

LẬP TRÌNH PHÂN TÁN THEO CHỦ ĐỀ

RPC

Cao Tuấn Dũng
Trịnh Tuấn Đạt



Nội dung

- 1. Đặt vấn đề
- 2. Lời gọi thủ tục thông thường
- 3. Thực thi RPC
- 4. Các vấn đề trong RPC
- 5. Lập trình với RPC



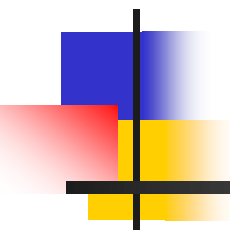
1. Đặt vấn đề

- Khó khăn khi lập trình socket:
 - Tương tác trong Socket là đơn giản:
 - [connect]
 - read/write
 - [disconnect]
 - NHƯNG ... bắt buộc có cơ chế Read/Write
 - Chúng ta thường sử dụng một "Procedure call"
 - Để lập trình phân tán trong suốt với lập trình viên, như lập trình tập trung:
 - I/O không thể được lựa chọn



1. Đặt vấn đề

- Lịch sử ra đời:
 - 1984: Birrell & Nelson: "*Mechanism to call procedures on other machines: Remote Procedure Call*"
 - Mục tiêu: RPC đối với lập trình viên chỉ như lời gọi hàm thông thường



Cách thức các lời gọi thủ tục
thông thường hoạt động?



2. Lời gọi thủ tục thông thường

- Quá trình gọi & trả về chia làm 3 giai đoạn:
 - Truyền tham số
 - Xử lý biến cục bộ
 - Trả về dữ liệu



2. Lời gọi thủ tục thông thường

- Ví dụ:

- Khi viết lệnh: `x = f(a, "test", 5);`
- Compiler phân tích và sinh code để:
 - Đẩy giá trị 5 vào stack
 - Đẩy địa chỉ của string "test" vào stack
 - Đẩy giá trị hiện tại của a vào stack
 - Sinh 1 lời gọi tới hàm f
- Khi biên dịch hàm f, compiler sinh code để:
 - Đẩy các thanh ghi sẽ bị ghi đè vào stack để lưu lại giá trị
 - Điều chỉnh stack, tạo không gian cho biến cục bộ & biến tạm thời
 - Trước khi trả về, để stack lại nguyên dạng như trước khi điều chỉnh, đặt giá trị trả về vào 1 thanh ghi và thực thi câu lệnh return.

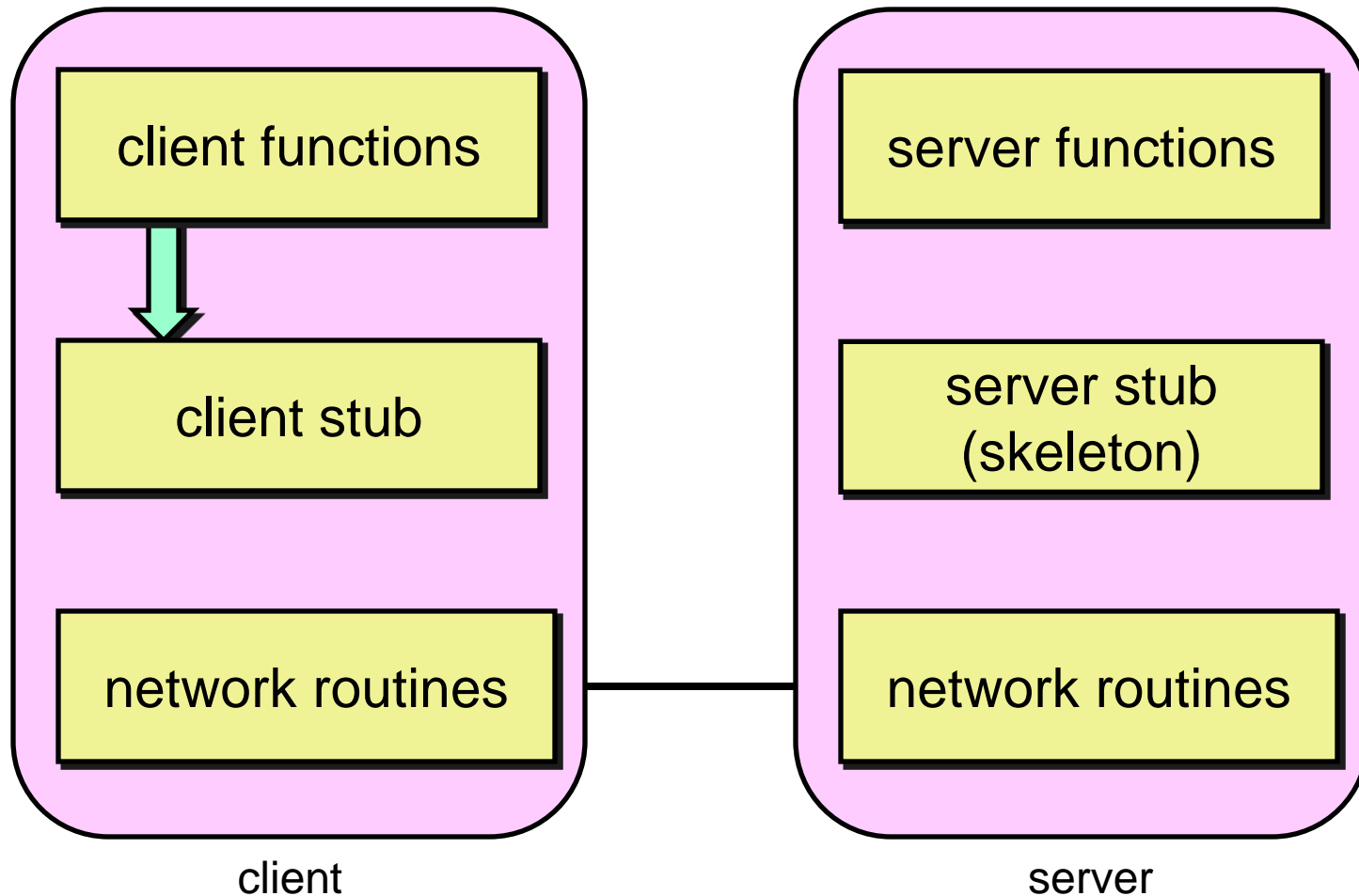


3. Thực thi RPC

- Không có kiến trúc nào trực tiếp hỗ trợ RPC
- Thủ thuật
 - Tạo các hàm stub, để lập trình viên thấy như 1 lời gọi local
 - Hàm stub có giao diện như của hàm ban đầu

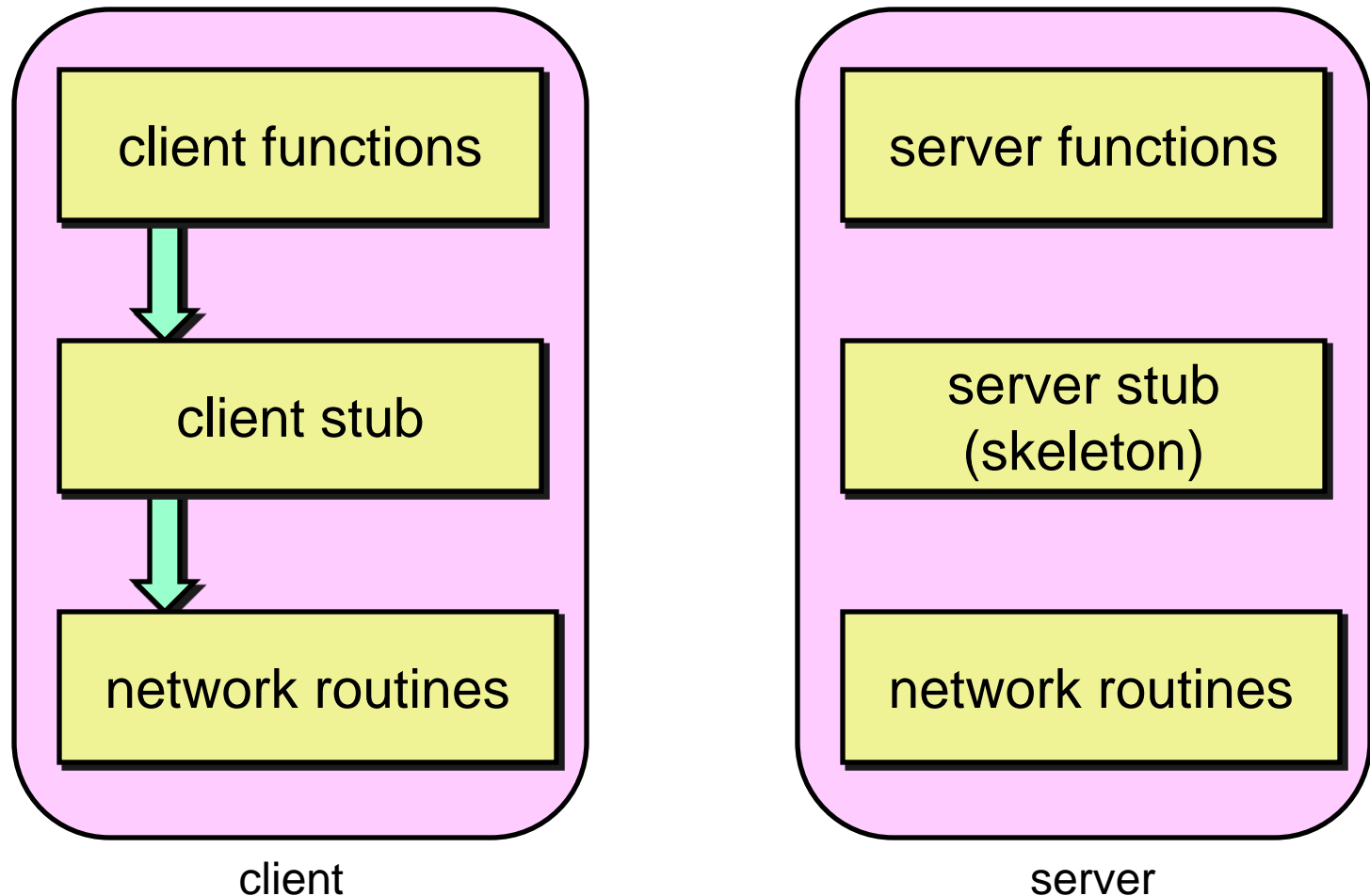
Hàm Stub

- 1. Client gọi hàm stub



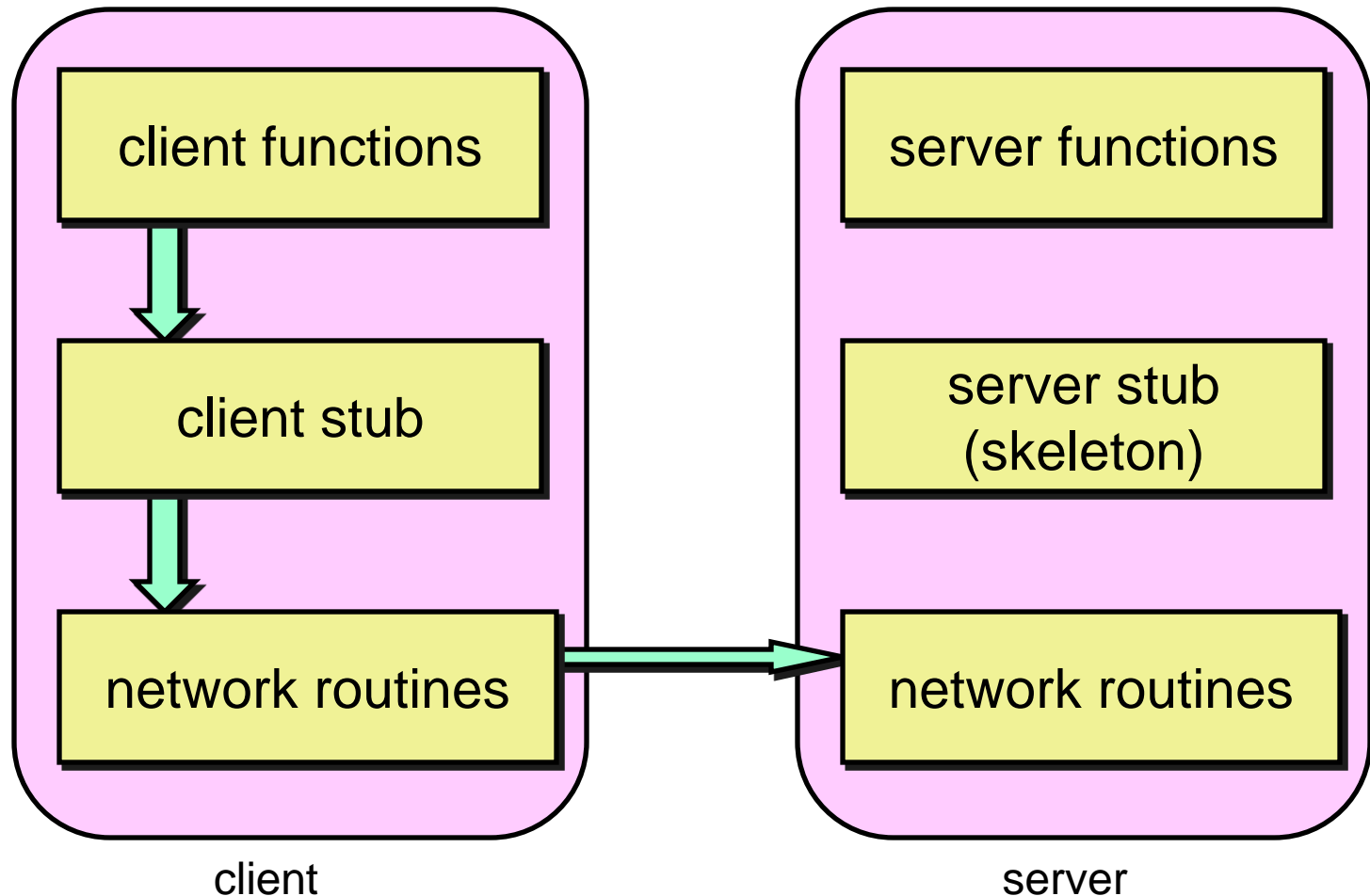
Hàm Stub

- 2. Stub (client) đóng gói tham số vào thông điệp mạng



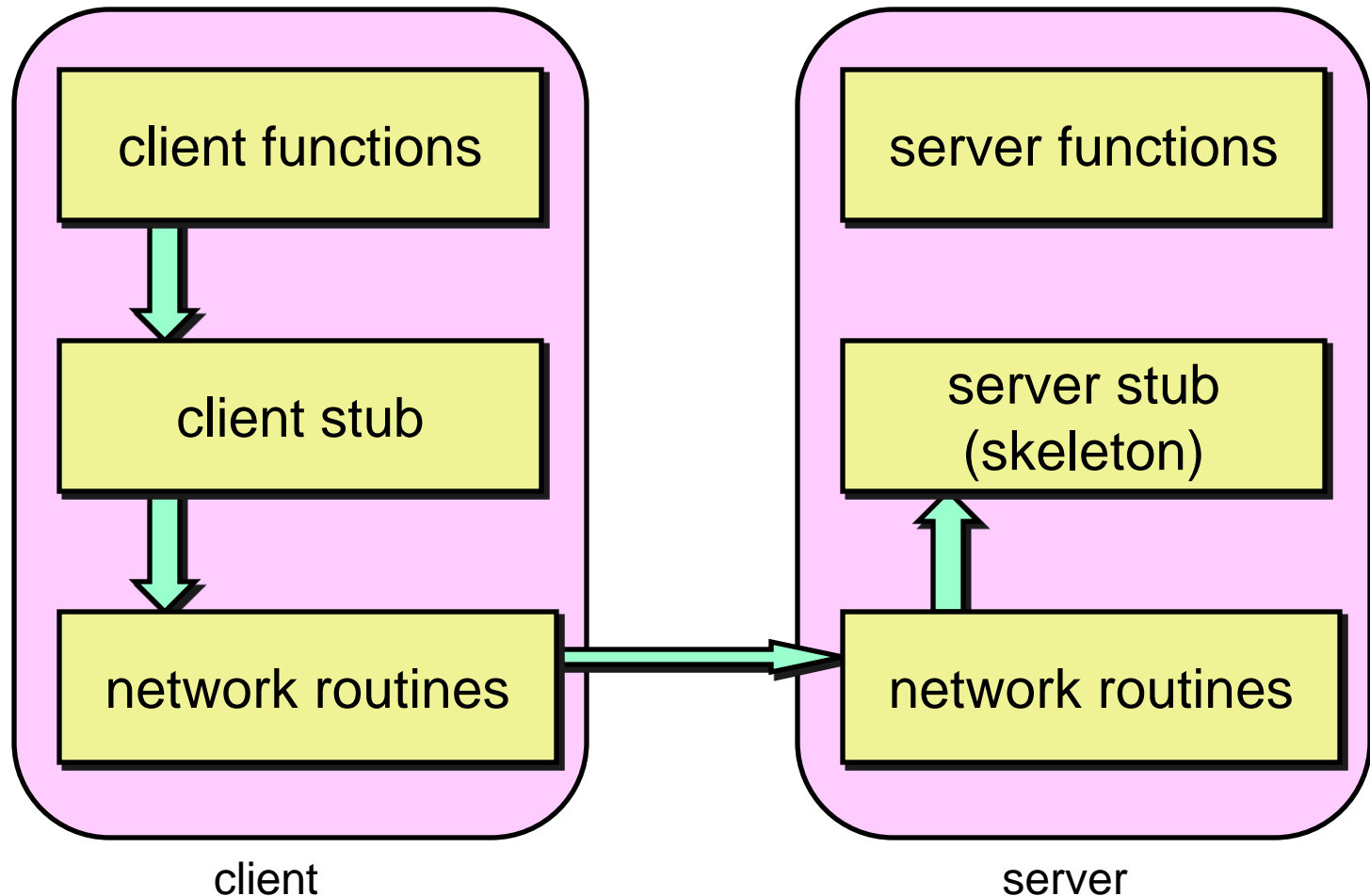
Hàm Stub

- 3. Thông điệp được gửi tới Server



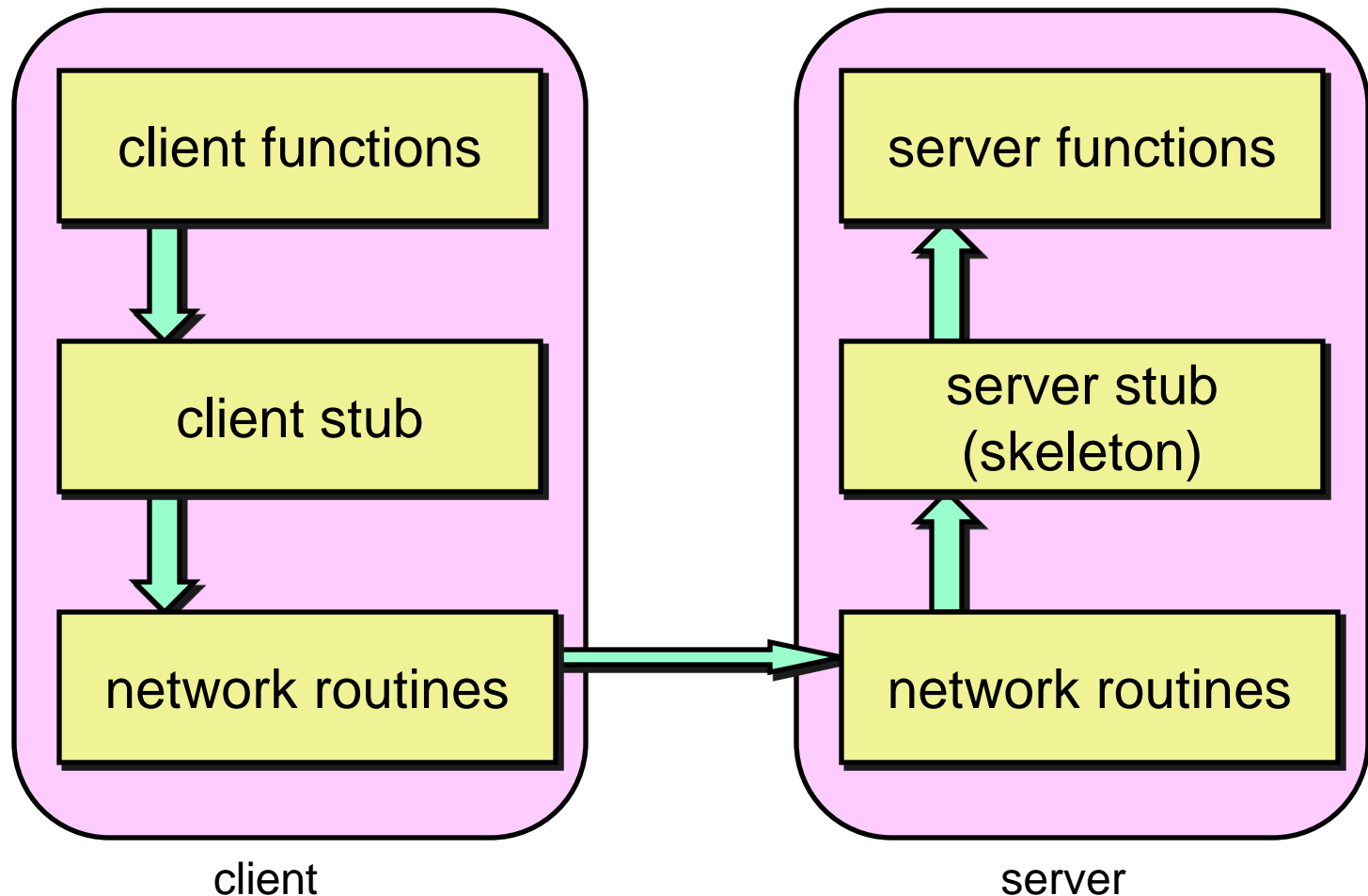
Hàm Stub

- 4. Receive message: send to stub



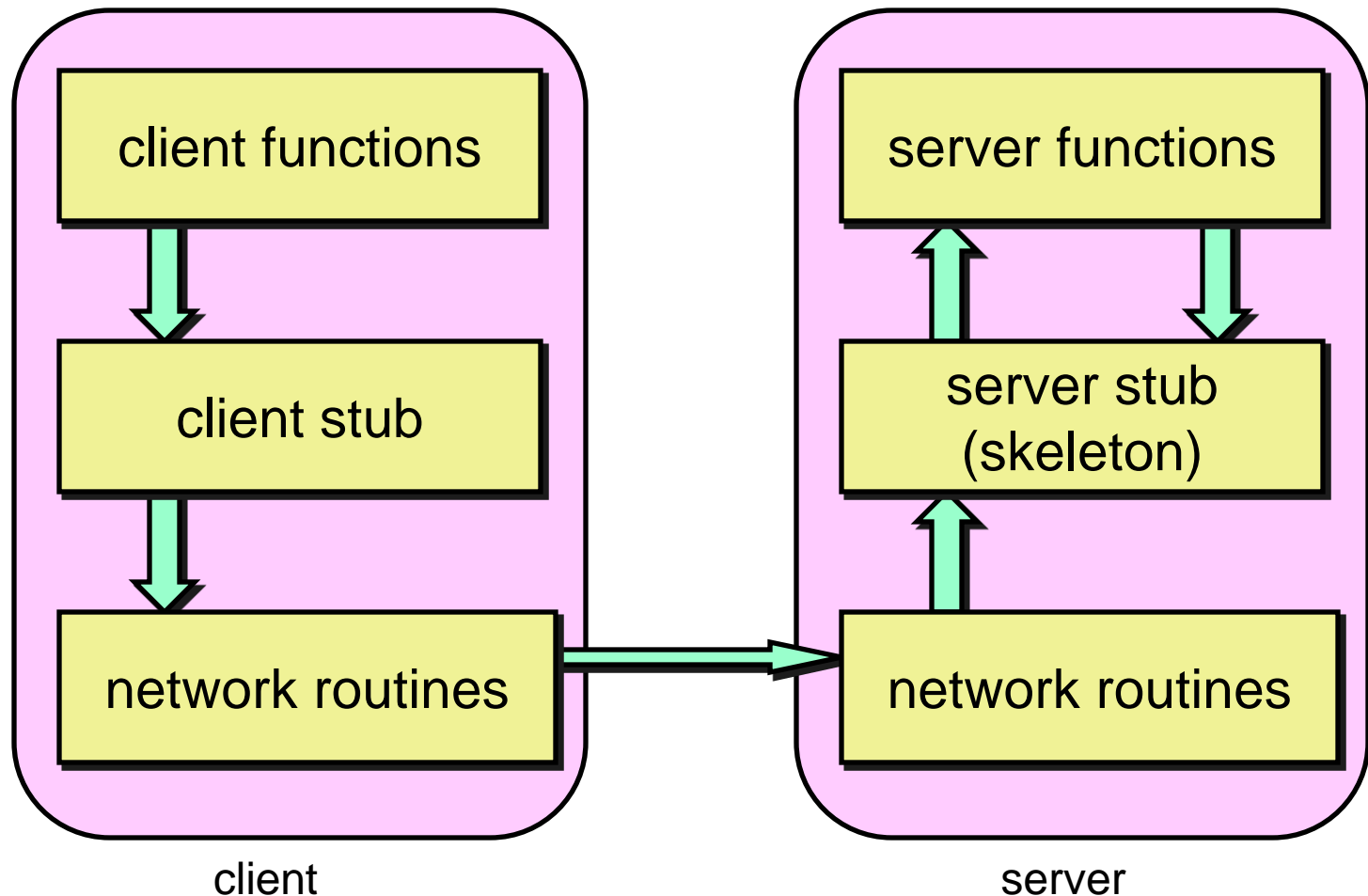
Hàm Stub

- 5. Mở gói lấy tham số, gọi thủ tục server



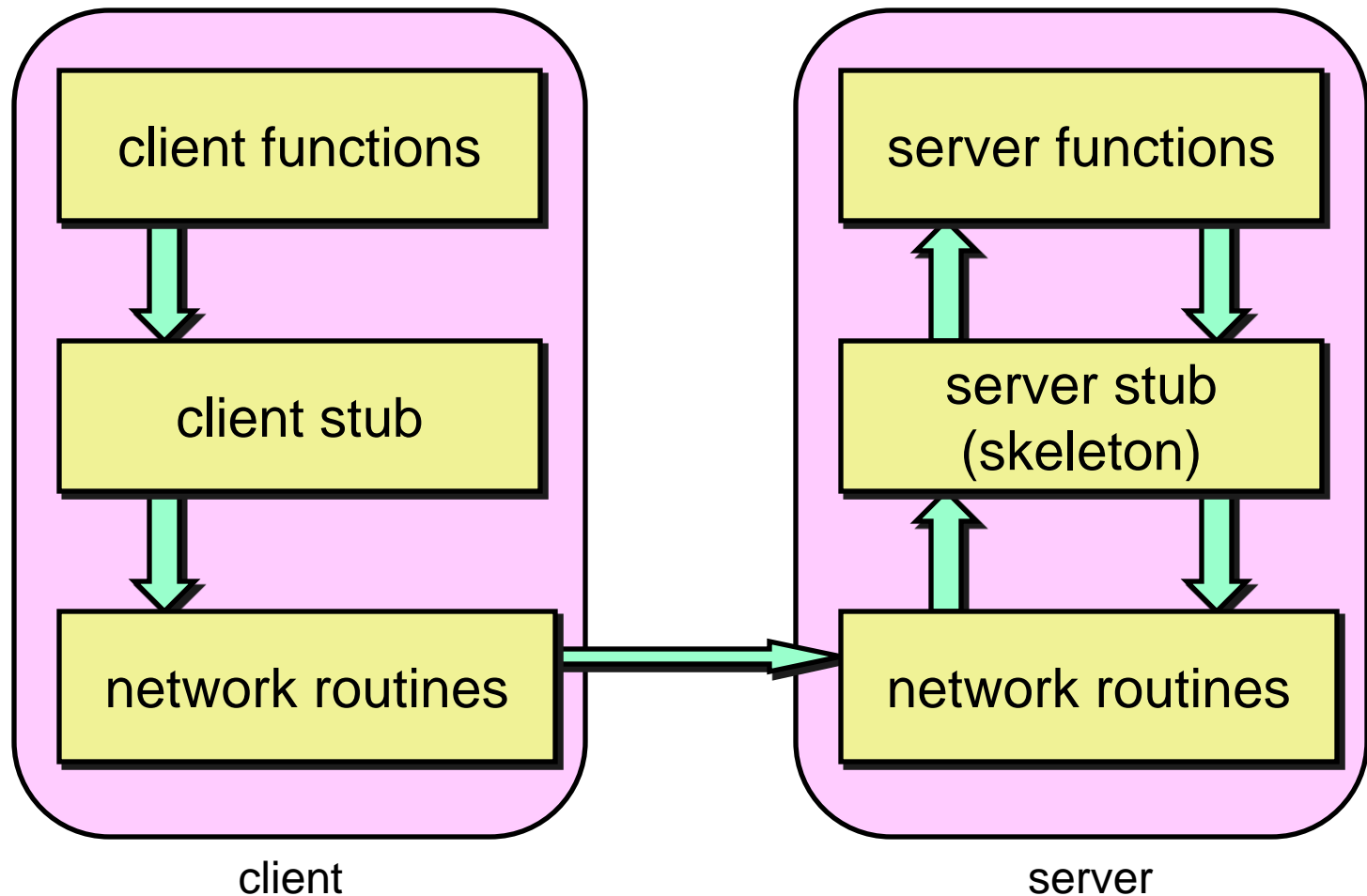
Hàm Stub

- 6. Hàm phía server trả về kết quả



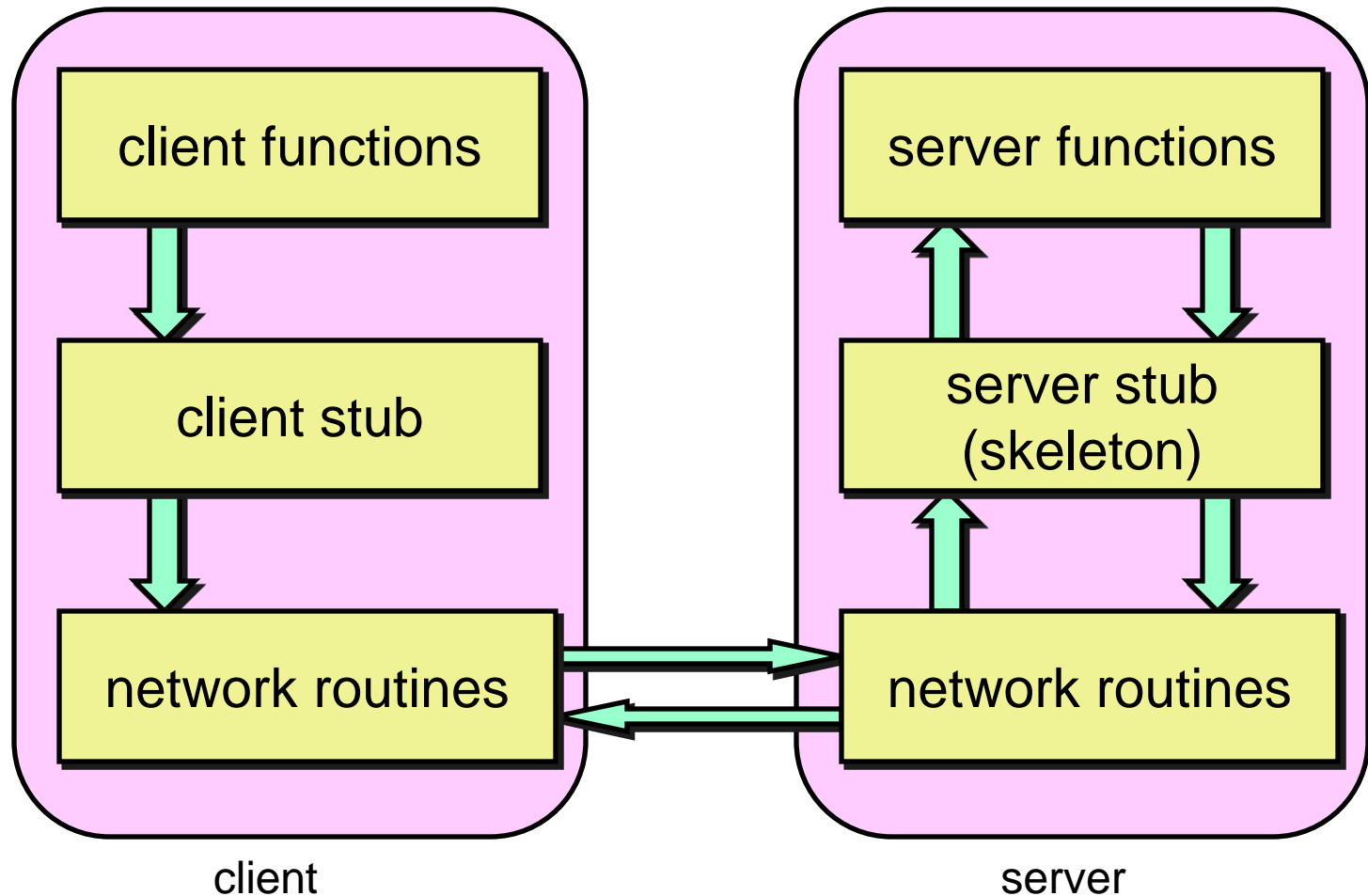
Hàm Stub

- 7. Đóng gói kết quả và gửi thông điệp



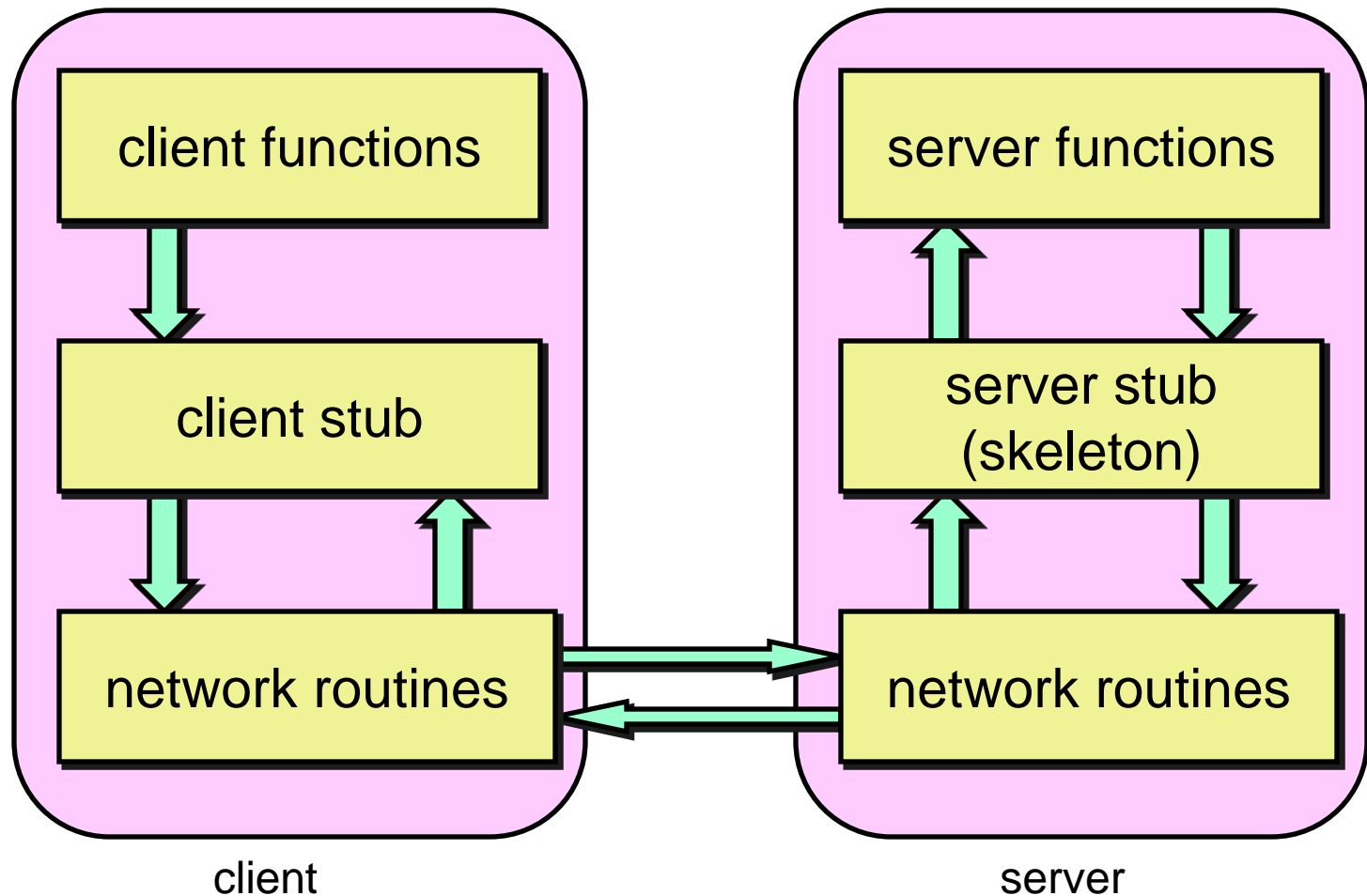
Hàm Stub

- 8. Chuyển thông điệp



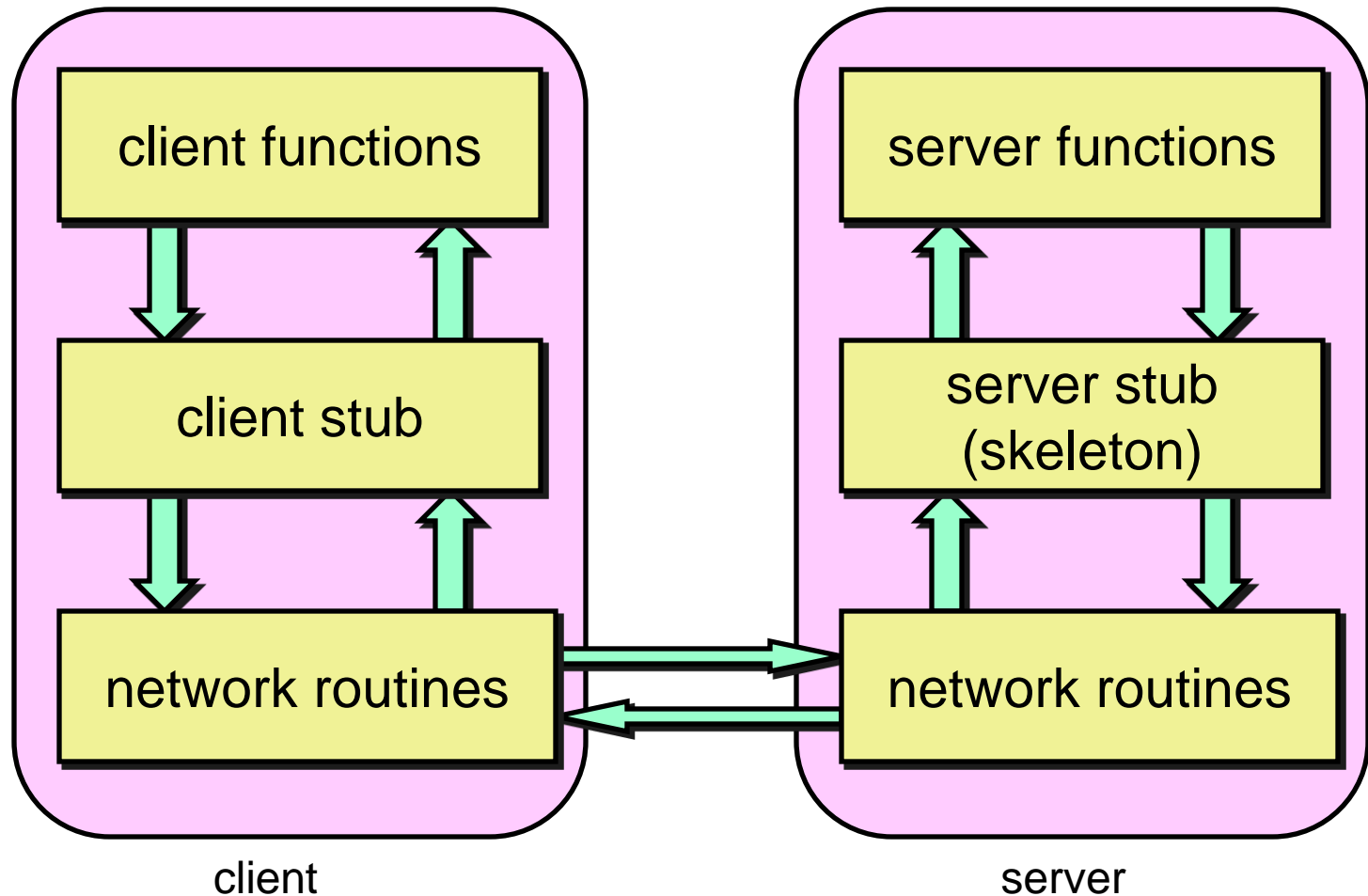
Hàm Stub

- 9. Nhận thông điệp – đưa về stub (client)



Hàm Stub

- 10. Mở gói, trả về kết quả tại hàm client





Lợi ích

- Giao diện : lời gọi hàm
- Viết các ứng dụng được đơn giản hóa
 - RPC ẩn tất cả các đoạn code mạng, truyền thông vào các hàm stub
 - Người lập trình ứng dụng không cần quan tâm chi tiết về:
 - Sockets, số hiệu cổng, thứ tự các bytes



4. Một số vấn đề trong RPC

■ 4.1. Truyền tham số

- Truyền tham trị: đơn giản, chỉ cần copy dữ liệu vào thông điệp (**network message**)
- Truyền tham chiếu: vô nghĩa nếu không có **bộ nhớ chia sẻ**



4. Các vấn đề trong RPC

- 4.2. Biểu diễn dữ liệu:
 - Hệ thống địa phương:
 - Không có “**Vấn đề về xung đột, không nhất quán**”
 - Từ xa
 - Khác biệt về thứ tự byte (đầu to, đầu nhỏ, ...)
 - Khác biệt về kích thước dữ liệu: integer, long, ...
 - Khác biệt về biểu diễn dấu phẩy động cho số thực
 - Khác biệt về tập ký tự



4.2. Biểu diễn dữ liệu:

- IP (headers) bắt buộc sử dụng kiểu **big endian (đầu to)** cho các giá trị 16 và 32 bit
 - Sparc, 680x0, MIPS, PowerPC G5: Big Endian
 - x86/Pentiums sử dụng kiểu Little endian

```
main() {  
    unsigned int n;  
    char *a = (char *)&n;
```

```
    n = 0x11223344;  
    printf("%02x, %02x, %02x, %02x\n",  
           a[0], a[1], a[2], a[3]);
```

```
}
```

Output on a Pentium:
44, 33, 22, 11

Output on a PowerPC:
11, 22, 33, 44



4.2. Biểu diễn dữ liệu:

- Cần có encoding chuẩn tắc để cho phép truyền thông giữa các hệ thống dị bộ
 - Ví dụ:
 - Sun's RPC uses XDR (eXternal Data Representation)
 - ASN.1 (ISO Abstract Syntax Notation)
- Định kiểu ẩn
 - Chỉ truyền đi giá trị, không truyền kiểu dữ liệu hoặc **thông tin về tham số**
 - VD: Sun XDR
- Định kiểu tường minh
 - Mỗi giá trị được kèm với kiểu dữ liệu khi truyền đi
 - Ví dụ: ISO's ASN.1, XML



4. Các vấn đề trong RPC

- 4.3. Giao thức truyền tải
 - Một số thực thi của RPC sử dụng duy nhất 1 giao thức (đại đa số: TCP)
 - Hầu hết hỗ trợ nhiều giao thức
 - Cho phép Lập trình viên lựa chọn



4. Các vấn đề trong RPC

- 4.4. Khi có lỗi
 - Lời gọi cục bộ
 - Nếu bị lỗi coredump, toàn bộ tiến trình sẽ hỏng.
 - Với RPC, sẽ có nhiều khả năng gặp lỗi hơn
 - Ứng dụng phải tự lường trước các lỗi của RPC
 - Các lỗi có thể:
 - 1. Client không thể định vị được server
 - 2. Client request bị lỗi
 - 3. Hỏng Server
 - 4. Trả lời từ Server bị lỗi
 - 5. Hỏng Client



4. Các vấn đề trong RPC

■ 4.4. Khi có lỗi:

■ Ngữ nghĩa của lời gọi thủ tục từ xa:

- Lời gọi cục bộ: Chỉ 1 lần
- Một RPC có thể được gọi
 - 0 lần: server bị crash hoặc tiến trình server chết trước khi thực hiện code phía server
 - 1 lần: mọi giai đoạn OK
 - 1 hoặc nhiều: quá độ trễ, hoặc mất phản hồi từ server → truyền lại



4.4. Khi có lỗi- Ngữ nghĩa của RPC

- Hầu hết các hệ thống RPC hỗ trợ hoặc:
 - ít nhất một (At-least-once-semantics)
 - hoặc nhiều nhất một (At-most)
- Trong 1 ứng dụng, phải nhận biết
 - idempotent functions: có thể chạy 1 số lần bất kỳ mà không gây hại
 - Non-idempotent functions:



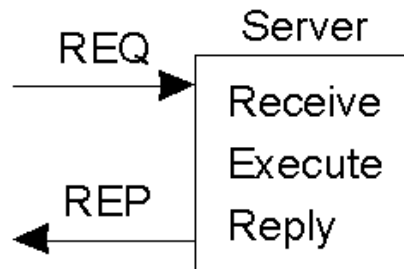
4. Các vấn đề trong RPC-Khắc phục lỗi

- 1. Client không định vị được server:
 - Giải pháp đơn giản: thông báo lại cho ứng dụng client → Mất tính trong suốt
- 2. Client mất request:
 - Giải pháp: gửi lại message
 - Sử dụng message ID, phân biệt giữa các message

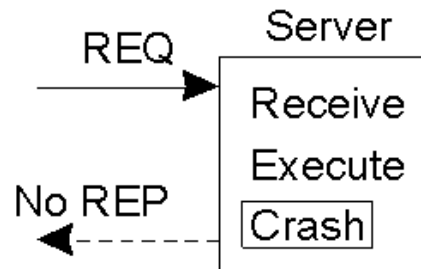
4. Các vấn đề trong RPC-Khắc phục lỗi

■ 3. Server crashes:

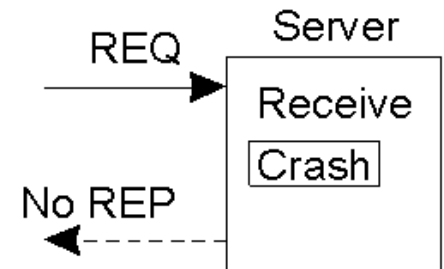
- Khó xử lý vì không biết server thực hiện đến đâu



(a)



(b)



(c)

- Cần quyết định cần kiểu server nào :

- Ít nhất một
- Nhiều nhất một



4. Các vấn đề trong RPC-Khắc phục lỗi

■ 4. Mất các phản hồi

- Khó phát hiện, vì có thể nguyên nhân là do server chậm. Ta không thể biết server có phải đang thực hiện không
- Giải pháp: Không có giải pháp tổng quát
 - Cố gắng tạo ra các thao tác **idempotent**



4. Các vấn đề trong RPC-Khắc phục lỗi

■ 5. Client crashes

- → Server thực hiện công việc và tài nguyên vô nghĩa (gọi là **orphan computation**).
- Giải pháp:
 - Orphan bị hủy bởi Client khi Client hoạt động trở lại
 - Chi phí cao, và ... có thể không thực hiện được
 - Client broadcast số epoch mới khi khôi phục trở lại → server sẽ hủy orphan
 - Yêu cầu các tính toán phải thực hiện trong T đơn vị thời gian, nếu không, sẽ bị hủy



4. Các vấn đề trong RPC

- 4.5. Một số vấn đề khác
 - Hiệu năng:
 - Security:
 - Message truyền đi là **visible** trên network
 - Xác thực Client?
 - Xác thực Server?



5. Lập trình với RPC

- Hỗ trợ ngôn ngữ:

- Hầu hết các ngôn ngữ lập trình (C, C++, Java, ...) không có khái niệm về RPC
- Bộ biên dịch ngôn ngữ không sinh ra client và server stubs

- Giải pháp truyền thống:

- Sử dụng các bộ biên dịch độc lập để sinh ra các stubs (pre-compiler)
 - VD: rmic trong jdk.



Mô hình truyền thống

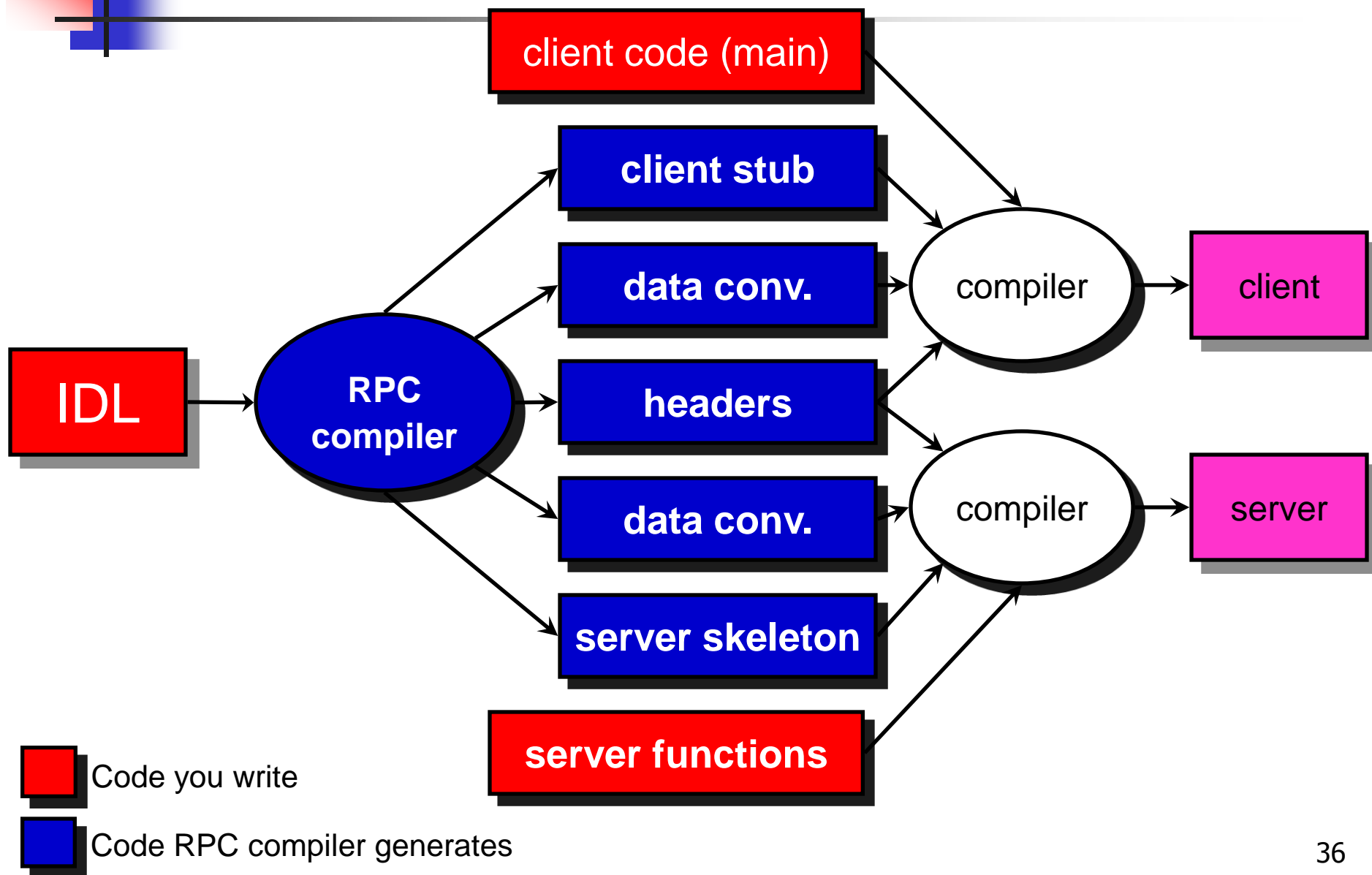
- Server định nghĩa giao diện dịch vụ sử dụng IDL
 - đặc tả tên, tham số, kiểu trả về cho các thủ tục
- Bộ biên dịch stub đọc khai báo IDL và sinh ra các cặp hàm stub tương ứng
 - *Server-side* và *client-side*



Interface Definition Language-IDL

- Bộ biên dịch stub có thể sử dụng IDL để sinh ra các client và server stubs:
 - Mã nguồn liên quan đến Marshaling
 - Mã nguồn liên quan đến Unmarshaling
 - Các thủ tục truyền tin trên mạng
 - Tương thích với giao diện định nghĩa
- Tương tự với **function prototypes**

RPC Compiler





Viết chương trình

- Code ở client cần có điều chỉnh:
 - Khởi tạo chuẩn bị cho RPC
 - Dự đoán, xử lý lỗi của RPC
- Server functions:
 - Thông thường không cần hoặc ít có điều chỉnh