

Socket Programming

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH



Contents

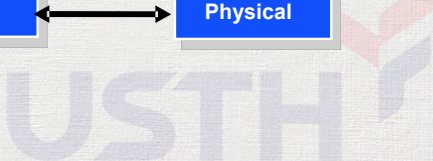
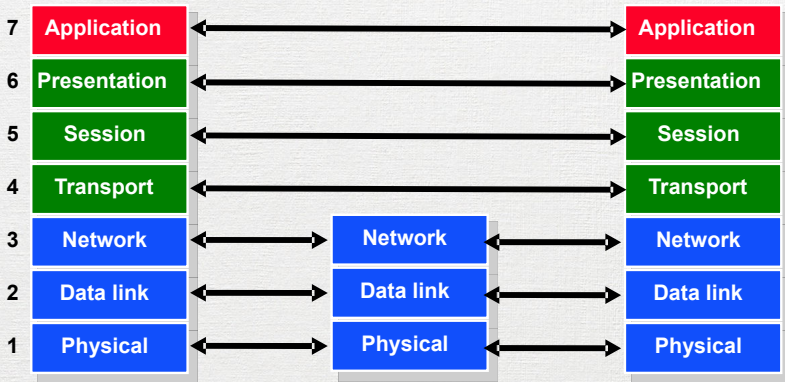
- What's a socket?
- Why socket?
- What's in a socket?
- How to use sockets?



What & Why?



Layers



Why layering?

- Similar to application layering
- Application programmer
 - Doesn't care about routing
 - Doesn't care about Ethernet frame
 - Doesn't care about WiFi WPA2 encryption
 - Doesn't care about reliability implementations



Why layering?

- Application programmer
 - Passes the data down
 - Focus on the application



Why layering?

Lower layers:

- What does they need to know?



Why layering?

Lower layers:

- What does they need to know?
- Destination
 - Where to?
 - Hostname («resolved» with `gethostbyname()`)
 - IP address
 - Which service?
 - Indicated by port number



Socket: What?

- Endpoint of a two-way communication link between two networked programs
- Represented by a file descriptor after creation
 - Unix philosophy: Everything is a file
- *De facto* standard for TCP/UDP, replaced
 - NetBIOS / NetBEUI
 - IPX / SPX



Socket: Applications

Most network applications use sockets

- Send messages
- Share data: image, music, video, “cast”
- Interprocess Communication (IPC)



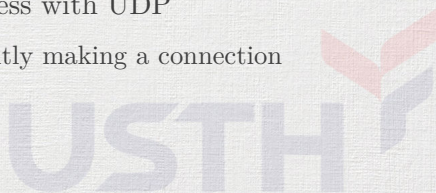
Socket: Which types?

- Stream sockets: connection-oriented with TCP
 - Make a connection
 - Transfer data
 - Close connection
 - Ensure sequence, error checking, etc. . .
 - Example: youtube video



Socket: Which types?

- Stream sockets: connection-oriented with TCP
 - Make a connection
 - Transfer data
 - Close connection
 - Ensure sequence, error checking, etc. . .
 - Example: youtube video
- Datagram sockets: connectionless with UDP
 - Transfer data without explicitly making a connection
 - Example: DNS



Socket: Why?

- The **standard** API for connecting processes
 - Local
 - Networked



Socket: Why?

- The **standard** API for connecting processes
 - Local
 - Networked
- Compatibility: widely supported
 - Linux / UNIX
 - Windows
 - macOS



Socket: Why?

- The **standard** API for connecting processes
 - Local
 - Networked
- Compatibility: widely supported
 - Linux / UNIX
 - Windows
 - macOS
- Low level
 - Minimize amount of data transfer
 - Fast, very little overhead
 - Customizable, flexible, self-defined protocol

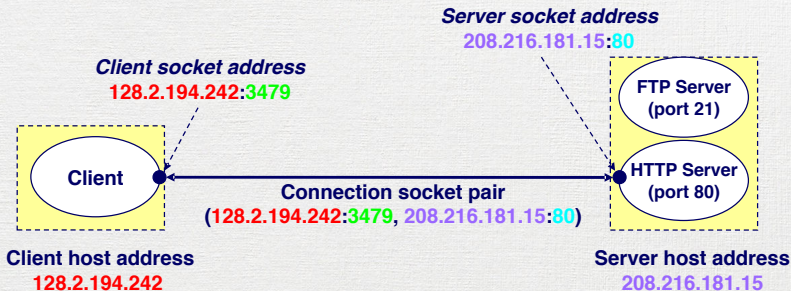


Why NOT socket over higher level libraries?

- Low level: more efforts
 - Define protocol
 - Message boundaries
 - Data representation
 - Security
- Session control
 - Authentication, etc.



What's in a socket?



Overview & Setup



Overview

Server

- Passively waits
- Passive socket

Client

- Initiates the connection
- Active socket



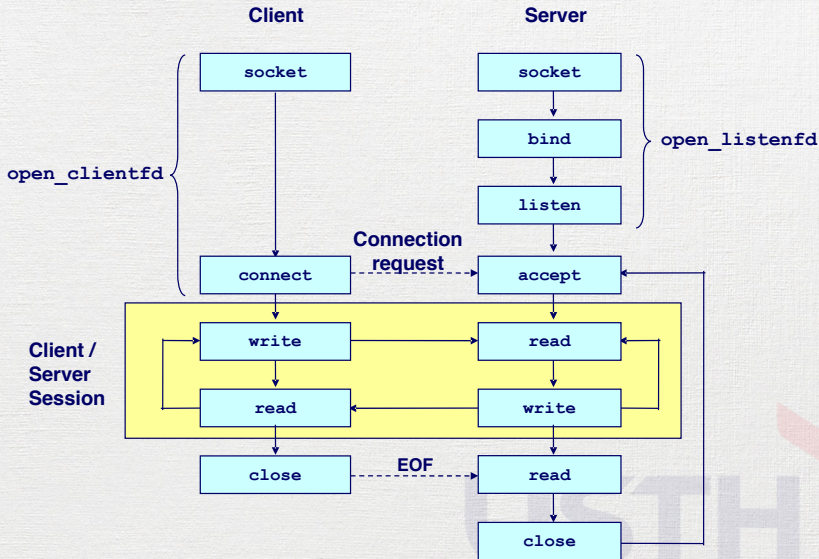
Overview

Steps

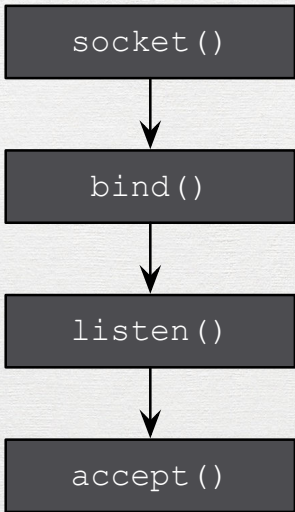
- Setup
 - Where is the remote host?
 - What service?
- Transfer Data
 - Send/Receive ~ write() / read()
- Close



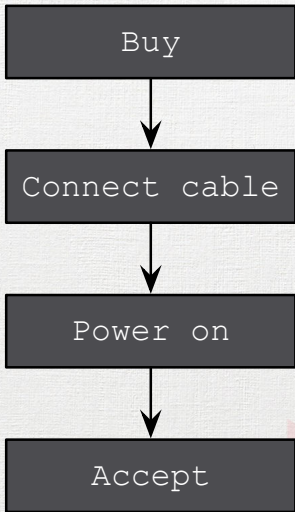
Socket Overview



Server: Overview



Server socket



Landline phone

Socket: Important struct

```
struct sockaddr_in {
    short          sin_family;    // e.g. AF_INET
    unsigned short sin_port;     // e.g. htons(3490)
    struct in_addr sin_addr;     // see struct in_addr, below
    char          sin_zero[8];  // zero this if you want to
};

struct in_addr {
    unsigned long s_addr; // load with inet_aton()
};
```



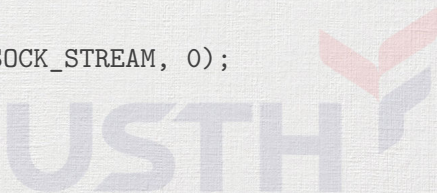
Server: Setup socket()

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

- domain: AF_INET (IPv4) or AF_INET6 (IPv6)
- type: SOCK_STREAM (TCP) or SOCK_DGRAM (UDP)
- protocol: 0

For example:

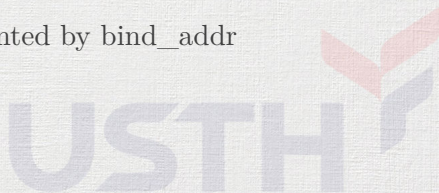
```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```



Server: Binding `bind()`

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

- `sockfd`: file descriptor that `socket()` returned
- `bind_addr`: a «`struct sockaddr_in`» for IPv4
- `addrlen`: size of the struct pointed by `bind_addr`



Server: Example for `socket()` and `bind()`

```

struct sockaddr_in saddr;
int sockfd;
unsigned short port = 80;
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    printf("Error creating socket\n");
    ...
}

memset(&saddr, 0, sizeof(saddr));
saddr.sin_family = AF_INET;
saddr.sin_addr.s_addr = htonl(INADDR_ANY);
saddr.sin_port = htons(port);
if (bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) {
    printf("Error binding\n"); ...
}

```

Server: Listen to incoming connections `listen()`

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

- `sockfd`: file descriptor that `socket()` returned
- `backlog`: number of pending connections to queue

For example:

```
listen(sockfd, 10);
```



Server: Accept an incoming connection `accept()`

- Server must explicitly accept incoming connections

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

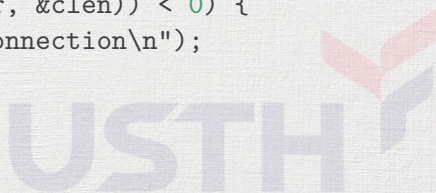
- `sockfd`: file descriptor that `socket()` returned
- `addr`: pointer to store client address
- `addrlen`: size of `addr`
- Returns a file descriptor for the connected socket

For example:

```
int client = accept(sockfd,
                   (struct sockaddr_in *) &caddr, &crlen);
```

Server: Example for `listen()` and `accept()`

```
if (listen(sockfd, 5) < 0) {  
    printf("Error listening\n");  
    ...  
}  
  
clen=sizeof(caddr);  
if ((clientfd=accept(sockfd,  
    (struct sockaddr *) &caddr, &clen)) < 0) {  
    printf("Error accepting connection\n");  
    ...  
}
```



Server: Complete Example

```

int sockfd, clen, clientfd;
struct sockaddr_in saddr, caddr;
unsigned short port = 80;
if ((sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    printf("Error creating socket\n");
    ...
}

memset(&saddr, 0, sizeof(saddr));
saddr.sin_family = AF_INET;
saddr.sin_addr.s_addr = htonl(INADDR_ANY);
saddr.sin_port = htons(port);

if ((bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) {
    printf("Error binding\n");
    ...
}

if (listen(sockfd, 5) < 0) {
    printf("Error listening\n");
    ...
}

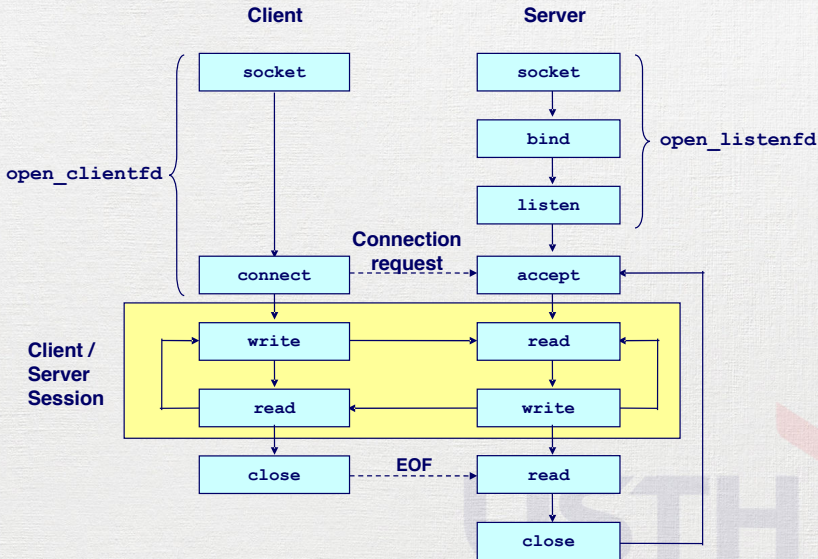
clen=sizeof(caddr);
if ((clientfd=accept(sockfd, (struct sockaddr *) &caddr, &clen)) < 0) {
    printf("Error accepting connection\n");
    ...
}

```

Practical Work 3: Server setup

- Write a new program in C
 - Name it « 03.practical.work.server.setup.c »
 - Write a server that:
 - listens to **TCP** port **8784** [USTH in a T9 dial pad!]
 - binds to all possible interfaces
 - prints a message when a client connects to it
- Deploy to your shiny VPS
- Test the connection
 - Use «telnet» or «nc»
- Push your C program to corresponding forked Github repository

Remind: Socket Overview



Client: Setup socket()

- Similar to server's

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

- domain: AF_INET (IPv4) or AF_INET6 (IPv6)
- type: SOCK_STREAM (TCP) or SOCK_DGRAM (UDP)
- protocol: 0

For example:

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Client: Connect to server `connect()`

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

- `sockfd`: file descriptor that `socket()` returned
- `addr`: pointer to store **server** address
- `addrlen`: size of `addr`

Example:

```
connect(sockfd, (struct sockaddr *) &saddr, sizeof(saddr));
```



Client: Complete Example

```

struct sockaddr_in saddr;
struct hostent *h;
int sockfd;
unsigned short port = 80;
if ((sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    printf("Error creating socket\n");
    ...
}
if ((h=gethostbyname("ict.usth.edu.vn")) == NULL) {
    printf("Unknown host\n");
    ...
}
memset(&saddr, 0, sizeof(saddr));
saddr.sin_family = AF_INET;
memcpy((char *) &saddr.sin_addr.s_addr, h->h_addr_list[0], h->h_length)
saddr.sin_port = htons(port);
if (connect(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) {
    printf("Cannot connect\n");
    ...
}

```

Practical Work 4: Client setup

- Write a new program in C
 - Name it « 04.practical.work.client.setup.c »
 - Write a client that:
 - gets server hostname from program arguments
 - in case no argument, asks hostname from STDIN
 - resolves its IP address, print to STDOUT
 - connects to that server, **TCP** port **8784** [It's **USTH**]
 - prints a message if it connects to server successfully
- Test the connection from your client to your server on VPS
- Push your C program to corresponding forked Github repository

Data Transfer



Data Transfer: Overview

- Two common ways for data transfer
 - `send()` / `recv()`
 - Original socket functions
 - Specific to sockets with «flags»
 - `read()` / `write()`
 - Consider socket as a file
 - Generic functions
- Use either of the two pairs



Data Transfer: `recv()` and `send()`

```
ssize_t recv(int socket, void *buffer, size_t length, int flags);  
ssize_t send(int socket, const void *buffer, size_t length, int flags);
```

- `socket`: use socket file descriptor returned by `socket()` or `accept()`
- `buffer`: buffer to read from / write to
- `length`: size of the allocated buffer (`recv()`) and length of the content (`send()`)
- `flags`: specific “settings” for the request

Example

```
recv(sockfd, buffer, sizeof(buffer), 0);  
send(sockfd, "hello world!\n", 13, 0);
```

Data Transfer: `read()` and `write()`

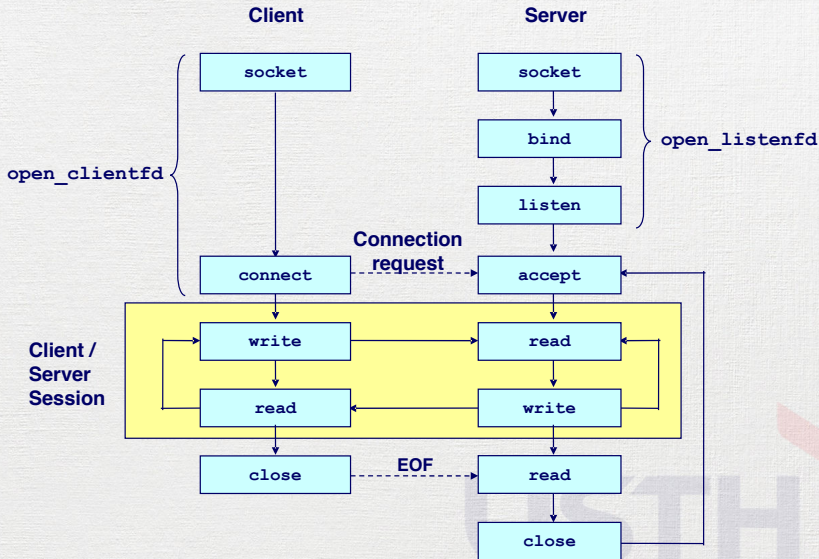
```
ssize_t read(int fd, void *buf, size_t len);  
ssize_t write(int fd, const void *buf, size_t len);
```

- `fd`: file descriptor, use socket file descriptor returned by `socket()` or `accept()`
- `buf`: buffer to read from / write to
- `len`: size of the allocated buffer (`read()`) and length of the content (`write()`)

Example

```
read(sockfd, buffer, sizeof(buffer));  
write(sockfd, "hello world!\n", 13);
```


Data Transfer: Taking Turn



Data Transfer: Taking Turn

- Client

```
while (condition) {  
    scanf() from STDIN;  
    send() to server;  
    recv() from server;  
    printf() to STDOUT;  
}
```

- Server

```
while (condition) {  
    recv() from client;  
    printf() to STDOUT;  
    scanf() from STDIN;  
    send() to client;  
}
```



Practical Work 5: Data Transfer, Taking Turn

- Copy your client and server code from 4th practical work to
 - « 05.practical.work.server.turn.c »
 - « 05.practical.work.client.turn.c »
 - Improve the client and server to build a chat system
 - input from STDIN
 - send to other side
 - output received data to STDOUT
 - client and server take turn
- Test the system between your laptop and VPS
- Push your C programs to corresponding forked Github repository

Message Framing



Problem

- Problem: Message boundaries are not preserved
 - One «send» may result in many «receive»s
 - Many «send»s may result in one «receive»
 - Usually not considered by developers
 - Especially when testing on 127.0.0.1
- For example:
 - sends: “hell”, “o wo”, “rld!\n”
 - receive: “hello world!\n”



Problem

- Why?
 - Applications work with messages
 - TCP sockets work with streams
 - Network is unstable and unpredictable
 - ACK is not delivered



Solution

- Use a buffer
- Make your own protocol



«Protocolization»



**KEEP
CALM**

&

**FOLLOW
THE DII EC**

Message Framing

- Possible approaches
 - Delimiter character (`\n`, `\0`)
 - Structured data
 - Define your own with headers...
 - Reuse predefined representations & validations

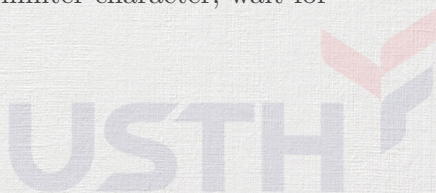


Delimiter character

- Define a rare (or unused) character to indicate a termination of message
 - Usually `\0` or `\n`



- If `recv()` does not have the delimiter character, wait for more data



Delimiter character

Pros

- Easy to implement
- Flexible

Cons

- Easy to mess up
- Escaping / unescaping delimiters
- How much to allocate buffer?



Delimiter character

```
while (condition) {  
    // read messages until we have delimiter  
    while (no_delimit_char) {  
        read();  
        append_to_buffer();  
    }  
    // process the message  
    // and write back  
    write();  
}
```

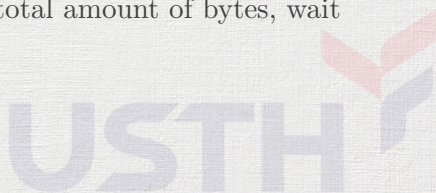


Structured Data

- Use several bytes for length
- The rest for message
- [Optional] Validation: hash, checksum, etc..



- If `recv()` does not return the total amount of bytes, wait for more data



Structured Data

Pros

- Flexible, can handle large or small message frames

Cons

- Manage allocating buffers based on the length-prefix
- Waste memory



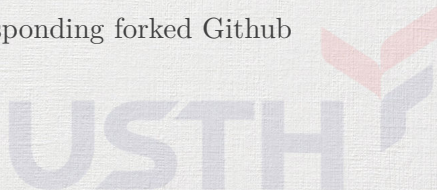
Structured Data

```
while (condition) {  
    // read the message until we have termination  
    while (not_enough) {  
        read();  
        append_to_buffer();  
    }  
    // process the message  
    // and write  
    write();  
}
```



Practical Work 6: Delimiter character

- Copy your client and server code from 5th practical work to
 - « 06.practical.work.server.turn.delim.c »
 - « 06.practical.work.client.turn.delim.c »
 - Improve the client and server to integrate delimiter character
 - Can use existing null termination scheme ☺
- Test the system between your laptop and VPS
- Push your C program to corresponding forked Github repository



Disconnect



What?

- Terminate the connection willingly
- From either side
- Example
 - Client quits
 - Server disconnects because of unauthorization



How?

- Shutdown the socket
- Close the socket



How?

- Shutdown: block communication in one or both directions

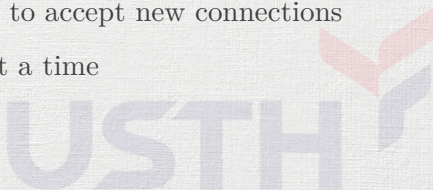
```
int shutdown(int sockfd, int how);
```
- Server: `sockfd` = client's socket file descriptor returned by `accept()`
- Client: `sockfd` = socket file descriptor returned by `socket()`
- how:
 - SHUT_RD: block further receives
 - SHUT_WR: block further sends
 - SHUT_RDWR: block all

Example:

How?

```
int close(int sockfd);
```

- Server: `sockfd` = client's socket file descriptor returned by `accept()`
- Client: `sockfd` = socket file descriptor returned by `socket()`
- Server now can repeat the loop to accept new connections
- Currently : single connection at a time



How?

Question:

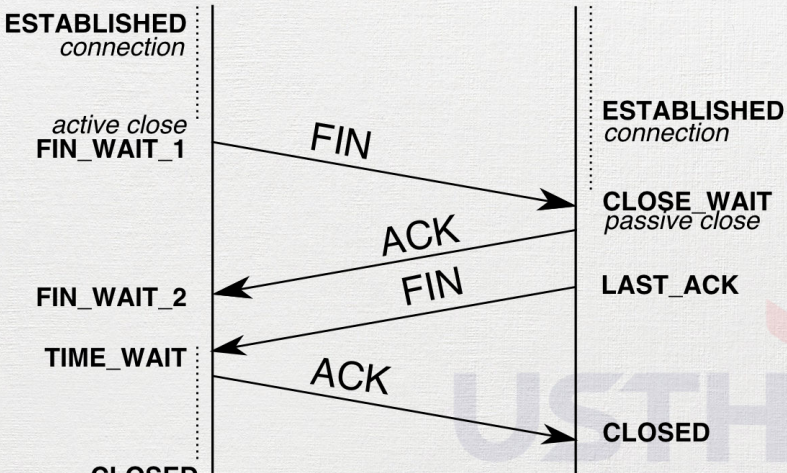
- What happens to the port after server quits while client is still connected?
 - Use CTRL-C from the server while both are connected
- Check with

```
netstat -nat | grep 8784
```



How?

Initiator Receiver



Practical Work 7: Disconnect

- Copy your client and server code from 6th practical work to
 - « 07.practical.work.server.turn.delim.close.c »
 - « 07.practical.work.client.turn.delim.close.c »
 - Improve server and client
 - Client: a command “/quit” to disconnect and exit
 - Server: a command “/dc” to disconnect current client, continue `accept()` loop for another client
 - Both: properly close sockets after disconnecting
- Test the system between your laptop and VPS
- Push your C program to corresponding forked Github repository