Linux Socket Tutorial

1. Development Environment

- Ubuntu 22.04 (Linux kernel 6.2) with GCC 11.4

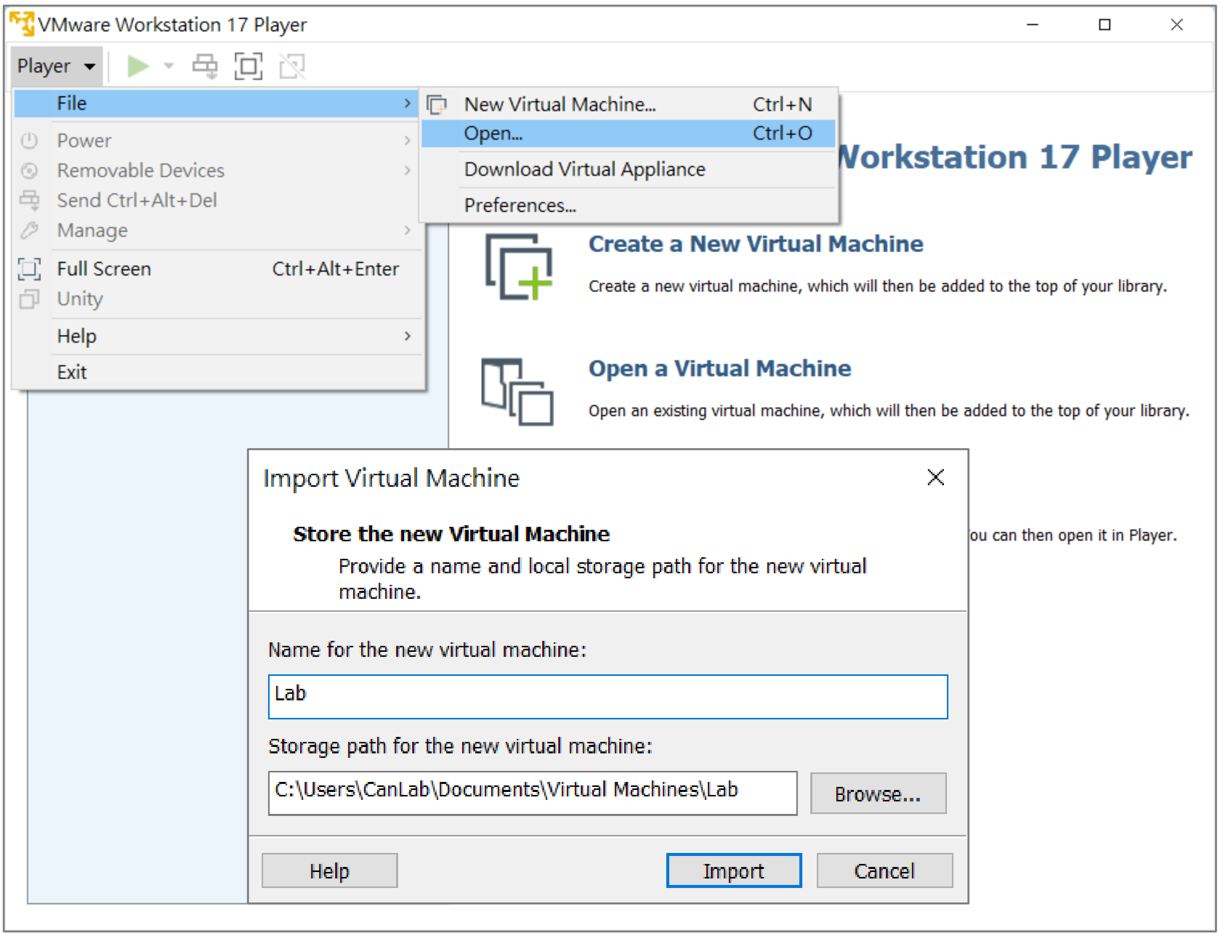
Because your own development environment might not be the same as ours, we may not be able to successfully compile or run your program due to differences in the system environment or compiler versions. Therefore, we highly recommend using the virtual machine we provide for you, or at least testing your code/program on the virtual machine before submitting your project.

(a) For Windows users, download and install VMware Workstation Player for Windows
For Linux users, download and install VMware Workstation Player for Linux (video)
For Intel-based Mac users, download and install VMware Fusion (free, registration required)
For M-series Mac users, we recommend setting up an ARM-based virtual machine using either QEMU-based UTM (video), VMware Fusion (video) or Parallels Desktop on your own.

(b) Download the OVF file (x64-based) prepared by TA, then unzip it to a folder.

(c) Open VMware Workstation Player, click on [File] → [Open...], and select the OVF file from the folder you just extracted. Follow the prompts to import the virtual machine.

Password: canlab



Create a new virtual machine, which will then be added to the top of your library.

Open an existing virtual machine, which will then be added to the top of your library.

ou can then open it in Player.

Import Virtual Machine

Store the new Virtual Machine

Provide a name and local storage path for the new virtual machine.

Name for the new virtual machine: Lab

Storage path for the new virtual machine: C:\Users\CanLab\Documents\Virtual Machines\Lab

2. Shell

For students who are not familiar with command-line operations, we recommend that you watch the video "Lecture 1: Course Overview + The Shell" (with English subtitles available) from the course "The Missing Semester of Your CS Education" recorded by MIT. Follow along with the course step by step to practice the operations.

If you're interested in further learning how to write scripts, use Vim (editor), and Git (version control), this course also provides excellent introductory learning resources.

3. Editor

You can use any tool you are familiar with to do this lab. A smart editor such as Visual Studio Code (VS Code) is recommended.

4. Compile & Run

Tutorial video: How to Compile and Run C program Using GCC on Ubuntu (Linux)

Compile and Run C Program

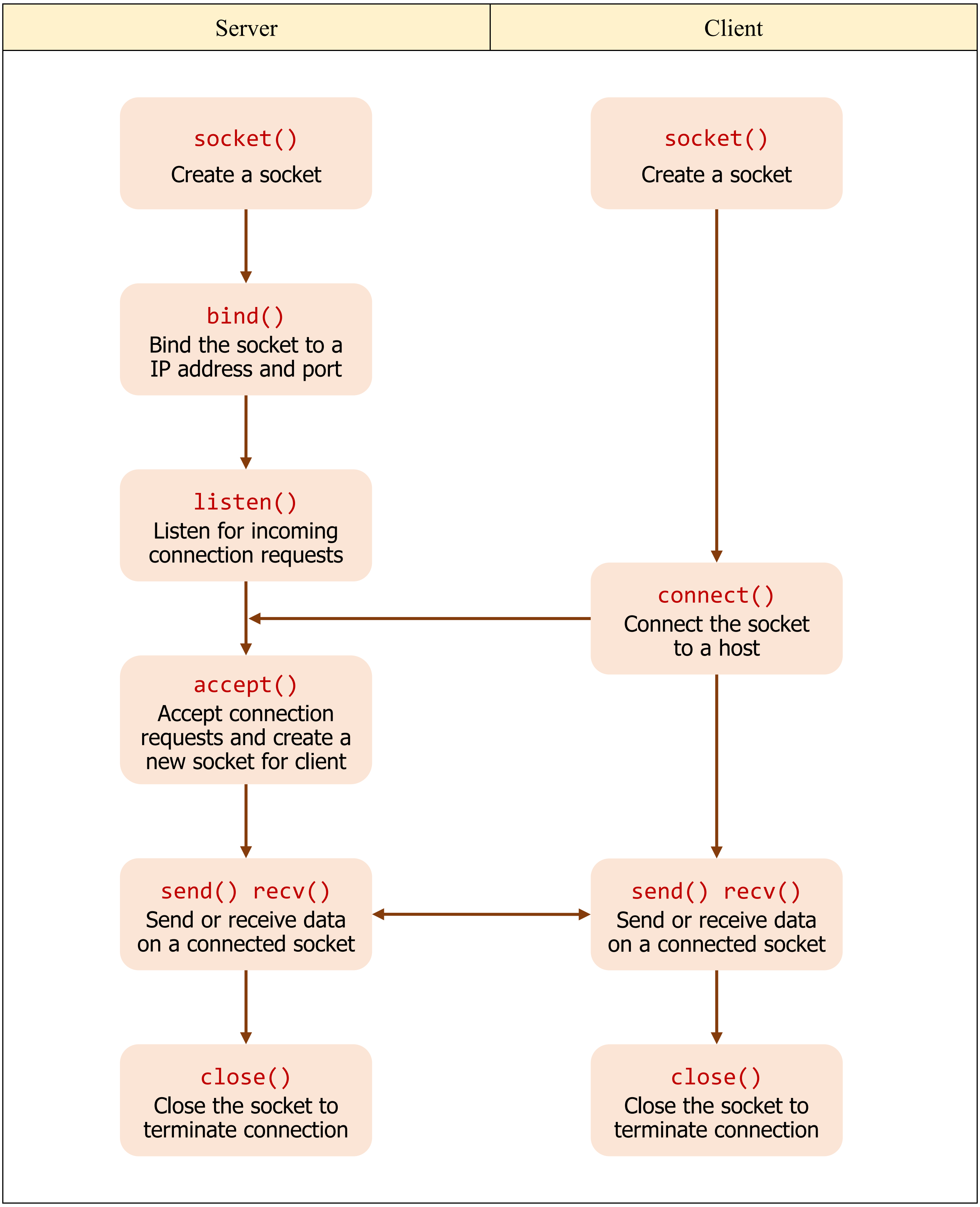
```
$ gcc lab.c -o lab
$ ./lab
```

Note:

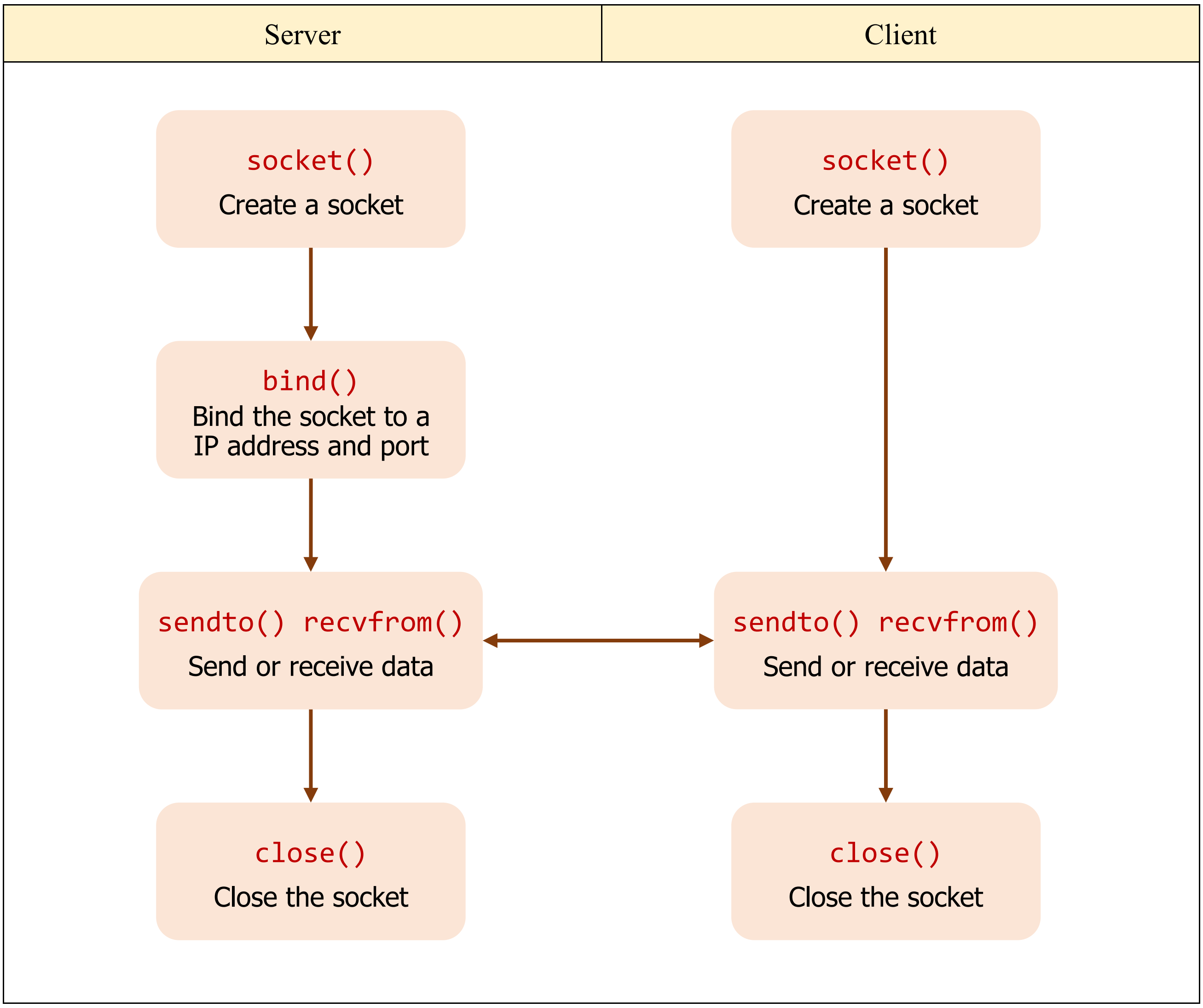
- The dollar sign `$` represents the shell prompt; you only need to enter the command after the dollar sign.
- A "dot slash" is a dot followed immediately by a forward slash (`./`). It is used in Linux and Unix to execute a program or script in the current directory.

5. Socket Programming

(a) TCP socket flow chart


Server
Client
socket()
Create a socket
socket()
Create a socket
bind()
Bind the socket to a IP address and port
listen()
Listen for incoming connection requests
connect()
Connect the socket to a host
accept()
Accept connection requests and create a new socket for client
send() recv()
Send or receive data on a connected socket
send() recv()
Send or receive data on a connected socket
close()
Close the socket to terminate connection
close()
Close the socket to terminate connection

(b) UDP socket flow chart



6. Linux Programmer’s Manual

In Unix and Linux, you can typically use the “man” command to access user manuals for specific programs or system calls. For example, “man socket” would provide you with the manual for the socket system call.

Linux Programmer’s Manual

```
$ man socket
```

```
SOCKET(2)                   Linux Programmer's Manual                  SOCKET(2)

NAME
      socket - create an endpoint for communication

SYNOPSIS
      #include <sys/types.h>          /* See NOTES */
      #include <sys/socket.h>

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

DESCRIPTION
      socket()  creates  an  endpoint  for communication and returns a file descriptor
      that refers to that endpoint.  The file descriptor returned by a successful call
      will be the lowest-numbered file descriptor not currently open for the process.

      The domain argument specifies a communication domain; this selects the protocol
      family which will be used for communication.
      These families are defined in <sys/socket.h>.  The formats currently understood
      by the Linux kernel include:
      ......
Manual page socket(2) line 1 (press h for help or q to quit)
```

The numeric value (2) within parentheses refers to the “System Calls” section. This section contains documentation related to system calls and functions provided by the operating system kernel.

You need to learn how to query these manuals to understand what parameters should be passed to a system call. However, for beginners, navigating these manuals through the shell might be challenging. Therefore, you can use web-based documentation instead. Here are some examples:

- `socket()`
- `bind()`
- `listen()`
- `connect()`
- `accept()`
- `send() / sendto()`
- `recv() / recvfrom()`
- `close()`

7. Examples

(a) TCP socket example

server.c

```c
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define BUFFER_SIZE 1024
#define PORT 8080

int main() {
    int sockfd, client_sockfd;
    struct sockaddr_in address;
    socklen_t addrlen = sizeof(address);

    char *message = "Hello from server";
    unsigned char buffer[BUFFER_SIZE] = {'\0'};

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket()");
        exit(EXIT_FAILURE);
    }

    int on = 1; // Forcefully attaching socket to the port 8080
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) == -1) {
        perror("setsockopt()");
        exit(EXIT_FAILURE);
    }

    if (bind(sockfd, (struct sockaddr *)&address, sizeof(address)) == -1) {
        perror("bind()");
        exit(EXIT_FAILURE);
    }

    if (listen(sockfd, 1) == -1) {
        perror("listen()");
        exit(EXIT_FAILURE);
    }

    printf("Waiting for a client...\n");

    if ((client_sockfd = accept(sockfd, (struct sockaddr *)&address, &addrlen)) == -1) {
        perror("accept()");
        exit(EXIT_FAILURE);
    }

    // Consider using recv() in a loop for large data to ensure complete message reception
    recv(client_sockfd, buffer, BUFFER_SIZE, 0);
    printf("%s\n", buffer);

    send(client_sockfd, message, strlen(message), 0);
    printf("Hello message sent\n");

    close(client_sockfd);
    close(sockfd);

    return 0;
}
```

client.c

```
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define BUFFER_SIZE 1024
#define PORT 8080

int main() {
    int sockfd;
    struct sockaddr_in server_addr;
    socklen_t addrlen = sizeof(server_addr);

    char *message = "Hello from client";
    unsigned char buffer[BUFFER_SIZE] = {'\0'};

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    server_addr.sin_port = htons(PORT);

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket()");
        exit(EXIT_FAILURE);
    }

    if (connect(sockfd, (struct sockaddr *)&server_addr, addrlen) == -1) {
        perror("connect()");
        fprintf(stderr, "Please start the server first\n");
        exit(EXIT_FAILURE);
    }

    printf("Connected to server\n");

    send(sockfd, message, strlen(message), 0);
    printf("Hello message sent\n");

    // Consider using recv() in a loop for large data to ensure complete message reception
    recv(sockfd, buffer, BUFFER_SIZE, 0);
    printf("%s\n", buffer);

    close(sockfd);

    return 0;
}
```

Terminal 1

```
$ gcc server.c -o server
$ ./server

Waiting for a client...
Hello from client
Hello message sent
```

Terminal 2

```
$ gcc client.c -o client
$ ./client

Connected to server
Hello message sent
Hello from server
```

(b) UDP socket example

server.c

```c
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define BUFFER_SIZE 1024
#define PORT 8080

int main() {
    int sockfd;
    struct sockaddr_in address, client_addr;
    socklen_t client_addrlen = sizeof(client_addr);

    char *message = "Hello from server";
    unsigned char buffer[BUFFER_SIZE] = {'\0'};

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    memset(&client_addr, 0, sizeof(client_addr));

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket()");
        exit(EXIT_FAILURE);
    }

    // Forcefully attaching socket to the port 8080
    int on = 1;
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) == -1) {
        perror("setsockopt()");
        exit(EXIT_FAILURE);
    }

    if (bind(sockfd, (struct sockaddr *)&address, sizeof(address)) == -1) {
        perror("bind()");
        exit(EXIT_FAILURE);
    }

    if (recvfrom(sockfd, buffer, BUFFER_SIZE, 0, (struct sockaddr *)&client_addr,
        &client_addrlen) == -1) {
        perror("recvfrom()");
        exit(EXIT_FAILURE);
    }

    printf("%s\n", buffer);

    if (sendto(sockfd, message, strlen(message), 0, (struct sockaddr *)&client_addr,
        client_addrlen) == -1) {
        perror("sendto()");
        exit(EXIT_FAILURE);
    }

    printf("Hello message sent\n");

    close(sockfd);

    return 0;
}
```

client.c

```c
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define BUFFER_SIZE 1024
#define PORT 8080

int main() {
    int sockfd;
    struct sockaddr_in server_addr;
    socklen_t addrlen = sizeof(server_addr);

    char *message = "Hello from client";
    unsigned char buffer[BUFFER_SIZE] = {'\0'};

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    server_addr.sin_port = htons(PORT);

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket()");
        exit(EXIT_FAILURE);
    }

    if (sendto(sockfd, message, strlen(message), 0, (struct sockaddr *)&server_addr,
        addrlen) == -1) {
        perror("sendto()");
        exit(EXIT_FAILURE);
    }

    printf("Hello message sent\n");

    if (recvfrom(sockfd, buffer, BUFFER_SIZE, 0, (struct sockaddr *)&server_addr, &addrlen)
        == -1) {
        perror("recvfrom()");
        exit(EXIT_FAILURE);
    }

    printf("%s\n", buffer);

    close(sockfd);

    return 0;
}
```

Terminal 1	Terminal 2

`$ gcc server.c -o server`
`$ ./server`
 	
`Hello from client`
`Hello message sent`
	`$ gcc client.c -o client`
	`$ ./client`
	
	`Hello message sent`
	`Hello from server`