ĐẠI HỌC BÁCH KHOA HÀ NỘI

TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

****

BÁO CÁO

Bài tập thực hành :

Chương 6: UDP Sockets

Học phần: Thực hành Lập Trình Mạng

Giảng viên hướng dẫn : Trần Hải Anh

Mã lớp : 151907

Sinh viên thực hiện : Phạm Vân Anh

Mã số sinh viên : 20214988

Hà Nội, tháng 10 năm 2024

Link github: github

Phần 1: Thiết lập UDP Sockets với mã hóa

1.1 Chương trình server (Xử lý nhiều client, các hàm cơ bản, mã hóa)

- Sử dụng **socket()** để tạo socket UDP cho server:

```
// Tạo socket UDP
if ((server_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("Socket creation failed");
    exit(EXIT_FAILURE);
}
```

  - **socket()**: hàm hệ thống dùng để tạo một socket.
  - **SOCK_DGRAM** chỉ định rằng socket sử dụng giao thức UDP, không kết nối.

    ➔ Hàm trả về một mô tả (descriptor) của socket nếu thành công, ngược lại sẽ trả về giá trị âm (lỗi).

- Sử dụng **bind()** để liên kết socket với một địa chỉ IP và cổng:

```
    // Liên kết socket với địa chỉ IP và cổng
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(server_fd, (const struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        perror("Bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }
```

  - **bind()** dùng để gắn (liên kết) socket với một địa chỉ IP cụ thể và một cổng trên server.
  - **INADDR_ANY** cho phép server lắng nghe trên mọi địa chỉ IP của máy chủ.
  - **htons(PORT)** đảm bảo rằng cổng được chuyển đổi đúng từ byte order của host thành byte order của mạng.
  - Nếu **bind()** thất bại (trả về giá trị âm), chương trình sẽ báo lỗi và thoát

- Sử dụng **recvfrom()** và **sendto()** để giao tiếp với các client:

```
        // Nhận dữ liệu từ client
        int recv_len = recvfrom(server_fd, buffer, BUFFER_SIZE, 0,
(struct sockaddr *)&client_addr, &addr_len);
        if (recv_len == -1) {
            perror("Failed to receive data");
            continue;
        }
        buffer[recv_len] = '\0'; // Kết thúc chuỗi

        printf("Received message from client: %s\n", buffer);
```

  - **recvfrom()** nhận dữ liệu từ client, sử dụng giao thức UDP.
  - **buffer** chứa dữ liệu nhận được và n trả về số byte đã nhận.
  - **cliaddr** là cấu trúc chứa thông tin địa chỉ của client gửi đến.
  - Sau khi nhận dữ liệu, cần đảm bảo rằng chuỗi được kết thúc bằng ký tự null ('\0') để tránh lỗi xử lý chuỗi.

- Sử dụng **XOR** để mã hóa và giải mã các thông điệp:

```
        // Mã hóa thông điệp
        xor_cipher(buffer, XOR_KEY);
        printf("Sending encrypted message to client: %s\n", buffer);
```

  - Hàm **xor_cipher()** sử dụng XOR để mã hóa hoặc giải mã một chuỗi.
  - Thuật toán này có thể được sử dụng để mã hóa và giải mã bằng cách áp dụng nó hai lần với cùng một khóa (mã hóa -> gửi đi -> giải mã).

- Gửi dữ liệu tới client:

```
// Gửi dữ liệu mã hóa đến client
        if (sendto(server_fd, buffer, strlen(buffer), 0, (struct
sockaddr *)&client_addr, addr_len) == -1) {
            perror("Failed to send data");
        }
```

  - **sendto()** gửi dữ liệu tới client qua giao thức UDP.
  - Địa chỉ của client (lưu trong cliaddr) được sử dụng để gửi dữ liệu.

- Sử dụng **select()** để xử lý timeout cho các thông điệp:

```
        FD_ZERO(&read_fds);
        FD_SET(client_fd, &read_fds);
        timeout.tv_sec = TIMEOUT_SEC;
        timeout.tv_usec = 0;

        // Chờ dữ liệu từ server với timeout
        int activity = select(client_fd + 1, &read_fds, NULL, NULL,
&timeout);
        if (activity == -1) {
            perror("Select error");
            continue;  // Tiếp tục vòng lặp nếu gặp lỗi
        } else if (activity == 0) {
            printf("Timeout, no data received from server.\n");
            continue;  // Tiếp tục vòng lặp nếu không nhận được phản
hồi
        }
```

- **select()** cho phép server chờ cho đến khi một sự kiện xảy ra (dữ liệu từ client), hoặc hết thời gian chờ (timeout).
- **FD_SET(sockfd, &readfds)** thêm **sockfd** vào tập hợp các file descriptor để theo dõi hoạt động.
- **tv** chứa giá trị timeout, ở đây là 5 giây.
- **select()** trả về 0 nếu hết thời gian chờ mà không có hoạt động, trả về số lượng file descriptor sẵn sàng nếu có hoạt động, và trả về -1 nếu có lỗi.

1.2 Chương trình Client (Xác minh địa chỉ server bằng memcmp(), mã hóa)

- Giải mã thông điệp bằng thuật toán XOR

```
        // Giải mã thông điệp nhận được
        xor_cipher(buffer, XOR_KEY);
        printf("Decrypted message from server: %s\n", buffer);
```

- Sau khi nhận được thông điệp mã hóa từ server, sử dụng hàm **xor_cipher()** để giải mã thông điệp.
- Hàm **xor_cipher()** thực hiện phép XOR trên từng ký tự của chuỗi buffer với khóa key, giải mã thông điệp đã nhận.

- Sau khi giải mã, thông điệp sẽ được in ra để xác minh.

- Xác minh rằng thông điệp đến từ server mong đợi

```
        // So sánh địa chỉ IP của server với địa chỉ mong đợi
        if (memcmp(&response_addr.sin_addr, &expected_server_addr,
sizeof(expected_server_addr)) != 0) {
            printf("Error: Received message from unknown IP
address: %s\n", inet_ntoa(response_addr.sin_addr));
            continue;  // Tiếp tục vòng lặp nếu IP không khớp
        }

        printf("Received message from verified server IP: %s\n",
inet_ntoa(response_addr.sin_addr));
```

- Client cần xác minh rằng thông điệp đến từ đúng server (tức là từ địa chỉ IP và cổng đã biết).
- **memcmp()** so sánh **servaddr** (địa chỉ của server mà client mong đợi) và **recv_servaddr** (địa chỉ của server gửi phản hồi).
- Nếu kết quả của **memcmp()** bằng 0, nghĩa là địa chỉ IP và cổng của server nhận được phản hồi khớp với địa chỉ mà client mong đợi. Do đó, thông điệp được xác minh là từ đúng server.

- Xử lý timeout cho thông điệp bằng cách sử dụng **select()**

```
        // Đặt timeout để nhận phản hồi từ server
        FD_ZERO(&read_fds);
        FD_SET(client_fd, &read_fds);
        timeout.tv_sec = TIMEOUT_SEC;
        timeout.tv_usec = 0;

        // Chờ dữ liệu từ server với timeout
        int activity = select(client_fd + 1, &read_fds, NULL, NULL,
&timeout);
        if (activity == -1) {
            perror("Select error");
            continue;  // Tiếp tục vòng lặp nếu gặp lỗi
        } else if (activity == 0) {
            printf("Timeout, no data received from server.\n");
            continue;  // Tiếp tục vòng lặp nếu không nhận được phản
hồi
        }
```

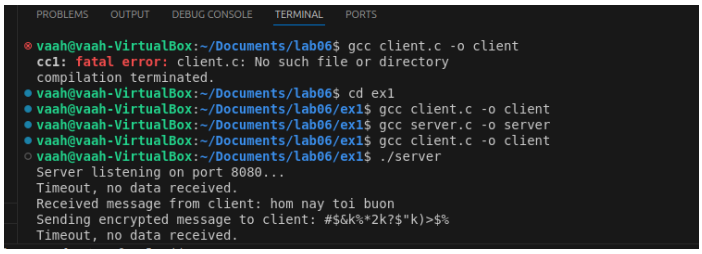
- **select()** được sử dụng để kiểm tra liệu có bất kỳ dữ liệu nào có thể đọc được từ socket (tức là từ server) trong một khoảng thời gian giới hạn (timeout).
- **FD_ZERO()** khởi tạo tập hợp file descriptors trống.
- **FD_SET()** thêm socket sockfd của client vào tập hợp các file descriptors cần theo dõi.
- **tv** chứa thông tin về thời gian chờ (timeout), được đặt thành 5 giây (thông qua tv.tv_sec).
- **select()** sẽ chờ tối đa 5 giây để xem có dữ liệu từ server được gửi đến socket hay không.

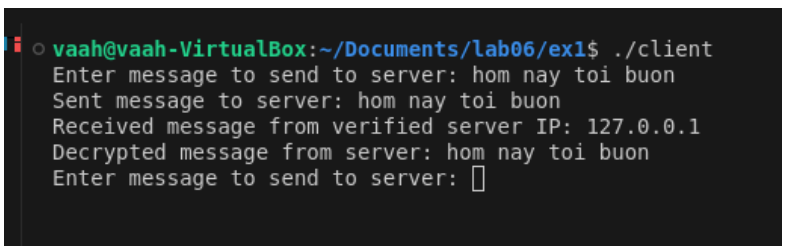
- Sử dụng **recvfrom()** dùng để nhận dữ liệu từ socket

```
        // Nhận phản hồi từ server
        int recv_len = recvfrom(client_fd, buffer, BUFFER_SIZE, 0,
(struct sockaddr *)&response_addr, &addr_len);
        if (recv_len == -1) {
            perror("Failed to receive data");
            continue;  // Tiếp tục vòng lặp nếu gặp lỗi
        }
```

- Nhận dữ liệu từ server qua socket bằng hàm **recvfrom()**
- Nếu việc nhận dữ liệu thành công, dữ liệu sẽ được lưu vào bộ đệm **(buffer)**
- Nếu gặp lỗi (hàm trả về -1), chương trình sẽ in ra thông báo lỗi và tiếp tục vòng lặp, tránh làm gián đoạn chương trình.

1.3 Kết quả





```
vaah@vaah-VirtualBox:~/Documents/lab06/ex1$ ./client
Enter message to send to server: helo
Sent message to server: helo
Received message from verified server IP: 127.0.0.1
Decrypted message from server: helo
Enter message to send to server: hnay tui buon
Sent message to server: hnay tui buon
Received message from verified server IP: 127.0.0.1
Decrypted message from server: hnay tui buon
Enter message to send to server:
```

Phần 2: Cải tiến

- Thêm tính năng xử lý nhiều client trong server: Sử dụng **select()** để theo dõi các socket cho nhiều client và nhận thông điệp từ nhiều client khác nhau.
- Server sẽ tiếp tục chạy và xử lý nhiều client cùng lúc mà không bị gián đoạn khi chờ thông điệp từ một client cụ thể.

```
    while (1) {
        char response[MAXLINE];

        // Lấy tin nhắn
        printf("Enter message to send to server: ");
        fgets(response, MAXLINE, stdin);
        response[strcspn(response, "\n")] = '\0';

        sendto(sockfd, response, strlen(response), 0, NULL, 0);
        printf("Message sent to server.\n");

        FD_ZERO(&readfds);
        FD_SET(sockfd, &readfds);
        tv.tv_sec = TIMEOUT;
        tv.tv_usec = 0;

        activity = select(sockfd + 1, &readfds, NULL, NULL, &tv);
        if (activity == -1) {
            perror("select error");
            close(sockfd);
            exit(EXIT_FAILURE);
        } else if (activity == 0) {
            printf("Timeout: No response from server.\n");
            continue;
        }

        n = recvfrom(sockfd, buffer, MAXLINE, 0, (struct sockaddr
*)&recv_servaddr, &len);
        buffer[n] = '\0';
```

```
        //
        improved_xor_cipher(buffer, key, key_len, n);
        printf("Decrypted message from server: %s\n", buffer);
    }
```

- Cải thiện XOR: Sử dụng hàm **improved_xor_cipher** để mã hóa thông điệp dựa trên khóa XOR. Chuỗi khóa được sử dụng theo dạng tuần hoàn (cyclic), giúp tăng độ phức tạp và tính bảo mật của thông điệp.

```
// Update XOR
void improved_xor_cipher(char *data, const char *key, int key_len, int
data_len) {
    for (int i = 0; i < data_len; i++) {
        data[i] ^= key[i % key_len];  // Use key in a cyclic manner
    }
}
```

- Triển khai xử lý lỗi tốt hơn cho timeout của thông điệp và phản hồi từ server
  - Server

```
        // Đợi socket
        int activity = select(sockfd + 1, &readfds, NULL, NULL, &tv);

        if (activity == -1) {
            perror("select error");
            close(sockfd);
            exit(EXIT_FAILURE);
        } else if (activity == 0) {
            printf("Timeout: No activity from clients in the last %d
seconds.\n", TIMEOUT);
            continue;
        }
```

  - Client

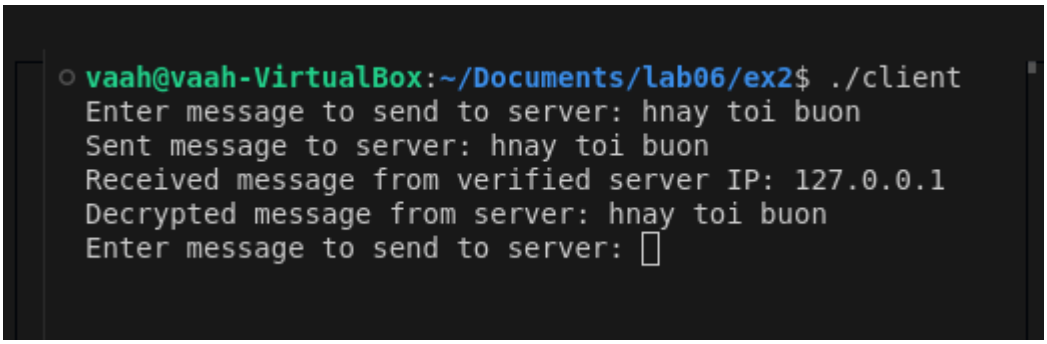
```
    // Đợi server phản hồi
    int activity = select(sockfd + 1, &readfds, NULL, NULL, &tv);
    if (activity == -1) {
        perror("select error");
        close(sockfd);
```

```
        exit(EXIT_FAILURE);
    } else if (activity == 0) {
        printf("Timeout: No response from server.\n");
        close(sockfd);
        exit(EXIT_FAILURE);
    }
```

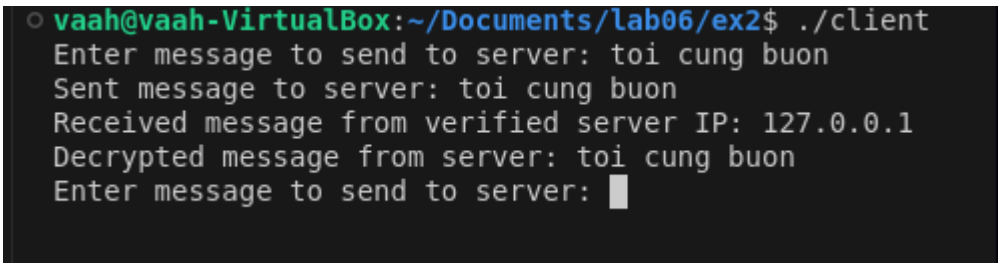
- Cả client và server đều sử dụng select() để thiết lập khoảng thời gian timeout (5 giây). Nếu không có hoạt động trong khoảng thời gian này, server sẽ tiếp tục chờ các client khác mà không bị treo, còn client sẽ tiếp tục gửi các thông điệp tiếp theo mà không chờ mãi cho phản hồi.

Kết quả:

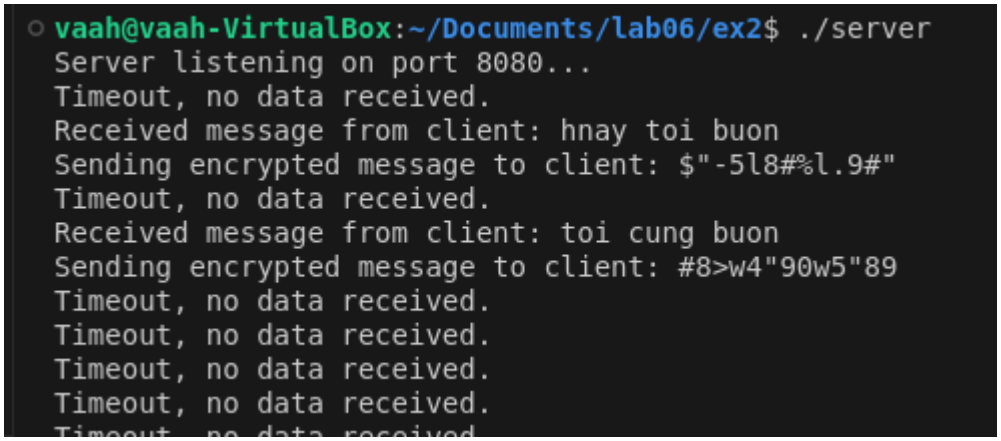
- Client:



- Client:



- Server:



Thảo luận về cách triển khai mã hóa, xác minh server bằng memcmp(), và sử dụng connect().

- Cách triển khai mã hóa: Vì UDP không có cơ chế bảo mật tích hợp, bạn cần thực hiện mã hóa dữ liệu trước khi gửi qua socket để đảm bảo bảo mật. Mã hóa và giải mã với XOR là đối xứng, có nghĩa là nếu một thông điệp được mã hóa bằng cách sử dụng XOR với một khóa, có thể giải mã nó bằng cách thực hiện lại XOR với cùng một khóa.
- Xác minh server bằng **memcmp()**: Vì UDP không có cơ chế kết nối như TCP, client không thể đảm bảo rằng gói tin đến từ server mong muốn. Để xác minh server, bạn có thể sử dụng một chữ ký hoặc một token bí mật để đảm bảo tính xác thực của gói tin.
- Sử dụng **connect()**: Trong UDP, hàm connect() không thực sự thiết lập kết nối, nhưng có thể được dùng để liên kết một địa chỉ đích cố định với socket. Sau khi gọi connect(), các gói tin từ client sẽ chỉ gửi đến và nhận từ địa chỉ đã kết nối này.