

25 YEARS ANNIVERSARY
SICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

CÁC HỆ THỐNG PHÂN TÁN VÀ ỨNG DỤNG

Chương 2: Tiến trình và trao đổi thông tin

Nội dung

1. Tiến trình và luồng
2. Khái niệm về trao đổi thông tin
3. Lời gọi thủ tục từ xa
4. Trao đổi thông tin hướng thông điệp
5. Trao đổi thông tin hướng dòng



HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

1. Tiến trình và luồng

- 1.1. Khái niệm
- 1.2. Luồng trong hệ thống tập trung
- 1.3. Luồng trong hệ thống phân tán

1.1. Tiến trình và luồng

- Tiến trình
 - Chương trình đang hoạt động
 - Tài nguyên:
 - Virtual Processor
 - Virtual Memory
 - Trong suốt tương tranh
 - Quá trình tạo 1 tiến trình
 - Chuyển ngữ cảnh giữa các tiến trình

Luồng

- Là một luồng thực thi của tiến trình.
- Tiến trình có nhiều luồng thực thi → Tiến trình đa luồng
- Các luồng của tiến trình dùng môi trường thực hiện chung của tiến trình: trạng thái của CPU
- Trao đổi thông tin giữa các luồng thông qua các biến chia sẻ
- An toàn và hợp lý của tương tác luồng do lập trình viên quyết định
- Luồng=> hiệu năng+chi phí lập trình

1.2. Đa tiến trình và đa luồng trong các hệ thống tập trung

- Lợi ích của xử lý song song với xử lý tuần tự
- Đa tiến trình vs. Đa luồng.
 - Chi phí lập trình
 - Chi phí chuyển ngữ cảnh
 - Lời gọi hệ thống dừng (blocking system calls)

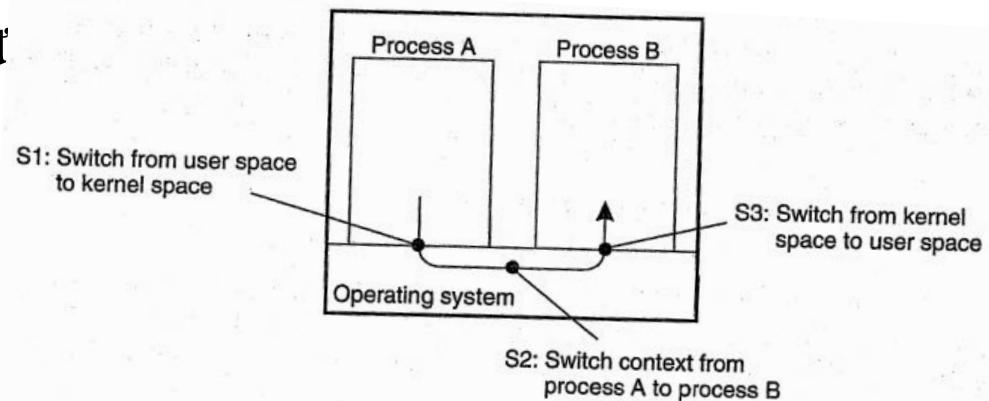


Figure 3-1. Context switching as the result of IPC.

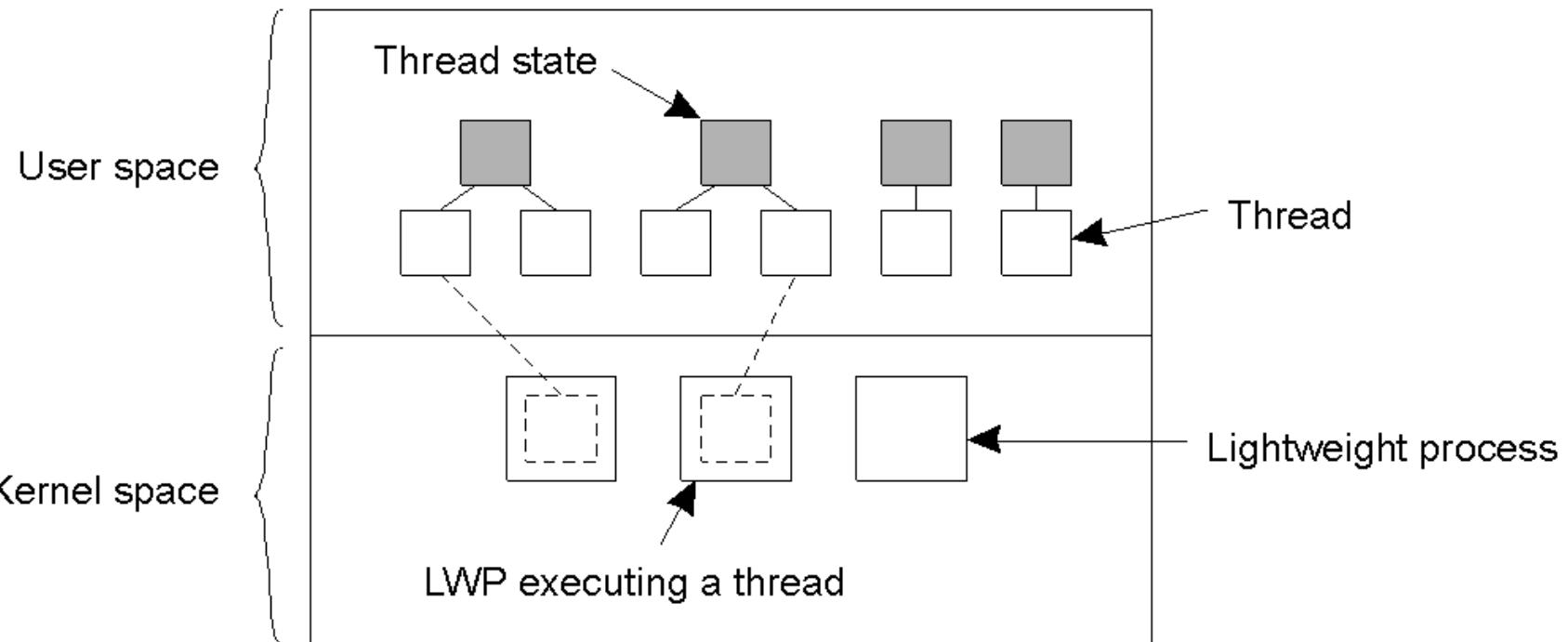
Cài đặt luồng

- Được quản lý bởi gói luồng (Thread package)
 - Khởi tạo luồng (1)
 - Giải phóng luồng (2)
 - Đồng bộ các luồng (3)
- (1), (2), (3) có thể thực hiện dưới chế độ NSD hoặc nhân
 - Chế độ NSD: có thao tác vào ra-> block cả tiến trình
 - Chế độ nhân: Tốn kém

| | Creation time | Synchronization Time using semaphore |
|-------------|---------------|--|
| User thread | 52 | 66 |
| LWP | 350 | 390 |
| Process | 1700 | 200 |

Cơ chế các tiến trình nhẹ trong Linux

- Combining kernel-level lightweight processes and user-level threads.



Cơ chế các tiến trình nhẹ trong Linux (2)

- Luồng ở mức user được xây dựng theo chuẩn POSIX (Portable Operating System Interface for uniX)
- Chạy ở 2 không gian thực thi phân biệt:
 - User space: sử dụng thư viện pthread
 - Kernel: các LWPs
- Ánh xạ 1-1 từ mỗi thread và 1 LWP
- Thay vì dùng fork(), LINUX dùng clone().

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void * function1(void *arg)
{
    pthread_t tid(pthread_self());
    printf("In thread %u and process %u\n",tid,getpid());
}

void * function2(void *arg)
{
    pthread_t tid(pthread_self());
    printf("In thread %u and process %u\n",tid,getpid());
}

int main()
{
    void *status;
    pthread_t tid1,tid2;
    pthread_attr_t attr;

    if(pthread_create(&tid1,NULL,function1,NULL)) {
        perror("Failure");
        exit(1);
    }

    if(pthread_create(&tid2,NULL,function2,NULL)) {
        perror("Failure");
        exit(2);
    }

    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("In main thread %u and process %u\n",pthread_self(),getpid());
}
```



Quản lý ID

```
In thread 3086625680 and process 5480
In thread 3076135824 and process 5480
In main thread 3086628544 and process 5480
```

1.3. Tiến trình và Luồng trong các hệ thống phân tán

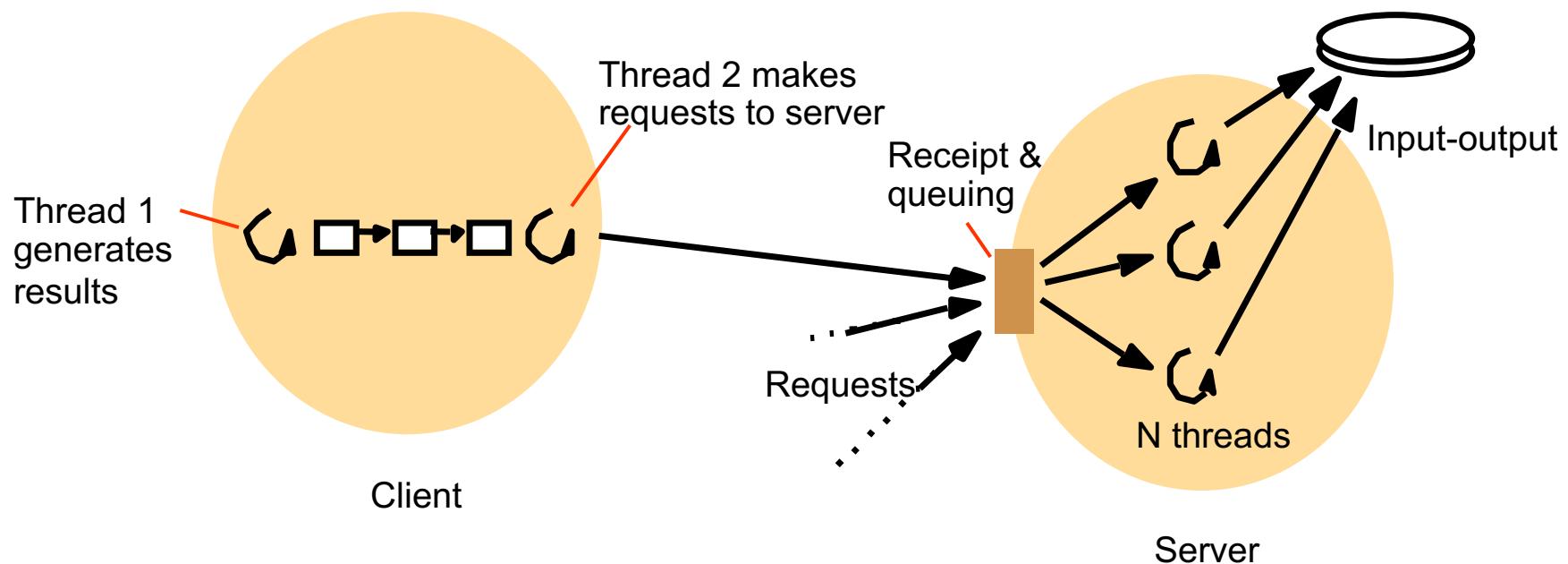
□ Server đơn luồng

- Chỉ xử lý được một yêu cầu tại một thời điểm
- Các yêu cầu có thể được xử lý tuần tự
- Các yêu cầu có thể được xử lý bởi các tiến trình khác nhau
- Không đảm bảo tính trong suốt



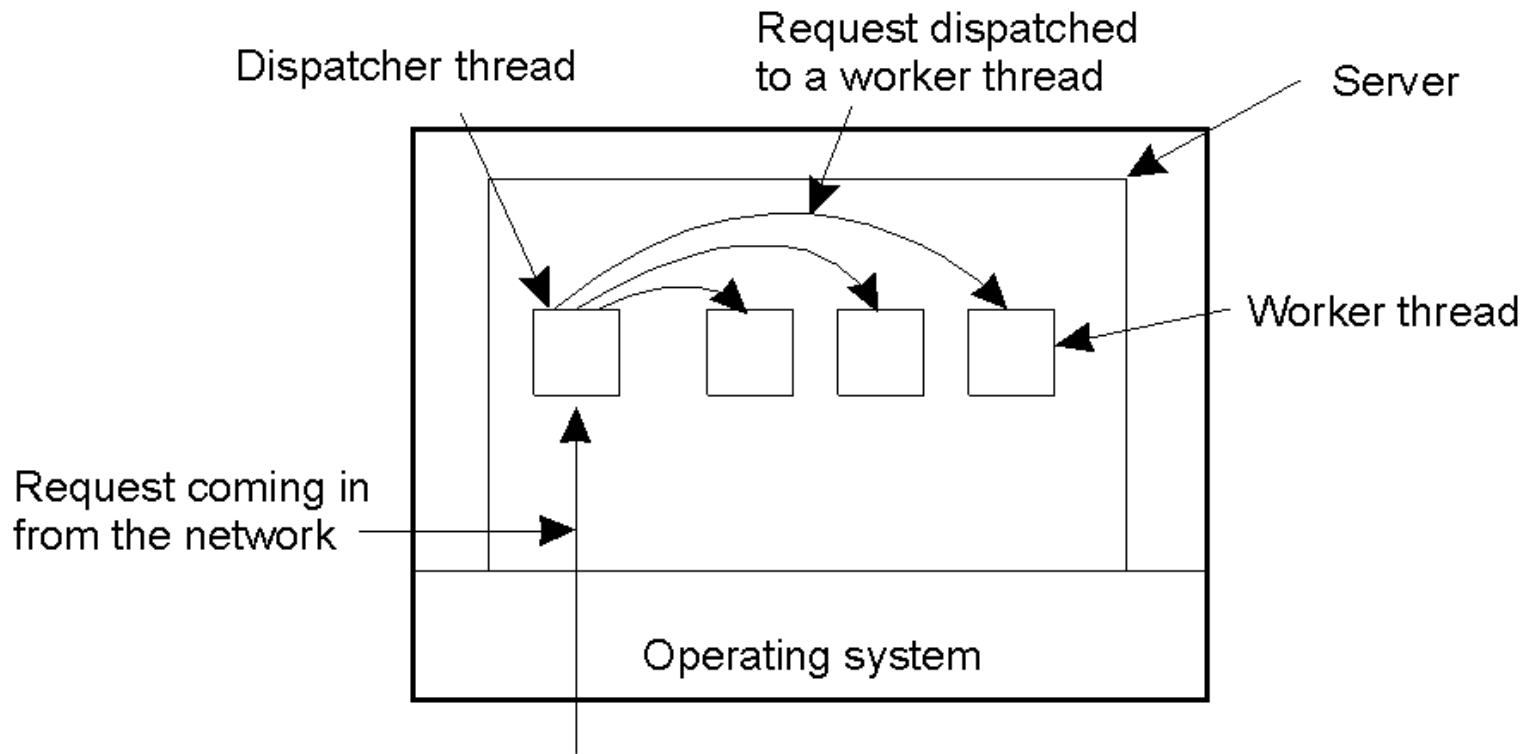
Server đa luồng

Client và server đa luồng

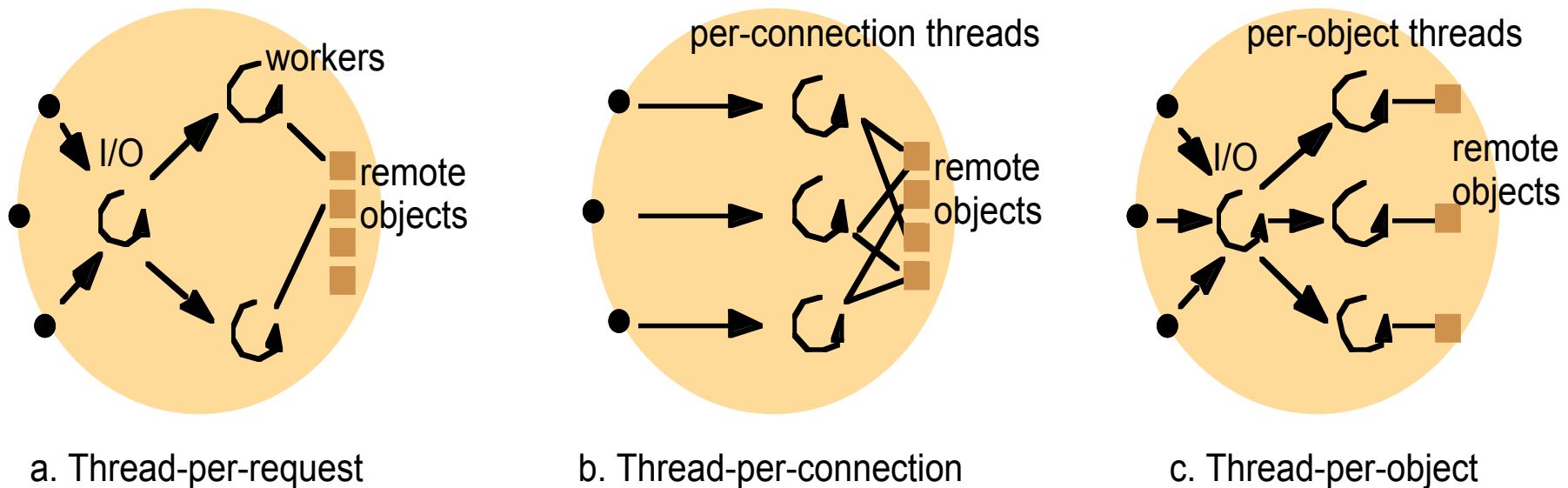


1.4. Server trong hệ phân tán

□ Mô hình server dispatcher



Server đa luồng



a. Thread-per-request

b. Thread-per-connection

c. Thread-per-object

Mô hình máy trạng thái hữu hạn

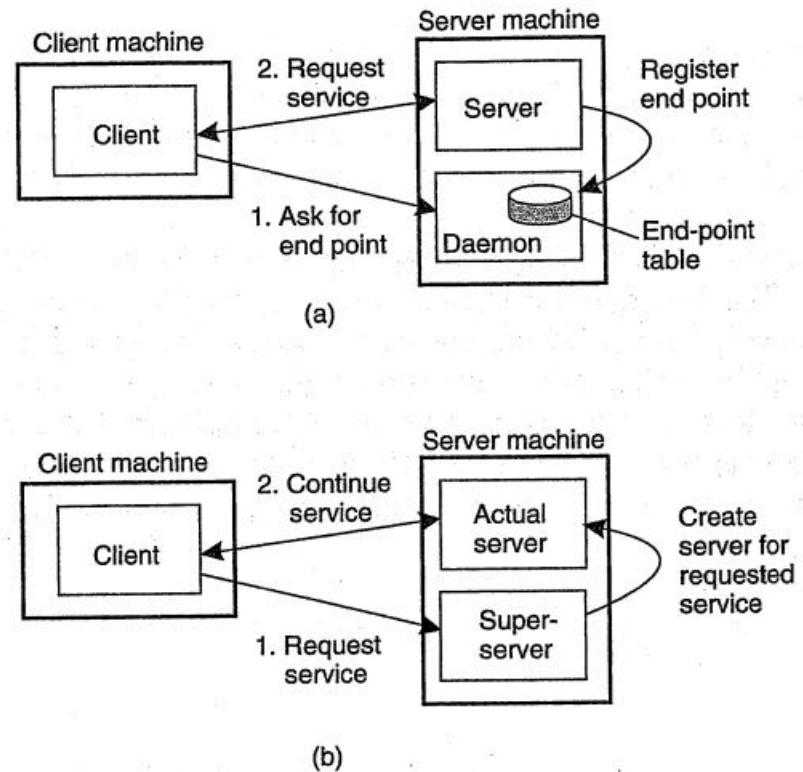
- Các yêu cầu từ client và xử lý được sắp hàng
- Tại một thời điểm server thực hiện thao tác trong hàng
- Không cần đa luồng
- Các lời gọi xử lý là các lời gọi “không dừng”
- VD: Node.js
 - ▣ Bất đồng bộ và hướng sự kiện
 - ▣ Đơn luồng nhưng khả năng co giãn cao

So sánh

| Mô hình | Đặc điểm |
|------------------------|--|
| Đa luồng | Song song, lời gọi hệ thống dừng |
| Đơn luồng | Không song song, lời gọi hệ thống dừng |
| Máy trạng thái hữu hạn | Song song, lời gọi hệ thống không dừng |

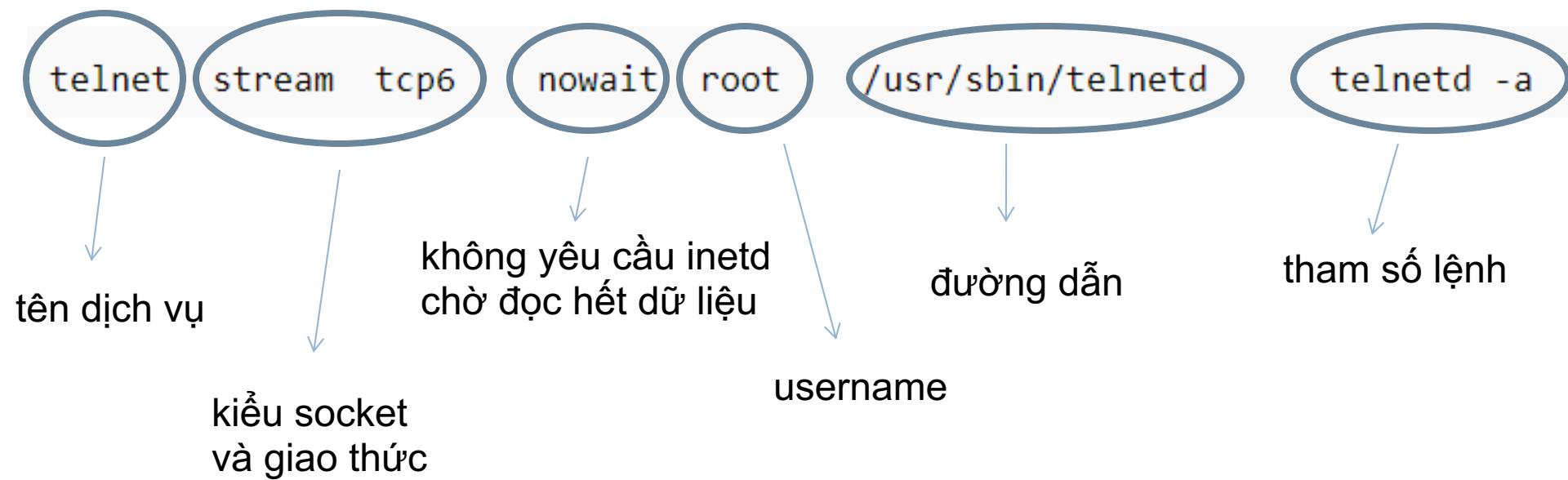
Các vấn đề thiết kế server

- Tổ chức server
 - Server lặp
 - Server đồng thời
- Vấn đề xác định server:
 - End-point (port)
 - Deamon
 - Superserver
- Vấn đề ngắt server
- Stateless & stateful server



Inetd

- Cấu hình của Inetd được lưu trong */etc/inetd.conf*



Xây dựng dịch vụ cho inetd

- Viết 1 chương trình errorLogger.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    const char *fn = argv[1];
    FILE *fp = fopen(fn, "a+");

    if(fp == NULL)
        exit(EXIT_FAILURE);

    char str[4096];
    //inetd passes its information to us in stdin.
    while(fgets(str, sizeof(str), stdin)) {
        fputs(str, fp);
        fflush(fp);
    }
    fclose(fp);
    return 0;
}
```

Xây dựng dịch vụ cho inetd

- Điene thêm vào /etc/services

errorLogger 9999/udp

- Điene thêm vào /etc/inetd.conf

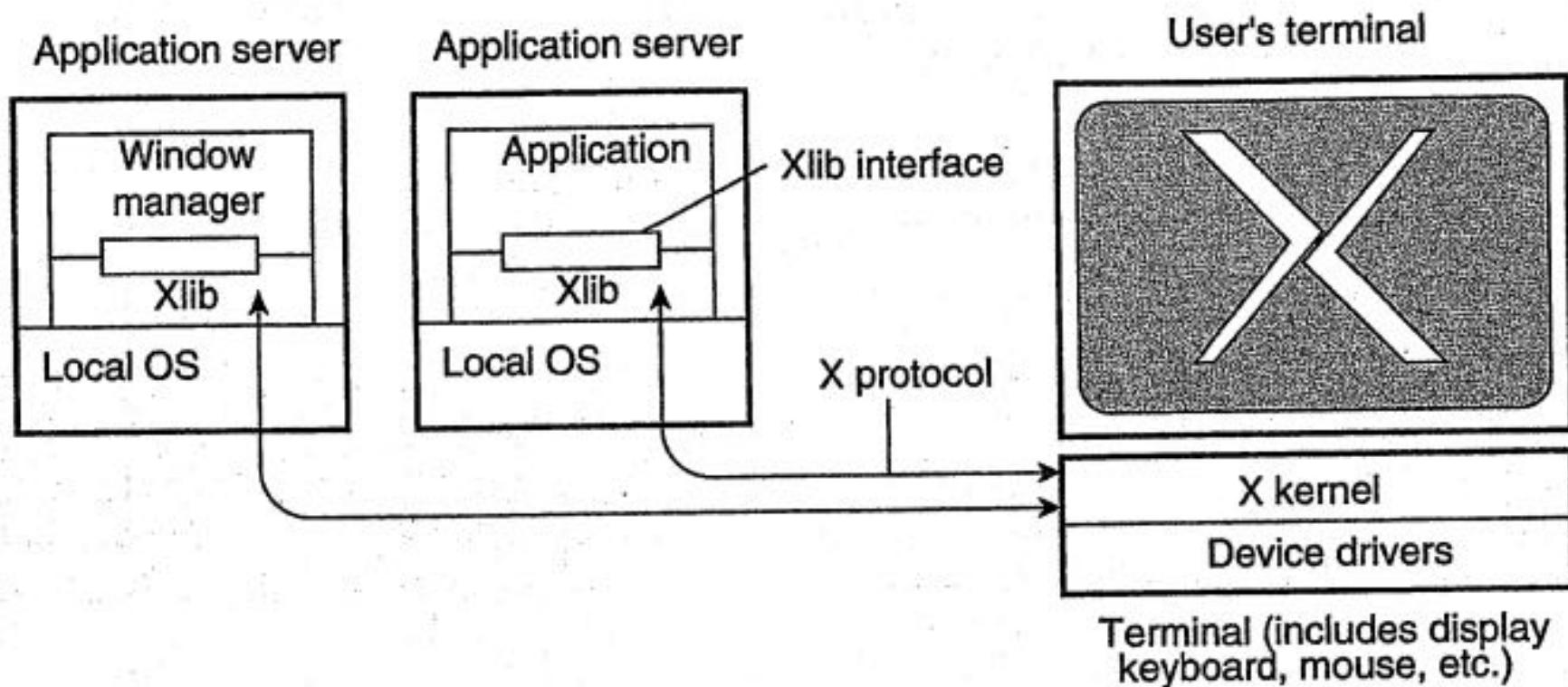
errorLogger dgram udp wait root
/usr/local/bin/errlogd errlogd
/tmp/logfile.txt

1.5. Client trong Hệ phân tán

❑ Cần Client đa luồng

- ❑ Tách biệt giao diện người sử dụng và xử lý
- ❑ Giải quyết vấn đề các thao tác chờ đợi lẫn nhau
- ❑ Tăng tốc độ khi làm việc với nhiều server khác nhau
- ❑ Ví dụ: Tải trang web

Giải pháp cho Thin-client: Hệ thống X Window



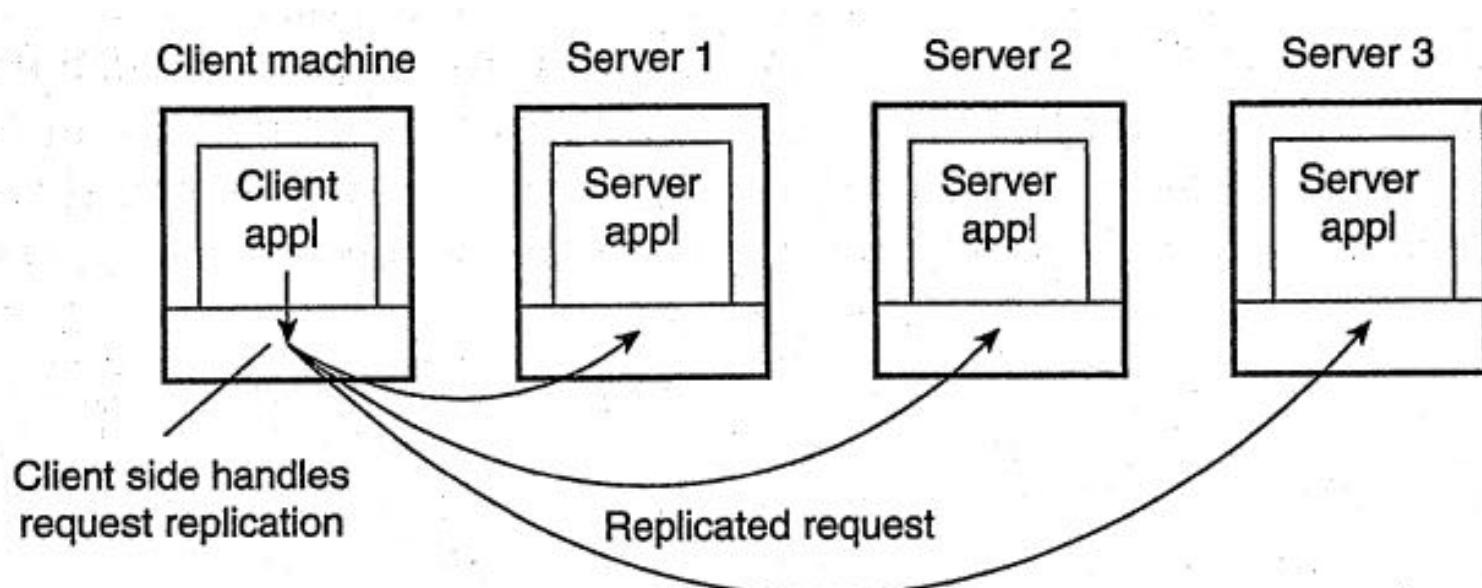
Thin-client Network Computing

- Phân biệt 2 khái niệm: X-client và X-server
- Các ứng dụng điều khiển màn hình bằng các lệnh chuyên dụng (cung ứng bởi X).
- Tách biệt về logic ứng dụng và các câu lệnh giao tiếp người dùng => Không thực hiện được
- Giải pháp: Nén thông điệp

Phần mềm client phục vụ trong suốt phân tán

❖ Trong suốt phân tán:

- ❖ Trong suốt truy cập
- ❖ Trong suốt di trú
- ❖ Trong suốt sao lưu
- ❖ Trong suốt che giấu lỗi



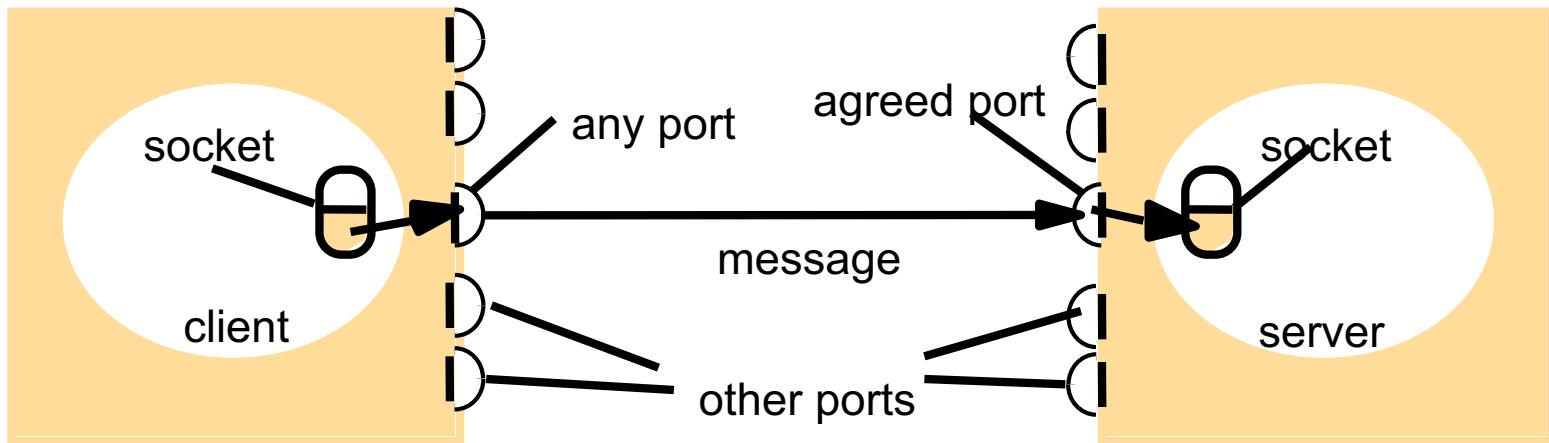
2. Trao đổi thông tin giữa các tiến trình

- 2.1. Khái niệm chung
- 2.2. Trao đổi thông tin với UDP
- 2.3. Trao đổi thông tin với TCP

2.1. Khái niệm

- Giao thức
 - ▣ Cấu trúc thông điệp
 - ▣ Kích cỡ thông điệp
 - ▣ Thứ tự gửi thông điệp
 - ▣ Cơ chế phát hiện thông điệp hỏng hay bị mất
 - ▣ V.v...
- Phân tầng
- Các loại giao thức
 - ▣ Hướng kết nối, không hướng kết nối, tin cậy, không tin cậy
- Các vấn đề của giao thức
 - ▣ Send, receive primitives
 - ▣ Đồng bộ/không đồng bộ, dùng, không dùng

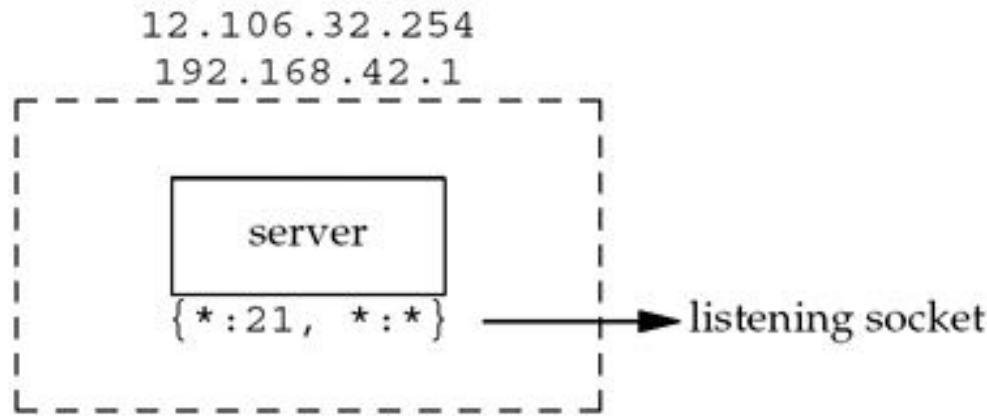
Socket-port



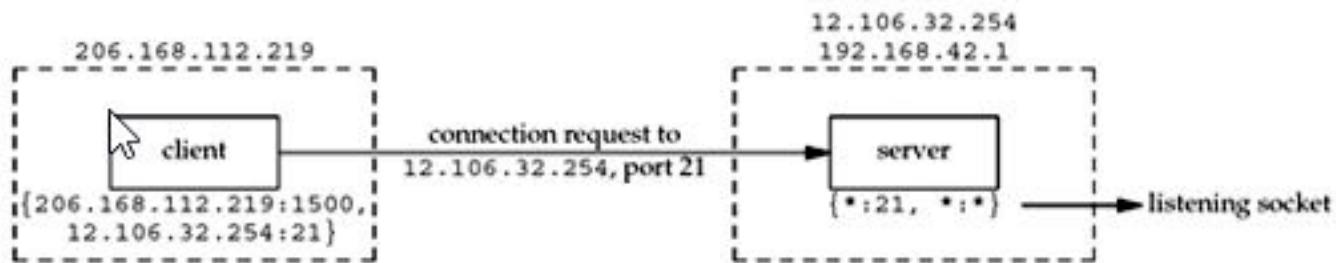
Internet address = 138.37.94.248

Internet address = 138.37.88.249

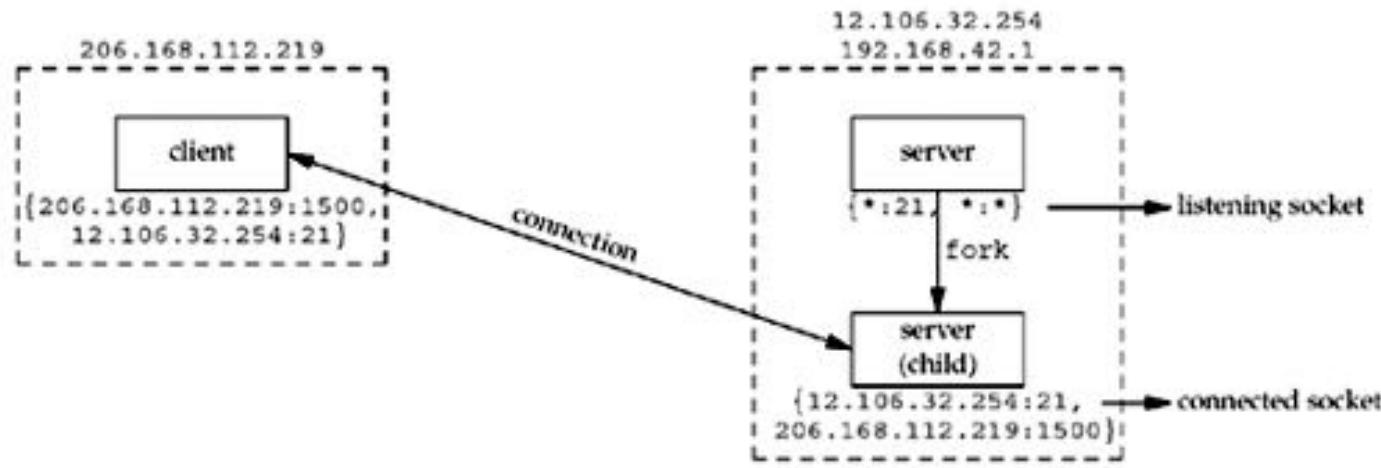
TCP Port Numbers and Concurrent Servers (1)



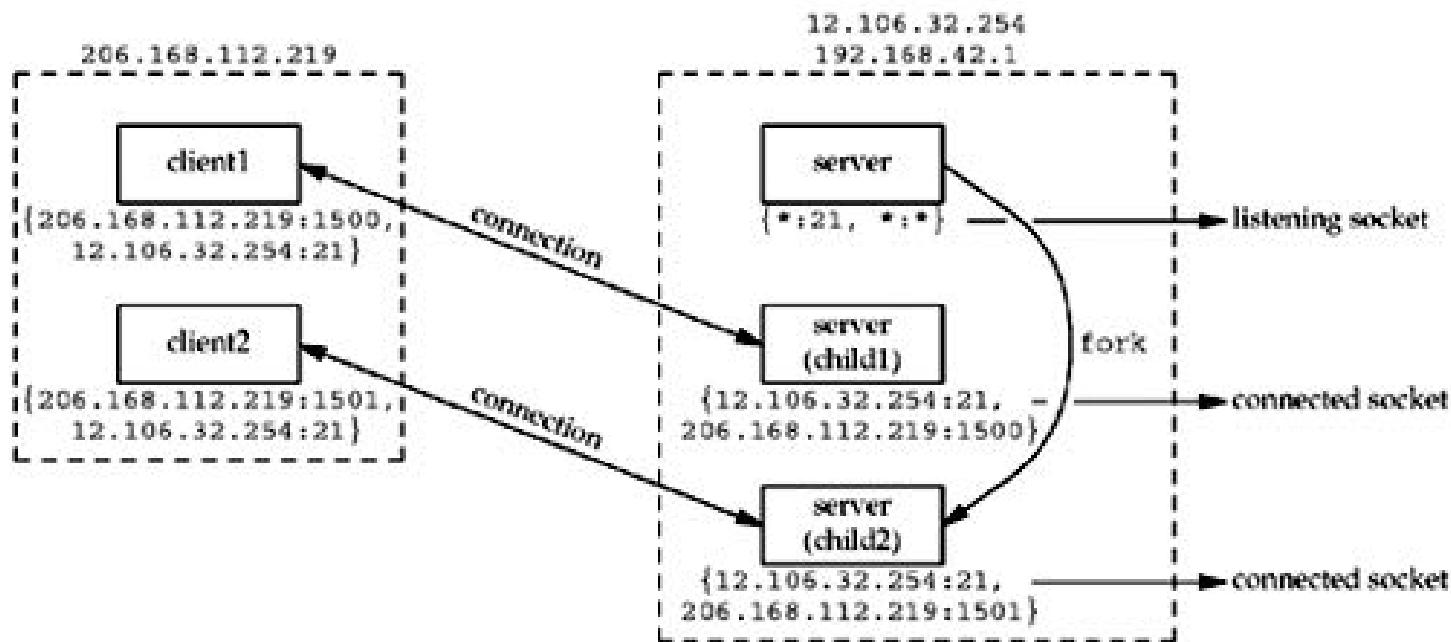
TCP Port Numbers and Concurrent Servers (2)



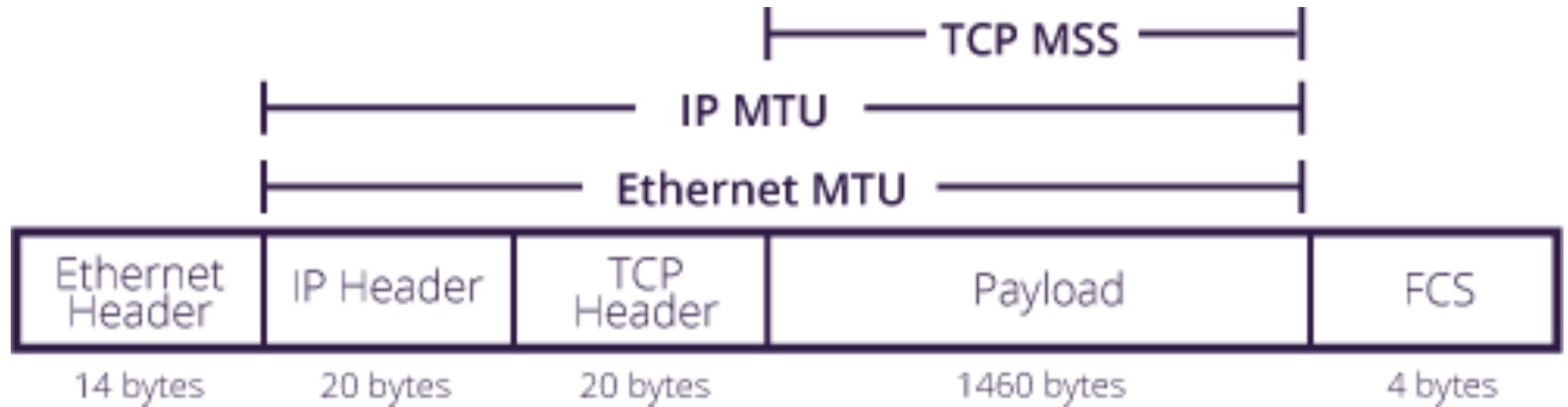
TCP Port Numbers and Concurrent Servers (3)



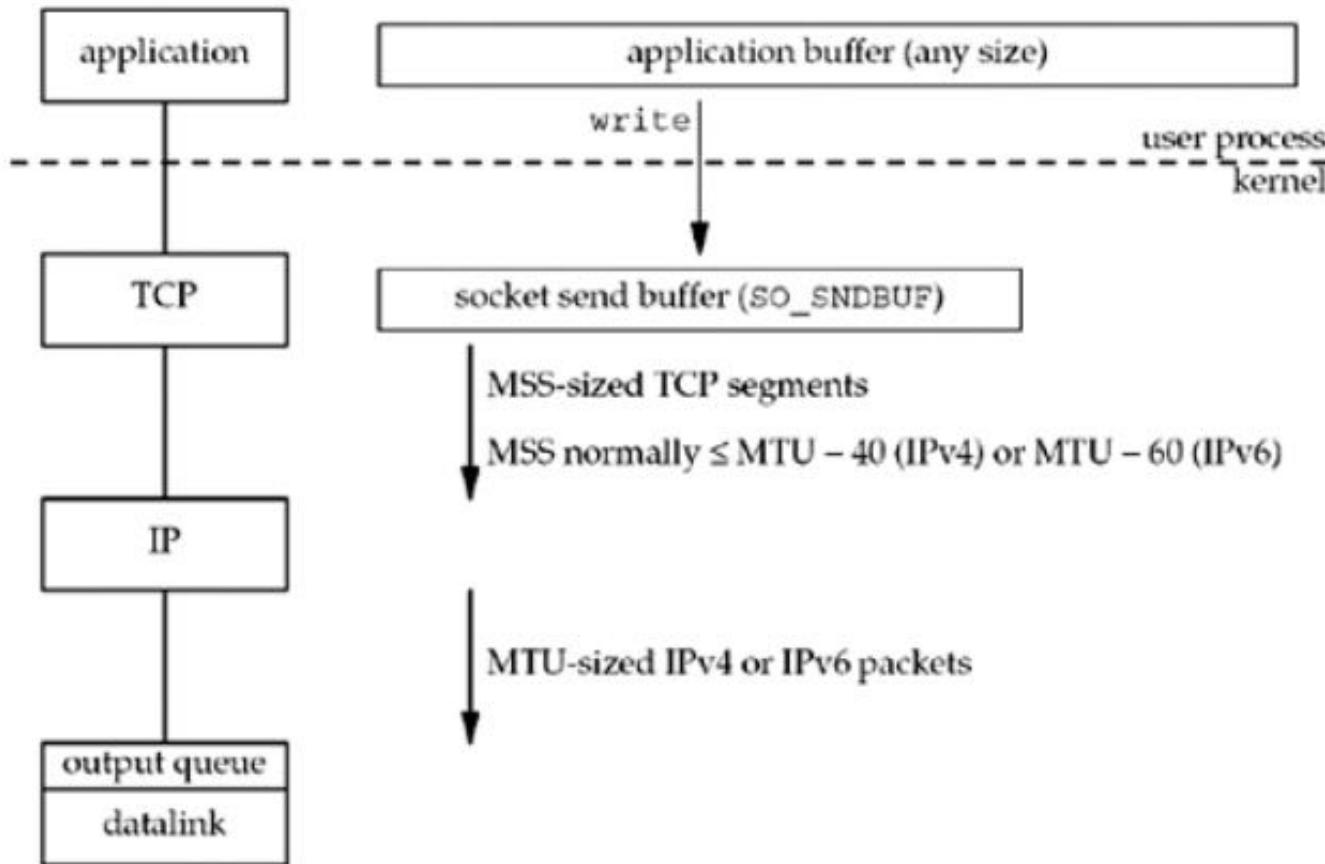
TCP Port Numbers and Concurrent Servers (4)



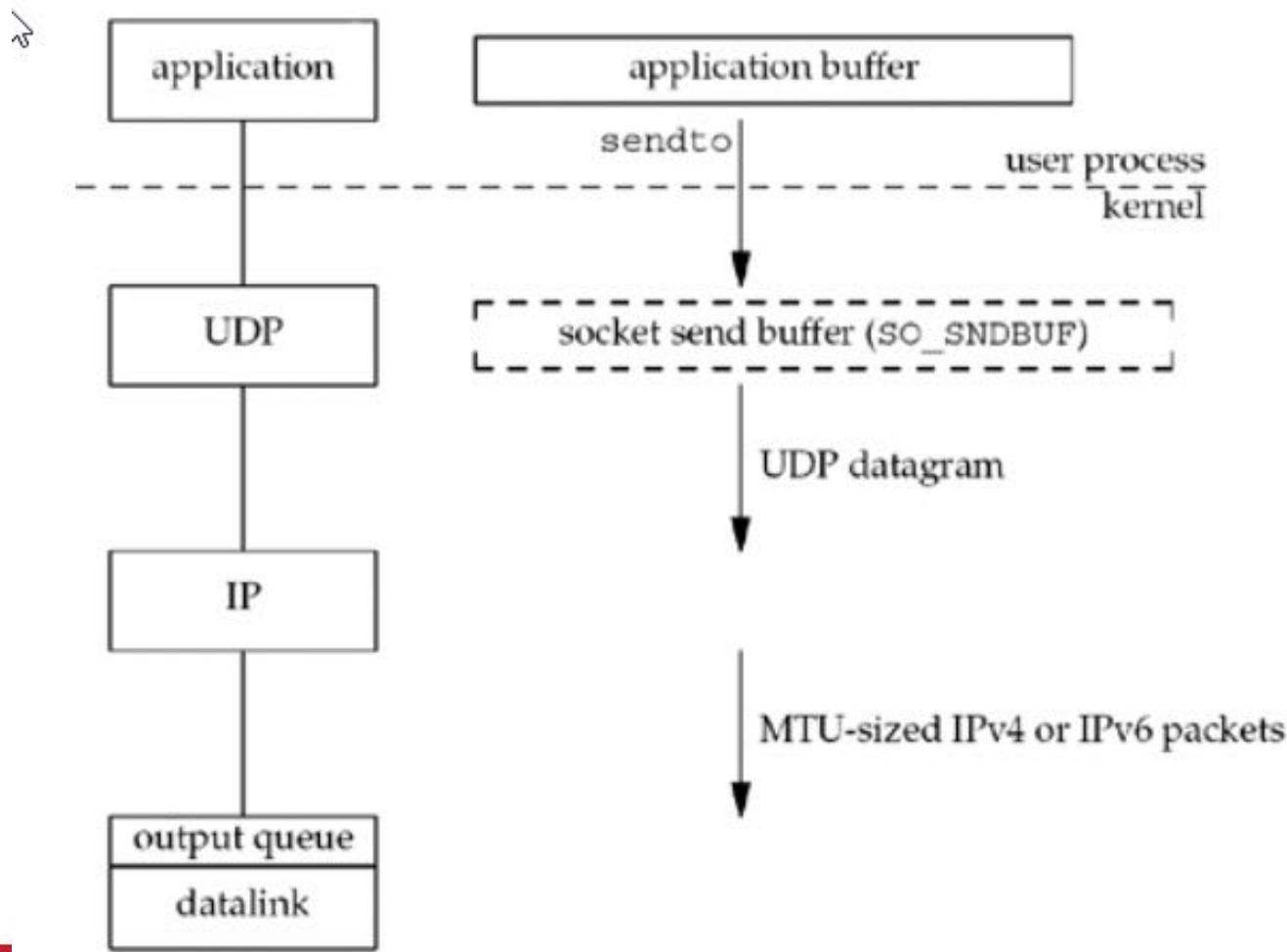
Buffer Sizes and Limitations



TCP output



UDP output



Hỗ trợ của Java

- Class InetAddress:
- Working with IP address and domain name
- ```
InetAddress aComputer =
InetAddress.getByName ("bruno.dcs.qmul.ac.
uk");
```

## 2.2. Trao đổi thông tin với UDP

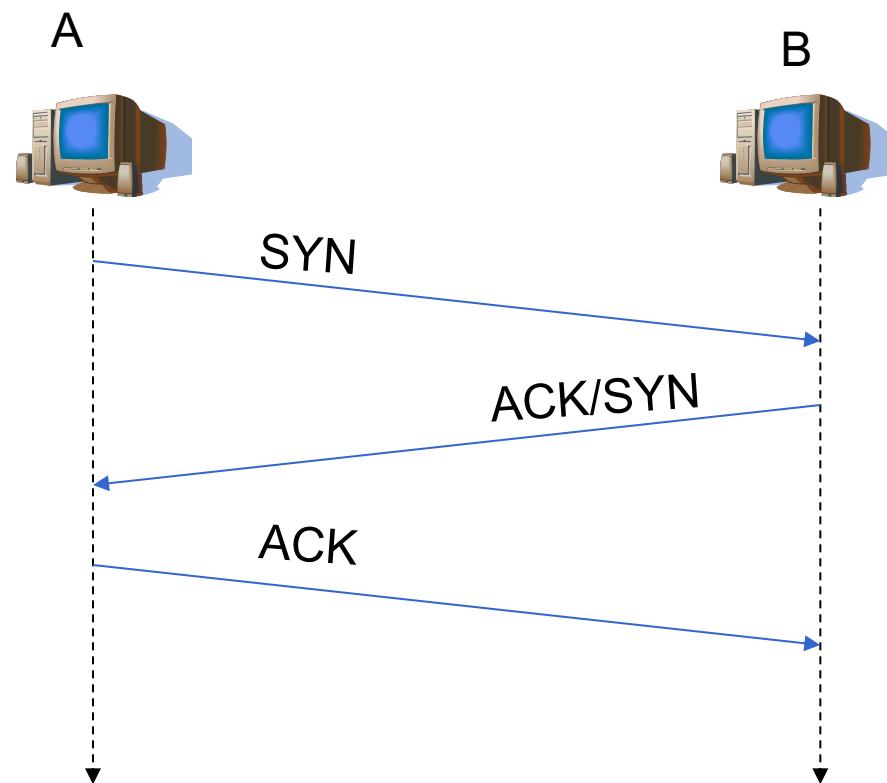
- Đặc điểm:
  - Không hướng kết nối
  - Không tin cậy
  - Không đồng bộ
- Vấn đề:
  - Kích cỡ thông điệp
  - Blocking (*send* không dừng; *receive* bị dừng)
  - Timeouts
  - Receive from any

```
import java.net.*;
import java.io.*;
public class UDPServer{
 public static void main(String args[]){
 DatagramSocket aSocket = null;
 try{
 aSocket = new DatagramSocket(6789);
 byte[] buffer = new byte[1000];
 while(true){
 DatagramPacket request = new DatagramPacket(buffer,
buffer.length);
 aSocket.receive(request);
 DatagramPacket reply = new DatagramPacket(request.getData(),
request.getLength(), request.getAddress(), request.getPort());
 aSocket.send(reply);
 }
 }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
 catch (IOException e) {System.out.println("IO: " + e.getMessage());}
 finally {if(aSocket != null) aSocket.close();}
 }
}
```

```
import java.net.*;
import java.io.*;
public class UDPClient{
 public static void main(String args[]){
 // args give message contents and server hostname
 DatagramSocket aSocket = null;
 try {
 aSocket = new DatagramSocket();
 byte [] m = args[0].getBytes();
 InetAddress aHost = InetAddress.getByName(args[1]);
 int serverPort = 6789;
 DatagramPacket request = new DatagramPacket(m, m.length, aHost,
serverPort);
 aSocket.send(request);
 byte[] buffer = new byte[1000];
 DatagramPacket reply = new DatagramPacket(buffer, buffer.length);

 aSocket.receive(reply);
 System.out.println("Reply: " + new String(reply.getData()));
 }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
 }catch (IOException e){System.out.println("IO: " + e.getMessage());}
 }finally {if(aSocket != null) aSocket.close();}
 }
}
```

## 2.3. Trao đổi thông tin với TCP-IP

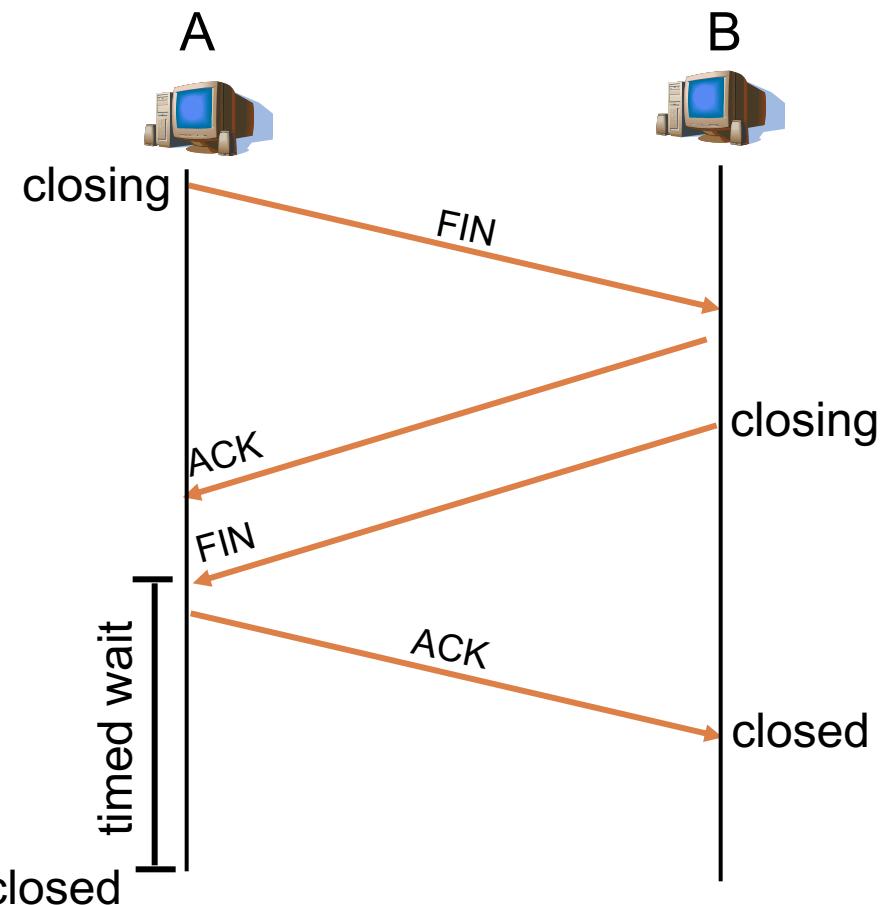


- **Bước 1:** A gửi SYN cho B
  - chỉ ra giá trị khởi tạo seq # của A
  - không có dữ liệu
- **Bước 2:** B nhận SYN, trả lời bằng SYNACK
  - B khởi tạo vùng đệm
  - chỉ ra giá trị khởi tạo seq. # của B
- **Bước 3:** A nhận SYNACK, trả lời ACK, có thể kèm theo dữ liệu

# Ví dụ về việc đóng liên kết

- Bước 1: Gửi FIN cho B
- Bước 2: B nhận được FIN, trả lời ACK, đồng thời đóng liên kết và gửi FIN.
- Bước 3: A nhận FIN, trả lời ACK, vào trạng thái “chờ”.
- Bước 4: B nhận ACK. đóng liên kết.

Lưu ý: Cả hai bên đều có thể chủ động đóng liên kết



```

import java.net.*;
import java.io.*;
public class TCPServer {
 public static void main (String args[]) {
 try{ int serverPort = 7896;
 ServerSocket listenSocket = new ServerSocket(serverPort);
 while(true) { Socket clientSocket = listenSocket.accept();
 Connection c = new Connection(clientSocket);}
 } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}}
class Connection extends Thread {
 DataInputStream in;
 DataOutputStream out;
 Socket clientSocket;
 public Connection (Socket aClientSocket) {
 try { clientSocket = aClientSocket;
 in = new DataInputStream(clientSocket.getInputStream());
 out =new DataOutputStream(clientSocket.getOutputStream());
 this.start();
 } catch(IOException e) {System.out.println("Connection:"+e.getMessage());}
 public void run(){
 try { // an echo server
 String data = in.readUTF();
 out.writeUTF(data);
 } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
 } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
 } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}}}

```

```
import java.net.*;
import java.io.*;
public class TCPClient {
 public static void main (String args[]) {
 // arguments supply message and hostname of destination
 Socket s = null;
 try{
 int serverPort = 7896;
 s = new Socket(args[1], serverPort);
 DataInputStream in = new DataInputStream(s.getInputStream());
 DataOutputStream out =
 new DataOutputStream(s.getOutputStream());
 out.writeUTF(args[0]); // UTF is a string encoding
 String data = in.readUTF();
 System.out.println("Received: "+ data) ;
 }catch (UnknownHostException e){
 System.out.println("Sock:"+e.getMessage());
 }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
 }catch (IOException e){System.out.println("IO:"+e.getMessage());}
 }finally {if(s!=null) try {s.close();}catch (IOException
e){System.out.println("close:"+e.getMessage());}}
 }
}
```

### 3. Lời gọi thủ tục từ xa

- 3.1. Giao thức yêu cầu-trả lời
- 3.2. RPC-Cơ chế lời gọi thủ tục từ xa
- 3.3. SUN-RPC và DCE-RPC
- 3.4. RMI

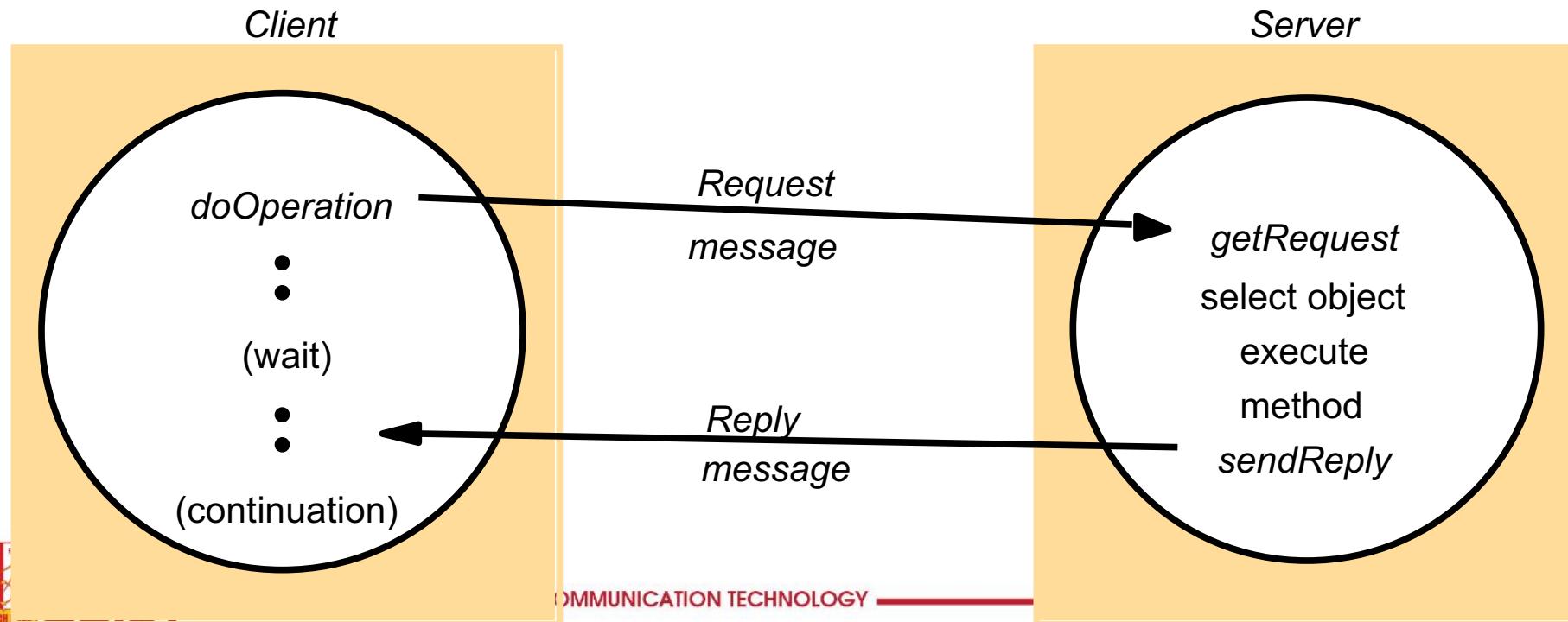
### 3.1. Giao thức yêu cầu-trả lời

- Là cơ chế bậc cao hơn truyền thông điệp, cho phép trao đổi thông tin giữa 2 tiến trình bằng 2 thông báo gửi nhận liên tiếp
- Hỗ trợ các lời gọi từ xa
- Đồng bộ
- Tin cậy

# Yêu cầu-trả lời

## ❑ Đặc điểm:

- ❑ Không cần báo nhận
- ❑ Không cần kiểm soát luồng



# Thủ tục

- *public byte[] doOperation (RemoteRefs, int operationId, byte[] arguments)*
- *public byte[] getRequest () ;*
- *public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);*

# Đơn vị dữ liệu

messageType

*int (0=Request, 1= Reply)*

requestId

*int*

remoteReference

*RemoteRef*

operationId

*int or Operation*

arguments

*array of bytes*

# HTTP: 1 vd của giao thức yêu cầu-trả lời

## HTTP *request message*

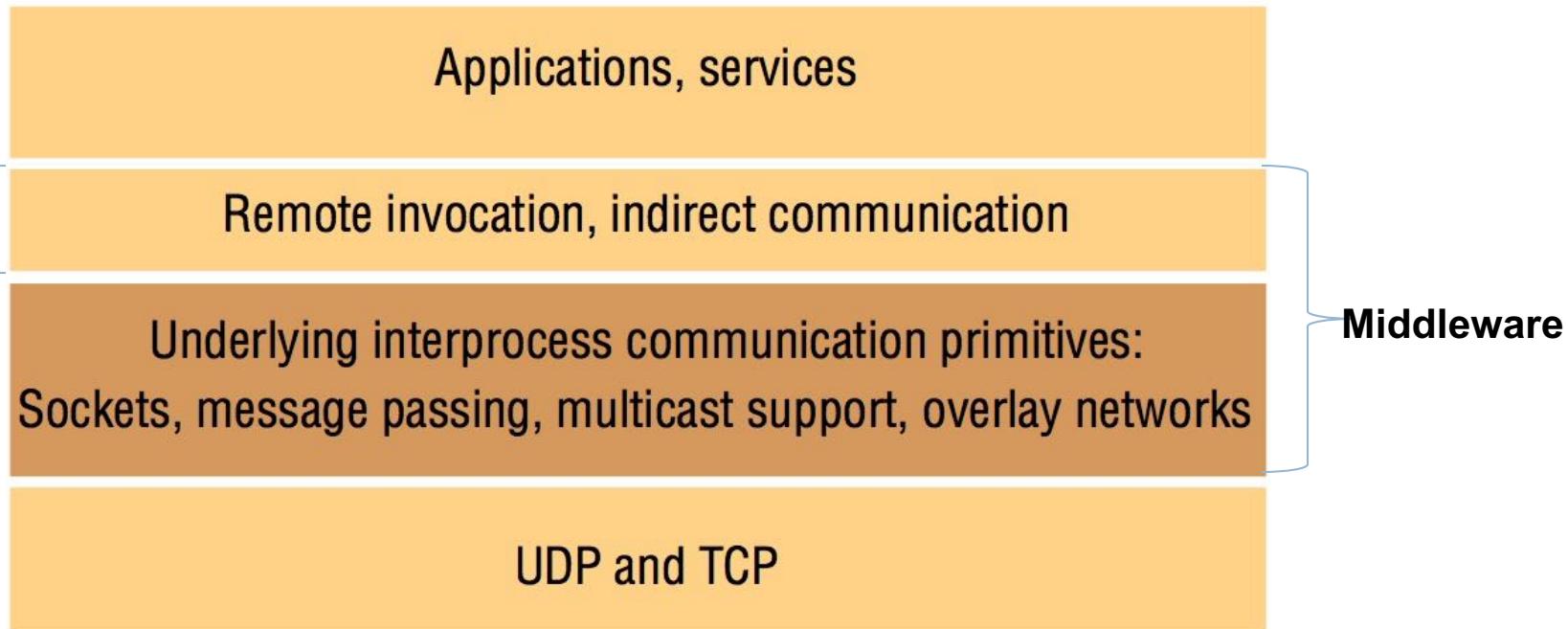
| <i>method</i> | <i>URL or pathname</i>        | <i>HTTP version</i> | <i>headers</i> | <i>message body</i> |
|---------------|-------------------------------|---------------------|----------------|---------------------|
| GET           | //www.dcs.qmw.ac.uk/index.htm | HTTP/ 1.1           |                |                     |

## HTTP *reply message*

| <i>HTTP version</i> | <i>status code</i> | <i>reason</i> | <i>headers</i> | <i>message body</i> |
|---------------------|--------------------|---------------|----------------|---------------------|
| HTTP/1.1            | 200                | OK            |                | resource data       |

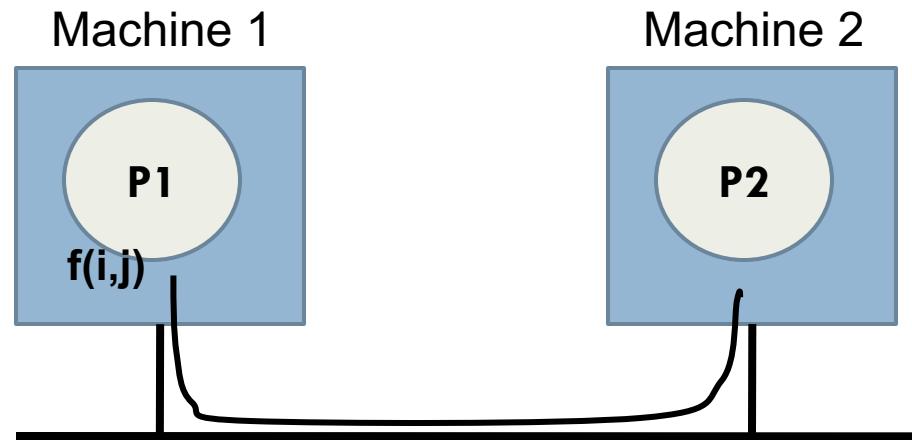
## 3.2. Lời gọi thủ tục từ xa RPC (Remote Procedure Call)

Nội  
dung  
nghiên  
cứu của  
chương  
học này

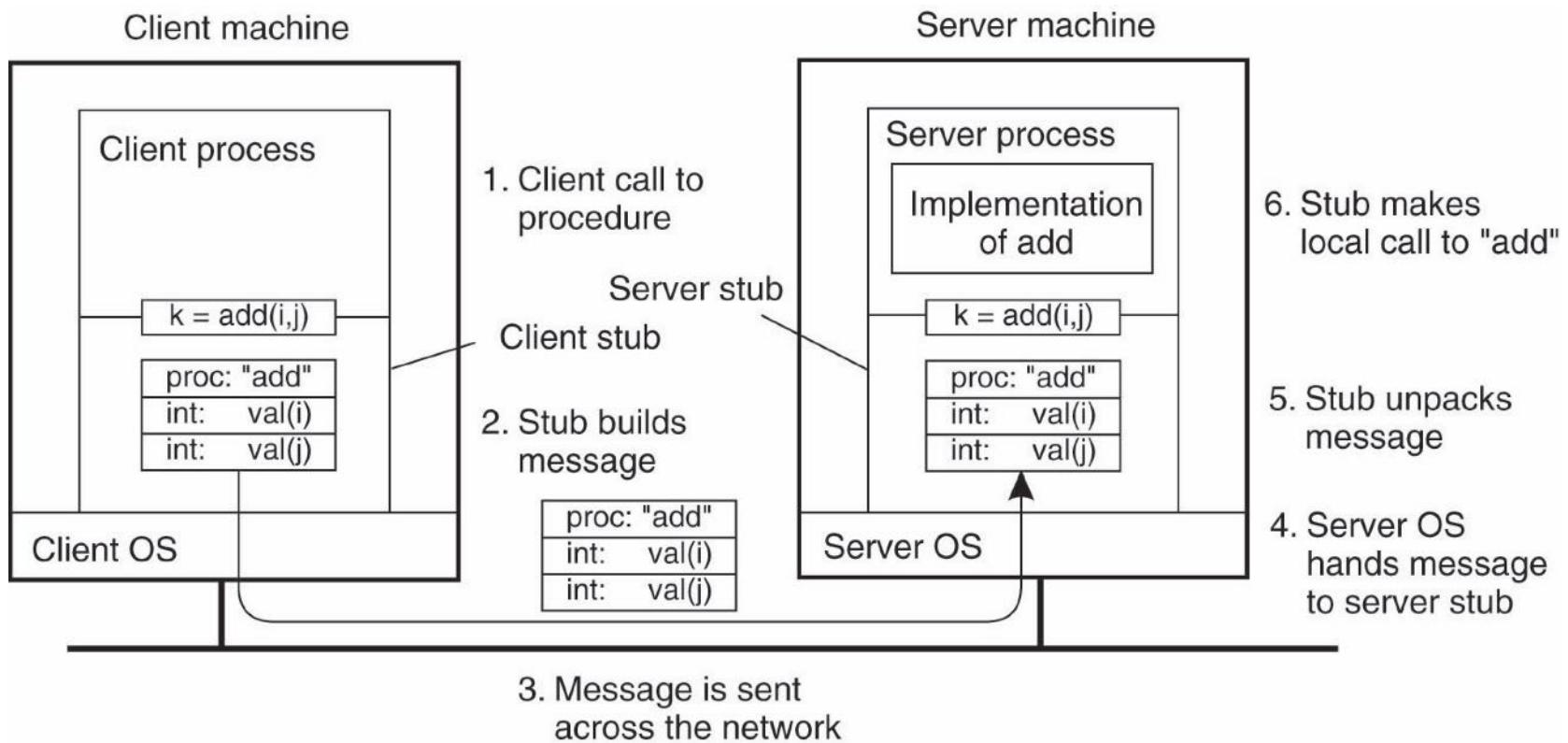


# Khái niệm lời gọi thủ tục từ xa

- Cơ chế truy cập trong suốt với người dùng
- Cơ chế
- Vấn đề:
  - ▣ Hệ thống không đồng nhất
    - Không gia nhó khác nhau
    - Cách biểu diễn thông tin khác nhau
  - ▣ Có lỗi xảy ra

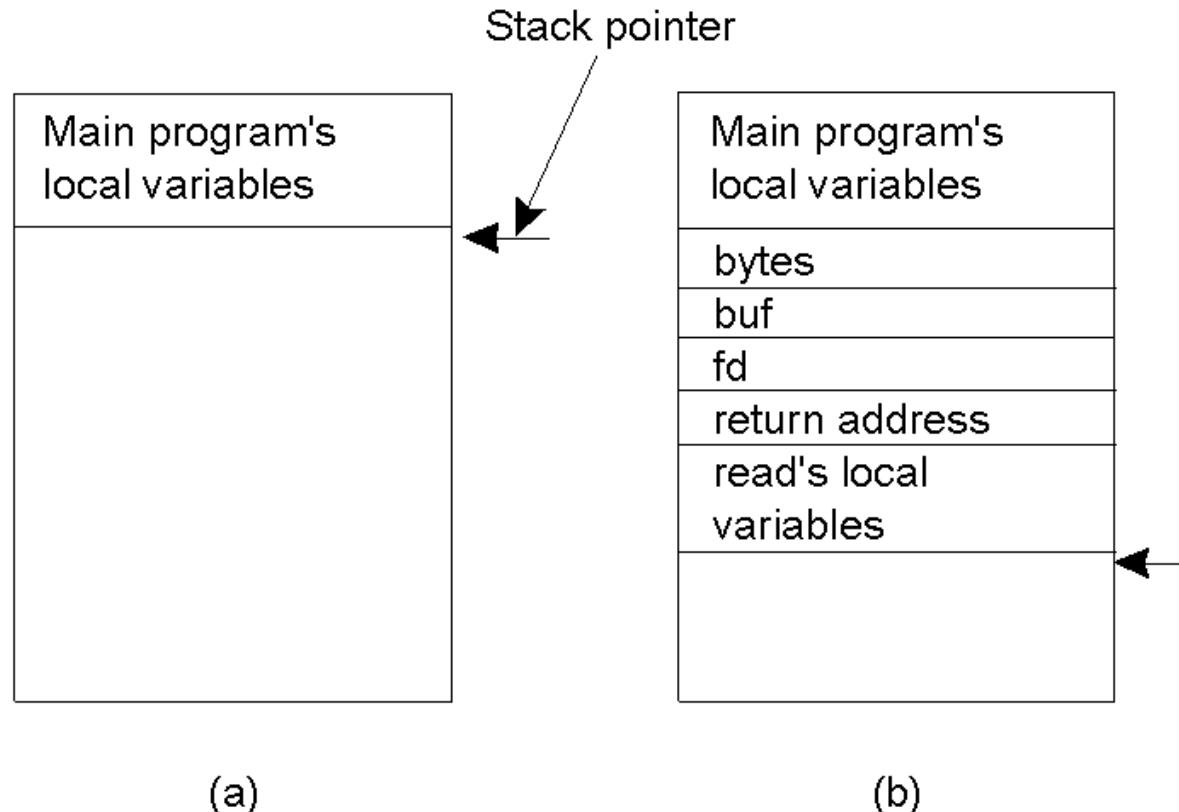


# Cơ chế RPC

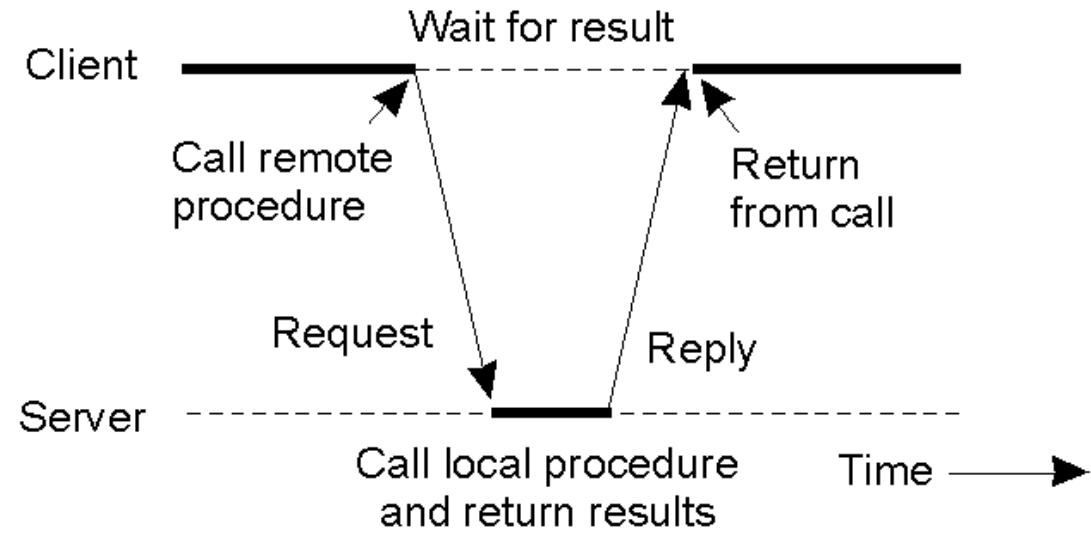
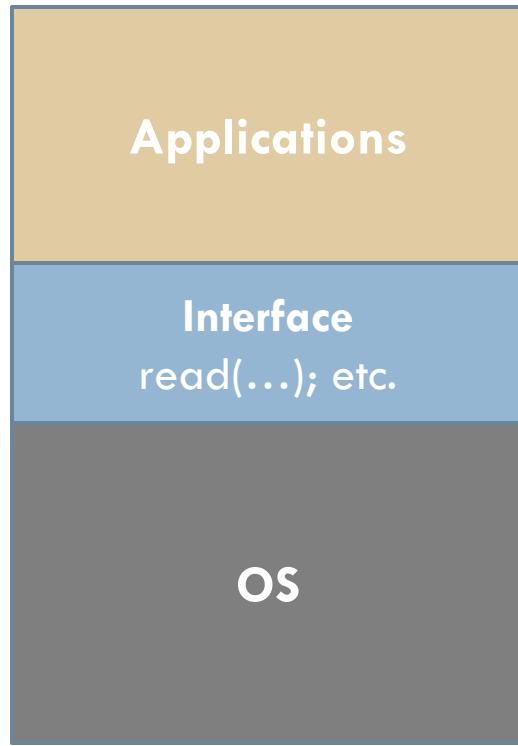


# Lời gọi thủ tục thông thường

**count = read(fd, buf, nbytes)**



# Cơ chế RPC



# Vấn đề với cơ chế truyền tham số

- Có 2 loại tham số: Tham biến và Tham chiếu
- Vấn đề với 2 loại tham số trên:
  - Tham biến
    - Vấn đề khi biểu diễn dữ liệu khác nhau
    - Các dữ liệu không thuộc cùng một kiểu, các kiểu dữ liệu khác nhau được biểu diễn khác nhau
  - Tham chiếu
    - Bộ nhớ phân tán
    - Giải pháp: Copy dữ liệu???
      - Tham chiếu thay bằng copy/restore
      - Vấn đề với giải pháp: tốn băng thông
  - Giải pháp: không dùng → không phù hợp
  - Vấn đề đối với dữ liệu có cấu trúc

# Ví dụ: Sai lệch trong truyền bằng tham biến

**Intel Pentium** (little endian)

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 3 | 2 | 1 | 0 |
| L | L | I | J |   |
| 0 | 5 | 0 | 0 | 0 |
| 4 | J | 5 | I | L |

(a)

**SPARC** (big endian)

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 5 | 0 | 0 | 0 |
| 4 | 5 | 6 | 7 |
| J | I | L | L |

(b)

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 0 | 0 | 5 |
| 4 | 5 | 6 | 7 |
| L | L | I | J |

(c)

# Nhu cầu: Đặc tả tham số

- 2 bên gửi và nhận cùng phải thống nhất về đặc tả tham số (tuân thủ 1 kiểu giao thức)
- Các yếu tố thống nhất:
  - Định dạng thông điệp
  - Cách biểu diễn cấu trúc dữ liệu cơ bản
  - Kiểu trao đổi thông điệp
  - Triển khai client-stub và server-stub

```
foobar(char x; float y; int z[5])
{

}
```

(a)

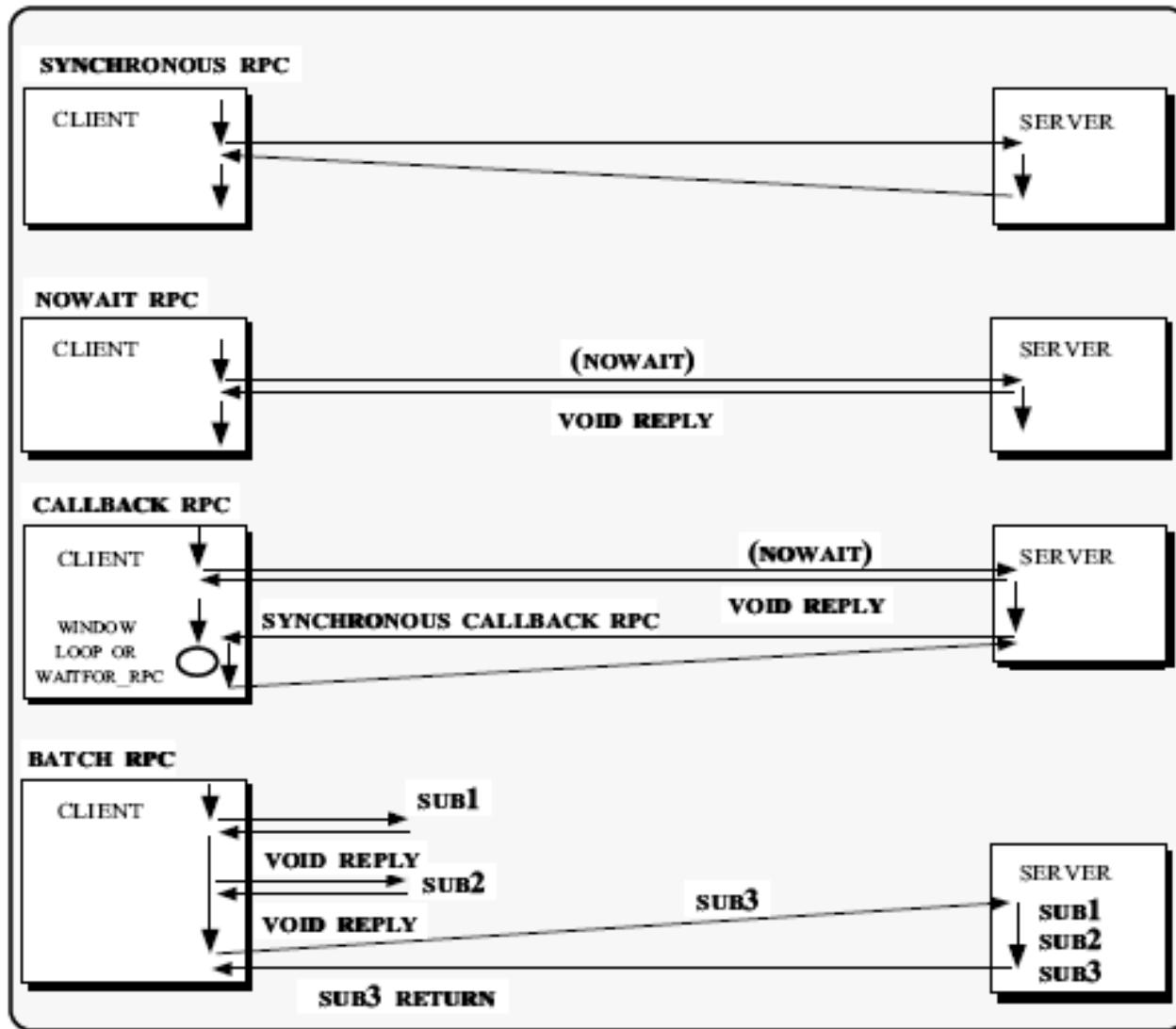
| foobar's local variables |  |
|--------------------------|--|
| x                        |  |
| y                        |  |
| 5                        |  |
| z[0]                     |  |
| z[1]                     |  |
| z[2]                     |  |
| z[3]                     |  |
| z[4]                     |  |

(b)

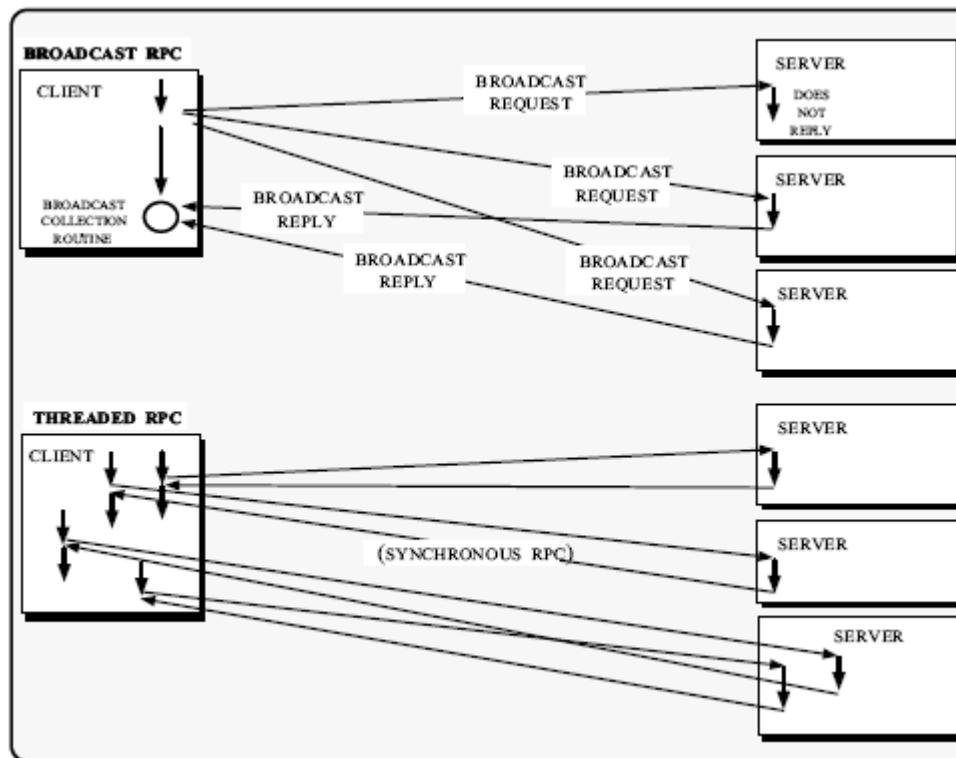
# Tính mở của RPC

- Client và Server được cài đặt bởi các NSX khác nhau
- Giao diện thống nhất client và server
  - ▣ Không phụ thuộc công cụ và ngôn ngữ lập trình
  - ▣ Mô tả đầy đủ và trung lập
  - ▣ Thường dùng ngôn ngữ định nghĩa giao diện

# Các mô hình RPC



# Các mô hình RPC



# Usecase: Hệ thống DCE RPC

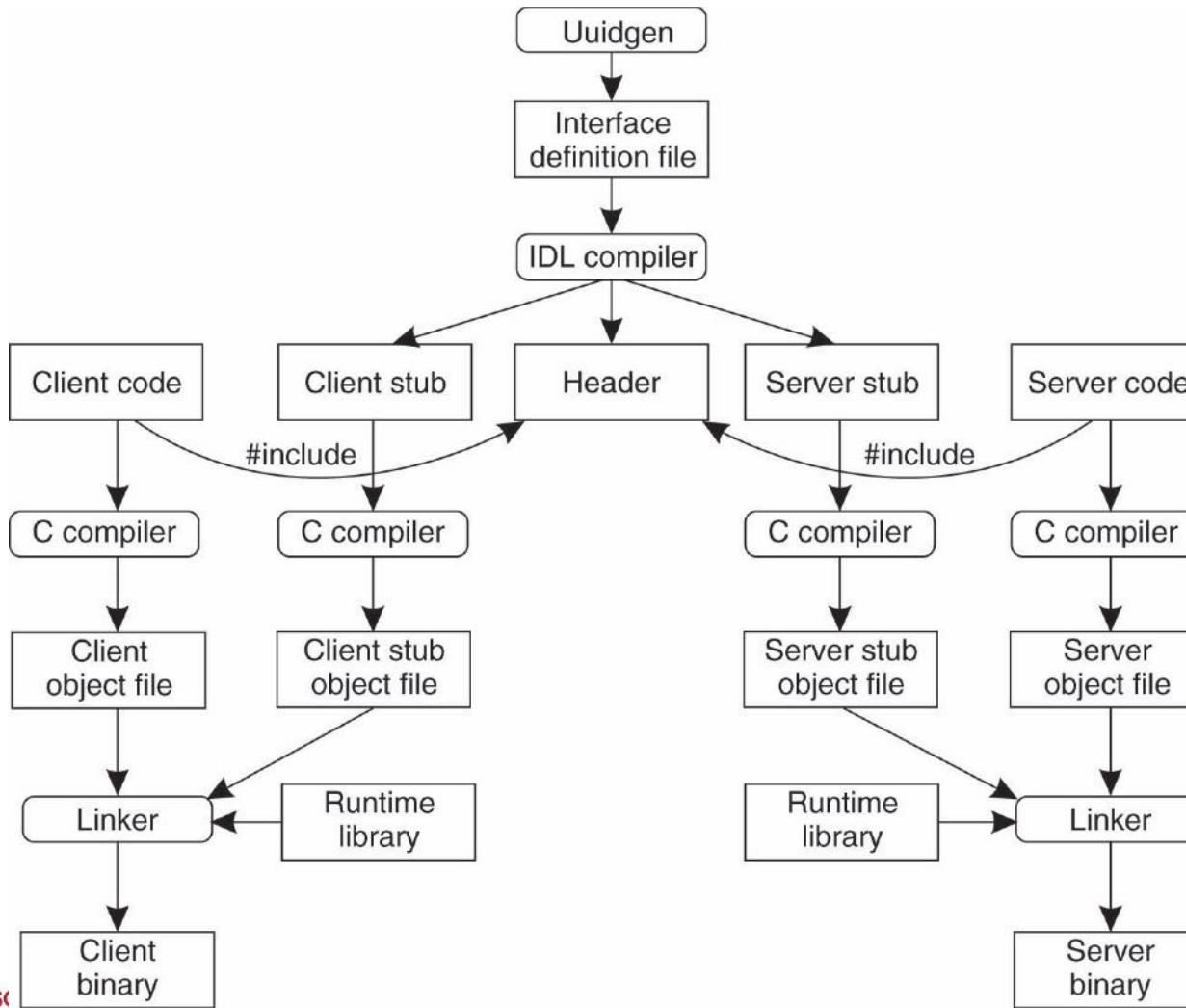
- Distributed Computing Environment (DCE) được phát triển bởi Open Group: <http://www.opengroup.org/dce/>
- Tầng middleware
- Mô hình client-server
- Giao tiếp được thực hiện thông qua RPCs
- Các dịch vụ:
  - Distributed file service
  - Directory service
  - Distributed time service

# Mục đích

- Sử dụng RPCs để truy cập các dịch vụ từ xa
- Đơn giản hóa việc lập trình
- Trong suốt

→ Client và server hoàn toàn độc lập

# Xây dựng chương trình bằng DCE-RPC



# Thực hành: Xây dựng RPC trên Windows

- Link: <https://docs.microsoft.com/en-us/windows/win32/rpc/tutorial>
- Create interface definition and application configuration files.
- Use the MIDL compiler to generate C-language client and server stubs and headers from those files.
- Write a client application that manages its connection to the server.
- Write a server application that contains the actual remote procedures.
- Compile and link these files to the RPC run-time library to produce the distributed application.

### 3.3. RMI (Remote Method Invocation)

- So sánh với RPC

- ▣ Giống:

- Cùng hỗ trợ lập trình với các giao diện
    - Dựa trên giao thức yêu cầu/trả lời
    - Mức độ trong suốt

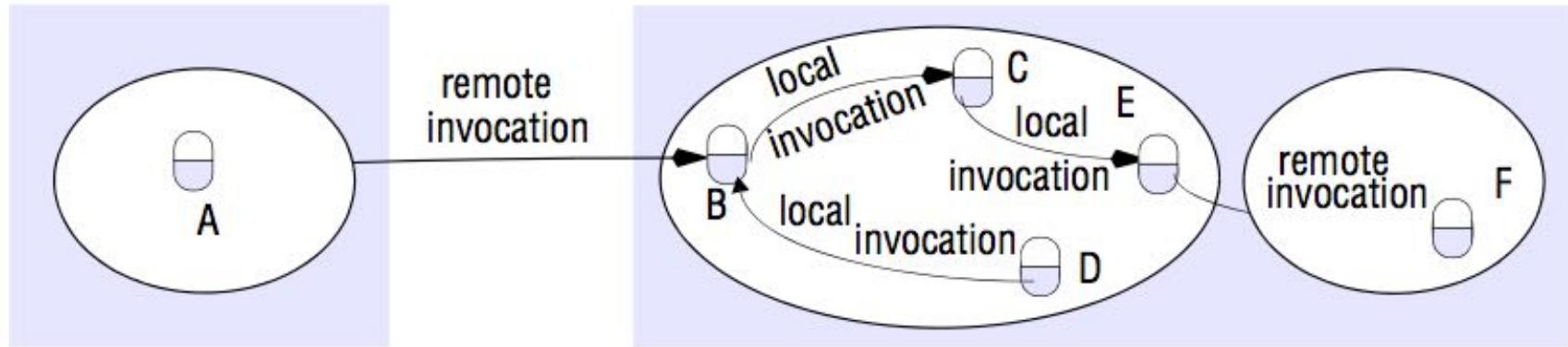
- ▣ Khác:

- Lập trình viên có thể sử dụng khai thác hết điểm mạnh của OOP
    - Định danh duy nhất → truyền tham chiếu đối tượng

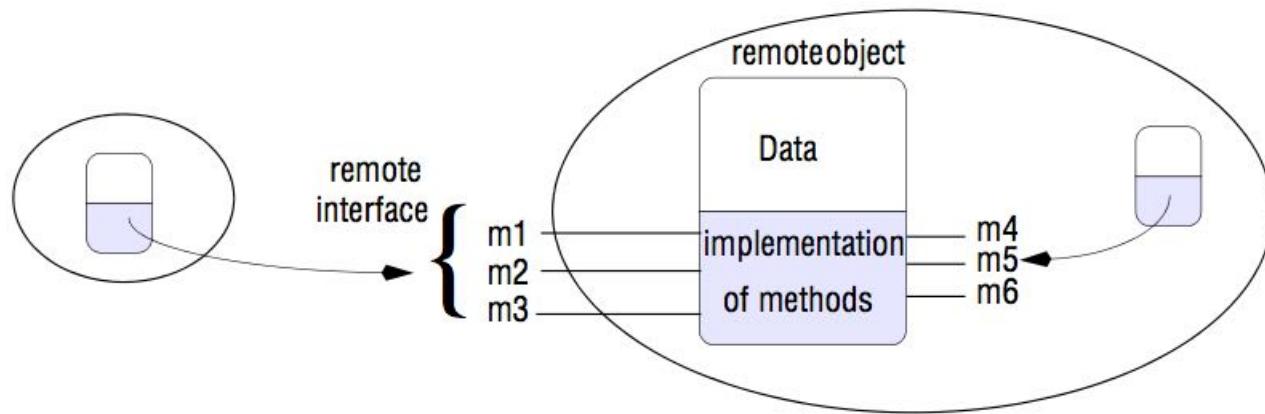
# RMI: Lời gọi phương thức từ xa

- Lập trình hướng đối tượng :
  - đối tượng từ xa, ứng dụng phân tán hướng đối tượng
- Các vấn đề cần giải quyết
  - Định vị đối tượng từ xa
  - Trao đổi thông tin với đối tượng
  - Gọi các phương thức của đối tượng
- RMI, T-RMI, DCOM, CORBA

# Mô hình đối tượng phân tán



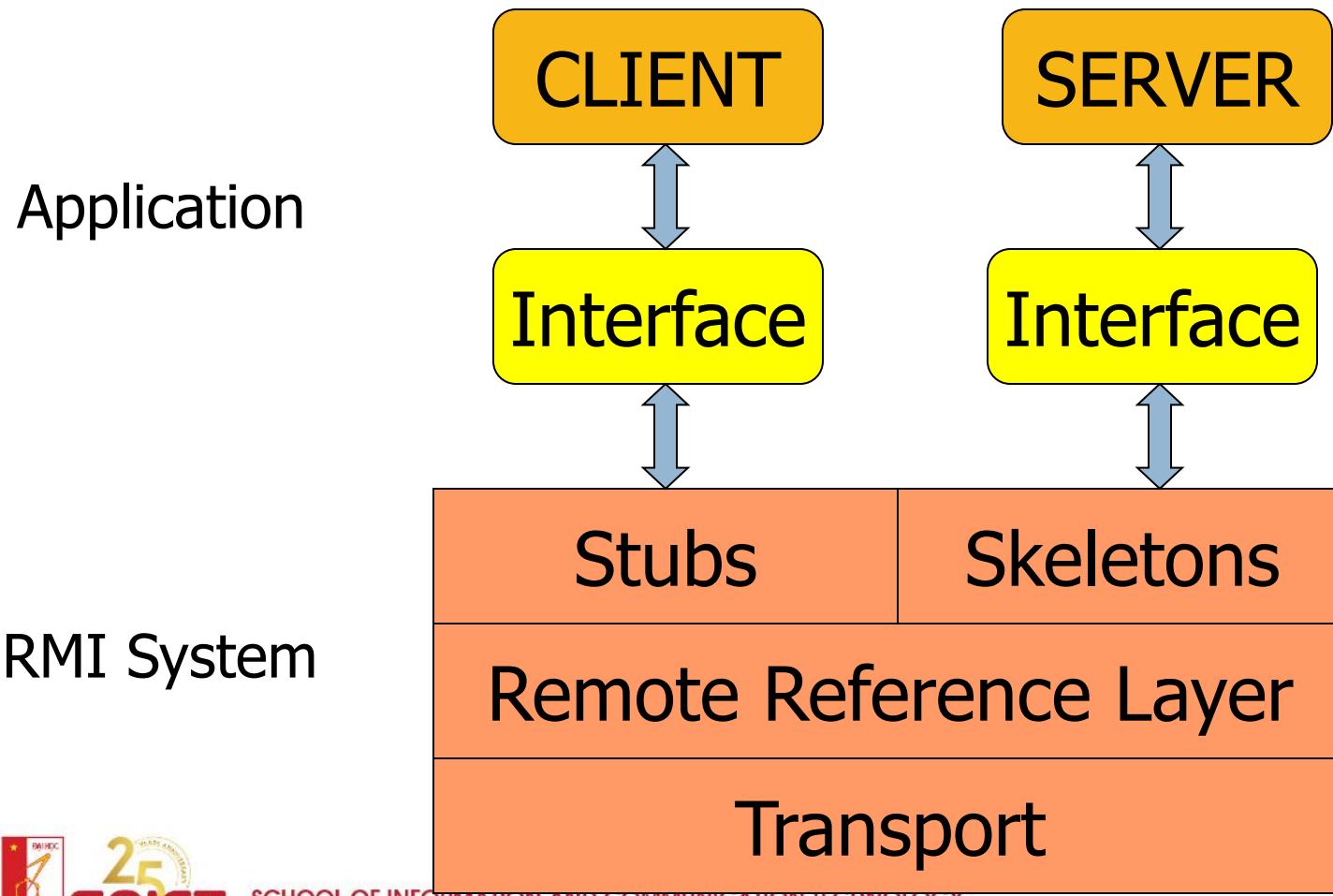
# Đối tượng từ xa và giao diện từ xa



# Ưu điểm

- Đơn giản, dễ sử dụng
- Trong suốt: lời gọi phương thức từ xa giống lời gọi phương thức cục bộ
- Độ tin cậy cao
- An toàn và bảo mật (do JVM cung cấp)
- Nhược điểm:
  - ▣ Chỉ dùng cho java

# Kiến trúc



# Thực hành: Xây dựng chương trình phân tán sử dụng RMI

- Link:  
<https://docs.oracle.com/javase/tutorial/rmi/index.html>
- Overview of RMI
- Writing RMI server
- Creating RMI client
- Compiling & Running

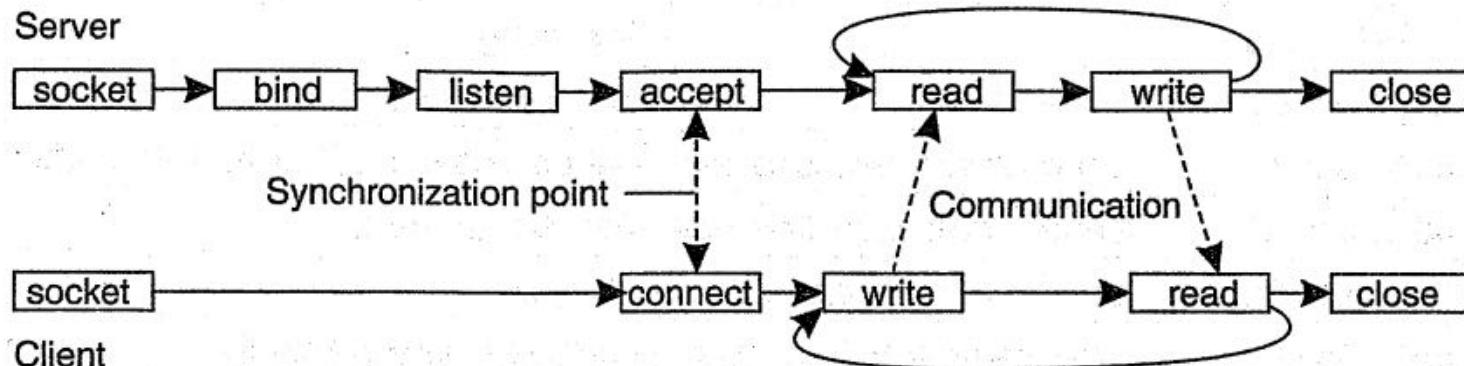
# 4. Trao đổi thông tin hướng thông điệp

- 3.1. Trao đổi thông tin hướng thông điệp tạm thời
- 3.2. Trao đổi thông tin hướng thông điệp bền vững

# 4.1. Trao đổi thông tin hướng thông điệp tạm thời

## □ Berkeley Sockets

| Primitive | Meaning                                         |
|-----------|-------------------------------------------------|
| Socket    | Create a new communication end point            |
| Bind      | Attach a local address to a socket              |
| Listen    | Announce willingness to accept connections      |
| Accept    | Block caller until a connection request arrives |
| Connect   | Actively attempt to establish a connection      |
| Send      | Send some data over the connection              |
| Receive   | Receive some data over the connection           |
| Close     | Release the connection                          |



# *socket* function

- To perform network I/O, the first thing a process must do is call the *socket* function

```
#include <sys/socket.h>
int socket (int family, int type, int protocol);
```

- Returns: non-negative descriptor if OK, -1 on error

| family   | Description                        |
|----------|------------------------------------|
| AF_INET  | IPv4 protocols                     |
| AF_INET6 | IPv6 protocols                     |
| AF_LOCAL | Unix domain protocols (Chapter 15) |
| AF_ROUTE | Routing sockets (Chapter 18)       |
| AF_KEY   | Key socket (Chapter 19)            |

**family**

| type           | Description             |
|----------------|-------------------------|
| SOCK_STREAM    | stream socket           |
| SOCK_DGRAM     | datagram socket         |
| SOCK_SEQPACKET | sequenced packet socket |
| SOCK_RAW       | raw socket              |

**socket**

| Protocol     | Description             |
|--------------|-------------------------|
| IPPROTO_TCP  | TCP transport protocol  |
| IPPROTO_UDP  | UDP transport protocol  |
| IPPROTO_SCTP | SCTP transport protocol |

**protocol**

# *connect* Function

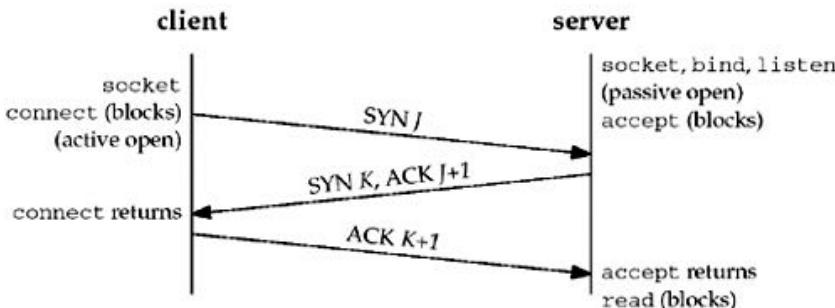
- The *connect* function is used by a TCP client to establish a connection with a TCP server.

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *servaddr,
 socklen_t addrlen);
```

- Returns: 0 if OK, -1 on error
- *sockfd* is a socket descriptor returned by the *socket* function
- The second and third arguments are a pointer to a socket address structure and its size.
- The client does not have to call *bind* before calling *connect*: the kernel will choose both an ephemeral port and the source IP address if necessary.

# Nhắc lại: Thiết lập liên kết TCP : Giao thức bắt tay 3 bước



- **Bước 1:** A gửi SYN cho B
  - chỉ ra giá trị khởi tạo seq # của A
  - không có dữ liệu
- **Bước 2:** B nhận SYN, trả lời bằng SYNACK
  - B khởi tạo vùng đệm
  - chỉ ra giá trị khởi tạo seq. # của B
- **Bước 3:** A nhận SYNACK, trả lời ACK, có thể kèm theo dữ liệu

# *connect* Function (2)

- Problems with *connect* function:
  1. If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned. (If no response is received after a total of 75 seconds, the error is returned).
  2. If the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for connections on the server host at the port specified (i.e., the server process is probably not running). Error: ECONNREFUSED.
  3. If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered a soft error. If no response is received after some fixed amount of time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as either EHOSTUNREACH or ENETUNREACH.

# *bind* Function

- The bind function assigns a local protocol address to a socket.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr,
 socklen_t addrlen);
```

- Returns: 0 if OK,-1 on error

- Example

```
struct sockaddr_in address;
/* type of socket created in socket() */
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
/* 7000 is the port to use for connections */
address.sin_port = htons(7000);
/* bind the socket to the port specified above */
```

# *listen* Function

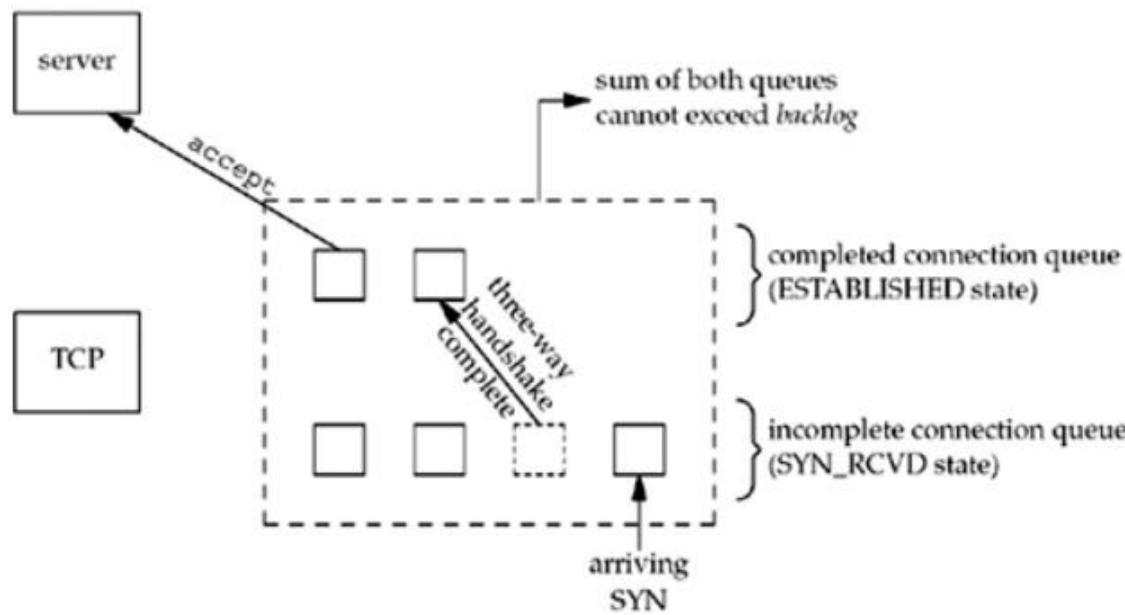
- The *listen* function is called only by a TCP server.
- When a socket is created by the *socket* function, it is assumed to be an active socket, that is, a client socket that will issue a *connect*.
- The *listen* function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
- Move the socket from the CLOSED state to the LISTEN state.

```
#include <sys/socket.h>
int listen (int sockfd, int backlog);
```

- Returns: 0 if OK, -1 on error

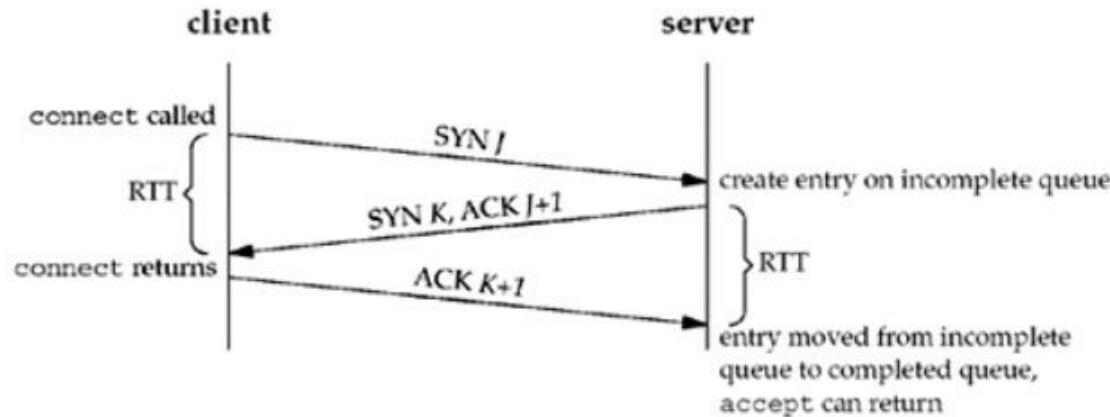
# *listen* Function (2)

- The second argument (*backlog*) to this function specifies the maximum number of connections the kernel should queue for this socket.



**The two queues maintained by TCP for a listening socket**

# *listen* Function (3)



**TCP three-way handshake and the two queues for a listening socket.**

# *accept* Function

- *accept* is called by a TCP server to return the next completed connection from the front of the completed connection queue.
- If the completed connection queue is empty, the process is put to sleep.

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t
 *addrlen);
```

- Returns: non-negative descriptor if OK, -1 on error
- The *cliaddr* and *addrlen* arguments are used to return the protocol address of the connected peer process (the client).
- *addrlen* is a value-result argument

# *accept* Function

## □ Example

```
int addrlen;
struct sockaddr_in address;

addrlen = sizeof(struct sockaddr_in);
new_socket = accept(socket_desc, (struct sockaddr *)&address, &addrlen);
if (new_socket<0)
 perror("Accept connection");
```

# *fork* and *exec* Functions

```
#include <unistd.h>
pid_t fork(void);
```

- Returns: 0 in child, process ID of child in parent, -1 on error
- *fork* function (including the variants of it provided by some systems) is the only way in Unix to create a new process.
- It is called once but it returns twice.
- It returns once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child). It also returns once in the child, with a return value of 0.
- The reason *fork* returns 0 in the child, instead of the parent's process ID, is because a child has only one parent and it can always obtain the parent's process ID by calling *getppid*.

# Example

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
 printf("--beginning of program\n");

 int counter = 0;
 pid_t pid = fork();

 if (pid == 0)
 {
 // child process
 int i = 0;
 for (; i < 5; ++i)
 {
 printf("child process: counter=%d\n", ++counter);
 }
 }
 else if (pid > 0)
 {
 // parent process
 int j = 0;
 for (; j < 5; ++j)
 {
 printf("parent process: counter=%d\n", ++counter);
 }
 }
 else
 {
 // fork failed
 printf("fork() failed!\n");
 return 1;
 }

 printf("--end of program--\n");

 return 0;
}
```

# Concurrent Servers

- fork a child process to handle each client

```
pid_t pid;
int listenfd, connfd;

listenfd = Socket(...);

/* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ...);
Listen(listenfd, LISTENQ);

for (; ;) {
 connfd = Accept (listenfd, ...); /* probably blocks */

 if((pid = Fork()) == 0) {
 Close(listenfd); /* child closes listening socket */
 doit(connfd); /* process the request */
 Close(connfd); /* done with this client */
 exit(0); /* child terminates */
 }

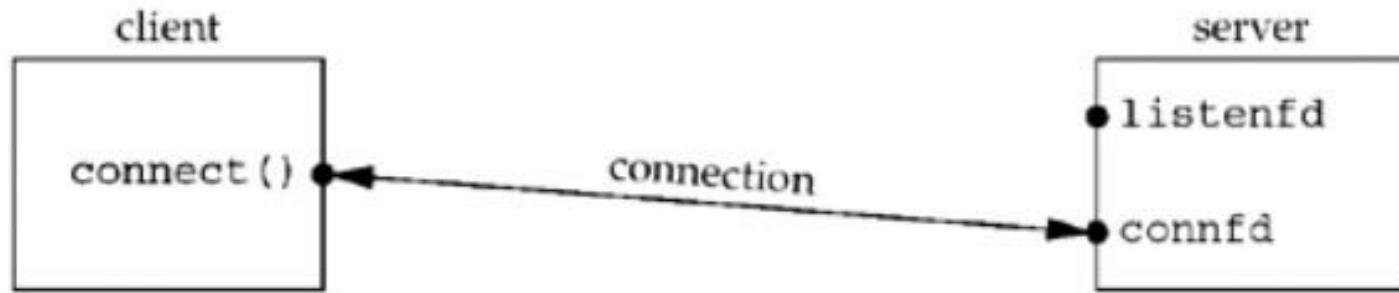
 Close(connfd); /* parent closes connected socket */ -
}
```



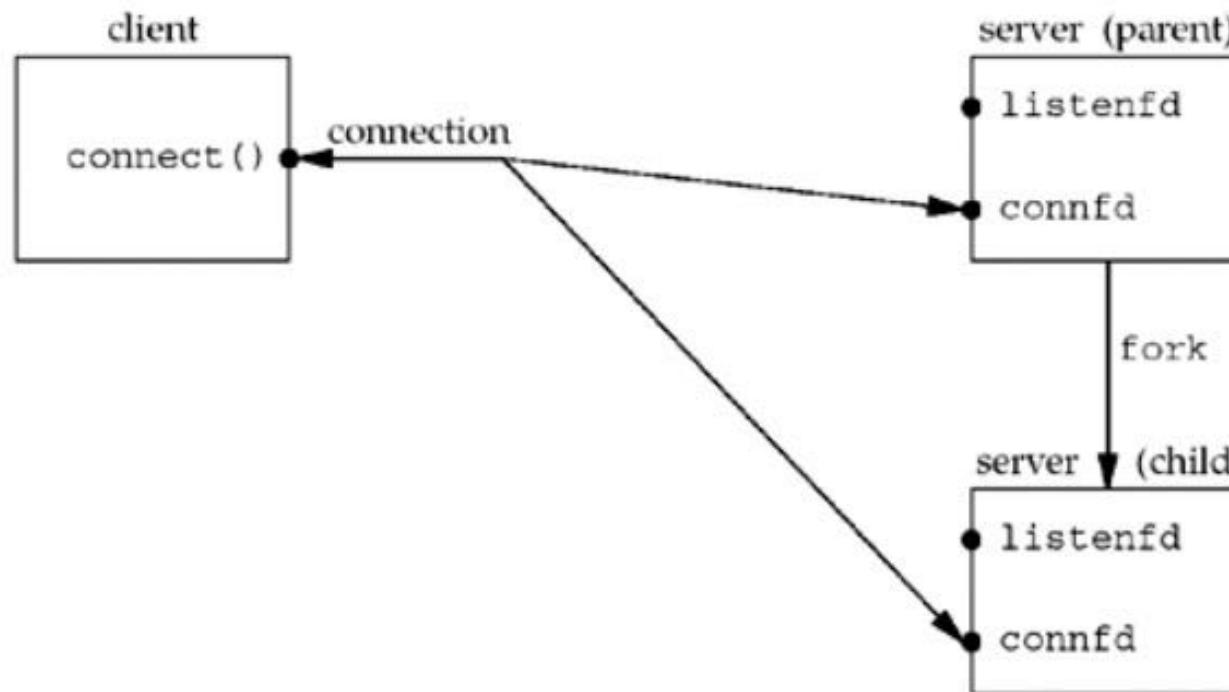
# Status of client/server before call to *accept* returns.



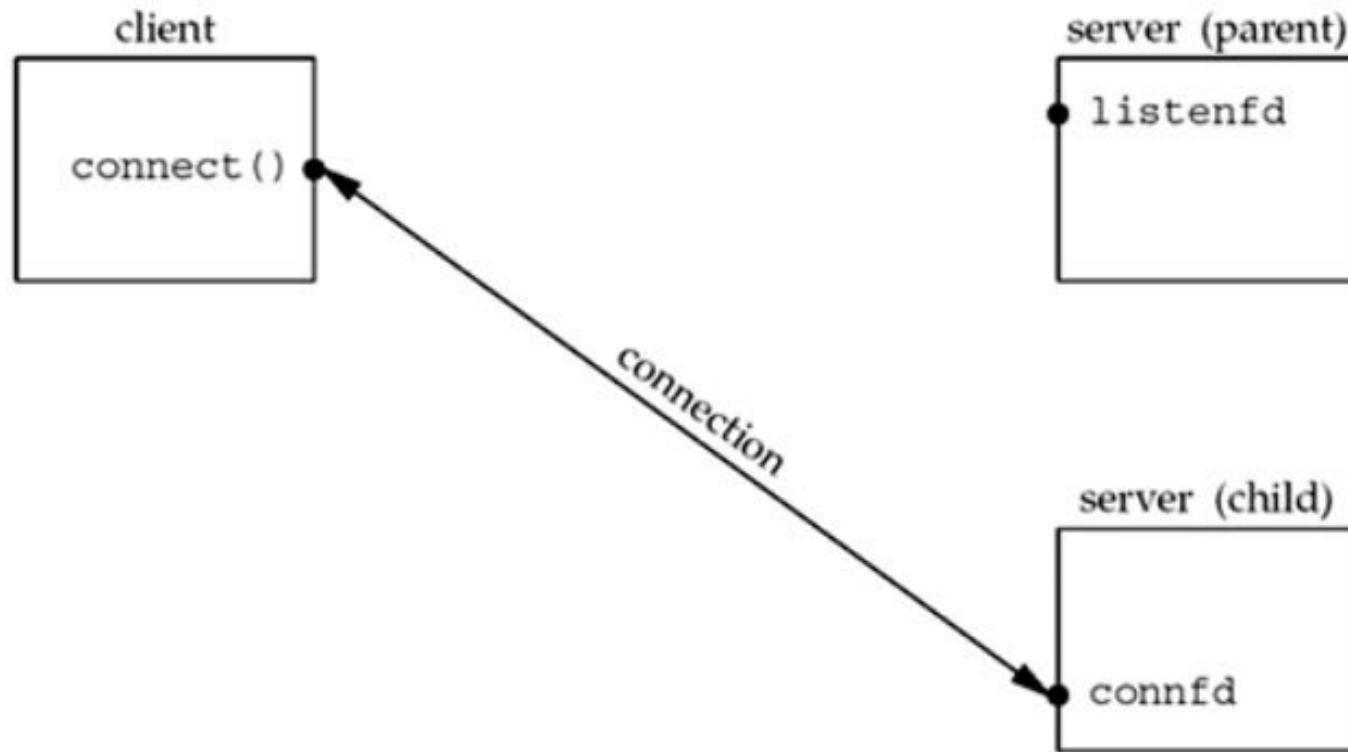
# Status of client/server after return from *accept*.



# Status of client/server after fork returns.



# Status of client/server after parent and child close appropriate sockets.



# *close* Function

- The normal Unix close function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>
int close (int sockfd);
```

- Returns: 0 if OK, -1 on error
- If the parent doesn't close the socket, when the child closes the connected socket, its reference count will go from 2 to 1 and it will remain at 1 since the parent never closes the connected socket. This will prevent TCP's connection termination sequence from occurring, and the connection will remain open.

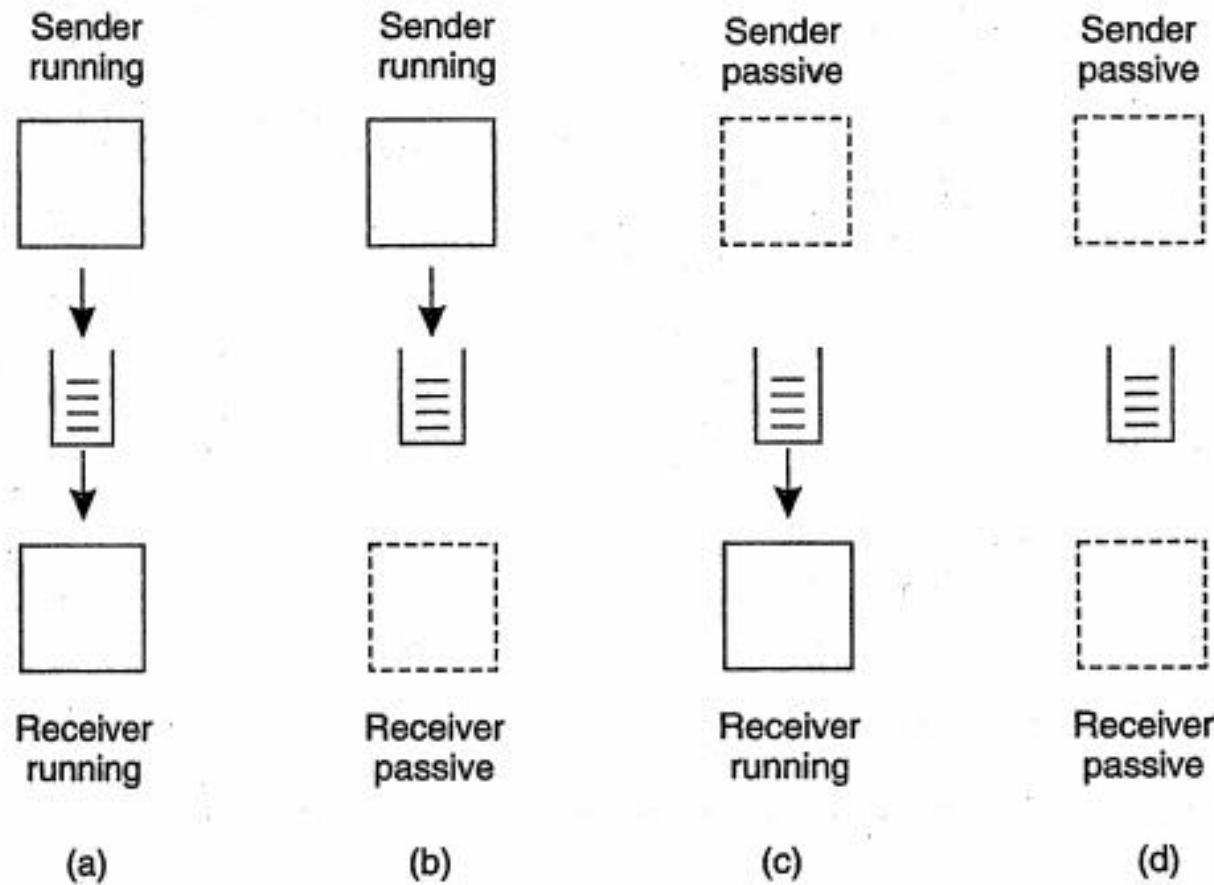
# Message-Passing Interface

| Primitive    | Meaning                                                           |
|--------------|-------------------------------------------------------------------|
| MPI_bsend    | Append outgoing message to a local send buffer                    |
| MPI_send     | Send a message and wait until copied to local or remote buffer    |
| MPI_ssend    | Send a message and wait until receipt starts                      |
| MPI_sendrecv | Send a message and wait for reply                                 |
| MPI_isend    | Pass reference to outgoing message, and continue                  |
| MPI_issend   | Pass reference to outgoing message, and wait until receipt starts |
| MPI_recv     | Receive a message; block if there is none                         |
| MPI_irecv    | Check if there is an incoming message, but do not block           |

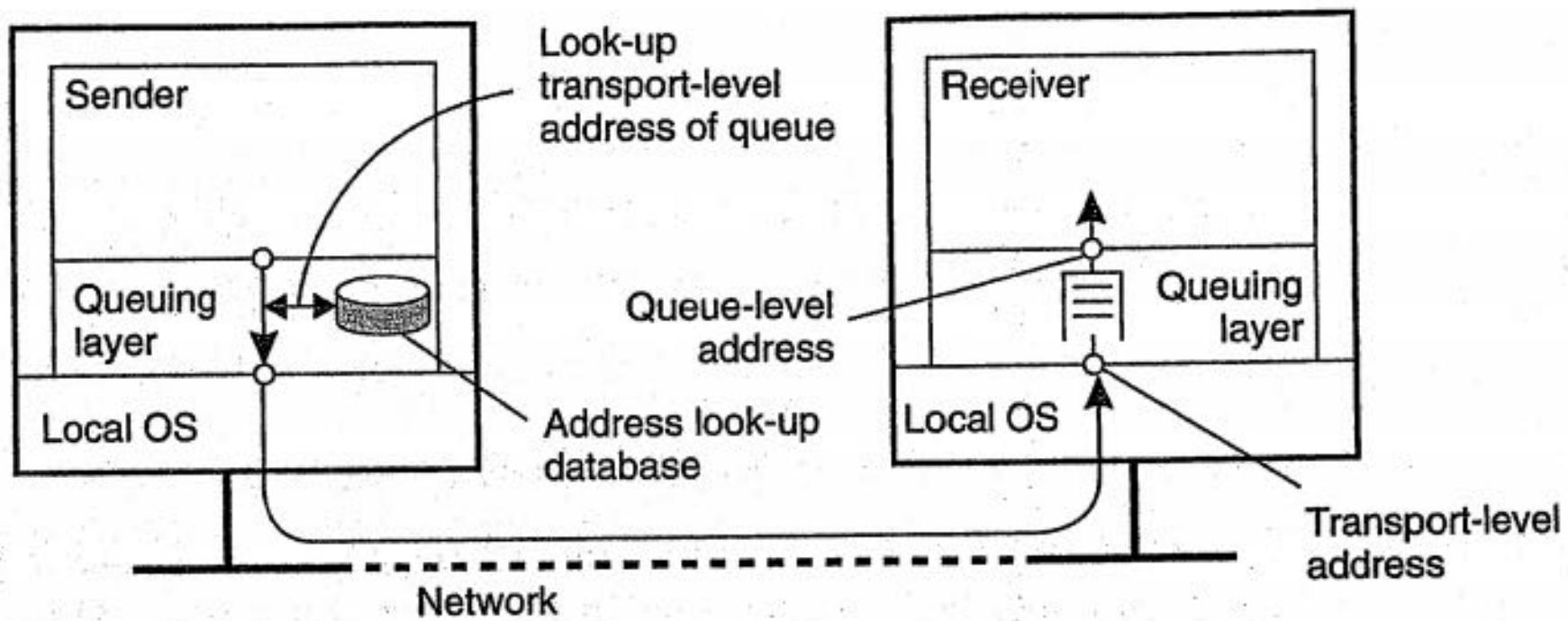
## 4.2. Trao đổi thông tin hướng thông điệp bền vững

- MOM (Message-Oriented Middleware)
- Hệ thống hàng đợi thông điệp hỗ trợ trao đổi thông tin không đồng bộ bền vững.
- Hỗ trợ khả năng lưu trữ trung gian cho thông điệp
- Chấp nhận độ trễ thời gian cao
- VD: hệ thống email

# Mô hình hàng đợi thông điệp

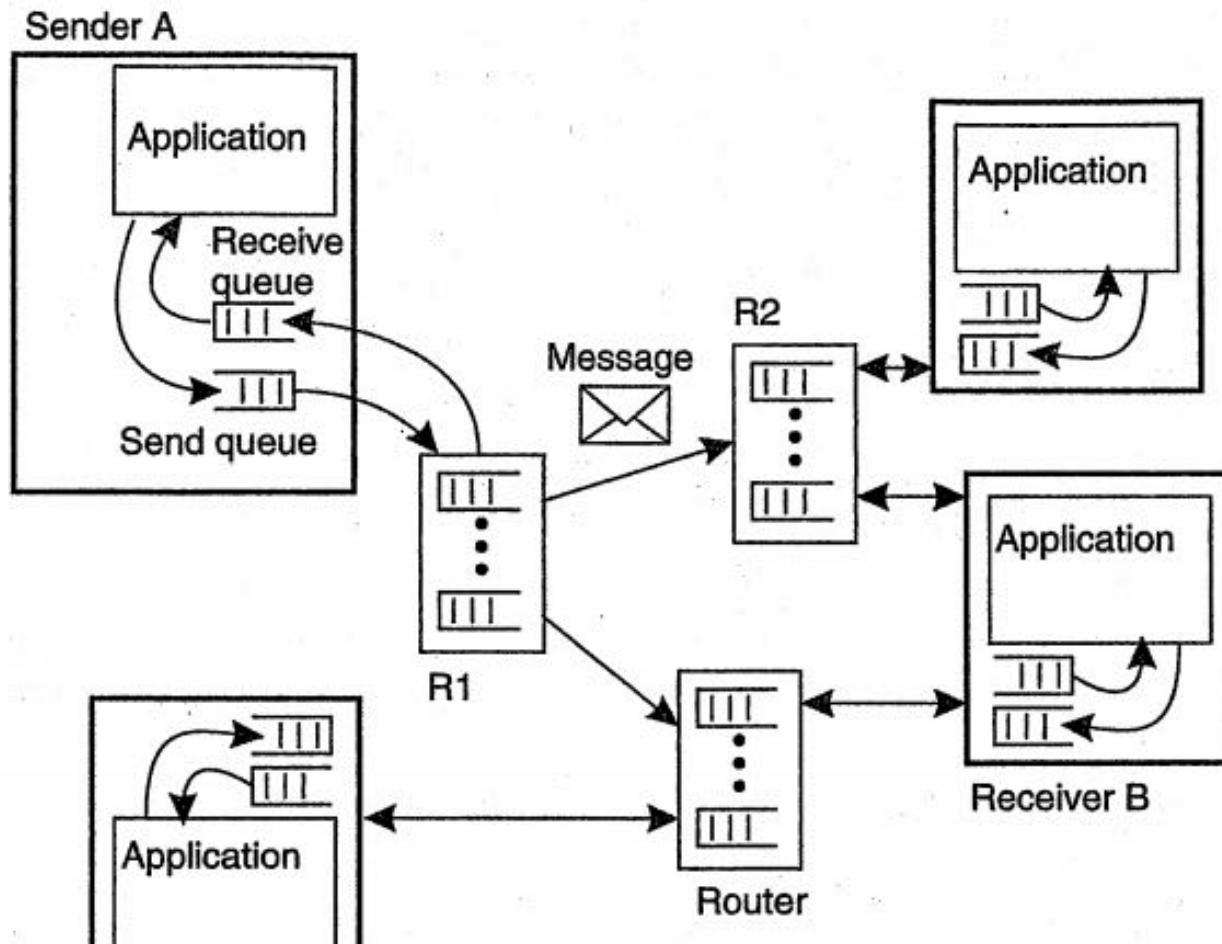


# Định địa chỉ

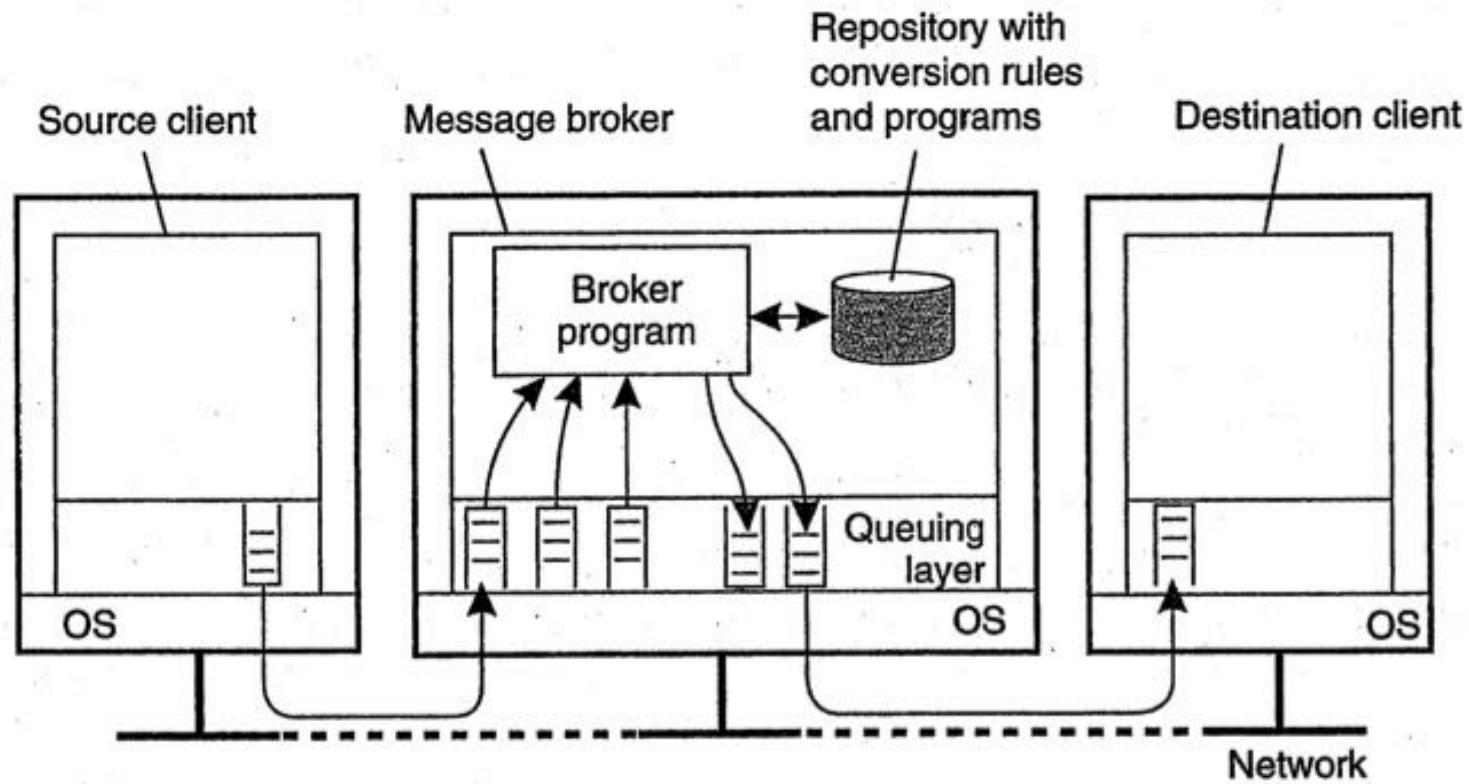


Mối quan hệ giữa định địa chỉ mức hàng đợi và mức mạng

# Hệ thống hàng đợi thông điệp với các routers



# Message Broker



# RabbitMQ

## 1 "Hello World!"

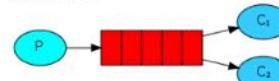
The simplest thing that does something



- > Python
- > Java
- > Ruby
- > PHP
- > C#
- > Javascript
- > Go
- > Elixir
- > Objective-C

## 2 Work queues

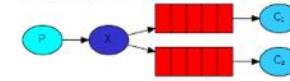
Distributing tasks among workers



- > Python
- > Java
- > Ruby
- > PHP
- > C#
- > Javascript
- > Go
- > Elixir
- > Objective-C

## 3 Publish/Subscribe

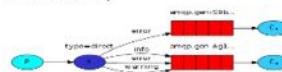
Sending messages to many consumers at once



- > Python
- > Java
- > Ruby
- > PHP
- > C#
- > Javascript
- > Go
- > Elixir
- > Objective-C

## 4 Routing

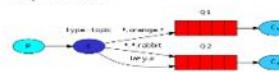
Receiving messages selectively



- > Python
- > Java
- > Ruby
- > PHP
- > C#
- > Javascript
- > Go
- > Elixir
- > Objective-C

## 5 Topics

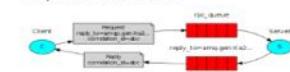
Receiving messages based on a pattern



- > Python
- > Java
- > Ruby
- > PHP
- > C#
- > Javascript
- > Go
- > Elixir
- > Objective-C

## 6 RPC

Remote procedure call implementation



- > Python
- > Java
- > Ruby
- > PHP
- > C#
- > Javascript
- > Go
- > Elixir

# 5. Trao đổi thông tin hướng dòng

- 5.1. Hỗ trợ cho phương tiện truyền thông liên tục
- 5.2. Dòng và QoS
- 5.3. Đồng bộ hóa dòng

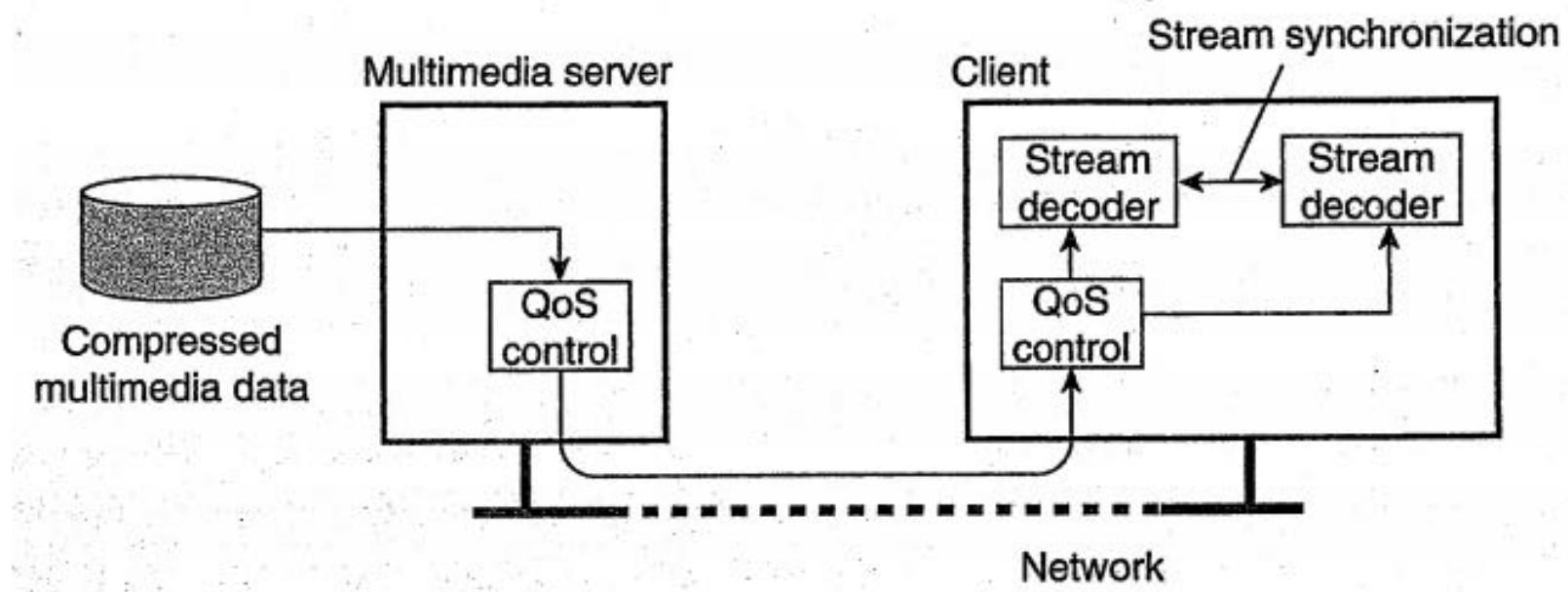
## 5.1. Hỗ trợ cho phương tiện truyền thông liên tục

- Phương tiện truyền đạt thông tin
  - Lưu trữ
  - Truyền tin
  - Biểu diễn (màn hình, v.v...)
- Phương tiện truyền thông liên tục/rồi rác

# Dòng dữ liệu

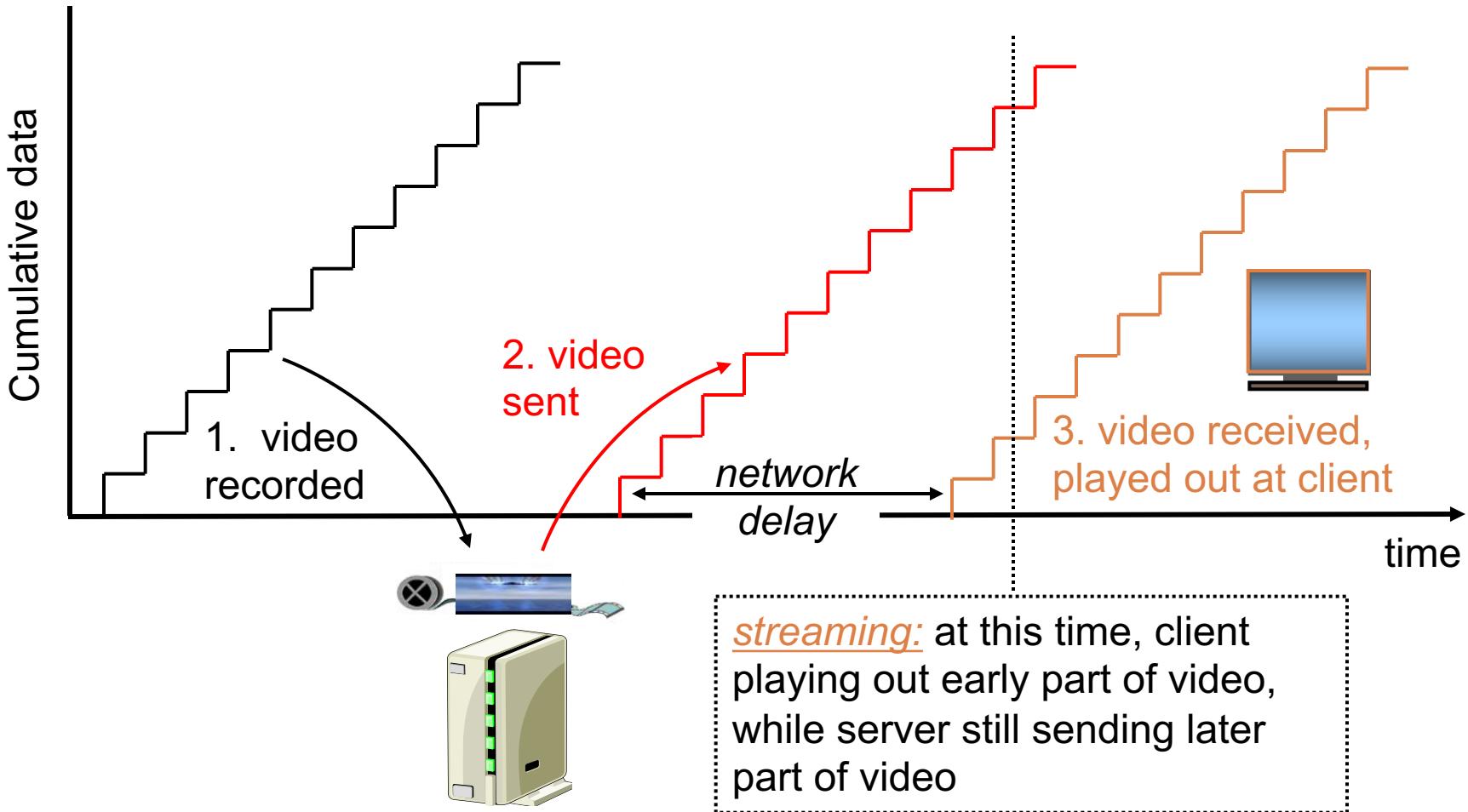
- Chuỗi các đơn vị dữ liệu liên tục
- Truyền tin không đồng bộ
- Truyền tin đồng bộ
- Truyền tin đẳng thời
- Dòng dữ liệu đơn & phức
- Dòng dữ liệu thời gian thực
- Vấn đề
  - ▣ Nén dữ liệu
  - ▣ Kiểm soát chất lượng đường truyền
  - ▣ Đồng bộ hóa

# Dòng dữ liệu (cont.)

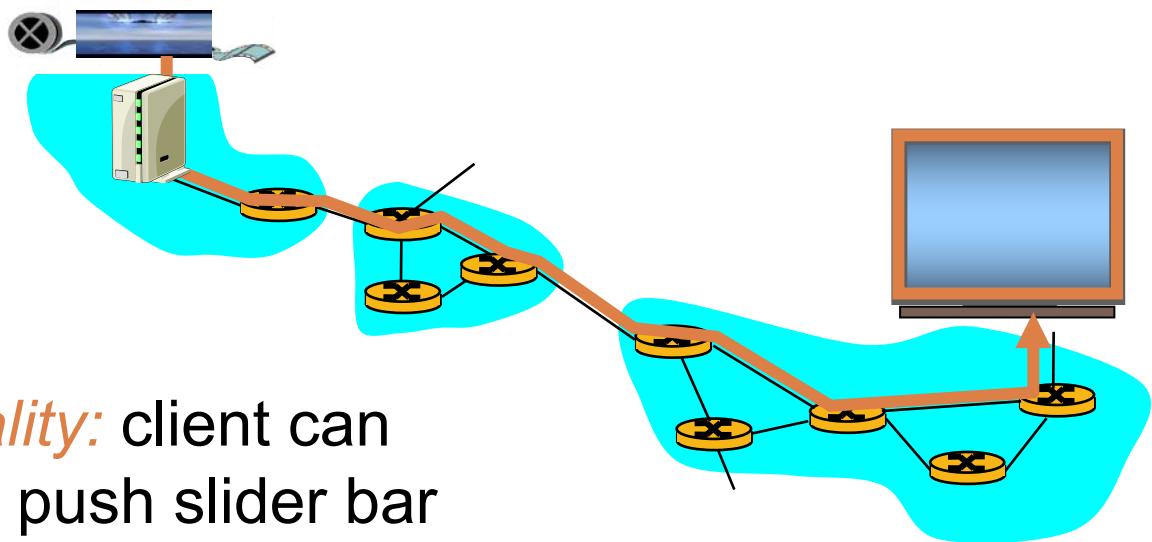


## Kiến trúc chung truyền dữ liệu đa phương tiện qua mạng

# Truyền dòng dữ liệu đa phương tiện lưu trữ

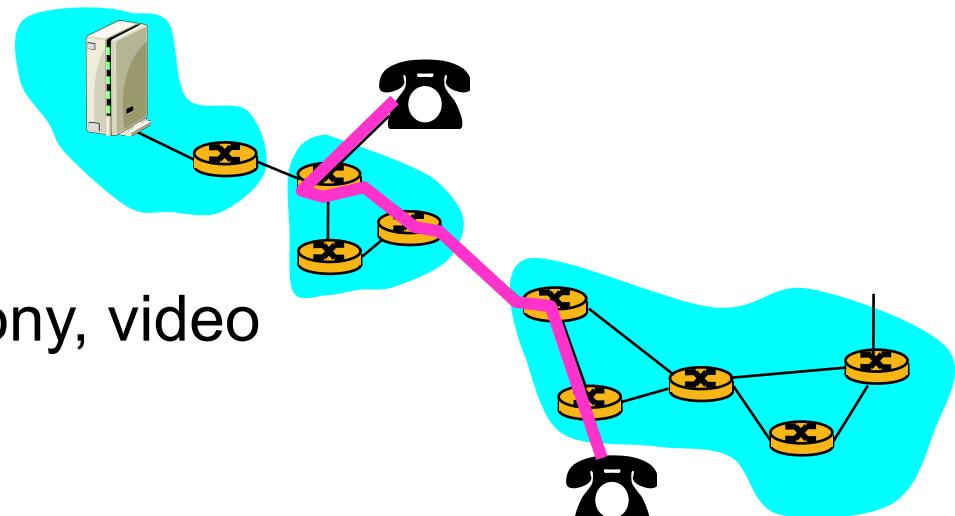


# Truyền dòng dữ liệu đa phương tiện: Có tương tác



- **VCR-like functionality:** client can pause, rewind, FF, push slider bar
  - 10 sec initial delay OK
  - 1-2 sec until command effect OK
- timing constraint for still-to-be transmitted data: in time for playout

# Real-Time Interactive Multimedia



- **applications:** IP telephony, video conference, distributed interactive worlds
- **end-end delay requirements:**
  - audio: < 150 msec good, < 400 msec OK
    - includes application-level (packetization) and network delays
    - higher delays noticeable, impair interactivity
- **session initialization**
  - how does callee advertise its IP address, port number, encoding algorithms?

# Nén dữ liệu audio

- analog signal sampled at constant rate
  - telephone: 8,000 samples/sec
  - CD music: 44,100 samples/sec
- each sample quantized, i.e., rounded
  - e.g.,  $2^8=256$  possible quantized values
- each quantized value represented by bits
  - 8 bits for 256 values
- example: 8,000 samples/sec, 256 quantized values --> 64,000 bps
- receiver converts bits back to analog signal:
  - some quality reduction

## Example rates

- CD: 1.411 Mbps
- MP3: 96, 128, 160 kbps
- Internet telephony: 5.3 kbps and up

# Nén dữ liệu video

- video: sequence of images displayed at constant rate
  - e.g. 24 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- redundancy
  - spatial (within image)
  - temporal (from one image to next)

## Examples:

- MPEG 1 (CD-ROM) 1.5 Mbps
- MPEG2 (DVD) 3-6 Mbps
- MPEG4 (often used in Internet, < 1 Mbps)

## Research:

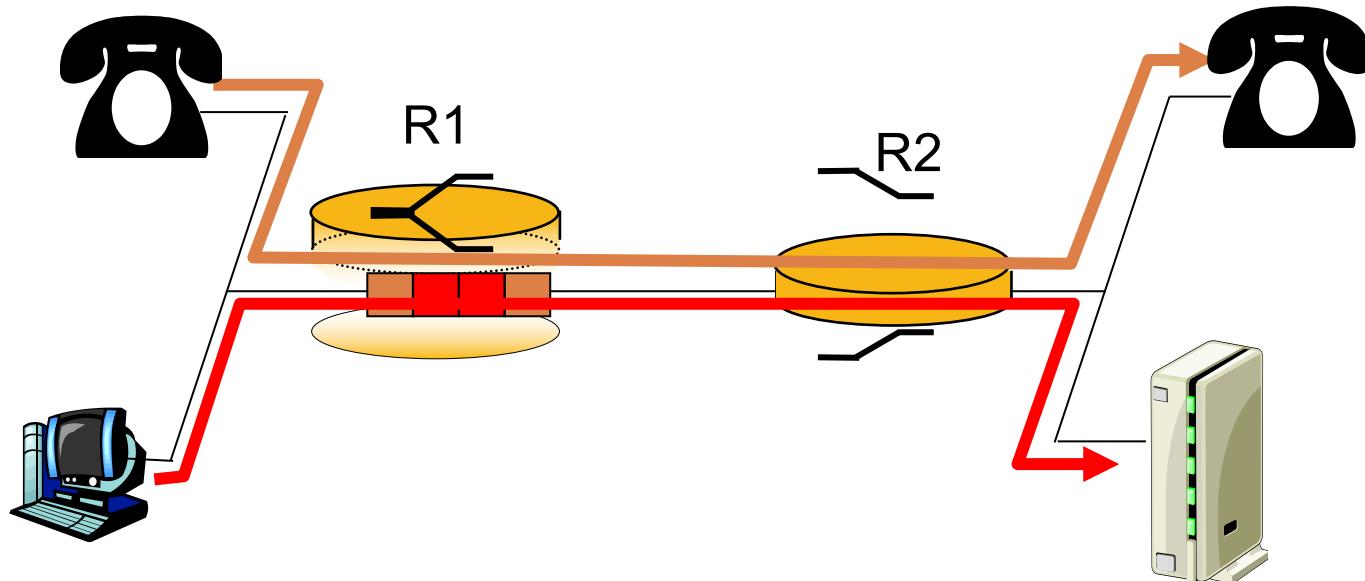
- layered (scalable) video
  - adapt layers to available bandwidth

## 5.2. Dòng dữ liệu và QoS

- Quality of Service (QoS):
  - bit-rate,
  - delay
  - e2e delay
  - jitter
  - round-trip delay
- Dựa trên tầng IP
  - Đơn giản, best-effort

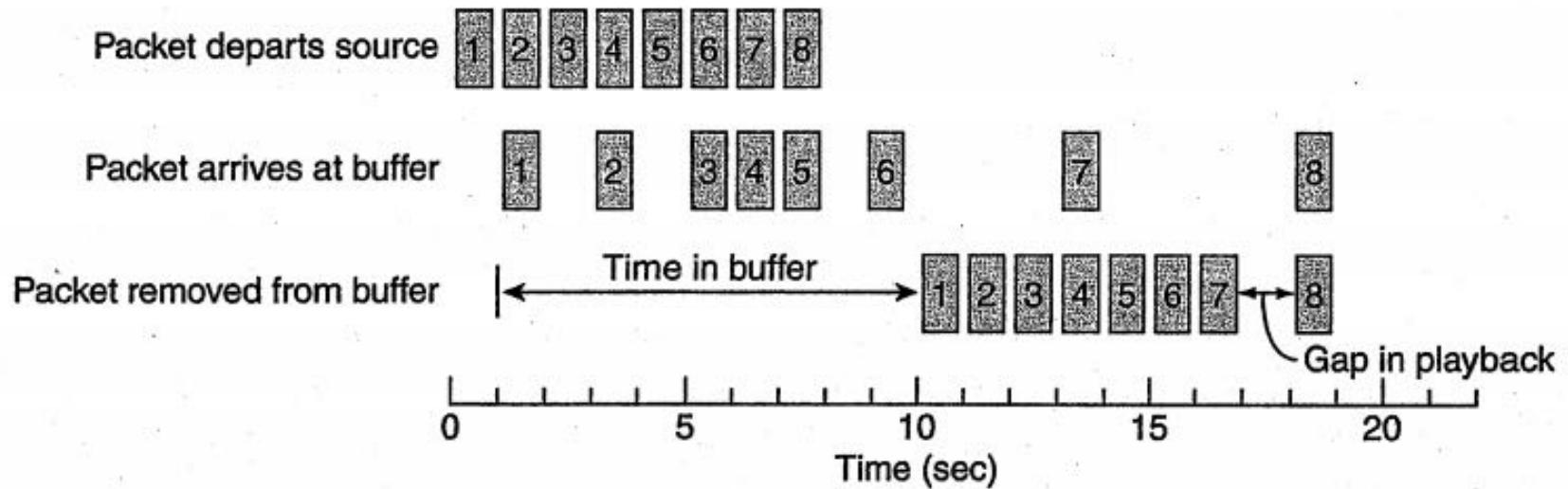
# Thực thi QoS

- Differentiated services



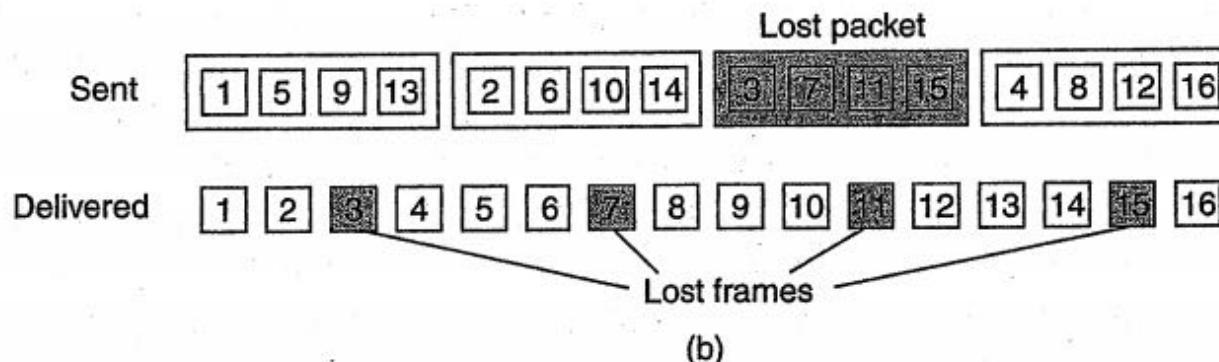
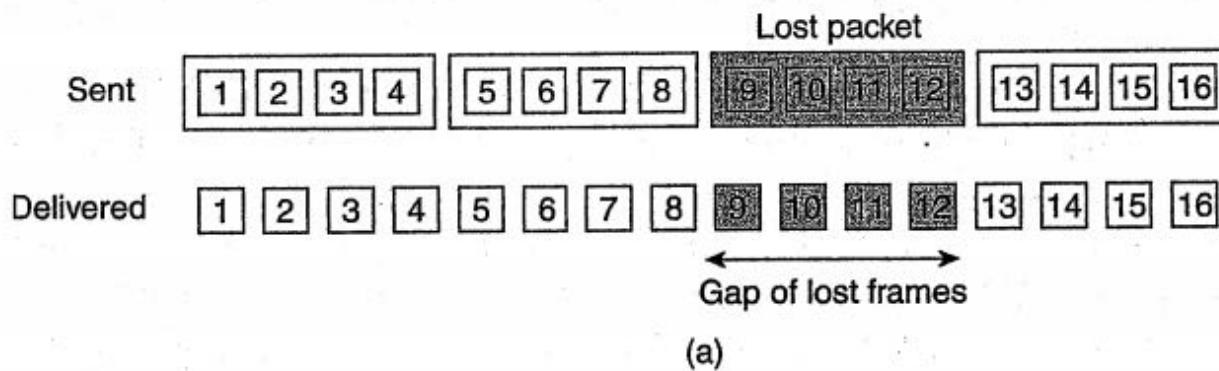
# Thực thi QoS (cont.)

- Sử dụng bộ đếm để giảm jitter

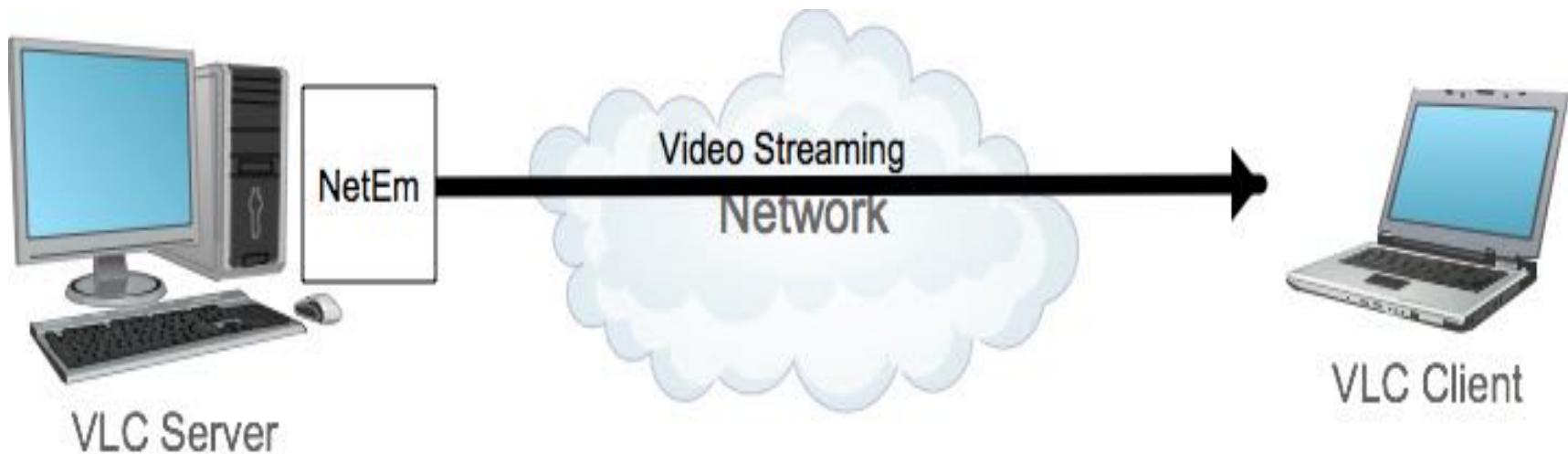


# Thực thi QoS (cont.)

- Forward error correction (FEC)
  - ▣ Interleaved transmission



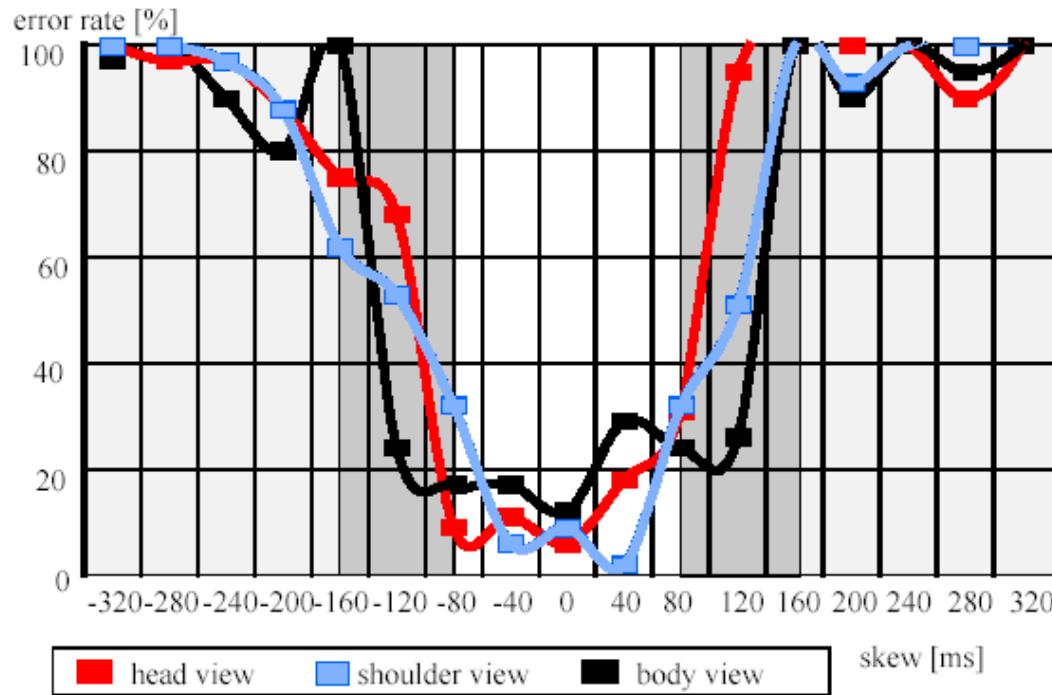
# Labwork



## 5.3. Đồng bộ hóa dòng

- Nhu cầu đồng bộ hóa các dòng dữ liệu
- 2 kiểu đồng bộ:
  - Đồng bộ *dòng dữ liệu rời rạc* và *dòng dữ liệu liên tục*.
  - Đồng bộ *2 dòng dữ liệu liên tục*.
- Dựa trên đơn vị dữ liệu

# Lip Synchronization



Not  
tolerable

Not  
detectable

Not  
tolerable

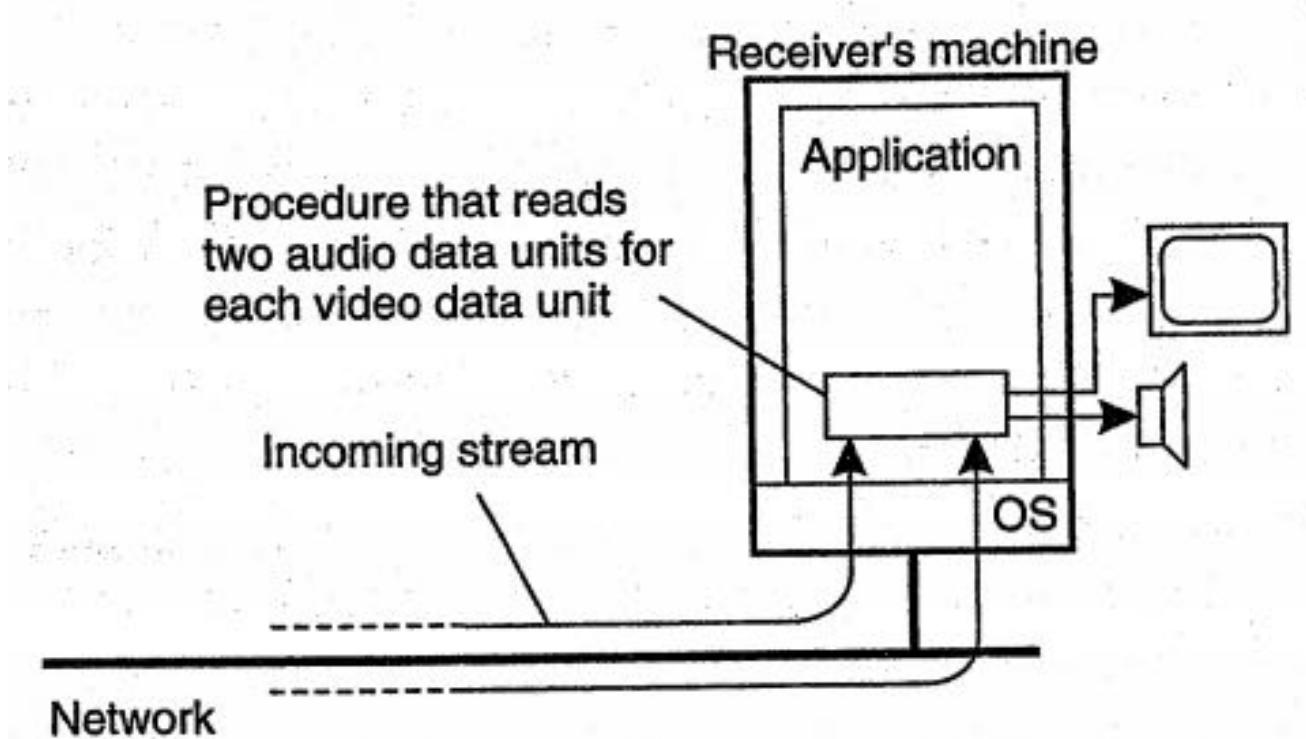
Tolerable

Tolerable

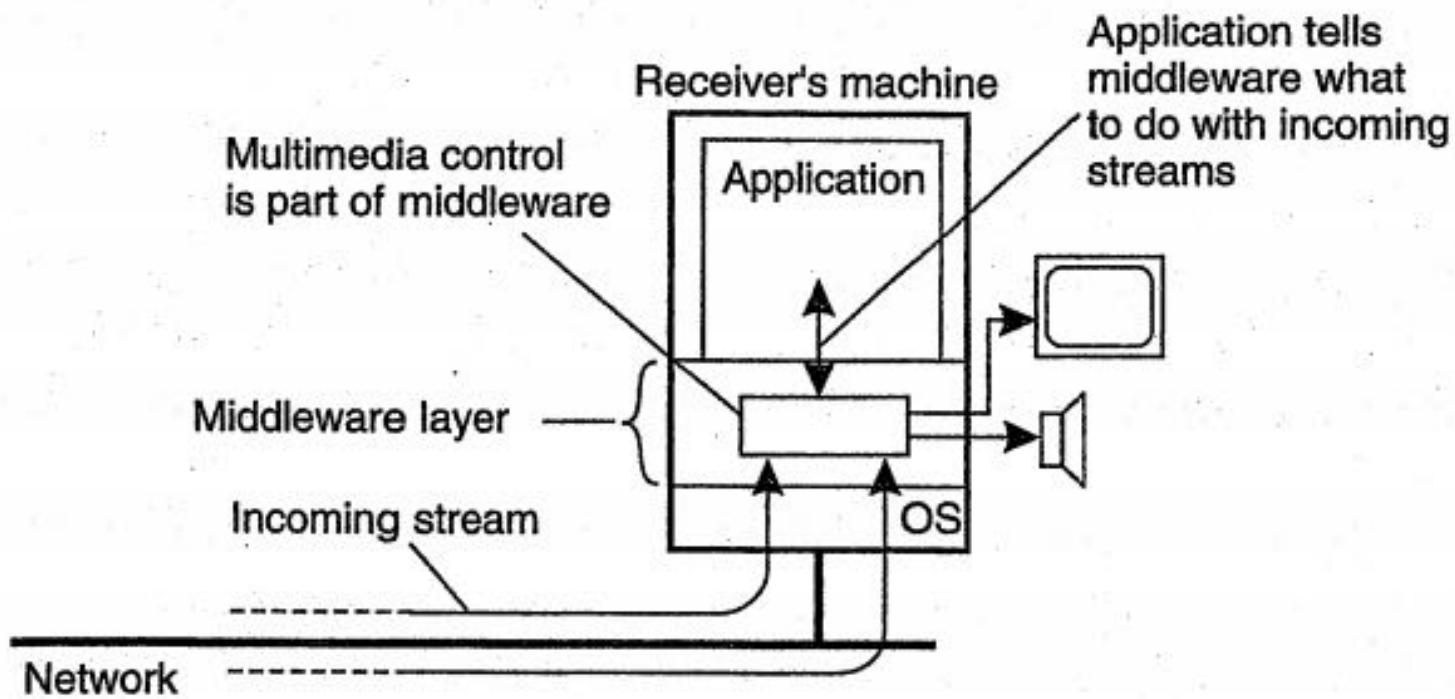
# Cơ chế đồng bộ hóa

- ❑ Cơ chế cơ bản để đồng bộ 2 dòng
- ❑ Sự phân bố các cơ chế đó trong môi trường mạng

# Đồng bộ hóa ở mức đơn vị dữ liệu



# Đồng bộ hóa có hỗ trợ của những giao diện mức cao





25  
YEARS ANNIVERSARY  
**SOICT**

**VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG**  
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

## Câu hỏi?

