Lab Introduction + Socket Programming

EE323 Spring 2021

ee323@nmsl.kaist.ac.kr

Lab Introduction

Caution

We do not recommend this course to the students…

- Who take OS this semester (or other tight projects)
- Who have not taken or taking EE209 <Programming Structure for EE>

You should be familiar with

- Linux environment (Ubuntu)
- C language

You should not try any kind of

- Plagiarism
- Cheating

Logistics

In lab sessions, we will give a brief introduction of upcoming project(s).

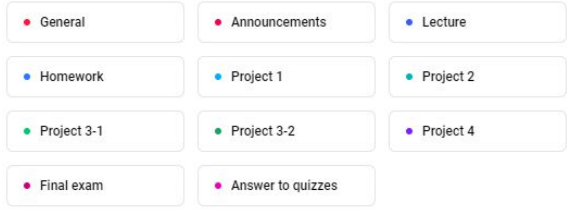
- (4.5%) Lab session #1: Socket Programming
  - Open at 3/10, Due 3/18 (11:55 pm) - 1 week
- (4.5%) Lab session #2: HTTP Proxy
  - Open at 3/19, Due 3/30 (11:55 pm) - 1.5 weeks
- (6+6%) Lab session #3: Simple TCP in Reliable Transport Layer
  - Open at 3/31, Due 4/13, 5/11 (11:55 pm) - 2/4 weeks for each (including Midterm period)
- (9%) Lab session #4: Simple Router
  - Open at 5/12, Due 6/1 (11:55 pm) - 3 weeks

Logistics (2)

No attendance check for lab (but please participate!)

- We will not answer questions covered in lab sessions

Use Campuswire to ask questions
https://campuswire.com/c/G9DCC7D11



- Post questions publicly in appropriate category
- We might change it to public if you post in private
- Anyone can answer any question - it counts to your **participant point**

Use email to ask private issues: ee323@nmsl.kaist.ac.kr

Logistics (3)

Course homepage: https://networking101.org

- Every course materials, slides, lab sessions, and project documents will be uploaded in Campuswire & homepage
- Lecture video links will be announced in Campuswire (not the homepage)
- Submissions (HW, Projects) will be accepted via Google Form
- Grades will be sent via email (your registered Gmail account)

Server (1)

You can freely use **Haedong Lounge Server** for this course, this semester.

- ID: {Student_ID}
- PW: G3JHZ6{Student_ID}
- Change your password immediately with `$ passwd`

You can directly access the machine @ Haedong Lounge (E3-4 Room #1412)

Project #1, #2, #3: You can use your local computing machine on your own responsibility; we will grade with the Haedong Lounge Server.

Project #4: You will install Virtual Machine in your own computing machine (we will grade with the same VM environment).

Server (2)

You can freely use **Haedong Lounge Server** for this course, this semester.

- ID: {Student_ID}
- PW: G3JHZ6{Student_ID}
- Change your password immediately with **`$ passwd`**

What is $? → Bash

How can I remotely access the server?

- **SSH** (**S**ecure **SH**ell Protocol)
  - In your terminal, type **`ssh`** and see what happens
  - You can use GUI programs (ex. PuTTY, VSCode, …)
- **`$ ssh <your_student_id>@eelab<5 or 6>.kaist.ac.kr`**
  - **`ex) $ ssh 20201234@eelab6.kaist.ac.kr`**

Server (3)

You can freely use **Haedong Lounge Server** for this course, this semester.

- ID: {Student_ID}
- PW: G3JHZ6{Student_ID}
- Change your password immediately with `$ passwd`

How can I upload/download the file?

- **SFTP** (**S**ecure **F**ile **T**ransfer **P**rotocol): based on SSH
  - In your terminal, type **sftp** and see what happens
  - You can use GUI programs (ex. Filezilla, ...)

Server (4)

You can freely use **Haedong Lounge Server** for this course, this semester.

- ID: {Student_ID}
- PW: G3JHZ6{Student_ID}
- Change your password immediately with `$ passwd`

I'm out of campus. How can I connect to the server?

- **KVPN** (**K**ASIT **V**irtual **P**rivate **N**etwork): https://kvpn.kaist.ac.kr/
- Login ID/PW would be same as Welcome_KAIST.

Project #1: Socket Programming

The Ultimate Guide

Primary Project Document

http://bit.ly/ee323-proj1-2021

This slide is based on the document above.

Please refer to this document first if there is any question.

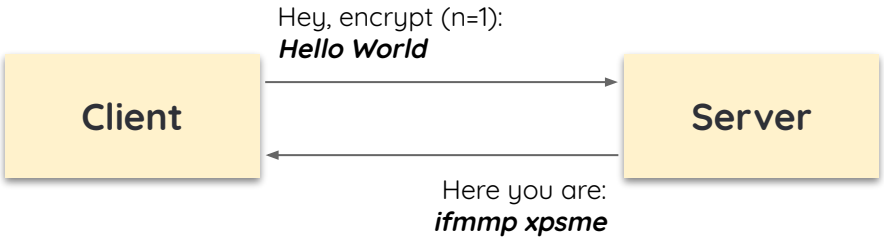
Still ongoing project - we need you help and participation!

You can view and comment on the document directly, so please participate.

(hey, it's a rich source of participation points!)

Objectives

- Review basic concepts and API of network socket
- Implement a connection-oriented, client/server protocol based on a given specification
- Send/Receive data via **sockets**
- Implement basic encryption scheme **(Caesar cipher)**



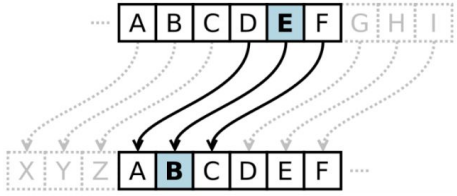
Caesar Cipher

Caesar cipher (a.k.a shift cipher): the simplest form of encryption

Each character is replaced with a character corresponding to n letters up (encryption) or down (decryption) in alphabet

- Ex: a → b when n=1

For better understanding of Caesar Cipher, check: https://cryptii.com/pipes/caesar-cipher



$$E_n(x) = (x + n) mod\ 26$$
$$D_n(x) = (x - n) mod\ 26$$

Caesar Cipher (2)

For this project, there might be characters with uppercase alphabet, or special characters.

- Convert uppercase characters to lowercase
  - `int tolower(int c)`
- After the conversion, encrypt/decrypt only lowercase alphabet chars
  - Ignore special characters (e.g., white spaces, line break, …)

Example)

Plaintext: The quick brown fox jumps over the lazy dog!

Ciphertext: qeb nrfzh yoltk clu grjmp lsbo qeb ixwv ald!

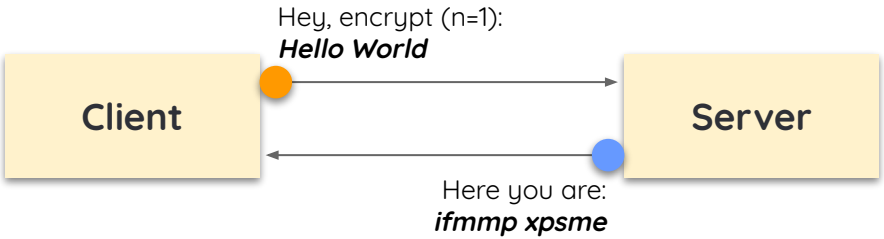
Socket (Stream Socket)

An interface between application and network

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

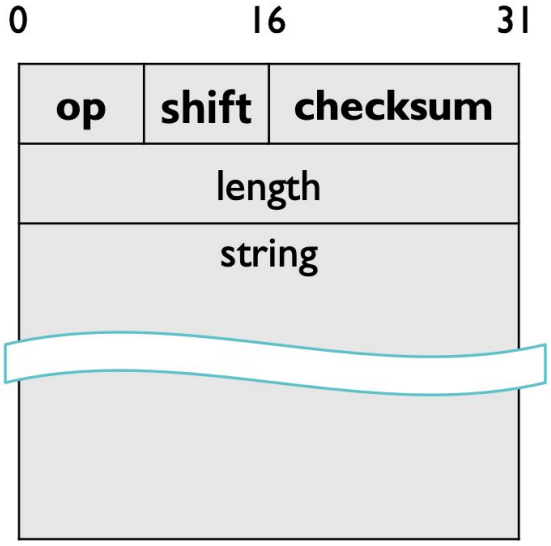
In UNIX-like system, a socket descriptor is just an another file descriptor

- File descriptor: An integer that is related to any I/O operation



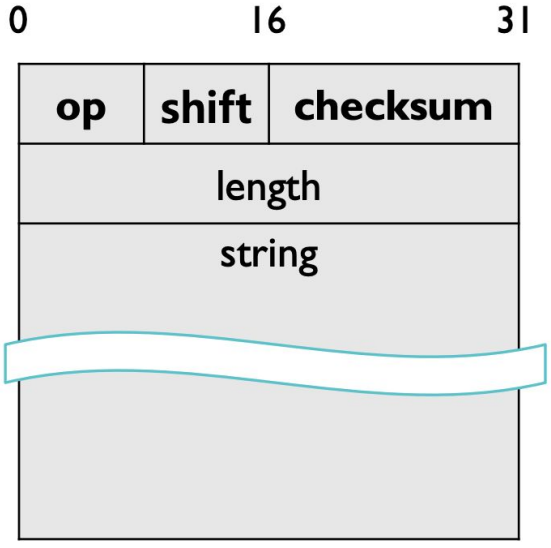
Protocol Specification

- **op** field
  - 8 bits, operation type
  - 0: encrypt / 1: decrypt
- **shift** field
  - 8 bits, # of shifts (n)
  - Must be treated as **unsigned (non-negative; 0~255)**
- **checksum** field
  - 16 bits
  - Used for error-checking of protocol fields
  - Checksum must be calculated in the same way as TCP checksum (1's complement)
  - Check the document; Check RFC 793 (TCP/IP)
    - **Request For Comments**: a document that describes the standards, protocols, and technologies related to the Internet

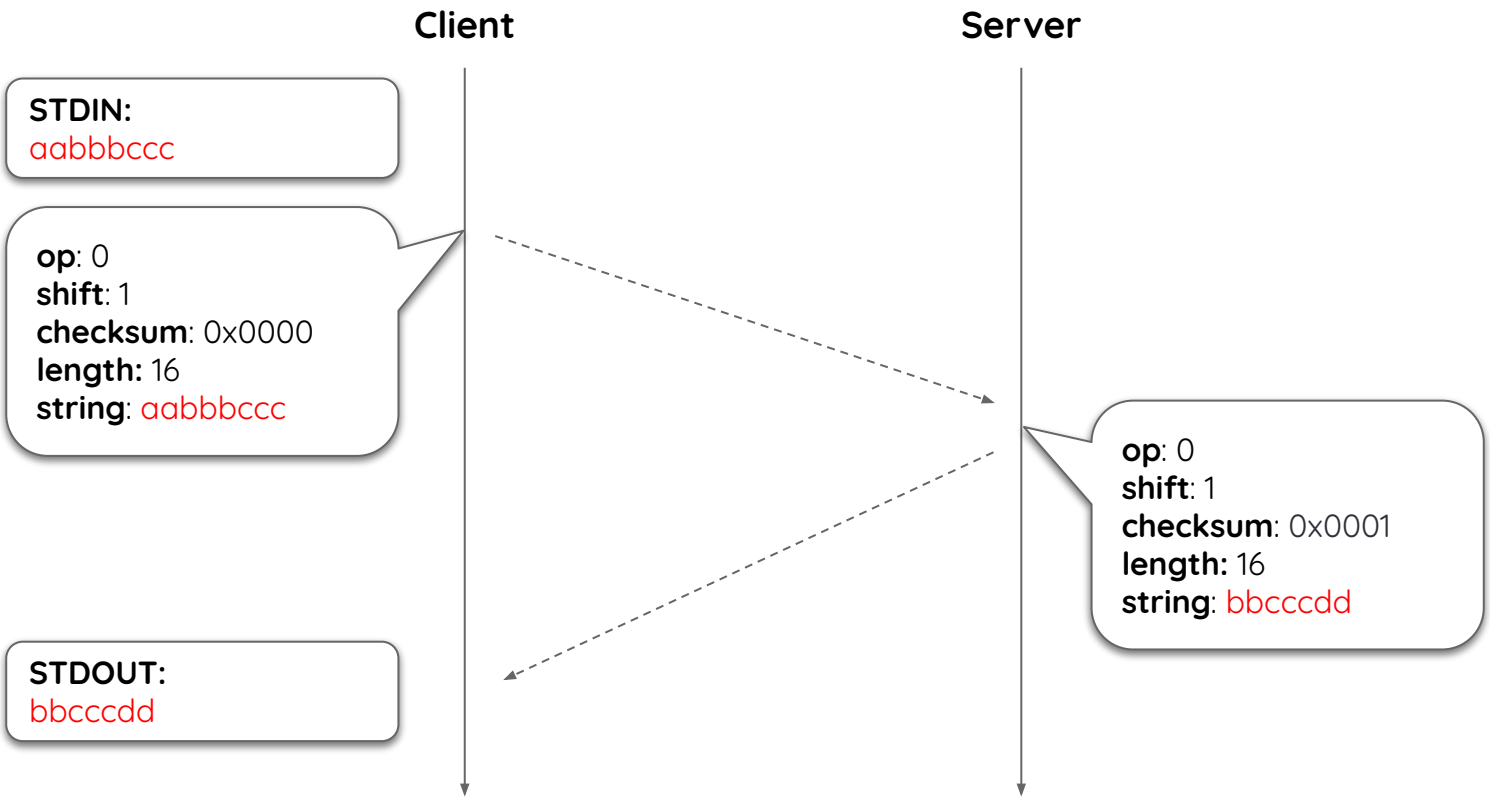


Protocol Specification (2)

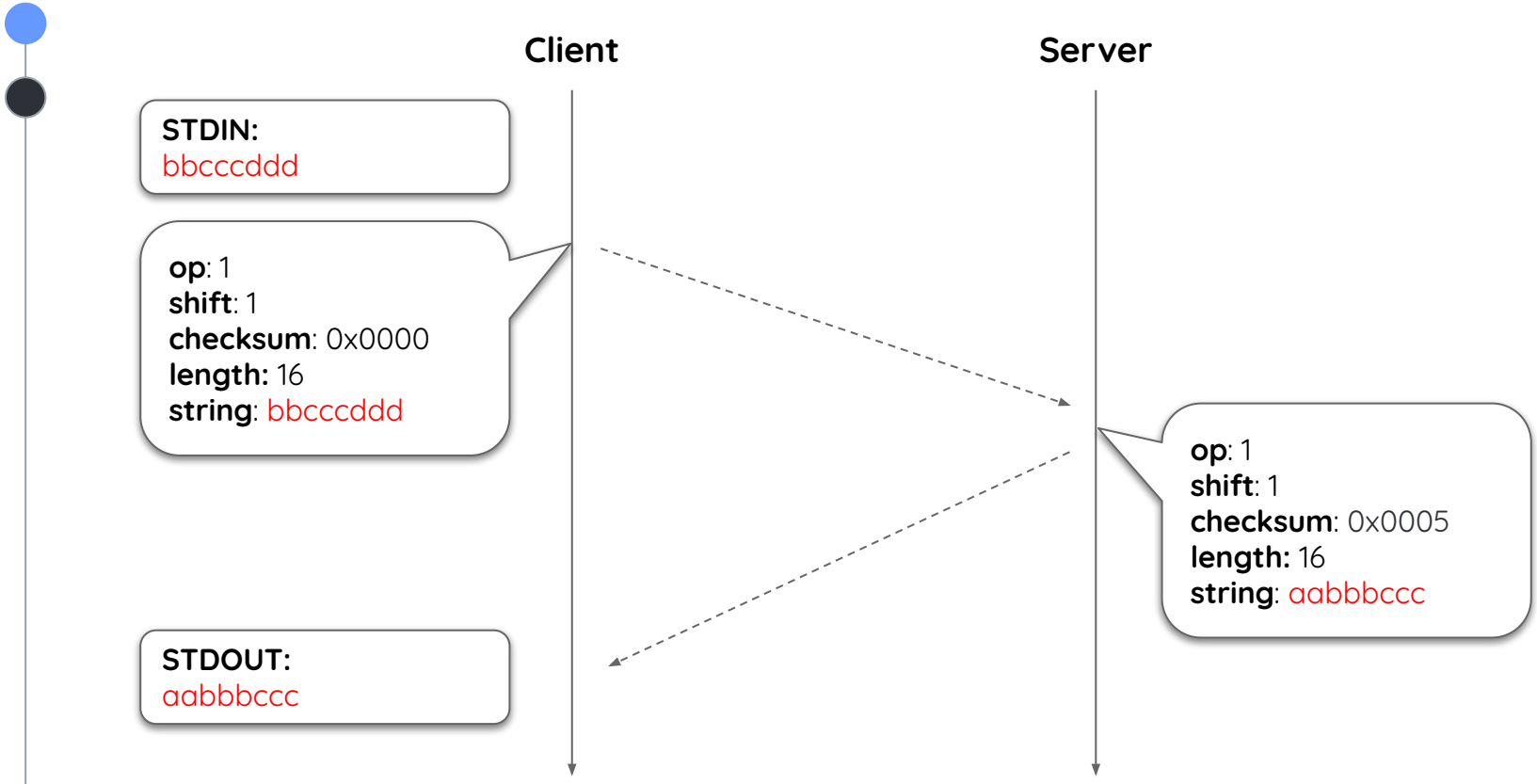
- **length** field
  - 32 bits, **network order**
  - Total length of a message
    - Including op, shift, checksum, length, and string
  - Unit in byte (not bit!)
  - **Should be in Network Order**
  - **Maximum length is 10MB** (= 10,000,000 bytes)
- **string** field
  - String to be transmitted
  - Not a fixed size; determined from length field.



Communication Example (Encryption)



Communication Example (Decryption)



Client Specifications

Client program should take command-line parameters:
IP address, **port**, **operation**, and **shift**

- `$ ./client -h 143.248.111.222 -p 1234 -o 0 -s 5`
  - -h option: Server IP address
  - -p option: Server port number
  - -o option: 0 for encrypt, 1 for decrypt
  - -s option: shift value (n)

Take input through STDIN, and print output through STDOUT

Terminate program when EOF (end-of-file) is received

Server Specifications

Server should take one command-line parameter

- `$ ./server -p 1234`
- -p option: server port number

Handle connections, receive messages and reply

Handle multiple connections in parallel

- Options that you can consider to use: `fork()` / `select()` / `epoll()`

Requirements

Implementation restrictions

- Only in C programming language & Linux environment
- Use C standard libraries & Linux system calls only
  - No other 3rd party libraries
- Follow network byte ordering (Big-endian)

Include `Makefile`

- We do not provide any skeleton codes
- `$ make all` should generate two executables: `client` and `server`

Write a report

- Briefly explain you implementation. **No more than 3 pages. PDF only.**

Testing

Sample input/output files:
https://drive.google.com/file/d/1kkWCUMJl8PQDYG_ttzLGy3AYnTo5tfxP/

Sample test servers:

- Server 1 (eelab5.kaist.ac.kr:12000): Reference solution
- Server 2 (eelab5.kaist.ac.kr:12001): No checksum validation
- Server 3 (eelab5.kaist.ac.kr:12002): Respond with wrong checksum values
- Server 4 (eelab5.kaist.ac.kr:12003): Laggy responses (should be handled)

We will grade at the same environment as the Haedong Server

We will not tolerate any issue related to the different grading environment

Grading Criteria

This project is worth **4.5%** of your total grade.

- (30%) Code: Client implementation
- (50%) Code: Server implementation
- (20%) Code: Makefile
- (0%) Report: We will use it when there's any grading issue

Any violations will result in penalty

- Wrong Makefile script
- Wrong file names
- Wrong report format
- And more...

Submission

Report + Source code of client & server (No skeleton code provided)

Example:

1. client.c
2. server.c
3. Makefile
4. report.pdf

Compress in a single ZIP file:

- {StudentID}_{Name (in English)}_project1.zip
  (ex. 20211234_HyungJunYoon_project1.zip)

Due March 18th, Thursday, 11:55 pm (No late submission)

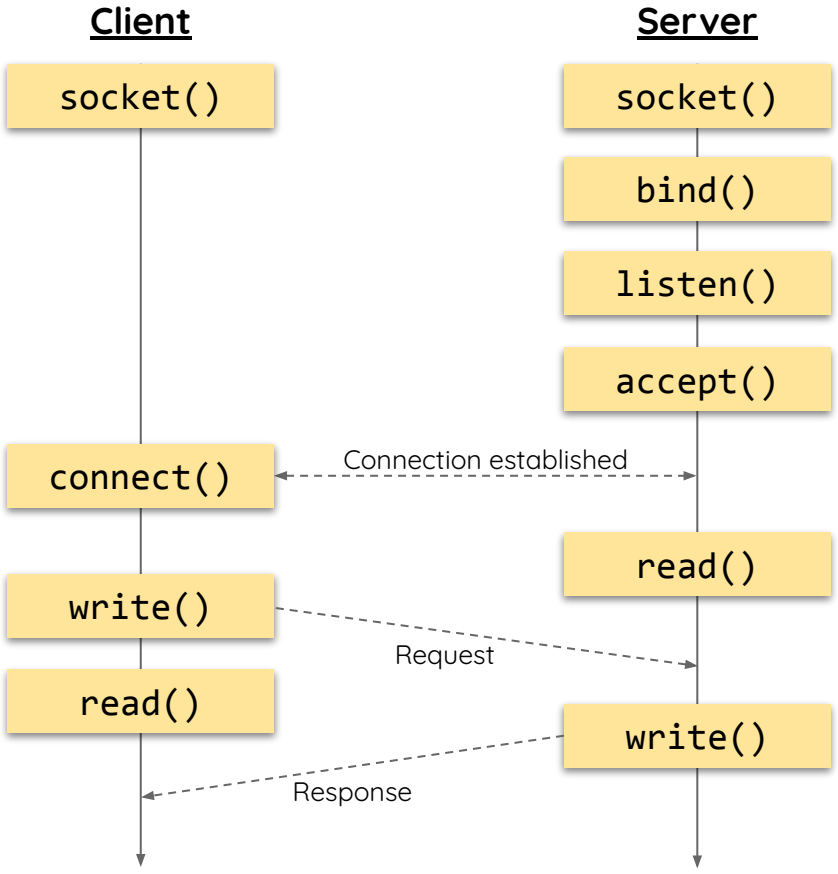
Submit here: https://bit.ly/ee323-proj1-2021-submit

Advices

1. Check Beej's Guide to Network Programming:
   a. https://beej.us/guide/bgnet/html//index.html
2. Be used to read function APIs; arguments and return values.
3. Think of corner cases and handling errors before writing the code.
4. Don't be afraid to ask questions in Campuswire & e-mail :D
5. **Start early - we only have about a week. NO LATE SUBMISSION!**

Supplement Slides for Socket APIs

Connection-oriented Data Transmission



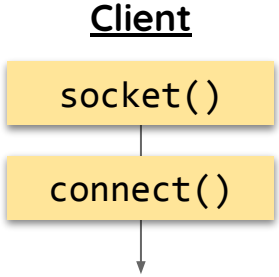
Client-side Basic Socket API

`socket()`

- Creates a new socket and returns its socket descriptor

`connect()`

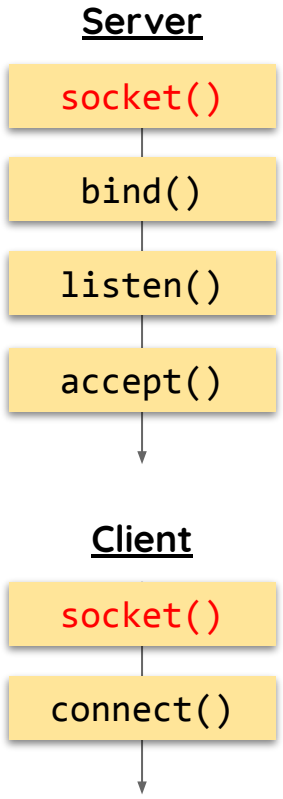
- Set a destination (IP/port) and connect



Socket API: `socket()`

```
int socket (int family, int type, int protocol)
```

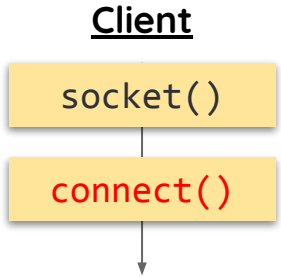
- Creates a socket descriptor
- `family` specifies the protocol family
  - `AF_UNIX`: Local Unix domain protocols
  - `AF_INET`: IPv4 Internet protocols
- `type` specifies the communication semantics
  - `SOCK_STREAM`: provides sequenced, reliable, two-way, connection-based byte streams
  - `SOCK_DGRAM`: 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
  - Usually zero



Socket API: `connect()`

```
int connect (int sockfd, const struct sockaddr *servaddr,
socklen_t addrlen)
```

- **`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



Server-side Basic Socket API

`socket()`

- Creates a new socket and returns its socket descriptor

`bind()`

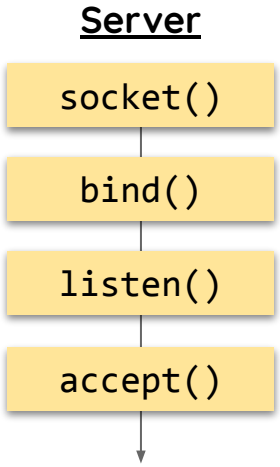
- Associates a socket with a local port number and IP addr.

`listen()`

- Make a socket ready for incoming connections

`accept()`

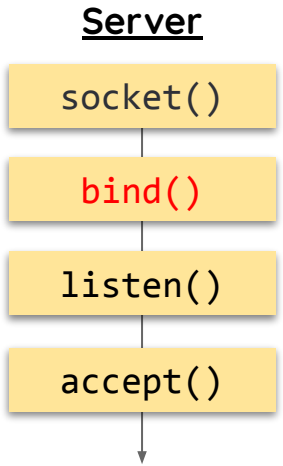
- Accepts a received incoming attempt from client
- Returns a new socket descriptor associated with a new connection



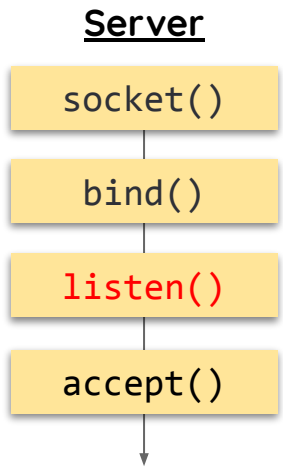
Socket API: `bind()`

```
int bind (int sockfd, const struct sockaddr *addr,
socklen_t addrlen)
```

- Associates a socket with a port number and IP addr. (check **`INADDR_ANY`**)
- Any packet arriving to server's IP address which matches 1) port number, and 2) incoming IP address would be accepted after **`listen()`** call



Socket API: `listen()`



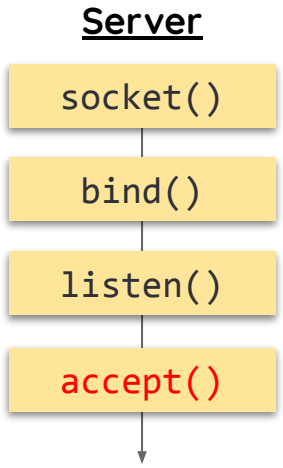

```
int listen (int sockfd, int backlog)
```

- 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
  - Setting to 10 would be sufficient (you should make program parallel)

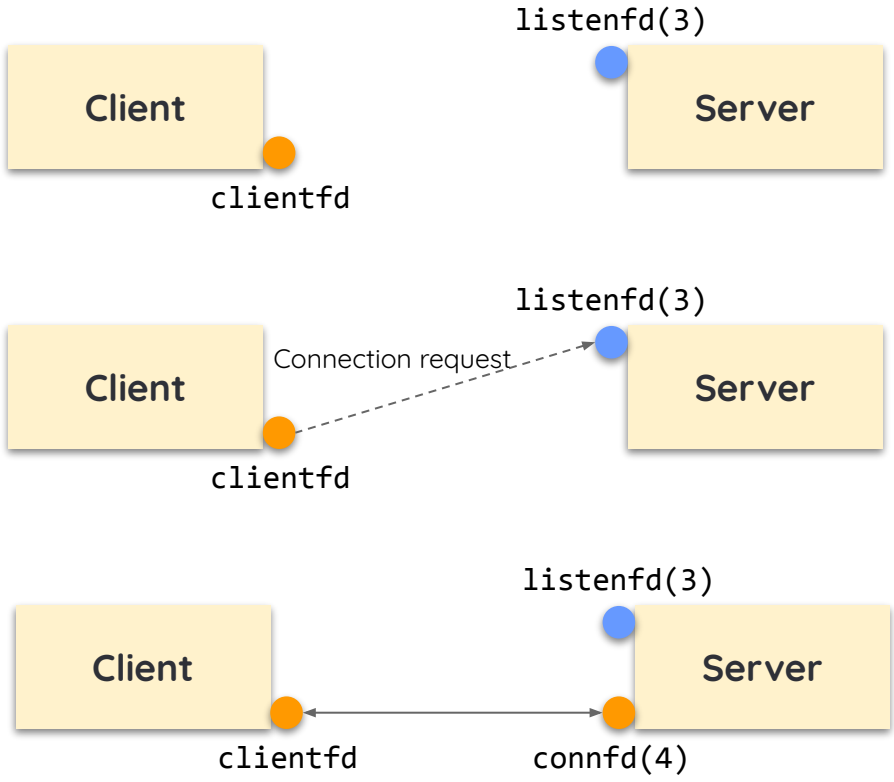
Socket API: `accept()`

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

- Blocks until a connection request arrives
- 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 connected socket
- The **`cliaddr`** and **`addrlen`** arguments are used to **return** the address of the connected peer process (the client)



Socket API: `accept()` (2)

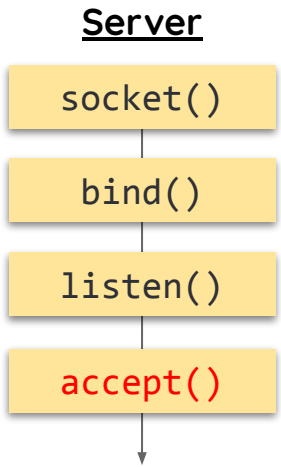


1. Server blocks in **`accept()`**, waiting for connection request on listening descriptor **`listenfd`**

2. Client makes connection request by calling and being blocked in **`connect()`**

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

Socket API: `accept()` (3)



Listening descriptor (passed as argument)

- Endpoint for client connection requests
- Created once and exists for lifetime of the server

Connected descriptor (returned from function)

- Endpoint of the connection between client and server
- A new descriptor is created each time the server accepts a new connection request
- Exists only as long as it takes to provide services for client

Why do we need to separate them?

- Allows concurrent servers to communicate over many client connections simultaneously