CS341 Spring 2018
Programming Assignment #1

Socket Programming

KAIST

Advanced Networking Laboratory

Notice

- Due: 9am, Wed, March 21st, 2018
- This is not a team project. Do it by yourself.

Overview

- In this assignment, you will learn to do socket programming on Linux via implementing a simple HTTP server and a client.

- Client:
  - Implement only GET and POST requests of HTTP 1.0.

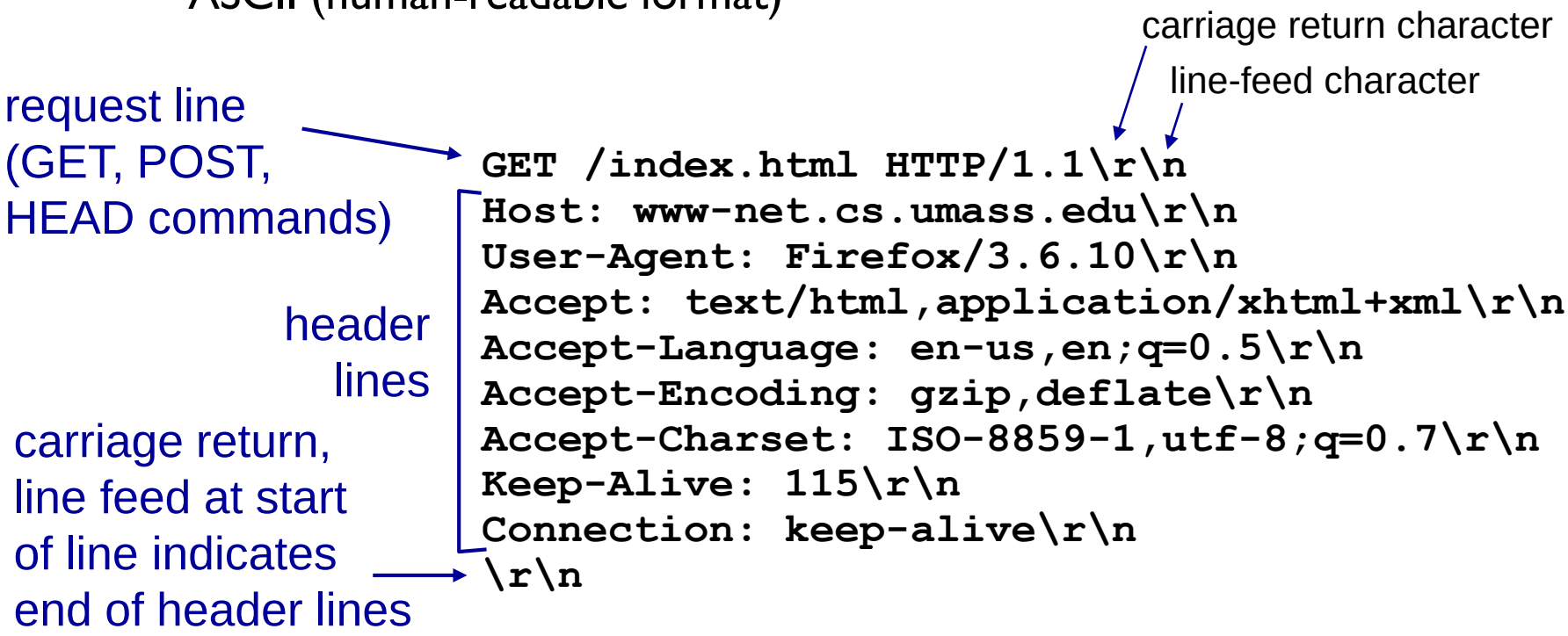
- Server:
  - Implement only the following RESPONSE status codes.
  - 200 OK / 400 Bad Request / 404 Not Found

Basic Socket API

- Server:
  - `socket()`
  - `bind()`
  - `listen()`
  - `select()`
  - `accept()`
  - `read()/write()`
  - `close()`
- Client:
  - `socket()`
  - `connect()`
  - `read()/write()`
  - `close()`

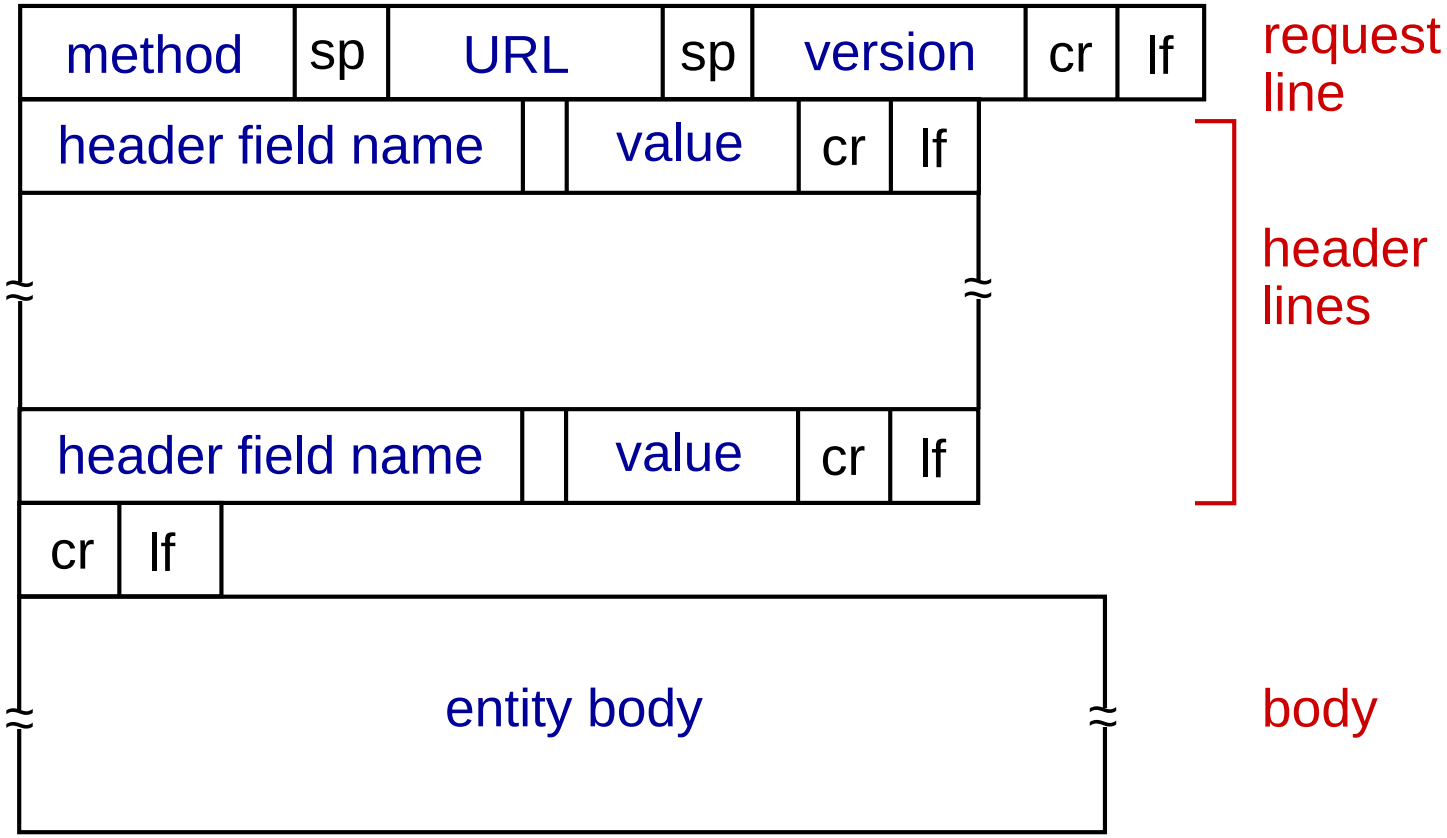
HTTP request message

- two types of HTTP messages: *request, response*
- HTTP request message:
  - ASCII (human-readable format)



slide borrowed from Kurose's Lecture slide

HTTP request message



slide borrowed from Kurose's Lecture slide

HTTP response message

status line (protocol status code status phrase)

header lines

data, e.g., requested HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
   GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
   1\r\n
\r\n
data data data data data ...
```

slide borrowed from Kurose's Lecture slide

Client

- Your client must be able to send GET and POST requests to the server.
- Write a program `client` such that it
  - Fetches payload by sending a GET request with a URL
    - ./client –G URL
  - Uploads payload by sending a POST request with a URL
    - ./client –P URL

Client

- Examples
  - ./client –G URL
    - This should print out the content of URL via standard output.
  - ./client –P URL
    - This should read standard input until the EOF appears.
    - Then, this should send out the data received from the standard input.
    - Hint 1: Don't forget "Content-Length:" in the header!
    - Hint 2: use redirection for easy test!
      - e.g.) ./client –P URL < aqours.jpg

URL

- http://www.kaist.ac.kr:8080/test.html
  - www.kaist.ac.kr must be replaced with your server IP address
  - 8080 must be replaced with your server port number
- In a "GET" request, the client should fetch "test.html" from the server.
- In a "POST" request, the body of the request the client sends should be stored in "test.html" on the server.

Port Numbers

- 0 to 1023: well-known ports
  - 20/21 FTP, 22 SSH, 25 SMTP, 80 HTTP, …
- 1024 to 49151: registered ports w/o superuser privileges
- 49152 to 65535: typically assigned sequentially thru connect()
- Pick a port number larger than 1024 and wish for no collision!

Server

- Your server must be able to handle a simple GET/POST HTTP requests.
- Write a program `server` such that
  - Returns corresponding payload to the given URL when a valid GET request arrives.
  - Stores the given payload to the file when a valid POST request arrives.
  - Handles error when an invalid request arrives
    - 400(Bad Request): A HTTP Request is not well-formatted
    - 404(Not found): There's nothing to serve for the given URL
- Your server must be executable with this command.
  - ./server –p [port_number]

Server

- Examples:
  - GET /xxx HTTP/1.0
    Host: [server_ip]:[server_port]
    - 404 Not Found
  - POST /xxx HTTP/1.0
    Host: [server_ip]:[server_port]

    wow
    - 200 OK
    - Stores file “xxx”, with content “wow”
  - GET /xxx HTTP/1.0
    Host: [server_ip]:[server_port]
    - 200 OK “wow”

Evaluation

- Server:
  - When a client requests the HTML file, the server must send the correctly formatted response.
  - When a client requests anything else, the server must send one of the remaining two response status codes: 400 or 404.
  - The server must use select() and be able to serve more than one client at any time.
  - We will use both a valid client and an invalid client when we test your server program.
- Client:
  - Your client will be tested with our valid server.
  - Note that only valid request will be accepted to our server.

Submission

- Submit a zip file that contains
  - Your code
  - A Makefile file that builds your code
  - readme.txt
- The name of the zip file must follow the format below.
  - [student_id]_[student_name]_PJ1.zip
  - e.g.) 20169997_SeokjuHong_PJ1.zip
- Submit the zip file to the link below.
  - https://www.dropbox.com/request/IDBoS3u4JBawck13Lsnz

Common Requirements for the Assignments

- Only in C programming language & Linux environment
- Use C standard libraries & Linux system calls only
  - No other 3rd party libraries!
- Implementation with blocking I/O will be given half credit.
  - To get full credit, implement non-blocking I/O with select().
- Data over the network must follow the network byte ordering (= Big-endian)
  - Refer to the functions such as htons(), htonl(), ntohs(), ntohl()
- **Provide a Makefile (make all) and follow the argument format** (Automated scripts will grade your program)

Recommended Links

- MDN Web docs
  - Before you begin the project, read the request and response format carefully.
  - Don't forget to include headers.
  - Content-Length header
    - https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Length
  - Host header
    - https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Host

Recommended Links

- These links contain useful info and example codes
  - Beej's Guide to Network Programming
    - http://beej.us/guide/bgnet/html/single/bgnet.html
  - Implementation of simple server with select()
    - https://en.wikipedia.org/wiki/Select_(Unix)
- VirtualBox
  - We recommend you to use VirtualBox to establish a virtualized Linux environment on Windows
  - https://www.virtualbox.org/

Supplement slides for socket APIs

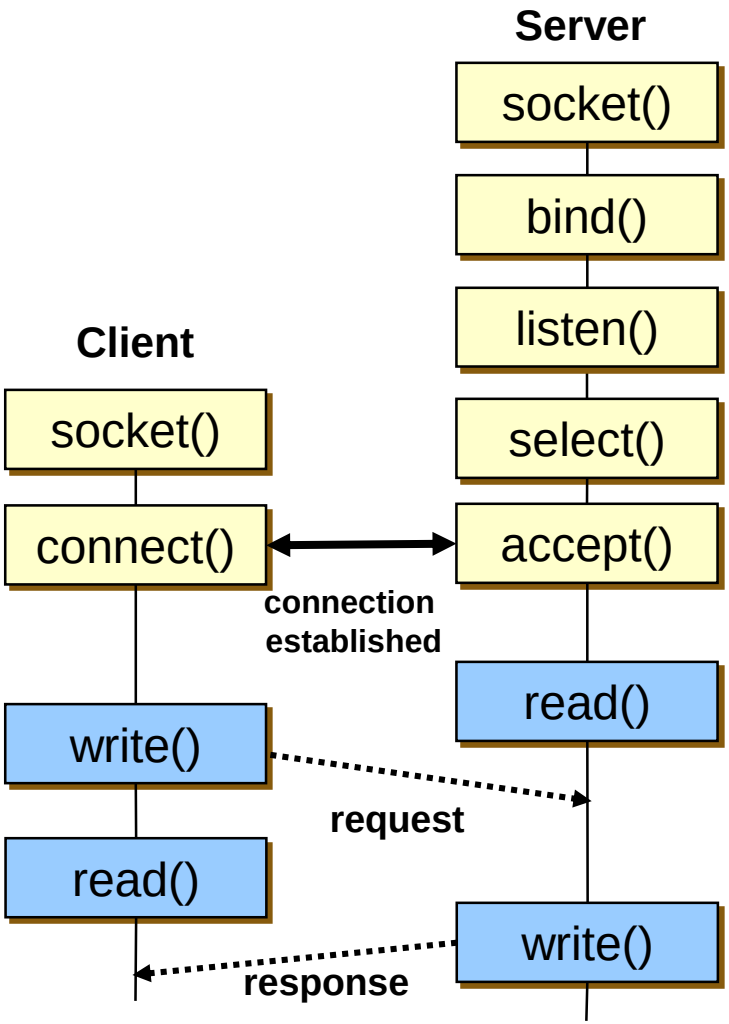
Socket

- An interface between application and network
- Unique identification to or from which information is transmitted in the network

- Clients and servers communicate with each other by reading from and writing to socket

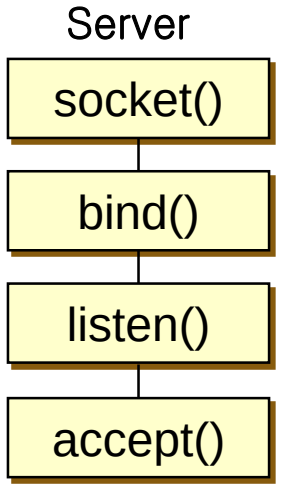
- In UNIX-like systems, a socket descriptor is just another file descriptor

How socket API calls proceed



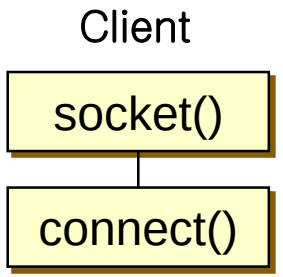
Server-side Basic Socket API

- socket()
  - Creates a new socket and returns its socket descriptor
- bind()
  - Associates a socket with a local port number and IP address
- listen()
  - Prepares a socket for incoming connections
- accept()
  - Accepts a received incoming attempt from client
  - Creates a new socket associated with a new TCP connection



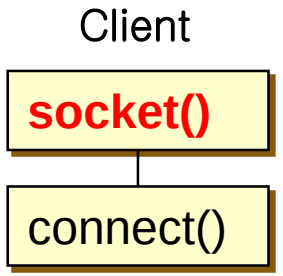
Client-side Basic Socket API

- socket()
  - Creates a new socket and returns its socket descriptor
- connect()
  - Binds a destination to a socket or set a connection



Socket API: socket()

- int socket (int family,
  int type,
  int protocol)
  - socket() creates a socket descriptor.
  - family specifies the protocol family.
    - AF_UNIX: Local Unix domain protocols
    - AF_INET: IPv4 Internet protocols
  - type specifies the communication semantics.
    - SOCK_STREAM: provides sequenced, reliable, two-way, connection-based byte streams
    - SOCK_DGRAM: supports datagrams (connectionless, unreliable messages of a fixed maximum length)
    - SOCK_RAW: provides raw network protocol access
  - protocol specifies a particular protocol to be used with the socket.

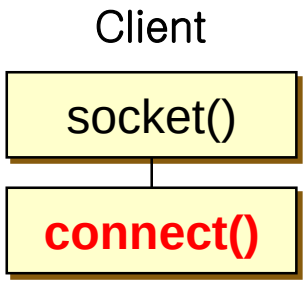


Socket API: connect()

- int connect (int sockfd,

  const struct sockaddr *servaddr,

  socklen_t addrlen)
  - TCP client uses to establish a connection with a TCP server.
  - servaddr contains *{IP address, port number}* of the server.
  - The client does not have to call bind() before calling connect().
    - The kernel will choose both an ephemeral port and the source IP address if necessary.
  - Client process suspends (blocks) until the connection is created.

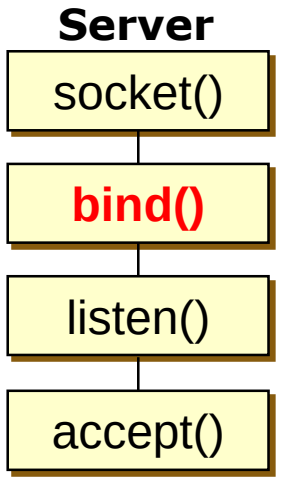


Socket API: bind()

- int bind (int sockfd,

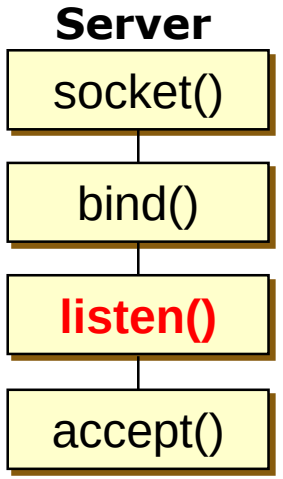
  struct sockaddr *myaddr,

  socklen_t addrlen)
  - bind() gives the socket sockfd the local address myaddr.
  - myaddr is addrlen bytes long.
  - Servers bind their well-known port when they start.
  - If a TCP server binds a specific IP address to its socket, this restricts the socket to receive incoming client connections destined only to that IP address.
  - Normally, a TCP client let the kernel choose an ephemeral port and a client IP address.

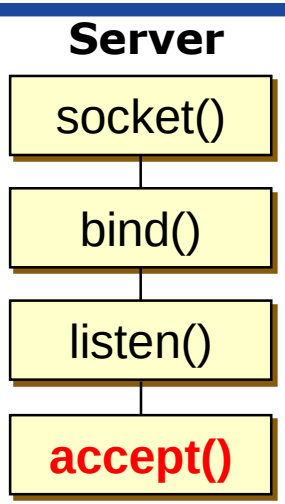


Socket API: listen()

- int listen (int sockfd,
  int backlog)
  - listen() converts an unconnected socket into a *passive socket*, indicating that the kernel should accept incoming connection requests.
    - When a socket is created, it is assumed to be an *active socket*, that is, a client socket that will issue a connect().
  - backlog specifies the maximum number of connections that the kernel should queue for this socket.
  - Historically, a backlog of 5 was used, as that was the maximum value supported by 4.2BSD.
    - Busy HTTP servers must specify a much larger backlog, and newer kernels must support larger values.

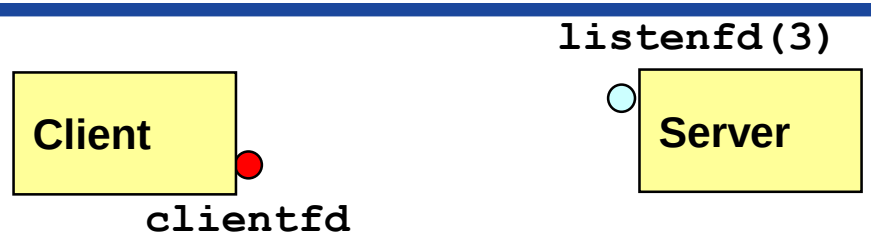


Socket API: accept()

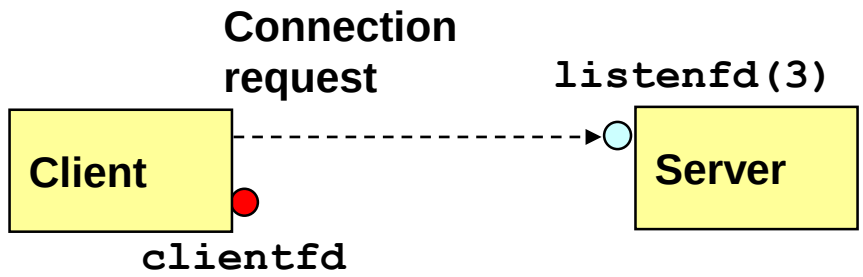


- int accept (int sockfd,
  struct sockaddr *cliaddr,
  socklen_t *addrlen)
  - accept() blocks waiting for a connection request.
  - accept() returns a connected descriptor with the same properties as the listening descriptor.
    - The kernel creates one connected socket for each client connection that is accepted.
    - 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.
  - The cliaddr and addrlen arguments are used to return the address of the connected peer process (the client)

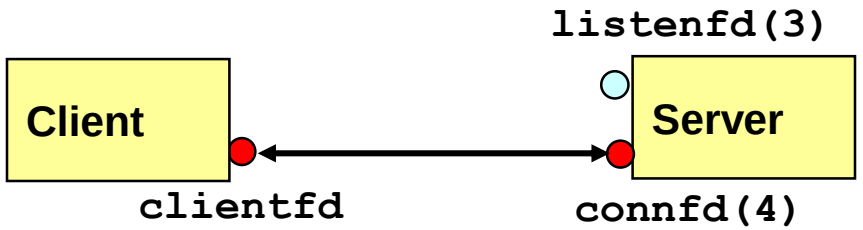
Socket API: accept()



1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`.



2. Client makes connection request by calling and blocking in `connect`.



3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`.

Socket API: accept()

- 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 provide services for client.
- Why the distinction?
  - Allows for concurrent servers that can communicate over many client connections simultaneously.