

1. Mạng máy tính (Computer Network)

- Bao gồm: **host**, **router**, **kênh truyền thông** (communication channels).
- **Host** chạy các ứng dụng (applications).
- **Router** chuyển tiếp thông tin (forwarding).

2. Giao thức (Protocol)

- Giao thức là quy định chung cho việc:
 - Định dạng (format),
 - Địa chỉ (address),
 - Truyền dẫn (transmit),
 - Định tuyến (route),
 - Nhận dữ liệu (receive).

3. TCP/IP Protocol Suite

- Là bộ giao thức chính được dùng trên Internet.
- Gồm các tầng:
 - Ứng dụng (Application)
 - Giao vận (Transport): TCP / UDP
 - Mạng (Internet): IP
 - Liên kết dữ liệu (Link layer)

🌐 So sánh TCP và UDP

Tiêu chí	TCP	UDP
Loại kết nối	Có kết nối (connection-oriented)	Không kết nối (connectionless)
Độ tin cậy	Đảm bảo dữ liệu đến đúng thứ tự	Không đảm bảo thứ tự, có thể mất gói
Ứng dụng	HTTP, FTP, Email, SSH	DNS, TFTP, VoIP, video stream
Port phổ biến	80 (HTTP), 443 (HTTPS)	53 (DNS), 69 (TFTP)

✳️ Socket là gì?

- Là **giao diện lập trình ứng dụng (API)** để giao tiếp giữa các tiến trình qua mạng.
- Một socket gồm:
 - Địa chỉ IP
 - Giao thức (TCP hoặc UDP)
 - Port

📌 Các loại socket:

- **Stream socket (SOCK_STREAM)** – dùng TCP.
- **Datagram socket (SOCK_DGRAM)** – dùng UDP.
- **Unix domain socket** – giao tiếp giữa các tiến trình cùng máy.

💻 Tạo socket trong C

c
Sao chépChỉnh sửa

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    • AF_INET: Giao tiếp qua IPv4
    • SOCK_STREAM: Dùng TCP
    • 0: Dùng giao thức mặc định (TCP)
```

💼 Các hàm quan trọng

Hàm	Chức năng
socket()	Tạo socket
bind()	Gán địa chỉ IP và port
listen()	Đợi kết nối đến (dành cho server)
accept()	Chấp nhận kết nối từ client
connect()	Client dùng để kết nối tới server
send()/recv()	Gửi/nhận dữ liệu với TCP
sendto()/recvfrom()	Dùng cho UDP
close()	Đóng socket

Mô hình Client - Server

- **Client:** Khởi tạo kết nối (active socket).
- **Server:** Chờ kết nối đến và xử lý (passive socket).

BÀI 2: GIAO THỨC HTTP

1. HTTP là gì?

- HTTP (*Hypertext Transfer Protocol*) là giao thức lớp ứng dụng dùng trong **World Wide Web (WWW)**.
- Dựa trên **mô hình Client – Server**:
 - **Client:** trình duyệt (Chrome, Firefox...) gửi yêu cầu.
 - **Server:** máy chủ web (Apache, Nginx...) trả về nội dung.

2. Cơ chế hoạt động của HTTP

- HTTP dùng **TCP port 80**.
- **Client khởi tạo kết nối TCP**, gửi yêu cầu HTTP (request).
- **Server nhận**, xử lý và gửi lại phản hồi HTTP (response).
- Kết nối TCP có thể được đóng sau đó (HTTP/1.0) hoặc giữ lại (HTTP/1.1).

3. Kết nối trong HTTP

- ◆ **Non-persistent (không duy trì):**
 - Mỗi request → một kết nối TCP riêng.
 - Mỗi kết nối chỉ truyền **1 đối tượng**, rồi đóng.
 - Nhược điểm: chậm, tốn tài nguyên.
- ◆ **Persistent (duy trì – HTTP/1.1):**
 - Nhiều đối tượng (ảnh, CSS, JS...) được truyền qua **1 kết nối TCP**.
 - Nhanh hơn, giảm độ trễ.

4. HTTP Request (Yêu cầu từ Client)

Cấu trúc:

vbnf

Sao chépChỉnh sửa

GET /index.html HTTP/1.1

Host: www.example.com

User-Agent: Mozilla/5.0

Accept: text/html

Connection: keep-alive

Các phương thức HTTP phổ biến:

Phương thức Mô tả

GET	Lấy dữ liệu từ server
POST	Gửi dữ liệu từ client lên server
HEAD	Yêu cầu chỉ phần header (không có body)
PUT	Tải lên file mới và ghi đè
DELETE	Xóa tài nguyên trên server

📦 5. HTTP Response (Phản hồi từ Server)

Cấu trúc:

yaml

Sao chépChỉnh sửa

HTTP/1.1 200 OK

Date: Tue, 22 Apr 2025 12:00:00 GMT

Content-Type: text/html

Content-Length: 4321

Một số mã trạng thái HTTP phổ biến:

Mã Ý nghĩa

200 OK – thành công

301 Moved Permanently

400 Bad Request

404 Not Found

500 Internal Server Error

🍪 6. Cookies – Duy trì trạng thái phiên làm việc

- HTTP là **stateless** → không nhớ thông tin lần truy cập trước.
- Cookies giúp lưu trạng thái người dùng:
 - ID đăng nhập
 - Giỏ hàng
 - Lịch sử duyệt web...

Cookies gồm:

1. Dòng header Set-Cookie từ server.
2. Client lưu trữ và gửi lại cookie trong lần truy cập sau.

📁 7. Web cache (proxy server)

- **Web cache** là máy chủ trung gian, giúp:
 - Giảm thời gian tải (cache gần client hơn).
 - Giảm lưu lượng mạng.
- Nếu dữ liệu đã có sẵn trong cache, server gốc không cần gửi lại.

🛠 8. Thử nghiệm HTTP đơn giản

Dùng lệnh netcat để gửi thử công HTTP request:

bash

Sao chépChỉnh sửa

nc gaia.cs.umass.edu 80

GET /index.html HTTP/1.1

9. Tài liệu và khuyến khích:

- Nên dùng Wireshark để xem luồng HTTP request/response thực tế.
 - Có thể dùng curl, telnet, hoặc nc để mô phỏng client.

BÀI 3: FILE DESCRIPTOR (FD) – TRÌNH MÔ TẢ TỆP TRONG UNIX

1. File Descriptor là gì?

- Là **một số nguyên không âm** (int) mà hệ điều hành cấp phát cho mỗi tài nguyên được mở (file, socket, pipe, v.v.).
 - Dùng để **tham chiếu và thao tác** với tài nguyên đó (đọc, ghi, đóng...).

Ví dụ:

C

Sao chépChỉnh sửa

```
int fd = open("/etc/passwd", O_RDONLY);
```

Nếu $fd \geq 0$ → mở file thành công. Nếu $fd < 0$ → lỗi.

2. Hàm liên quan đến FD trong C

Hàm	Chức năng
open()	Mở một file và trả về FD
read()	Đọc dữ liệu từ FD
write()	Ghi dữ liệu vào FD
close()	Đóng FD
dup() / dup2()	Nhân bản FD

3. Các FD mặc định

FD số	Tên	Mô tả
0	stdin	Nhập chuẩn
1	stdout	Xuất chuẩn
2	stderr	Xuất lỗi chuẩn

Mỗi tiến trình UNIX đều có 3 FD này mở mặc định.

4. Ví dụ đọc tệp bằng FD

6

Sao chépChỉnh sửa

```
int fd = open("file.txt", O_RDONLY);
char buf[100];
int n = read(fd, buf, sizeof(buf));
write(1, buf, n); // ghi ra stdout
close(fd);
```

5. Blocking vs Non-blocking

- **Blocking I/O:** Gọi `read()` sẽ **chặn** chương trình chờ dữ liệu đến.
 - **Non-blocking I/O:** Gọi `read()` sẽ **trả về ngay** nếu không có dữ liệu.

6. Chia sẻ FD giữa các tiến trình

- Khi một tiến trình gọi fork(), các FD đang mở sẽ được **chia sẻ cho tiến trình con**.
- Cả cha và con đều dùng chung tài nguyên, nhưng có **bảng FD riêng biệt**.

✚ 7. Nhân bản FD bằng dup()

c

Sao chépChỉnh sửa

```
int fd2 = dup(fd1);
```

- Cả hai FD (fd1 và fd2) **trở đến cùng tài nguyên**.
- Đóng fd1 chưa giải phóng tài nguyên nếu fd2 còn mở.

🔗 8. Pipe (ống dẫn)

- Pipe là công cụ giao tiếp giữa các tiến trình (IPC – interprocess communication).
- Có thể tạo **unnamed pipe** hoặc **named pipe (FIFO)**.

📌 Ví dụ:

c

Sao chépChỉnh sửa

```
int fd[2];
pipe(fd);
if (fork() == 0)
    read(fd[0], buf, sizeof(buf)); // con đọc
else
    write(fd[1], msg, strlen(msg)); // cha ghi
```

💡 9. Tình huống thực tế

- Giao tiếp giữa client-server trong lập trình mạng **cũng sử dụng FD** như socket() trả về.
- Dùng select() hoặc poll() để theo dõi nhiều FD cùng lúc.

✉️ BÀI 4: ĐÓNG KHUNG VÀ GỬI NHẬN THÔNG ĐIỆP (MESSAGE FRAMING)

🧠 1. Message Framing là gì?

- **Framing** là cách mà client/server định nghĩa **đầu và cuối của một thông điệp** trong luồng dữ liệu.
- Vì TCP là **giao thức hướng dòng (stream-oriented)**, nên:
 - Một lần send() có thể được nhận thành nhiều recv().
 - Một lần recv() có thể nhận nhiều lần send().

📌 **TCP không giữ ranh giới thông điệp**, lập trình viên **phải tự xử lý framing!**

⚙️ 2. Các phương pháp framing phổ biến

◆ A. Dùng ký tự kết thúc (Delimiter character)

- Ví dụ: dùng '\n', '\0' để đánh dấu kết thúc thông điệp.

c

Sao chépChỉnh sửa

```
sprintf(msgBuffer, "%d %d\n", x, y);
send(sock, msgBuffer, strlen(msgBuffer), 0);
```

Ưu điểm:

- Dễ cài đặt, phù hợp với dữ liệu dạng văn bản.

Nhược điểm:

- Cần xử lý "escaping" nếu dữ liệu chứa ký tự đặc biệt ('\n', '\0').

◆ B. Gửi kích thước thông điệp trước (Structured data + Header)

- Gửi độ dài trước, sau đó mới gửi nội dung thông điệp:

C

Sao chépChỉnh sửa

```
uint32_t len = htonl(strlen(data));  
send(sock, &len, sizeof(len), 0);  
send(sock, data, strlen(data), 0);
```

Ưu điểm:

- Tin cậy, phù hợp với dữ liệu nhị phân.

Nhược điểm:

- Cần cài đặt hàm recv_all() để đảm bảo nhận đúng số byte.

3. Kỹ thuật mã hóa dữ liệu (Encoding Data)

◆ A. Mã hóa dạng văn bản (ASCII):

C

Sao chépChỉnh sửa

```
sprintf(buffer, "%d %d", x, y);  
• Gửi dạng chuỗi dễ debug nhưng không hiệu quả.
```

◆ B. Mã hóa nhị phân (Binary struct):

C

Sao chépChỉnh sửa

```
typedef struct {  
    int x, y;  
} msgStruct;
```

```
send(sock, &msgStruct, sizeof(msgStruct), 0);
```

- Nhanh và gọn, nhưng **phụ thuộc vào kiến trúc máy**.

4. Byte Ordering (Endianess)

- Các máy tính dùng thứ tự byte khác nhau:
 - Little-endian: Intel, AMD
 - Big-endian: mạng TCP/IP dùng chuẩn này

☞ Dùng các hàm chuyển đổi:

C

Sao chépChỉnh sửa

```
htons(), htonl() // host to network  
ntohs(), ntohl() // network to host
```

5. Framing thực tế trong TCP

- Ví dụ:

C

Sao chépChỉnh sửa

// Gửi

```
uint32_t len = htonl(strlen(message));  
send(sock, &len, sizeof(len), 0);  
send(sock, message, strlen(message), 0);
```

// Nhận

```
uint32_t len;  
recv(sock, &len, sizeof(len), 0);  
len = ntohl(len);  
recv(sock, buffer, len, 0);
```

6. Buffering khi chưa đầy dữ liệu

- Nếu không nhận được đầy đủ nội dung, cần:
 - **Lưu trữ tạm** trong buffer.
 - **Ghép dữ liệu lại** và chỉ xử lý khi đầy đủ.

BÀI 5: I/O MULTIPLEXING – LẬP TRÌNH I/O ĐA KÊNH

1. I/O Multiplexing là gì?

- Là kỹ thuật cho phép **chương trình chờ nhiều nguồn dữ liệu cùng lúc** (nhiều socket, stdin, pipe...).
 - Thay vì gọi read() và bị block, ta dùng select() hoặc poll() để chờ **khi nào FD sẵn sàng** rồi mới xử lý.
-

2. Tại sao cần I/O multiplexing?

- Trong ứng dụng mạng:
 - Client có thể nhận dữ liệu từ **socket và bàn phím** cùng lúc.
 - Server cần **chờ nhiều client cùng lúc**.
 - Không dùng đa luồng hoặc đa tiến trình → tiết kiệm tài nguyên.
-

3. Các mô hình I/O phổ biến

Mô hình

Đặc điểm

Blocking I/O read() sẽ chờ đến khi có dữ liệu

Non-blocking I/O read() trả về ngay nếu chưa có dữ liệu

I/O multiplexing select()/poll() chờ nhiều FD

Signal-driven I/O FD sẵn sàng → gửi tín hiệu SIGIO

Asynchronous I/O OS tự xử lý và gửi kết quả sau

4. Hàm select()

c

Sao chépChỉnh sửa

```
int select(int maxfdp1,  
          fd_set *readfds,  
          fd_set *writefds,  
          fd_set *exceptfds,  
          const struct timeval *timeout);  
• maxfdp1: số lớn nhất trong FD + 1  
• readfds: tập các FD cần chờ đọc  
• timeout: thời gian chờ (NULL nếu chờ vô hạn)
```

Kết quả trả về:

- 0: số FD sẵn sàng
 - 0: hết thời gian chờ
 - -1: lỗi
-

5. Các macro hỗ trợ fd_set

C

Sao chépChỉnh sửa

```
FD_ZERO(&fdset); // Xóa toàn bộ  
FD_SET(fd, &fdset); // Thêm fd  
FD_CLR(fd, &fdset); // Xóa fd  
FD_ISSET(fd, &fdset); // Kiểm tra fd có sẵn sàng không
```

6. Ví dụ: client xử lý stdin + socket

C

Sao chépChỉnh sửa

```
FD_SET(fileno(stdin), &rset);  
FD_SET(sockfd, &rset);  
maxfdp1 = max(fileno(stdin), sockfd) + 1;  
select(maxfdp1, &rset, NULL, NULL, NULL);  
  
if (FD_ISSET(fileno(stdin), &rset)) {  
    // đọc từ bàn phím  
}  
if (FD_ISSET(sockfd, &rset)) {  
    // đọc từ socket  
}
```

7. Hàm shutdown() – đóng 1 chiều kết nối TCP

C

Sao chépChỉnh sửa

```
shutdown(sockfd, SHUT_WR); // Ngừng gửi
```

Các chế độ:

- SHUT_RD: đóng chiều nhận
 - SHUT_WR: đóng chiều gửi (gửi FIN)
 - SHUT_RDWR: đóng cả hai
-

8. select() vs poll()

Tiêu chí select() poll()

Hạn chế FD 1024 (FD_SETSIZE) Không giới hạn

Gọn hơn Không Có (mảng pollfd)

Cũ hơn Có Không

9. Lưu ý khi dùng select()

- Phải gọi lại FD_SET mỗi lần trước khi select().
- Các bit trong fd_set bị **xóa sau khi select() trả về**.
- Kết hợp tốt với socket để viết **server đa kết nối một luồng**.

BÀI 6: SOCKET OPTION – TÙY CHỈNH SOCKET TRONG LẬP TRÌNH MẠNG

1. Socket option là gì?

- Là các **thuộc tính cấu hình** cho socket, giúp điều khiển hành vi ở nhiều tầng:
 - **Tầng socket (SOL_SOCKET)**

- Tầng IP (IPPROTO_IP)
- Tầng TCP (IPPROTO_TCP)

☞ Dùng để:

- Bật chế độ keep-alive
- Cho phép reuse địa chỉ (SO_REUSEADDR)
- Thiết lập thời gian timeout
- Kiểm soát kích thước buffer...

2. Hàm sử dụng

c

Sao chépChỉnh sửa

```
int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);  
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

- level: Tầng áp dụng option (SOL_SOCKET, IPPROTO_TCP, ...)
- optname: Tên tùy chọn (ví dụ SO_RCVBUF, TCP_NODELAY)
- optval: Con trỏ đến giá trị
- optlen: Kích thước của giá trị

3. Một số socket option quan trọng

◆ A. SO_REUSEADDR

- Cho phép reuse lại port cũ (khi server restart không bị “Address already in use”).

c

Sao chépChỉnh sửa

```
int opt = 1;  
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

◆ B. SO_KEEPALIVE

- Bật chế độ giữ kết nối TCP sống, tự gửi gói “keep-alive” sau một thời gian không hoạt động.

◆ C. SO_LINGER

- Điều khiển hành vi của close():
 - Mặc định: close() trả về ngay.
 - Bật linger: chờ gửi hết dữ liệu hoặc hết thời gian linger.

c

Sao chépChỉnh sửa

```
struct linger sl;  
sl.l_onoff = 1; // bật  
sl.l_linger = 10; // tối đa 10s  
setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &sl, sizeof(sl));
```

◆ D. SO_RCVBUF / SO_SNDBUF

- Thiết lập kích thước buffer cho việc nhận/gửi.

c

Sao chépChỉnh sửa

```
int size = 65536;  
setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));
```

◆ E. SO_RCVTIMEO / SO_SNDFTIMEO

- Đặt **timeout** cho recv() hoặc send(). Nếu hết thời gian → trả lỗi EWOULDBLOCK.

c

```
Sao chépChỉnh sửa  
struct timeval tv;  
tv.tv_sec = 5;  
tv.tv_usec = 0;  
setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
```

◆ F. TCP_NODELAY

- Tắt **Nagle's algorithm** để gửi dữ liệu nhỏ lập tức, phù hợp với ứng dụng thời gian thực.

💡 4. Kiểm tra option

- Dùng getsockopt() để đọc giá trị hiện tại của socket option.
- Một số option có thể undefined trên vài hệ điều hành, cần #ifdef.

📌 5. Option kế thừa

- Khi socket accept() tạo kết nối mới → socket con **kế thừa option** từ socket lắng nghe:
 - SO_KEEPALIVE, TCP_NODELAY, SO_LINGER, ...

🧠 Bài 7: TCP Out-of-Band Data (OOB Data)

📌 Mục tiêu bài học

- Hiểu **Out-of-Band Data (OOB)** là gì.
- Cách hoạt động của OOB trong **TCP**.
- Cách lập trình gửi và nhận OOB data trong C.
- Một số **lưu ý quan trọng** và ví dụ minh họa.

✳️ 1. Out-of-Band Data là gì?

- **OOB** (Out-of-Band) hay còn gọi là **expedited data** là một dạng dữ liệu đặc biệt có độ ưu tiên cao hơn so với dữ liệu thông thường (**in-band**).
- Trong TCP, khi gửi OOB, dữ liệu này sẽ được **báo trước đến phía nhận** trước cả những dữ liệu thông thường đang xếp hàng gửi.

💬 UDP không hỗ trợ OOB.

💬 TCP hỗ trợ thông qua chế độ gọi là **urgent mode**, nhưng thực chất vẫn đi cùng luồng dữ liệu (không hoàn toàn tách biệt).

⚙️ 2. Cách TCP xử lý OOB

- Khi gọi hàm send() với flag MSG_OOB, bạn gửi một byte "khẩn":

c

```
Sao chépChỉnh sửa  
send(sockfd, "X", 1, MSG_OOB);
```

- TCP sẽ:
 - Đặt **URG flag** trong header.
 - Cài đặt **urgent pointer** để đánh dấu vị trí **sau** byte OOB trong luồng dữ liệu.

💡 Lưu ý: TCP chỉ hỗ trợ gửi một byte OOB duy nhất tại một thời điểm.

💡 3. Phía nhận xử lý OOB như thế nào?

- Khi có OOB, phía nhận:
 - **Nhận tín hiệu SIGURG** (nếu đã thiết lập xử lý tín hiệu).

- Có thể dùng select() để chờ **exception condition**.
- Dùng recv(..., MSG_OOB) để đọc byte OOB.

Cấu hình:

C

Sao chépChỉnh sửa
fcntl(sockfd, F_SETOWN, getpid()); // Nhận SIGURG

Đọc dữ liệu OOB:

C

Sao chépChỉnh sửa
char buf;
recv(sockfd, &buf, 1, MSG_OOB);

⚠ Nếu bạn không đọc kịp OOB trước khi byte tiếp theo tới, bạn **mất byte đó**.

⚙ 4. Tùy chọn SO_OOBINLINE

- Mặc định, byte OOB **không nằm trong** buffer nhận thường.
- Dùng setsockopt() để thay đổi:

C

Sao chépChỉnh sửa
int on = 1;
setsockopt(sockfd, SOL_SOCKET, SO_OOBINLINE, &on, sizeof(on));
• Nếu bật SO_OOBINLINE, OOB sẽ được **nhét vào luồng dữ liệu thường** và ta xác định vị trí bằng:

C

Sao chépChỉnh sửa
sockatmark(sockfd); // Trả về 1 nếu đang ở "mark"

💡 5. Một số ví dụ lập trình quan trọng

- tcpsend01.c & tcprecv01.c: Gửi và nhận OOB, xử lý SIGURG.
- tcprecv02.c: Dùng select() để phát hiện OOB (trên một số hệ điều hành sẽ lỗi nếu gọi recv(MSG_OOB) hai lần).
- tcprecv03.c: Phiên bản khắc phục lỗi EINVAL bằng cách đọc hết in-band trước khi check OOB.

✳ 6. Tổng kết

Thành phần Vai trò

MSG_OOB Gửi/nhận OOB

SIGURG Tín hiệu báo có OOB

fcntl(F_SETOWN) Đăng ký nhận SIGURG

SO_OOBINLINE Đưa OOB vào luồng thường

sockatmark() Kiểm tra đang đọc đến OOB chưa

🧠 Bài 8: Broadcasting và Multicasting trong IP

📌 Mục tiêu bài học

- Hiểu khái niệm **Broadcast** và **Multicast**.
- Phân biệt **Unicast - Broadcast - Multicast**.
- Cách lập trình gửi nhận Multicast trong C.
- Nắm được vai trò của **IGMP** và **Định tuyến Multicast**.

1. Giới thiệu

- **Broadcast** và **Multicast** là hai kỹ thuật truyền dữ liệu đến **nhiều thiết bị** cùng lúc trong mạng.
- Thường dùng với **UDP** hoặc **Raw IP**, không dùng với TCP.
- **IPv4** hỗ trợ cả hai; **IPv6** chỉ hỗ trợ **Multicast** (không có broadcast).

2. Broadcast trong IPv4

- Địa chỉ Broadcast = {subnetid, hostid = -1}
 - Ví dụ: 192.168.1.255

Cấu hình Broadcast Socket:

c

Sao chépChỉnh sửa

int on = 1;

setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

- **Không được phân mảnh** (fragmentation bị cấm trong broadcast).

3. Multicast – Giao tiếp nhiều điểm

- Giao tiếp từ **một đến nhiều**, hoặc **nhiều đến nhiều**.
- Cho phép **một tập con** của các máy chủ tham gia (hơn hẳn broadcast).
- Có hai kiểu tổ chức logic:
 - **Rooted** (có node gốc)
 - **Peer-to-peer** (tất cả ngang hàng)

4. Cấu trúc hệ thống Multicast

Control Plane:

- Quản lý việc tạo session multicast.
- Máy chủ (c_root) khởi tạo, các máy khách (c_leaf) tham gia.

Data Plane:

- Quản lý luồng dữ liệu trong phiên multicast.
- Dữ liệu đi từ d_root đến d_leaf (không có truyền trực tiếp giữa leafs).

5. Multicast IP (RFC 1112)

- Dùng địa chỉ **Class D**: từ 224.0.0.0 đến 239.255.255.255.
- Dựa trên **UDP**, không dùng TCP.
- Mỗi nhóm multicast có một địa chỉ IP riêng.
- Hỗ trợ nhờ **IGMP** và các router có hỗ trợ multicast.

6. IGMP Protocol – Internet Group Management Protocol

- IGMP là cầu nối giữa **host** và **router** để quản lý nhóm multicast.
- Phiên bản phổ biến: **IGMPv2, IGMPv3**

Các loại thông điệp:

Loại Message Ý nghĩa

Membership Query Router hỏi host đang tham gia nhóm nào

Membership Report Host trả lời đang tham gia nhóm nào

Leave Group Host thông báo rời nhóm

7. Định tuyến Multicast

- Router không lưu danh sách các thành viên, chỉ lưu **địa chỉ nhóm** theo từng interface.
- Dữ liệu chỉ được gửi đến những interface có thành viên tham gia.

Hai kỹ thuật định tuyến chính:

- **Source-based Trees**
- **Group-shared Trees**

💡 8. Mapping địa chỉ Multicast IP → Ethernet MAC

- IP Class D 32-bit được ánh xạ sang MAC 48-bit như sau:

Thành phần Giá trị

24 bit đầu MAC 01:00:5E

1 bit kế tiếp 0

23 bit cuối Từ địa chỉ IP

Ví dụ:

224.0.1.88 → 01:00:5E:00:01:58

🛠 9. Lập trình Multicast (Gửi & Nhận)

Tạo socket Multicast:

c

Sao chépChỉnh sửa

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

Gia nhập nhóm:

c

Sao chépChỉnh sửa

```
struct ip_mreq mreq;  
mreq.imr_multiaddr.s_addr = inet_addr("224.0.0.1");  
mreq.imr_interface.s_addr = htonl(INADDR_ANY);  
setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

Gửi dữ liệu Multicast:

c

Sao chépChỉnh sửa

```
sendto(sockfd, msg, strlen(msg), 0, (struct sockaddr*)&dest, sizeof(dest));
```

✓ Tổng kết nhanh

Kỹ thuật Địa chỉ IP Giao thức Phạm vi

Unicast Một máy duy nhất TCP/UDP Point-to-point

Broadcast subnetID.255 UDP Toàn subnet

Multicast 224.x.x.x → 239.x.x.x UDP Một nhóm

🧠 Bài 9: Virtual Private Network (VPN) & Giao diện TUN/TAP

📌 Mục tiêu bài học

- Hiểu cơ bản về **VPN** là gì và lý do cần VPN.
- Nắm được khái niệm **IP tunneling**.
- Phân biệt **TUN** và **TAP** interface.
- Biết cách lập trình tạo và sử dụng tunnel với giao diện ảo TUN.
- Ứng dụng: **bypass firewall**.

🔒 1. Virtual Private Network (VPN) là gì?

- Mạng riêng ảo (VPN) giúp kết nối một cách **an toàn** giữa thiết bị bên ngoài với mạng nội bộ (**private network**).
- VPN sử dụng **mạng công cộng (Internet)** để tạo đường hầm truyền thông tin **bảo mật**.

Lợi ích:

- Ẩn IP.
- Truy cập nội dung bị chặn.
- Truy cập tài nguyên mạng nội bộ từ xa.

2. Cơ chế hoạt động VPN

- Máy khách (client) kết nối tới **VPN Server**.
- Server xác thực và cấp quyền truy cập mạng nội bộ.
- Mọi lưu lượng được **gói** (encapsulated) trong **IP tunneling**.

3. IP Tunneling

- Là kỹ thuật nhúng (encapsulate) một gói tin IP **bên trong một gói tin IP khác**.
- Lớp ngoài là IP công cộng, lớp trong là gói tin riêng tư.

Có 2 kiểu tunneling:

Loại

Mô tả

IPSec tunneling Làm việc ở kernel, bảo mật thấp tầng

TLS/SSL tunneling Làm việc ở ứng dụng, dựa vào TCP/UDP

 Bài này tập trung vào **TLS/SSL tunneling**.

4. Giao diện TUN và TAP

- TUN** = Virtual Network Interface ở **Layer 3 (IP)**
- TAP** = Virtual Network Interface ở **Layer 2 (Ethernet)**

Interface Tầng OSI Gửi/nhận

TUN IP Layer (L3) IP packets

TAP Ethernet (L2) Ethernet frames

 **Giao diện TUN** giúp ứng dụng đọc/gửi gói tin IP giống như thật.

5. Tạo và cấu hình TUN Interface

c

Sao chépChỉnh sửa

struct ifreq ifr;

ifr.ifr_flags = IFF_TUN | IFF_NO_PI; // TUN device, no packet info

strcpy(ifr.ifr_name, "tun0");

// Tạo TUN device

int fd = open("/dev/net/tun", O_RDWR);

ioctl(fd, TUNSETIFF, &ifr);

Cấu hình IP:

bash

Sao chépChỉnh sửa

ip addr add 10.4.2.1/24 dev tun0

ip link set tun0 up

6. Thiết lập định tuyến (routing)

- Để gói tin đi qua tunnel, cần chỉnh bảng định tuyến:

bash

Sao chépChỉnh sửa

```
ip route add 192.168.60.0/24 dev tun0
```

- Gói tin gửi đến 192.168.60.x sẽ được **gửi qua TUN interface**, và xử lý bởi chương trình VPN.

7. Giao tiếp với TUN Interface

Đọc gói tin từ TUN:

c

Sao chépChỉnh sửa

```
read(tun_fd, buffer, BUFSIZE);
```

Gửi gói tin tới TUN:

c

Sao chépChỉnh sửa

```
write(tun_fd, buffer, BUFSIZE);
```

Gói tin này sẽ được "nhét" vào hệ thống mạng như thật!

8. Tạo VPN với TLS/SSL Tunnel

Bên Client:

- Đọc gói từ tun0.
- Gói lại và mã hóa bằng SSL.
- Gửi đến server qua TCP/UDP.

Bên Server:

- Nhận dữ liệu qua SSL.
- Giải mã và lấy gói IP.
- Gửi gói vào tun0.

9. Giám sát giao tiếp TUN <-> Socket

Sử dụng select() để giám sát cả TUN và socket:

c

Sao chépChỉnh sửa

```
select(maxfd + 1, &readfds, NULL, NULL, NULL);
```

Khi có dữ liệu ở một trong hai, xử lý chuyển tiếp sang phía còn lại.

10. Bypass Firewall với VPN

Tình huống:

- Người dùng bị **chặn truy cập Facebook** từ mạng công ty.

Giải pháp:

- Dùng VPN client gửi gói Facebook tới VPN server.
- VPN server **phát tán ra Internet**.
- Nhận phản hồi từ Facebook → gửi ngược lại qua tunnel.

 **Firewall không thể nhận ra vì gói Facebook nằm trong gói VPN!**

Tổng kết nhanh

Thành phần Mô tả

VPN Tạo mạng riêng ảo bảo mật qua Internet

TUN Interface ảo xử lý gói IP (L3)

Thành phần	Mô tả
TAP	Interface ảo xử lý gói Ethernet (L2)
IP Tunneling	Bao gói gói IP bên trong IP khác
TLS/SSL Tunnel	Mã hóa gói tin ở tầng ứng dụng
Firewall Bypass	Giấu gói tin thật trong gói VPN

Bài 10: Các Mô Hình Thiết Kế Client/Server trong Lập Trình Mạng

Mục tiêu bài học

- Hiểu các cách xây dựng server xử lý nhiều client.
- So sánh các mô hình: **iterative, fork, select, thread, pre-fork, pre-thread**.
- Biết cách đo lường hiệu năng giữa các mô hình.
- Hiểu về vấn đề đồng bộ hóa khi có nhiều tiến trình/luồng gọi accept().

1. Các mô hình server phổ biến

Mô hình Mô tả

Iterative Xử lý 1 client tại 1 thời điểm

Fork-based Tạo tiến trình con cho mỗi client

Thread-based Tạo thread mới cho mỗi client

Select-based Dùng select() để quản lý nhiều kết nối

Pre-fork Tạo sẵn N tiến trình con khi server khởi động

Pre-thread Tạo sẵn N thread khi server khởi động

2. Mô hình kiểm thử: TCP Test Client

bash

Sao chépChỉnh sửa

```
client <IP> <port> <#children> <#loops/child> <#bytes/request>
```

Ví dụ:

bash

Sao chépChỉnh sửa

```
client 192.168.1.20 8888 5 500 4000
```

- Tạo **2,500 kết nối TCP** từ 5 tiến trình con.
- Gửi "4000\n" để yêu cầu server gửi lại 4000 byte.

3. Iterative Server

- Xử lý từng client một cách tuần tự (blocking accept()).
- Mã nguồn: serv00.c

▼ **Hiệu suất thấp nhất** – dùng làm mốc so sánh CPU time.

4. Fork-based Concurrent Server

- Tạo một tiến trình con mới (fork()) cho mỗi client.
- Mã nguồn: serv01.c, web_child.c
- Sử dụng SIGCHLD để thu dọn zombie process.
- Sử dụng pr_cpu_time.c để đo CPU time.

 **Tốn nhiều CPU nhất**, do chi phí tạo tiến trình lớn.

5. Pre-forked Server (No Locking)

- Tạo sẵn N tiến trình khi khởi động.
- Tất cả cùng gọi accept() → hệ điều hành đánh thức tất cả (thundering herd).
- Mã nguồn: serv02.c, child02.c

⚠ Một số hệ điều hành cần dùng kỹ thuật khóa quanh accept().

🔒 6. Pre-forked Server (File Locking)

- Dùng fcntl() để đảm bảo chỉ 1 tiến trình thực sự gọi accept() tại 1 thời điểm.
- Mã nguồn: serv03.c, child03.c, lock_fcntl.c

🔒 **Tốt cho hệ thống không hỗ trợ accept() đồng thời.**

📝 7. Pre-forked Server (Thread Locking)

- Dùng mutex để đồng bộ accept() giữa các thread.
- Mã nguồn: serv04.c, child04.c

✓ Nhanh hơn file locking vì không phụ thuộc filesystem.

䎂 8. Pre-forked Server (Descriptor Passing)

- Chỉ **cha gọi accept()**, rồi gửi socket cho tiến trình con qua UNIX socket.
- Cần theo dõi tiến trình nào rảnh để phân phối.
- Mã nguồn: serv05.c, child05.c

❗ Chậm hơn do chi phí giao tiếp nội bộ.

📝 9. Thread-per-client Server

- Tạo một thread mới cho mỗi kết nối client.
- Mã nguồn: serv06.c

✓ **Nhanh nhất**, ít overhead.

䎂 10. Pre-threaded Server (Each thread accept)

- Tạo pool thread, mỗi thread gọi accept() và xử lý client riêng.
- Dùng mutex bảo vệ accept().
- Mã nguồn: serv07.c

⌚ Cân bằng giữa hiệu suất và kiểm soát.

䎂 11. Pre-threaded Server (Main thread accept)

- Chỉ main thread gọi accept() → đẩy socket vào queue chung.
 - Các thread còn lại lấy socket ra xử lý.
 - Dùng mutex + condition variable để đồng bộ queue.
 - Mã nguồn: serv08.c
- ♦ Chậm hơn per-thread accept do chi phí đồng bộ queue.
-

📊 12. Tổng kết hiệu năng (Theo CPU Time)

STT	Mô hình	Thời gian CPU (so với Iterative)
0	Iterative	0.00 (baseline)
1	Fork mỗi client	20.90
2	Pre-fork (no lock)	1.80
3	Pre-fork (file lock)	2.07
4	Pre-fork (thread lock)	1.75

STT	Mô hình	Thời gian CPU (so với Iterative)
5	Pre-fork + pass descriptor	2.58
6	Thread mỗi client	0.99
7	Pre-thread (mutex accept)	1.93
8	Pre-thread (main accept)	2.05

Tổng kết nhanh

Mô hình	Ưu điểm	Nhược điểm
Fork	Dễ hiểu	Chậm, tốn tài nguyên
Thread	Nhanh	Cần đồng bộ
Select	Không tạo tiến trình	Phức tạp khi xử lý logic
Pre-fork/thread	Cân bằng tốt	Cần quản lý pool

Bài 10: Raw Sockets và Giao Thức ICMP

Mục tiêu bài học

- Hiểu khái niệm **Raw Socket** và khi nào cần dùng.
 - Nắm được cách hoạt động của giao thức **ICMP**.
 - Biết cách lập trình các công cụ như **Ping** và **Traceroute**.
 - Nhận biết các rủi ro bảo mật liên quan đến raw socket.

1. Raw Sockets là gì?

- Thông thường, socket dùng để xây dựng ứng dụng trên nền TCP hoặc UDP.
 - Nhưng nếu bạn muốn làm việc **trực tiếp với IP**, cần dùng **Raw Socket**.

Raw socket cho phép:

- Gửi/nhận gói tin IP **tự thiết kế** (gồm cả header).
 - Truy cập các giao thức không phải TCP/UDP như **ICMP, IGMP**.
 - **Bypass kernel TCP/UDP processing** → Lập trình ở tầng thấp.

Yêu cầu:

Chỉ người dùng có quyền root/sudo mới tạo được raw socket.

2. Tao Raw Socket

6

Sao chépChỉnh sửa

```
int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

Tham số	Ý nghĩa
AF_INET	IPv4
SOCK_RAW	Raw Socket
IPPROTO_ICMP	Sử dụng giao thức ICMP

3. Gửi gói tin qua Raw Socket

Dùng `sendto()` hoặc `sendmsg()` như UDP:

C

Sao chépChỉnh sửa

```
sendto(sockfd, packet, len, 0, (struct sockaddr*)&addr, sizeof(addr));
```

Cho phép tự tạo IP header:

C

```
Sao chépChỉnh sửa  
int on = 1;  
setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
```

4. Nhận gói tin qua Raw Socket

Dùng recvfrom():

c

```
Sao chépChỉnh sửa  
recvfrom(sockfd, buffer, sizeof(buffer), 0, ...);
```

5. Giao thức ICMP – Internet Control Message Protocol

- **ICMP** là giao thức điều khiển hoạt động bên trên IP.
- Dùng để:
 - Kiểm tra kết nối (Ping).
 - Báo lỗi mạng (Destination Unreachable, TTL Exceeded).

ICMP nằm trên IP nhưng dưới TCP/UDP.

6. Cấu trúc ICMP Message

Trường Ý nghĩa

Type Loại ICMP message

Code Mã chi tiết phụ

Checksum Kiểm tra lỗi

Data Tùy theo loại message

Các loại phổ biến:

Type Code Ý nghĩa

0	-	Echo Reply
8	-	Echo Request (Ping)
3	0-3	Destination Unreachable
11	0	Time Exceeded (trong Traceroute)

7. Ví dụ 1: Ping (Echo Request/Reply)

Ping gửi ICMP Echo Request (Type = 8), host đích trả về Echo Reply (Type = 0).

Cấu trúc ICMP:

c

Sao chépChỉnh sửa

```
struct icmp {  
    u_char type;  
    u_char code;  
    u_short checksum;  
    u_short id;  
    u_short seq;  
    char data[];  
};
```

id thường là PID của tiến trình. seq để đánh số thứ tự gói gửi.

8. Ví dụ 2: Traceroute

- Traceroute gửi nhiều gói ICMP (hoặc UDP) với TTL tăng dần.
- Mỗi router giảm TTL, nếu TTL = 0 → trả về ICMP Time Exceeded.

Vòng lặp:

1. Gửi gói với TTL = 1 → nhận phản hồi từ router đầu tiên.
2. TTL = 2 → nhận từ router thứ hai.
3. ... đến khi tới đích → nhận ICMP Echo Reply.

9. Lưu ý bảo mật

- Raw sockets có thể bị **lợi dụng để giả mạo gói tin (spoofing)**.
- Ví dụ: giả mạo địa chỉ nguồn trong TCP SYN attack.
- Linux từ kernel 2.2 giới hạn raw socket không gửi được TCP.

 **Luôn chạy raw socket bằng user root và chỉ trong môi trường kiểm soát.**

Tổng kết nhanh

Thành phần Mô tả

Raw Socket Giao tiếp trực tiếp với IP

ICMP Giao thức kiểm tra và báo lỗi mạng

Ping Gửi ICMP Echo Request

Traceroute Gửi gói với TTL tăng dần

SOCK_RAW Dùng để tạo socket tầng thấp

IP_HDRINCL Cho phép tự xây IP header

Bài 11: OpenSSL & TLS – Bảo mật truyền thông mạng

Mục tiêu bài học

- Hiểu khái niệm **SSL/TLS** là gì và tại sao cần thiết.
- Giới thiệu về thư viện **OpenSSL**.
- Cách sử dụng **command-line** và **API** của OpenSSL.
- Các bước cấu hình một kết nối HTTPS đơn giản.
- Lưu ý bảo mật và hạn chế của OpenSSL.

1. SSL/TLS là gì?

- **SSL (Secure Socket Layer)** và kế thừa là **TLS (Transport Layer Security)** là giao thức **mã hóa dữ liệu** khi truyền trên mạng.
- Đảm bảo:
 - **Bảo mật (confidentiality)** – Dữ liệu được mã hóa.
 - **Xác thực (authentication)** – Đảm bảo đúng server.
 - **Toàn vẹn (integrity)** – Phát hiện thay đổi dữ liệu.

Cấu trúc SSL:

- **Handshake Protocol**: Xác thực & tạo khóa phiên.
- **Record Protocol**: Mã hóa và truyền dữ liệu.

2. OpenSSL là gì?

- **OpenSSL** là bộ công cụ mã nguồn mở hỗ trợ:
 - Thực hiện các giao thức SSLv2, SSLv3, TLSv1.
 - Các thuật toán mã hóa (AES, DES, RSA, SHA...).
 - Dùng được ở dòng lệnh và trong lập trình (API C, C++, Python...).

Tính năng:

- Tạo khóa RSA/DSA/DH
 - Tạo chứng chỉ số (X.509)
 - Mã hóa/giải mã file
 - Kiểm tra kết nối TLS
-

3. Dùng OpenSSL trên dòng lệnh

Tạo root certificate:

bash

Sao chépChỉnh sửa

```
openssl req -x509 -days 3650 -newkey rsa:2048 -keyout ca.key -out ca.crt
```

Tạo CSR (Certificate Signing Request):

bash

Sao chépChỉnh sửa

```
openssl req -newkey rsa:2048 -keyout server.key -out server csr
```

Ký CSR để cấp chứng chỉ:

bash

Sao chépChỉnh sửa

```
openssl x509 -req -in server csr -CA ca crt -CAkey ca key -CAcreateserial -out server crt
```

Xuất chứng chỉ dạng PKCS#12:

bash

Sao chépChỉnh sửa

```
openssl pkcs12 -export -in server crt -inkey server key -out server p12 -certfile ca crt
```

4. Cấu hình HTTPS với Apache + mod_ssl

Trong file httpd.conf:

apache

Sao chépChỉnh sửa

Listen 443

```
LoadModule ssl_module modules/mod_ssl.so
```

```
<VirtualHost _default_:443>
```

SSLEngine on

SSLCertificateFile "conf/ssl crt/server crt"

SSLCertificateKeyFile "conf/ssl key/server key"

SSLCACertificateFile "conf/ssl crt/ca crt"

```
</VirtualHost>
```

Bảo vệ thư mục:

apache

Sao chépChỉnh sửa

```
<Location /admin>
```

SSLVerifyClient require

SSLRequire %{SSL_CLIENT_S_DN_CN} eq "Administrator"

```
</Location>
```

5. S/MIME với OpenSSL

S/MIME = Secure MIME – Bảo mật email (ký & mã hóa).

- Ký mail:

bash

Sao chépChỉnh sửa

```
openssl smime -sign -in message.txt -out signed.eml -signer user.pem
```

- **Mã hóa mail:**

bash

Sao chépChỉnh sửa

```
openssl smime -encrypt -des3 -in message.txt -out encrypted.eml user.crt
```

- **Giải mã mail:**

bash

Sao chépChỉnh sửa

```
openssl smime -decrypt -in encrypted.eml -recip user.pem
```

✉ 6. Lập trình với OpenSSL (API)

Sử dụng hai thư viện:

Thư viện Chức năng

libssl Thực hiện SSL/TLS

libcrypto Mã hóa, hàm băm, khóa công khai

Một số thành phần API:

- SSL_new(), SSL_connect(), SSL_accept(), SSL_write(), SSL_read()
- BIO – Truyền dữ liệu
- EVP – API mã hóa/hàm băm cấp cao
- X509 – Chứng chỉ

OpenSSL API rất mạnh nhưng **khá phức tạp**, và **thiếu tài liệu chính thức rõ ràng**.

🚧 7. Hạn chế và rủi ro của OpenSSL

- Không thân thiện với người mới học (non-OOP).
- API phức tạp và tài liệu khó hiểu.
- Một số phiên bản cũ có lỗi nghiêm trọng (vd: Heartbleed).
- Vấn đề về **thư viện dùng chung** trên các nền tảng.

✓ Tổng kết nhanh

Thành phần Mô tả

TLS/SSL Giao thức bảo mật truyền thông

OpenSSL Công cụ mã hóa và chứng chỉ mã nguồn mở

Certificate (CRT) Chứng nhận máy chủ, máy khách

PKCS#12 Định dạng đóng gói khóa + chứng chỉ

API OpenSSL C/C++ interface để sử dụng SSL/TLS trong code

S/MIME Ký/mã hóa email

🧠 Bài 12: gRPC & Remote Procedure Call (RPC)

📌 Mục tiêu bài học

- Hiểu khái niệm và nguyên lý hoạt động của **RPC**.
- Phân biệt **Local Call** và **Remote Call**.
- Giới thiệu **gRPC**, một framework hiện đại để lập trình RPC.
- Biết quy trình xây dựng ứng dụng sử dụng gRPC (client & server).
- Thử nghiệm demo gRPC với Python.

📦 1. RPC là gì?

Remote Procedure Call (RPC) là cơ chế cho phép một chương trình gọi một hàm nằm trên

một máy khác như thể nó đang gọi một hàm cục bộ.

💡 Khái niệm này mô phỏng việc gọi hàm như trong lập trình thông thường, nhưng thực ra là gửi/nhận dữ liệu qua mạng.

⌚ 2. So sánh Local Call vs Remote Call

Đặc điểm	Local Call	RPC
Vị trí thực thi	Cùng tiến trình	Qua mạng
Dễ lập trình	✓	✓ (ẩn phức tạp)
Tính linh hoạt	Hạn chế	Cao
Có thể bị lỗi mạng	✗	✓

⚙️ 3. Cơ chế hoạt động của RPC

Gồm các bước:

1. Client gọi hàm local (stub).
2. Stub đóng gói dữ liệu (**serialize**), gửi qua mạng.
3. Server stub nhận và giải mã (**deserialize**), gọi hàm thực tế.
4. Kết quả trả về theo chiều ngược lại.

Stub là phần giúp **ẩn đi việc truyền dữ liệu** giữa client và server.

✳️ 4. gRPC là gì?

- gRPC là một framework RPC mã nguồn mở do Google phát triển.
- Sử dụng giao thức **HTTP/2** và ngôn ngữ định nghĩa dịch vụ **Protocol Buffers (protobuf)**.

Tính năng nổi bật:

- ✓ Hỗ trợ nhiều ngôn ngữ: C++, Python, Go, Java, C#, Rust,...
- ✓ Giao tiếp nhanh và nhẹ
- ✓ Hỗ trợ **streaming** hai chiều
- ✓ Có tính năng xác thực & mã hóa (SSL/TLS)

📚 5. Các kiểu gọi RPC trong gRPC

Kiểu gọi	Mô tả
Unary RPC	Gửi một yêu cầu và nhận một phản hồi
Server Streaming	Gửi một yêu cầu, nhận nhiều phản hồi
Client Streaming	Gửi nhiều yêu cầu, nhận một phản hồi
Bidirectional Streaming	Gửi và nhận nhiều yêu cầu/phản hồi đồng thời

🛠️ 6. Quy trình phát triển với gRPC

Các bước:

1. Viết file **.proto** định nghĩa dịch vụ và message.
2. Dùng protoc để **generate code stub** cho client & server.
3. **Implement service logic** trong server.
4. Tạo client và gọi các hàm như gọi hàm bình thường.
5. Chạy và test ứng dụng.

abc 7. Ví dụ .proto file

protobuf

```
Sao chépChỉnh sửa
syntax = "proto3";

service HelloService {
    rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
    string name = 1;
}

message HelloReply {
    string message = 1;
}
```

8. Demo gRPC với Python

Cài đặt:

bash

Sao chépChỉnh sửa

```
pip install grpcio grpcio-tools
```

Sinh mã:

bash

Sao chépChỉnh sửa

```
python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. hello.proto
```

Server:

python

Sao chépChỉnh sửa

```
class HelloServicer(hello_pb2_grpc.HelloServiceServicer):
    def SayHello(self, request, context):
        return hello_pb2.HelloReply(message=f"Hello, {request.name}!")
```

Client:

python

Sao chépChỉnh sửa

```
channel = grpc.insecure_channel('localhost:50051')
stub = hello_pb2_grpc.HelloServiceStub(channel)
response = stub.SayHello(hello_pb2.HelloRequest(name='Alice'))
print(response.message)
```

9. Ưu điểm của gRPC

-  Tối ưu hóa tốc độ với HTTP/2.
-  Data nhỏ gọn nhờ Protocol Buffers.
-  Hỗ trợ **bi-directional streaming** rất mạnh.
-  Rất phù hợp với **microservices**.

10. Nhược điểm của gRPC

-  Khó debug hơn REST.
-  Không dễ dùng trực tiếp trên trình duyệt (cần proxy).

- ✗ Yêu cầu client và server phải có cùng .proto.
-

Tổng kết nhanh

Thành phần Mô tả

RPC	Gọi hàm từ xa như gọi hàm cục bộ
gRPC	Framework RPC sử dụng protobuf + HTTP/2
.proto	Định nghĩa hàm và kiểu dữ liệu
Stub	Mã sinh ra giúp client/server giao tiếp
Streaming	Cho phép gửi/nhận nhiều gói dữ liệu