**ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG**

**\*\*\*\*\*\*\*\***

A picture containing icon

Description automatically generated

**BÁO CÁO**

**Bài tập thực hành :**

**Chương 5: I/O Multiplexing**

**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**

**Mục lục**

[Phần 1: Lập Trình Socket TCP - Thiết Lập Cơ Bản 3](#_Toc179331097)

[1.1 Triển Khai Server TCP 3](#_Toc179331098)

[1.2 Server phát tin nhắn từ 1 client và phát tới các client khác 3](#_Toc179331099)

[1.3 Kết quả: 3](#_Toc179331100)

[Phần 2: Logic Server và Phát Tin Nhắn 4](#_Toc179331101)

[2.1: Phát Tin Nhắn Đến Tất Cả Các Client 4](#_Toc179331102)

[2.2: Xử Lý Client Ngắt Kết Nối 5](#_Toc179331103)

[Phần 3: I/O Multiplexing để Xử Lý Nhiều Client 7](#_Toc179331104)

[3.1: Sử Dụng select() để Xử Lý Nhiều Client 7](#_Toc179331105)

[3.2: Sử Dụng pselect() để Thực Hiện I/O Multiplexing An Toàn Với Tín Hiệu 8](#_Toc179331106)

[3.3: Sử Dụng poll() để Xử Lý Nhiều Client 10](#_Toc179331107)

[3.3: Tổng kết 12](#_Toc179331108)

[Phần 4: Tính Năng Phòng Chat và Trải Nghiệm Người Dùng 16](#_Toc179331109)

[4.3: Cách mà chức năng phòng chat (phát tin nhắn, tên người dùng, v.v.) được triển khai 18](#_Toc179331110)

[4.4 Những khó khăn gặp phải trong quá trình triển khai và cách khắc phục: 19](#_Toc179331111)

Link github:[**github**](https://github.com/vah102/IT4062.Network_Programming.20214988.PhamVanAnh)

# **Phần 1:** Lập Trình Socket TCP - Thiết Lập Cơ Bản

* 1. **Triển Khai Server TCP**

A screenshot of a computer

Description automatically generated

* 1. **Server phát tin nhắn từ 1 client và phát tới các client khác**

A black screen with a black rectangular object

Description automatically generated

* 1. **Kết quả:**
* Server có thể xử lý nhiều client, phát tin nhắn từ một client tới client khác trong phòng chat.
* Mỗi client có thể gửi tin nhắn và nhìn thấy tin nhắn của các client khác trong thời gian thực.

# **Phần 2:** Logic Server và Phát Tin Nhắn

## **2.1: Phát Tin Nhắn Đến Tất Cả Các Client**

* Khi gửi tin nhắn từ 1 client, các client khác sẽ nhận được tin nhắn.

A screenshot of a computer

Description automatically generated

A computer screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

## **2.2: Xử Lý Client Ngắt Kết Nối**

* Client ngắt kết nối

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

* Server hủy kết nối

if (FD\_ISSET(sd, &readfds)) {

// Check if it was for closing, and read the incoming message

if ((valread = read(sd, buffer, BUFFER\_SIZE)) == 0) {

// Someone disconnected, get details

getpeername(sd, (struct sockaddr\*)&address, (socklen\_t\*)&addrlen);

printf("Client disconnected, ip %s, port %d\n", inet\_ntoa(address.sin\_addr), ntohs(address.sin\_port));

// Close the socket and mark as 0

close(sd);

client\_sockets[i] = 0;

// Inform other clients that a user has disconnected

char disconnect\_msg[BUFFER\_SIZE];

snprintf(disconnect\_msg, sizeof(disconnect\_msg), "Client at %s has disconnected.\n", inet\_ntoa(address.sin\_addr));

for (int j = 0; j < max\_clients; j++) {

if (client\_sockets[j] > 0) {

send(client\_sockets[j], disconnect\_msg, strlen(disconnect\_msg), 0);

}

}

Kết quả:

A screenshot of a computer screen

Description automatically generatedA screenshot of a computer screen

Description automatically generated

A screen shot of a computer

Description automatically generated

# **Phần 3:** I/O Multiplexing để Xử Lý Nhiều Client

## **3.1: Sử Dụng select() để Xử Lý Nhiều Client**

* Server

// Clear the socket set

FD\_ZERO(&readfds);

// Add server socket to set

FD\_SET(server\_fd, &readfds);

int max\_sd = server\_fd;

// Add client sockets to set

for (i = 0; i < max\_clients; i++) {

sd = client\_sockets[i];

if (sd > 0) FD\_SET(sd, &readfds);

if (sd > max\_sd) max\_sd = sd;

}

// Wait for activity on one of the sockets

activity = select(max\_sd + 1, &readfds, NULL, NULL, NULL);

// Incoming connection

if (FD\_ISSET(server\_fd, &readfds)) {

if ((new\_socket = accept(server\_fd, (struct sockaddr\*)&address, (socklen\_t\*)&addrlen)) < 0) {

perror("Accept failed");

exit(EXIT\_FAILURE);

}

printf("New connection, socket fd is %d, ip is: %s, port: %d\n", new\_socket, inet\_ntoa(address.sin\_addr), ntohs(address.sin\_port));

// Add new socket to array

for (i = 0; i < max\_clients; i++) {

if (client\_sockets[i] == 0) {

client\_sockets[i] = new\_socket;

printf("Adding to list of sockets as %d\n", i);

break;

}

}

}

* Client

FD\_ZERO(&read\_fds);

FD\_SET(STDIN\_FILENO, &read\_fds); // Monitor user input

FD\_SET(sockfd, &read\_fds); // Monitor server messages

// Wait for activity

int activity = select(sockfd + 1, &read\_fds, NULL, NULL, NULL);

// Check for input from the server

if (FD\_ISSET(sockfd, &read\_fds)) {

int valread = read(sockfd, buffer, BUFFER\_SIZE);

if (valread == 0) {

printf("Server disconnected.\n");

close(sockfd);

break;

}

buffer[valread] = '\0';

printf("Message from server: %s\n", buffer);

}

// Check for user input

if (FD\_ISSET(STDIN\_FILENO, &read\_fds)) {

fgets(buffer, BUFFER\_SIZE, stdin);

send(sockfd, buffer, strlen(buffer), 0);

}

}

## **3.2: Sử Dụng pselect() để Thực Hiện I/O Multiplexing An Toàn Với Tín Hiệu**

* Server

// Signal mask setup

sigset\_t block\_mask, orig\_mask;

sigemptyset(&block\_mask);

sigaddset(&block\_mask, SIGINT);

sigprocmask(SIG\_BLOCK, &block\_mask, &orig\_mask); // Block SIGINT temporarily

while (!stop\_server) {

FD\_ZERO(&readfds);

FD\_SET(server\_fd, &readfds);

int max\_sd = server\_fd;

for (i = 0; i < max\_clients; i++) {

sd = client\_sockets[i];

if (sd > 0) FD\_SET(sd, &readfds);

if (sd > max\_sd) max\_sd = sd;

}

// Use pselect for signal-safe I/O multiplexing

activity = pselect(max\_sd + 1, &readfds, NULL, NULL, NULL, &orig\_mask);

// Handle signal stop\_server

if (activity < 0 && stop\_server) {

break;

}

* Client

// Setup signal mask

sigset\_t block\_mask, orig\_mask;

sigemptyset(&block\_mask);

sigaddset(&block\_mask, SIGINT);

sigprocmask(SIG\_BLOCK, &block\_mask, &orig\_mask); // Block SIGINT temporarily

while (!stop\_client) {

FD\_ZERO(&read\_fds);

FD\_SET(STDIN\_FILENO, &read\_fds); // Monitor user input

FD\_SET(sockfd, &read\_fds); // Monitor server messages

// Use pselect to handle signals safely

int activity = pselect(sockfd + 1, &read\_fds, NULL, NULL, NULL, &orig\_mask);

// If interrupted by signal, stop client

if (activity < 0 && stop\_client) {

break;

}

// Check for input from the server

if (FD\_ISSET(sockfd, &read\_fds)) {

int valread = read(sockfd, buffer, BUFFER\_SIZE);

if (valread == 0) {

printf("Server disconnected.\n");

close(sockfd);

break;

}

buffer[valread] = '\0';

printf("Message from server: %s\n", buffer);

}

// Check for user input

if (FD\_ISSET(STDIN\_FILENO, &read\_fds)) {

fgets(buffer, BUFFER\_SIZE, stdin);

send(sockfd, buffer, strlen(buffer), 0);

}

}

printf("Shutting down client...\n");

close(sockfd);

## **3.3: Sử Dụng poll() để Xử Lý Nhiều Client**

* Server
* Xử lý phía server:

// Function to broadcast a message to all clients except the sender

void broadcast\_message(int sender\_fd, struct pollfd \*fds, int num\_clients, char \*message) {

for (int i = 1; i < num\_clients; i++) { // Start at 1 to skip server socket

int client\_fd = fds[i].fd;

if (client\_fd != sender\_fd && client\_fd > 0) {

send(client\_fd, message, strlen(message), 0);

}

}

}

* Xử lý kết nối mới:

// Check for new connection on server socket

if (fds[0].revents & POLLIN) {

new\_socket = accept(server\_fd, (struct sockaddr\*)&address, (socklen\_t\*)&addrlen);

if (new\_socket < 0) {

perror("Accept failed");

exit(EXIT\_FAILURE);

}

printf("New connection, socket fd is %d, ip is: %s, port: %d\n", new\_socket, inet\_ntoa(address.sin\_addr), ntohs(address.sin\_port));

// Add new client socket to the pollfd array

for (i = 1; i < MAX\_CLIENTS; i++) {

if (fds[i].fd == -1) {

fds[i].fd = new\_socket;

fds[i].events = POLLIN; // Monitor this socket for read events

break;

}

}

if (i == MAX\_CLIENTS) {

printf("Too many clients connected. Connection rejected.\n");

close(new\_socket); // Too many clients connected, reject the new one

}

}

* Client
* Xử lý data từ client

/ Check for data from clients

for (i = 1; i < MAX\_CLIENTS; i++) {

if (fds[i].revents & POLLIN) {

// Read incoming message

valread = read(fds[i].fd, buffer, BUFFER\_SIZE);

if (valread == 0) {

// Client disconnected

getpeername(fds[i].fd, (struct sockaddr\*)&address, (socklen\_t\*)&addrlen);

printf("Client disconnected, ip %s, port %d\n", inet\_ntoa(address.sin\_addr), ntohs(address.sin\_port));

close(fds[i].fd);

fds[i].fd = -1; // Mark the slot as available

} else {

// Broadcast message to all other clients

buffer[valread] = '\0';

broadcast\_message(fds[i].fd, fds, MAX\_CLIENTS, buffer);

}

}

}

* Xử lý phái client:

while (1) {

int poll\_count = poll(fds, MAX\_CLIENTS, -1); // Wait indefinitely until an event occurs

if (poll\_count < 0) {

perror("Poll failed");

exit(EXIT\_FAILURE);

}

// Check for new connection on server socket

if (fds[0].revents & POLLIN) {

new\_socket = accept(server\_fd, (struct sockaddr\*)&address, (socklen\_t\*)&addrlen);

if (new\_socket < 0) {

perror("Accept failed");

exit(EXIT\_FAILURE);

}

printf("New connection, socket fd is %d, ip is: %s, port: %d\n", new\_socket, inet\_ntoa(address.sin\_addr), ntohs(address.sin\_port));

// Add new client socket to the pollfd array

for (i = 1; i < MAX\_CLIENTS; i++) {

if (fds[i].fd == -1) {

fds[i].fd = new\_socket;

fds[i].events = POLLIN; // Monitor this socket for read events

break;

}

}

if (i == MAX\_CLIENTS) {

printf("Too many clients connected. Connection rejected.\n");

close(new\_socket); // Too many clients connected, reject the new one

}

}

## **3.3: Tổng kết**

Cả select(), pselect(), và poll() đều có thể mang lại hiệu quả khi xử lý I/O multiplexing, nhưng mỗi phương pháp lại có cơ chế hoạt động và cách sử dụng khác nhau. Dưới đây là sự phân biệt giữa từng phương pháp:

* **select()**
* **Cách thức hoạt động**: theo dõi một danh sách các file descriptors (bao gồm các socket, pipe, stdin, stdout,...) để kiểm tra xem có sự kiện nào xảy ra (như dữ liệu sẵn sàng để đọc, sẵn sàng để ghi, hoặc lỗi) trên các file descriptors đó hay không.
* **Giới hạn số lượng file descriptors**: có giới hạn số lượng file descriptors mà nó có thể xử lý, thường là **1024** trên nhiều hệ điều hành. Điều này có thể gây hạn chế nếu cần giám sát rất nhiều kết nối đồng thời.
* **Cách hoạt động**:
  + Cần tính giá trị max\_fd (file descriptor lớn nhất) và thêm các file descriptor vào fd\_set (cấu trúc dữ liệu lưu trữ danh sách file descriptors cần giám sát).
  + Sau khi select() hoàn tất, cần kiểm tra thủ công từng file descriptor để xác định xem nó có sẵn sàng cho I/O hay không.
* **Hạn chế**: Khi số lượng file descriptors lớn, select() có thể không hoạt động hiệu quả do việc kiểm tra từng file descriptor thủ công và sự giới hạn số lượng file descriptors mà nó hỗ trợ.
* **pselect()**
* **Cách thức hoạt động**: là một phiên bản mở rộng của select() với khả năng xử lý tín hiệu (signal). Nó cho phép tạm thời thay đổi và khôi phục mặt nạ tín hiệu (sigmask) khi đang thực hiện đợi I/O, giúp xử lý tín hiệu một cách an toàn hơn.
* **Tính năng bổ sung**:
  + Không chỉ giám sát I/O mà còn tích hợp khả năng quản lý tín hiệu một cách hiệu quả, pselect() còn có thể tạm thời khóa một số tín hiệu trong quá trình đợi I/O để tránh xung đột, và khôi phục lại mặt nạ tín hiệu sau khi pselect() kết thúc.
* **Cách hoạt động**:
  + Tương tự select(), pselect() theo dõi danh sách file descriptors để kiểm tra sự kiện I/O, nhưng với tính năng mở rộng là cung cấp mặt nạ tín hiệu (signal mask).
* **Hữu ích**: đặc biệt hữu ích trong các chương trình xử lý tín hiệu như SIGINT hoặc SIGTERM để dừng chương trình một cách an toàn.
* **poll()**
* **Cách thức hoạt động**: hoạt động tương tự select() nhưng không bị giới hạn về số lượng file descriptors. poll() sử dụng một mảng cấu trúc pollfd[] để lưu trữ danh sách các file descriptors và các sự kiện muốn theo dõi (chẳng hạn như POLLIN, POLLOUT, POLLERR,...).
* **Không cần tính max\_fd**: poll() không yêu cầu tính toán max\_fd như select(), thay vào đó chỉ cần khai báo một mảng pollfd[], trong đó mỗi phần tử của mảng chứa một file descriptor và sự kiện tương ứng cần theo dõi.
* **Tính linh hoạt cao**:
  + Có thể dễ dàng thêm hoặc xóa file descriptors khỏi mảng pollfd[] mà không bị giới hạn số lượng như select().
  + Khi số lượng file descriptors lớn, poll() hoạt động hiệu quả hơn vì không cần phải kiểm tra từng file descriptor theo cách thủ công.
* **Cách hoạt động**:
  + poll() kiểm tra tất cả các file descriptors trong mảng pollfd[], sau đó xác định file descriptor nào đã sẵn sàng cho sự kiện I/O.
* **Ưu điểm**: Hiệu quả hơn select() khi xử lý với số lượng lớn file descriptors do không có giới hạn số lượng và dễ dàng quản lý việc thêm hoặc xóa file descriptors.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tiêu chí** | **select()** | **pselect()** | **poll()** |
| **Cơ chế theo dõi** | Giám sát file descriptors bằng fd\_set | Giám sát file descriptors và tín hiệu | Giám sát file descriptors bằng pollfd[] |
| **Giới hạn file descriptors** | Thường bị giới hạn (1024 trên nhiều hệ thống) | Tương tự select() nhưng thêm quản lý tín hiệu | Không giới hạn số lượng |
| **Xử lý tín hiệu** | Không tích hợp xử lý tín hiệu | Tích hợp khả năng quản lý tín hiệu an toàn | Không tích hợp tín hiệu |
| **Hiệu quả** | Không hiệu quả khi số lượng file descriptors lớn | Tương tự select(), nhưng tốt hơn khi cần quản lý tín hiệu | Hiệu quả hơn khi xử lý số lượng lớn file descriptors |
| **Thêm/Xóa file descriptors** | Phải tính lại max\_fd khi thêm hoặc xóa | Tương tự select() | Dễ dàng thêm/xóa với mảng pollfd[] |
| **Độ phân giải thời gian (Time Resolution)** | Sử dụng **cấu trúc struct timeval** để chỉ định thời gian chờ (timeout), có độ phân giải là **micro giây** | Sử dụng **cấu trúc struct timespec**, có độ phân giải cao hơn, lên đến **nano giây (nanosecond)** | Cho phép đặt thời gian chờ bằng **mili giây (milliseconds)**. |

**Khi nào nên sử dụng từng phương pháp:**

* **select()**: Thích hợp cho các chương trình nhỏ, số lượng file descriptors giới hạn và không cần xử lý tín hiệu phức tạp.
* **pselect()**: Sử dụng khi cần xử lý đồng thời I/O và tín hiệu một cách an toàn, chẳng hạn như khi muốn dừng chương trình hoặc server khi nhận tín hiệu SIGINT.
* **poll()**: Phù hợp với các chương trình lớn, có số lượng lớn file descriptors, nơi cần linh hoạt trong việc quản lý thêm/xóa file descriptors mà không bị giới hạn về số lượng.

# **Phần 4:** Tính Năng Phòng Chat và Trải Nghiệm Người Dùng

Server hoạt động, có 4 client tham gia

Server thông báo có người tham gia mới vào kênh

A screenshot of a computer screen

Description automatically generated

Server nhận được tin nhắn từ 1 client, sau đó gửi đi các client khác

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

|  |  |
| --- | --- |
| A screen shot of a computer  Description automatically generated | A screen shot of a computer  Description automatically generated |

Khi 1 client disconnect, thông báo tới các client khác

A screenshot of a computer screen

Description automatically generated

A screen shot of a computer

Description automatically generated

|  |  |
| --- | --- |
|  |  |

Khi các client disconnect, server tắt.

|  |  |
| --- | --- |
|  |  |
|  |  |

## **4.3: Cách mà chức năng phòng chat (phát tin nhắn, tên người dùng, v.v.) được triển khai**

* **Kết nối ban đầu:**
* **Client**:
  + Client khởi tạo một kết nối TCP tới server bằng cách tạo socket và sử dụng hàm connect().
  + Sau khi kết nối thành công, client yêu cầu người dùng nhập **username**. Sau đó, tên người dùng được gửi tới server qua socket.
* **Server**:
  + Server lắng nghe các kết nối từ client bằng cách sử dụng hàm socket(), bind(), và listen().
  + Khi có kết nối mới, server sử dụng hàm accept() để chấp nhận kết nối và thêm client vào danh sách các kết nối đang hoạt động.
  + Server nhận **tên người dùng** từ client và lưu trữ thông tin này.
  + Server gửi thông báo tới tất cả các client khác về việc có một client mới tham gia phòng chat.
* **Phát tin nhắn giữa các client:**
* **Client**:
  + Client lắng nghe tin nhắn từ server bằng cách sử dụng cơ chế **poll()**. Khi có sự kiện tin nhắn từ server, client nhận tin nhắn và in ra màn hình.
  + Người dùng có thể nhập tin nhắn, và client sẽ gửi tin nhắn đó tới server thông qua socket bằng cách sử dụng hàm send().
* **Server**:
  + Server lắng nghe tin nhắn từ tất cả các client bằng cách sử dụng hàm select(). Khi có tin nhắn từ bất kỳ client nào, server đọc tin nhắn này.
  + Sau đó, server phát (broadcast) tin nhắn đến tất cả các client khác (ngoại trừ client đã gửi tin nhắn).
* **Ngắt kết nối:**
* **Client**:
  + Khi một client đóng kết nối (ngắt kết nối hoặc thoát), server sẽ phát hiện ngắt kết nối này qua hàm select() hoặc khi nhận được tín hiệu ngắt kết nối từ client.
* **Server**:
  + Khi phát hiện một client đã ngắt kết nối, server sẽ xóa thông tin của client đó khỏi danh sách các kết nối hoạt động và đóng socket của client.
  + Server cũng gửi thông báo đến tất cả các client khác về việc client đã rời khỏi phòng chat.
* **Dừng server (Ctrl + C):**
* **Server**:
  + Khi server nhận tín hiệu SIGINT (Ctrl + C), nó sẽ thực hiện việc dừng an toàn: thông báo cho tất cả các client rằng server đang dừng và ngắt kết nối các client.
  + Sau đó, server sẽ đóng tất cả các kết nối và kết thúc chương trình.

## **4.4 Những khó khăn gặp phải trong quá trình triển khai và cách khắc phục:**

**1. Quản lý nhiều client cùng một lúc**:

* Khó khăn: Theo dõi đồng thời nhiều kết nối từ client và nhận/gửi tin nhắn từ tất cả client.
* Cách khắc phục: Sử dụng poll() hoặc select() để xử lý nhiều socket cùng lúc. Điều này cho phép server lắng nghe trên nhiều kết nối mà không cần tạo luồng mới cho mỗi kết nối.

**2. Đảm bảo không phát tin nhắn lại cho client đã gửi**:

* Khó khăn: Tránh phát tin nhắn của client gửi đi quay lại chính client đó.
* Cách khắc phục: Kiểm tra client đang gửi tin với các client khác trong hàm broadcast\_message() bằng cách so sánh sender\_fd với các client khác.

**3. Quản lý tên người dùng**:

* Khó khăn: Khi nhận tên người dùng từ client, cần xử lý chính xác tên mà không bị lỗi mất ký tự cuối hoặc trùng lặp.
* Cách khắc phục: Dùng fgets() để nhận tên người dùng từ client, sau đó loại bỏ ký tự \n thừa bằng hàm strcspn(). Đảm bảo chuỗi luôn được kết thúc bởi ký tự null \0.

**4. Xử lý ngắt kết nối (SIGINT)**:

* Khó khăn: Khi server nhận tín hiệu ngắt (Ctrl + C), cần đảm bảo mọi client được ngắt kết nối an toàn.
* Cách khắc phục: Thiết lập signal handler để server có thể bắt tín hiệu và thực hiện các thao tác dọn dẹp (đóng các kết nối, in thông báo) trước khi dừng.

**5. Hiển thị giao diện đẹp**:

* Khó khăn: Hiển thị thông báo với màu sắc khác nhau và phân biệt tin nhắn giữa các client.
* Cách khắc phục: Sử dụng các mã escape ANSI để in màu, giúp người dùng dễ dàng phân biệt giữa các loại thông báo (tin nhắn, người mới tham gia, người rời phòng).