Creating a Web Server

CS 360 Internet Programming

Daniel Zappala

Brigham Young University
Computer Science Department
Overview

1. create a **socket**
2. **bind** the socket to an address and port
3. **listen** for incoming connections
4. **accept** a client
5. **send** and **recv** data
Bind the Socket

```c
1 int bind(int sockfd, const struct sockaddr *my_addr,
2     socklen_t addrlen);
3
4 sockfd = socket you created
5 my_addr = pointer to socket address structure
6 addrlen = length of socket address
```

- associates an address with a socket
- uses the socket address structure to pass an IP address and port
- on success returns zero, -1 and errno otherwise
Listen on Socket

```c
int listen(int sockfd, int backlog);
```

- sockfd = socket you created and bound
- backlog = maximum waiting connections

- converts a socket to a passive socket (one that accepts connections rather than connecting)
- backlog is the maximum number of waiting connections the kernel should hold in a queue
- maximum value in Linux is 128
- on success returns zero, -1 and errno otherwise
Accept a Connection

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

- sockfd = socket you created and bound
- addr = pointer to empty socket address structure
- addrlen = length of empty socket address structure

- accept a connection from a client; gets the next connection waiting in the queue
- if there are no pending connections, the process sleeps assuming this is a blocking socket (this is the default)
- on success returns a new socket descriptor
- on success, accept() fills in the address of the client
Putting it All Together

see example code on class web site
TCP Demultiplexing

- TCP sockets identified by four-tuple:
  
  (source IP address, source port, destination IP address, destination port)

- host delivers all TCP segments with a given tuple to the associated socket

- server creates one socket per client
TCP Demultiplexing

Applications

socket()  listen()  bind()  accept()

source address and port
128.187.81.236, 80

OS

socket()  connect()
123.214.3.2, 5681  36.21.2.80, 4538
destination address and port
Serving Multiple Clients

Web client host C

Web server B

Per-connection HTTP processes

Transport-layer demultiplexing

source port: 7532 dest. port: 80
source IP: C dest. IP: B

source port: 26145 dest. port: 80
source IP: C dest. IP: B

source port: 26145 dest. port: 80
source IP: A dest. IP: B

source port: 7532 dest. port: 80
source IP: C dest. IP: B

Creating a Server
Handling Multiple Clients
Creating a Web Server
Pipelined Requests
Time
Steps in Handling an HTTP Request

1. read and parse the HTTP request message
   - use supplied HTTP parser

2. translate the URI to a file name
   - need web server configuration to determine the document root

3. determine whether the request is authorized
   - check file permissions or other authorization procedure

4. generate and transmit the response
   - error code or file or results of script
   - must be a valid HTTP message with appropriate headers

5. log request and any errors
Handling Multiple Roots

- use the Host header to find the host name
- configuration file gives the root directory for each host served by the web server
- append the URI path to the root directory to get the complete path
Checking File Permissions

- call `open()` to determine whether you can access the file
- return value of -1 indicates failure
  - `errno` == `EACCESS` indicates you didn’t have the right permissions (403 Forbidden)
  - `errno` == `ENOENT` indicates the file doesn’t exist (404 Not Found)
  - any other error indicate a server failure (500 Internal Server Error)
Accessing File Attributes

- use `fstat()` to access file size and last modification time
- use in Content-Length and Last-Modified headers

```
1   int fstat(int filedes, struct stat *buf);
2
3   filedes = file descriptor from open()
4   buf = pointer to structure to hold statistics
```

- on success returns zero
- on error returns -1 and sets errno
- `stat(2)` provides details on the structure
- the `st_size` member gives the file size in bytes
- the `st_mtime` member gives the time of last modification
Sending a File

```c
ssize_t sendfile(int out_fd, int in_fd, off_t *offset,
                 size_t count);
```

- `out_fd` = socket file descriptor
- `in_fd` = file descriptor of file to send
- `offset` = NULL
- `count` = size of file in bytes (from `fstat()`)

- on success, returns number of bytes written
- on error returns -1 and sets `errno`
Handling Pipelined Requests

- loop forever
  - read until sentinel (\r\n\r\n) using recv() loop
  - process HTTP message
  - read body if needed (only for POST)
- break out of loop for
  - recv() error
  - socket closed
- how do you handle a client that connects but doesn’t send a request?
Socket Timeout

- need to recycle idle sockets, avoid denial-of-service attacks
- use setsockopt()

```
1 int setsockopt(int s, int level, int optname,
2       const void *optval, socklen_t optlen); 
3
4 s = socket
5 level = SOL_SOCKET (for socket options)
6 optname = name of option, see socket(7)
7 optval = pointer to appropriate data structure
8 optlen = length of data structure
```
Socket Timeout Example

```c
struct timeval tv;
tv.tv_sec = 1;
tv.tv_usec = 0;
if (setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)) < 0) {
    perror("Set socket option");
    close(sockfd);
}
```

- read actual timeout value from configuration file
- `recv()` will return -1 with `errno == EAGAIN` on timeout
Getting the Time

```c
1 time_t time(time_t *t);
```

- returns the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds
- usually pass NULL as the argument to time()
- see time(2) for details
Converting to GMT

1. `struct tm *gmtime(const time_t *timep);`
2. `timep = time given by time() or fstat()`

- on success returns `struct tm` pointer
- on error returns `NULL`
- details of structure given in `ctime(3)`
Converting to RFC 822, 1123 Time Format

- the recommended date format for HTTP
- used in the Date and Last-Modified headers

```c
size_t strftime(char *s, size_t max, const char *format, const struct tm *tm);
```

- on success returns number of characters stored in buffer
- on error returns 0
- the magic format string for RFC 1123 time:

```plaintext
%a, %d %b %Y %H:%M:%S GMT
```
From Time to Time

```c
1  string date(time_t t)
2  {
3       struct tm *gmt;
4       char buf[200];
5
6       memset(buf,0,200);
7       gmt = gmtime(&t);
8       if (gmt == NULL)
9           return "";
10      if (strftime(buf,sizeof(buf),"%a, %d %b %Y %H:%M:%S GMT",gmt) == 0)
11         return "";
12      return string(buf);
13  }
```

- pass in either time(NULL) for current time or the file modification time from fstat()