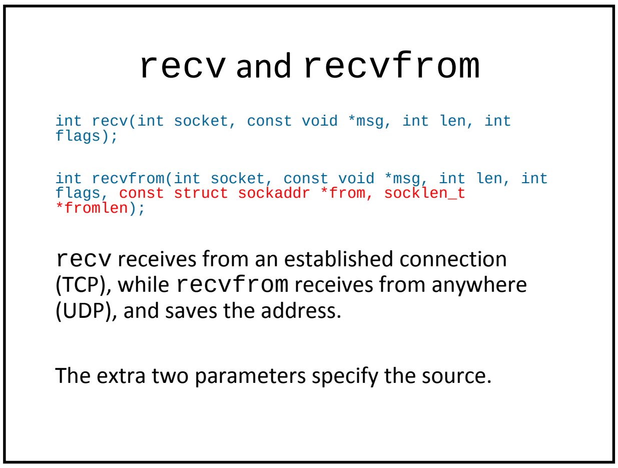


## `send` and `sendto`

```
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);
```

`send` sends along an established connection (TCP), while `sendto` sends to an address (UDP).

The extra two parameters specify the destination.

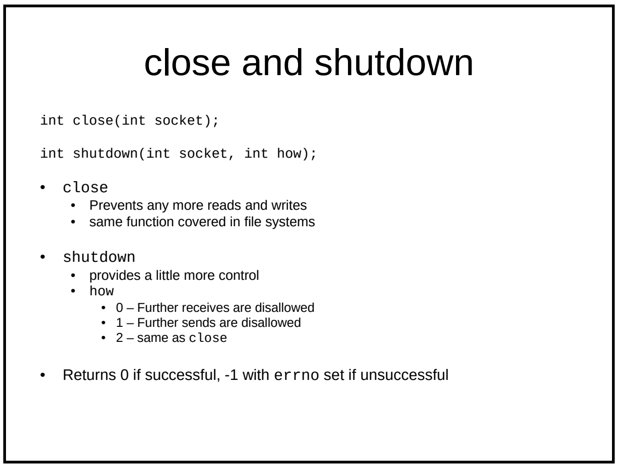


## `recv` and `recvfrom`

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

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

`recv` receives from an established connection (TCP), while `recvfrom` receives from anywhere (UDP), and saves the address.

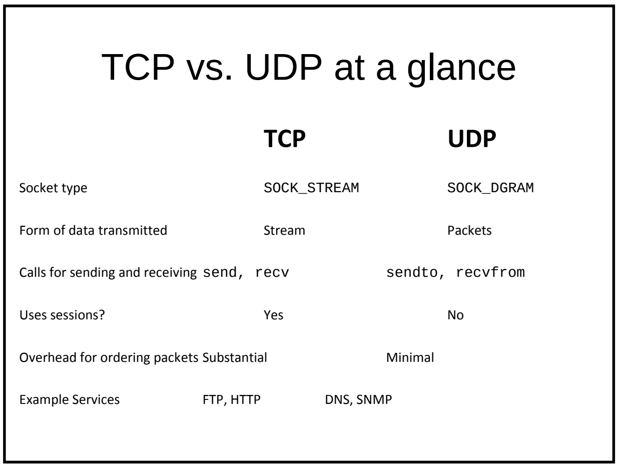
The extra two parameters specify the source.



## close and shutdown

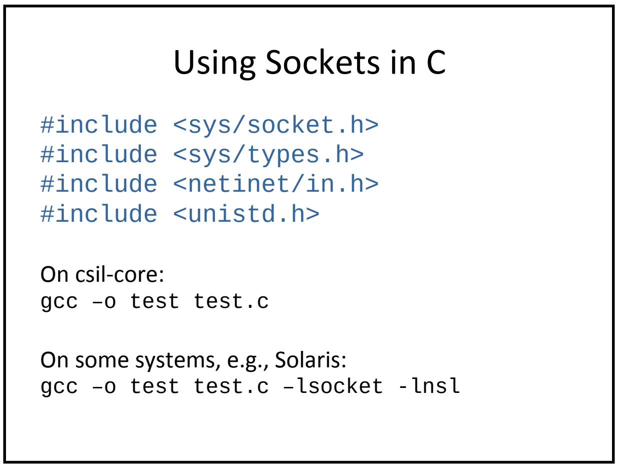
```
int close(int socket);

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

- `close`
  - Prevents any more reads and writes
  - same function covered in 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



## TCP vs. UDP at a glance

| | **TCP** | **UDP** |
|---|---|---|
| Socket type | `SOCK_STREAM` | `SOCK_DGRAM` |
| Form of data transmitted | Stream | Packets |
| Calls for sending and receiving | `send, recv` | `sendto, recvfrom` |
| Uses sessions? | Yes | No |
| Overhead for ordering packets | Substantial | Minimal |
| Example Services | FTP, HTTP | DNS, SNMP |



## Using Sockets in C

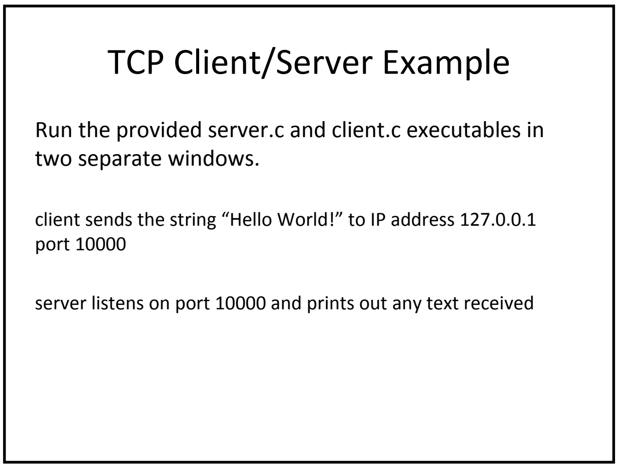
```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <unistd.h>
```

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 server.c and client.c executables in two separate windows.

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

server listens on port 10000 and prints out any text received

## HyperText Transfer Protocol

## HTTP

- Hypertext Transfer Protocol
  - Delivers virtually all files and resources on the World Wide Web
  - Uses Client-Server Model
- HTTP transaction
  - HTTP client opens a connection and sends a request to HTTP server
  - HTTP server returns a response message

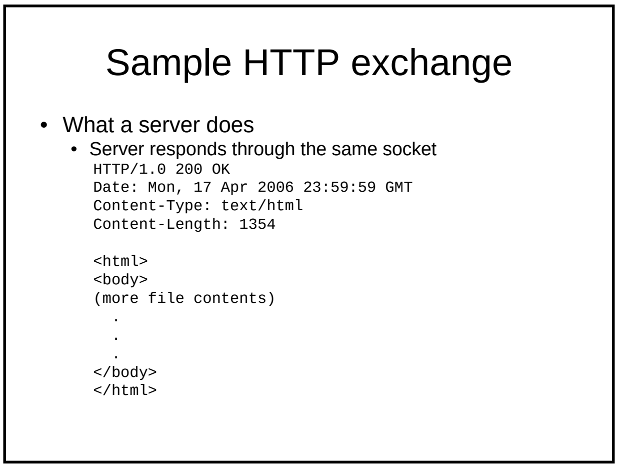
## HTTP (continued)

- Request
  - `GET /path/to/file/index.html HTTP/1.0`
  - Other methods (POST, HEAD) possible for request
- 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 file at the following URL (http://www.somehost.com/path/file.html)
- What a client does
  - Client opens a socket to the host www.somehost.com, port 80
  - Client sends the following message through the socket

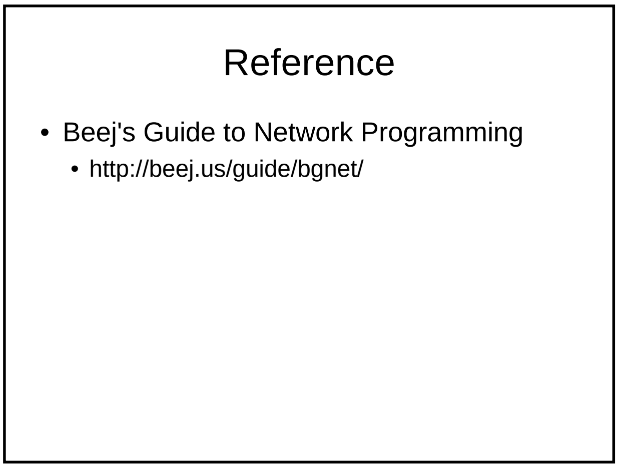
```
GET /path/file.html HTTP/1.0
From: someuser@uiuc.edu
User-Agent: HTTPTool/1.0
[blank line here]
```



## Sample HTTP exchange

- What a server does
  - Server responds through the same socket

```
HTTP/1.0 200 OK
Date: Mon, 17 Apr 2006 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354

<html>
<body>
(more file contents)
  .
  .
  .
</body>
</html>
```



## Reference

- Beej's Guide to Network Programming
  - http://beej.us/guide/bgnet/