Lập trình Socket

Mục đích

Chương này nhằm giới thiệu về cách thức xây dựng ứng dụng Client-Server trên mạng TCP/IP theo cả hai chế độ Có nối kết (TCP) và Không nối kết (UDP).

Yêu cầu

Sau khi hoàn tất chương này, bạn có thể:

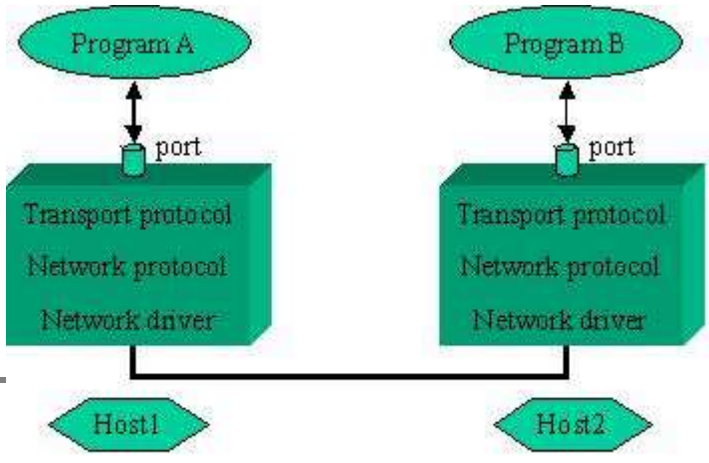
- Giải thích được Socket là gì, vai trò của số hiệu cổng (Port) và địa chỉ IP trong cơ chế Socket.
- Phân biệt được sự khác biệt của hai loại Protocol TCP và UDP.
- Trình bày được các bước xây dựng một chương trình Client-Server sử dụng Socket làm phương tiện giao tiếp trong cả hai chế độ: Có nối kết và không nối kết.
- Liệt kê các lớp hỗ trợ lập trình Socket của Java.
- Xây dựng được các chương trình Client sử dụng Sokcet ở chế độ có nối kết bằng ngôn ngữ Java.
- Xây dựng được các chương trình Server sử dụng Sokcet ở chế độ có nối kết phục vụ tuần tự và phục vụ song song bằng ngôn ngữ Java.
- Xây dựng được các chương trình Client-Server sử dụng Sokcet ở chế độ không nối kết bằng ngôn ngữ Java.
- Tự xây dựng được các Protocol mới cho ứng dụng của mình.

1.1. Giới thiệu về socket

1.1.1. Giới thiệu

Socket là một giao diện lập trình ứng d ụng (API-Application Programming Interface). Nó được giới thiệu lần đầu tiên trong ấn bản UNIX - BSD 4.2. dướ i dạng các hàm h ệ thống theo cú pháp ngôn ngữ C (socket(), bind(), connect(), send(), receive(), read(), write(), close() ,..). Ngày nay, Socket được hỗ trợ trong hầu hết các hệ điều hành như MS Windows, Linux và được sử dụ ng trong nhiều ngôn ngữ lập trình khác nhau: như C, C++, Java, Visual Basic, Visual C++, . . .

Socket cho phép thiết lập các kênh giao tiếp mà hai đầu kênh được đánh dấu bởi hai cổng (port). Thông qua các cổng này một quá trình có thể nhận và gởi dữ liệu với các quá trình khác.



Hình 4.1 – Mô hình Socket

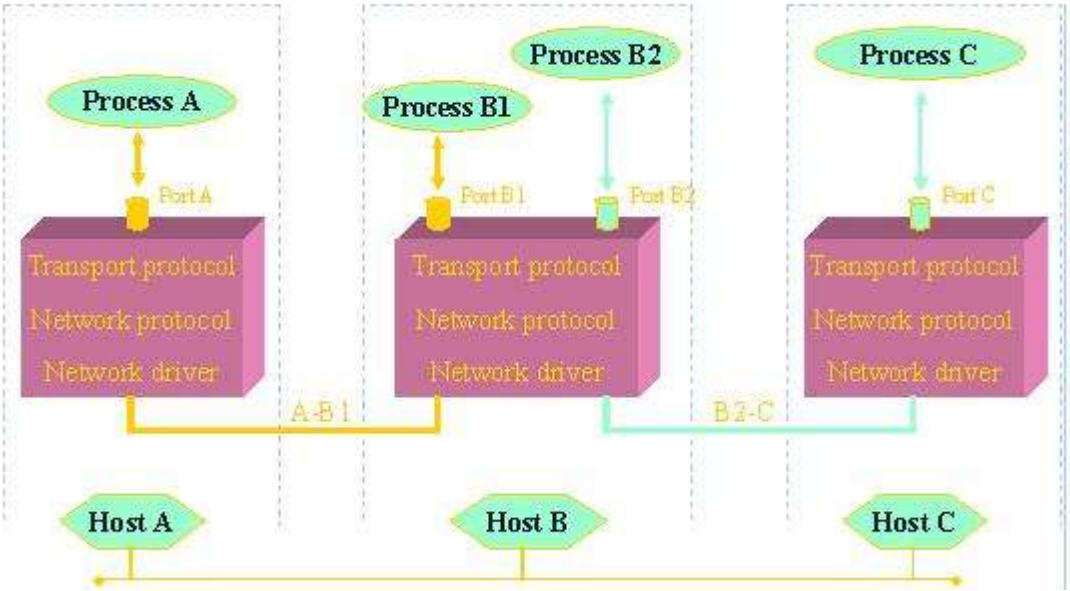
Có hai kiểu socket:

1. Socket kiểu AF_UNIX chỉ cho phép giao tiếp giữa các quá trình trong cùng một máy tính
2. Socket kiểu AF_INET cho phép giao tiếp giữa các quá trình trên những máy tính khác nhau trên mạng.

1.1.2. Số hiệu cổng (Port Number) của socket

Để có thể thự c hiện các cuộc giao tiếp, một trong hai quá trình phải công bố số hiệu cổng của socket mà mình sử dụng. Mỗ i cổng giao tiếp thể hiện một địa chỉ xác định trong hệ thống. Khi quá trình được gán một số hiệu cổng, nó có thể nhận dữ liệu gởi đến cổng này từ các quá trình khác. Quá trình còn lại cũng được yêu cầu tạo ra một socket.

Ngoài số hiệu cổng, hai bên giao tiếp còn phải biết địa chỉ IP của nhau. Địa chỉ IP giúp phân biệt máy tính này với máy tính kia trên mạng TCP/IP. Trong khi số hiệu cổng dùng để phân biệt các quá trình khác nhau trên cùng một máy tính.



Hình 4.2 – Cổng trong Socket

Trong hình trên, địa chỉ của quá trình B1 được xác định bằng 2 thông tin: (Host B, Port B1):

Địa chỉ máy tính có thể là địa chỉ IP dạng 203.162.36.149 hay là địa chỉ theo dạng tên miền như www.cit.ctu.edu.vn

Số hiệu cổng gán cho Socket phải duy nhất trên phạm vi máy tính đó, có giá trị trong khoảng từ 0 đến 65535 (16 bits). Trong đó, các cổng từ 1 đến 1023 được gọi là cổng

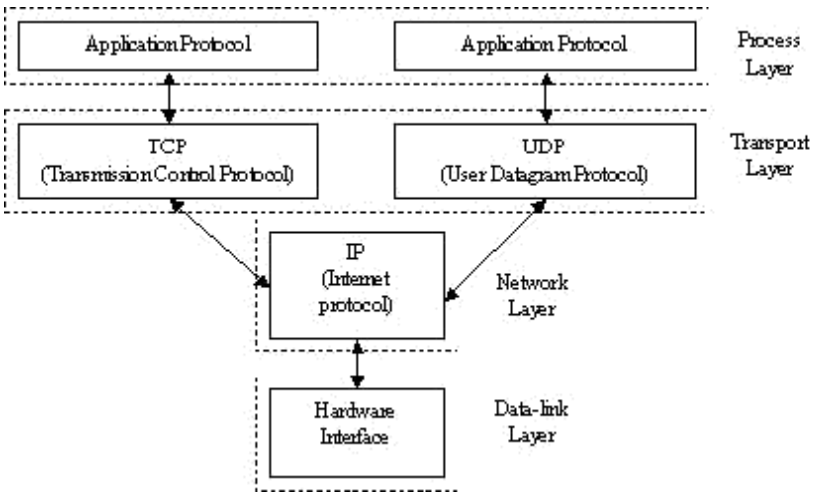
hệ thống được dành riêng cho các quá trình của hệ thống.

Các cổng mặc định của 1 số dịch vụ mạng thông dụng:

Số hiệu cổng	Quá trình hệ thống
7	Dịch vụ Echo
21	Dịch vụ FTP
23	Dịch vụ Telnet
25	Dịch vụ E-mail (SMTP)
80	Dịch vụ Web (HTTP)
110	Dịch vụ E-mail (POP)

1.1.3. Các chế độ giao tiếp

Xét kiến trúc của hệ thống mạng TCP/IP



Hình 4.3 – Bộ giao thức TCP/IP

Tầng vận chuyển giúp chuyển tiếp các thông điệp giữa các chương trình ứng dụng với nhau. Nó có thể hoạt động theo hai chế độ:

- Giao tiếp có nối kết, nếu sử dụng giao thức TCP
- Hoặc giao tiếp không nối kết, nếu sử dụng giao thức UDP

Socket là giao diện giữa chương trình ứng dụng với tầng vận chuyển. Nó cho phép ta chọn giao thức sử dụng ở tầng vận chuyển là TCP hay UDP cho chương trình ứng dụng của mình.

Bảng sau so sánh sự khác biệt giữa hai chế độ giao tiếp có nối kết và không nối kết:

Chế độ có nối kết (TCP)	Chế độ không nối kết (UDP)
• Tồn tại kênh giao tiếp ảo giữa hai bên giao tiếp • Dữ liệu được gởi đi theo chế độ bảo đảm: có kiểm tra lỗi. truyền lại gói tin lỗi hay mất, bảo đảm thứ tự đến của các gói tin . . . • Dữ liệu chính xác, Tốc độ truyền chậm.	• Không tồn tại kênh giao tiếp ảo giữa hai bên giao tiếp • Dữ liệu được gởi đi theo chế độ không bảo đảm: Không kiểm tra lỗi, không phát hiện không truyền lại gói tin bị lỗi hay mất, không bảo đảm thứ tự đến của các gói tin . . . • Dữ liệu không chính xác, tốc độ truyền nhanh. • Thích hợp cho các ứng dụng cần tốc độ, không cần chính xác cao: truyền âm thanh, hình ảnh . . .

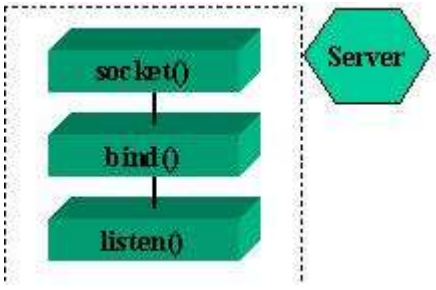
1.2. Xây dựng ứng dụng Client-Server với Socket

Socket là phương tiện hiệu quả để xây dựng các ứng dụng theo kiến trúc Client-Server. Các ứng dụng trên mạng Internet như Web, Email, FTP là các ví dụ điển hình.

Phần này trình bày các bướ c cơ bản trong việc xây dựng các ứng dụng Client-Server sử dụng Socket làm phương tiện giao tiếp theo cả hai chế độ: Có nối kết và không nối kết.

1.2.1. Mô hình Client-Server sử dụng Socket ở chế độ có nối kết (TCP)

Giai đoạn 1: Server tạo Socket, gán số hiệu cổng và lắng nghe yêu cầu nối kết.

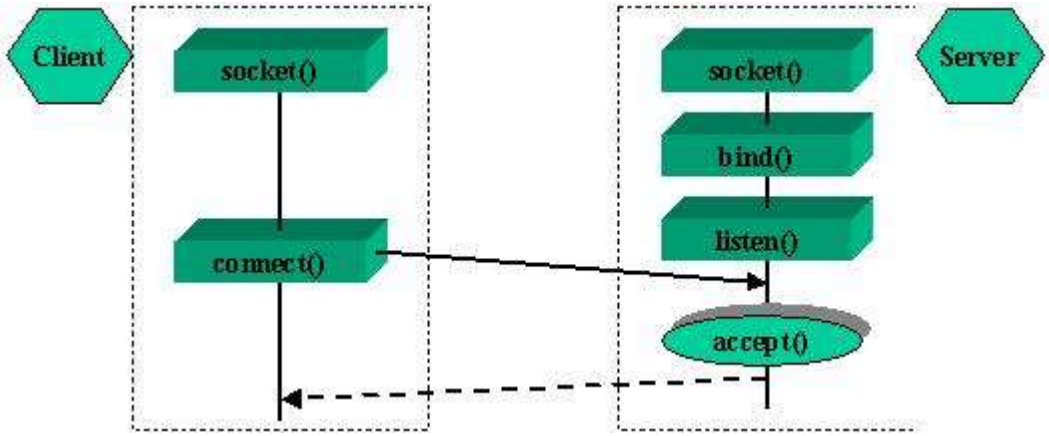


- socket(): Server yêu cầu tạo một socket để có thể sử dụng các dịch vụ của tầng vận chuyển.
- bind(): Server yêu cầu gán số hiệu cổng (port) cho socket.

- listen(): Server lắng nghe các yêu cầu nối kết từ các client trên cổng đã được gán.

Server sẵn sàng phục vụ Client.

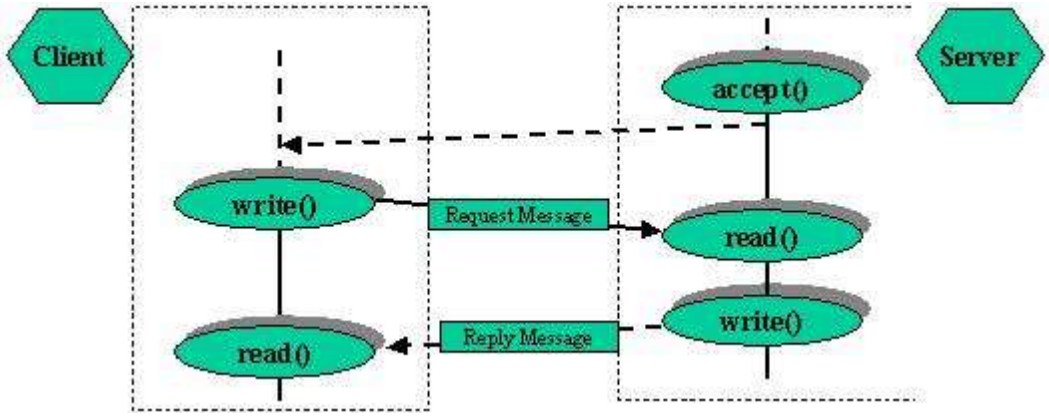
Giai đoạn 2: Client tạo Socket, yêu cầu thiết lập một nối kết với Server.



- socket(): Client yêu cầu tạo một socket để có thể sử dụng các dịch vụ của tầng vận chuyển, thông thường hệ thống tự động gán một số hiệu cổng còn rảnh cho socket của Client.
- connect(): Client gởi yêu cầu nối kết đến server có địa chỉ IP và Port xác định.

- accept(): Server chấp nhận nối kết của client, khi đó một kênh giao tiếp ảo được hình thành, Client và server có thể trao đổi thông tin với nhau thông qua kênh ảo này.

Giai đoạn 3: Trao đổi thông tin giữa Client và Server.

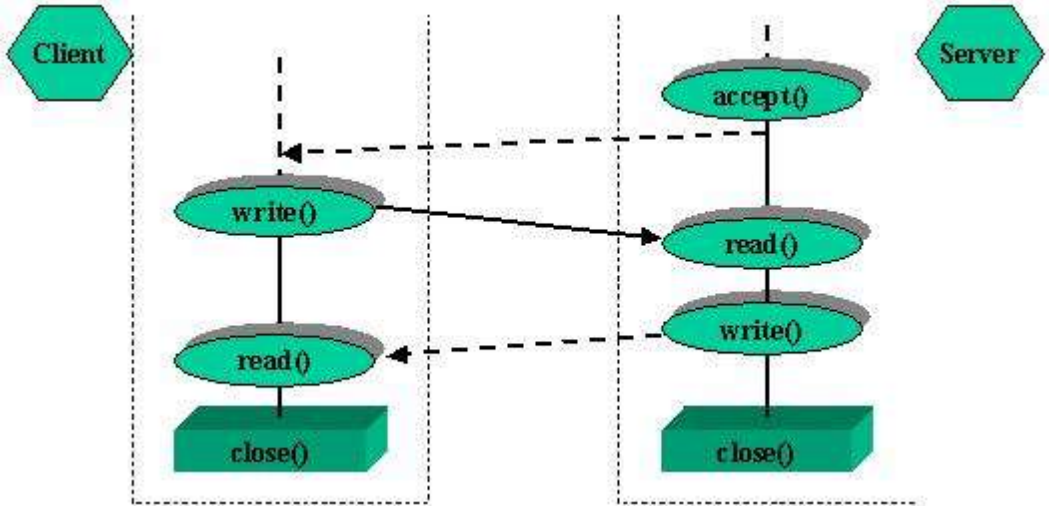


- Sau khi chấp nhận yêu cầu nối kết, thông thường server thực hiện lệnh read() và nghẽn cho đến khi có thông điệp yêu cầu (Request Message) từ client gởi đến.
- Server phân tích và thực thi yêu cầu. Kết quả sẽ được gởi về client bằng lệnh write().
- Sau khi gởi yêu cầu bằng lệnh write(), client chờ nhận thông điệp kết quả (ReplyMessage) từ server bằng lệnh read().

Trong giai đoạn này, việc trao đổi thông tin giữa Client và Server phải tuân thủ giao thức của ứng d ụng (D ạng thức và ý nghĩa của các thông điệp, qui tắc bắt tay, đồng bộ hóa,...). Thông thường Client sẽ là người gởi yêu cầu đến Server trước.

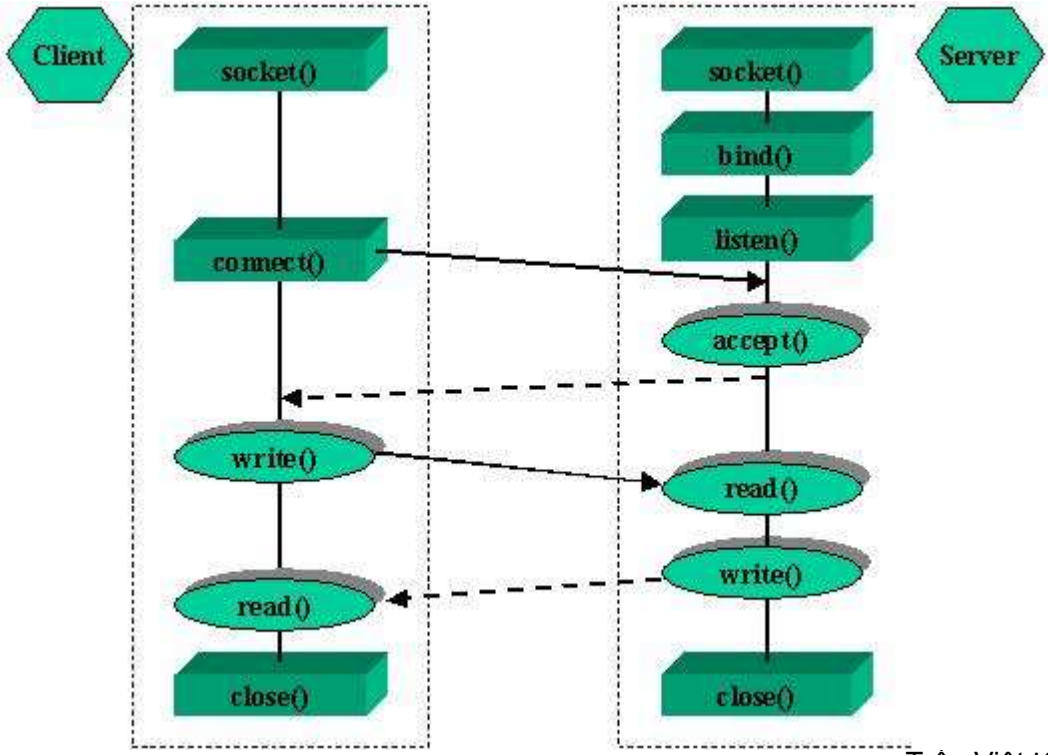
Nếu chúng ta phát triển ứng dụng theo các Protocol đã định nghĩa sẵn, chúng ta phải tham khảo và tuân thủ đúng những qui định của giao thức. Bạn có thể tìm đọc mô tả chi tiết của các Protocol đã được chuẩn hóa trong các tài liệu RFC (Request For Comments). Ngược lại, nếu chúng ta phát triển một ứ ng dụng Client-Server riêng của mình, thì công việc đầu tiên chúng ta phải thực hiện là đi xây dựng Protocol cho ứng dụng.

Giai đoạn 4: Kết thúc phiên làm việc.



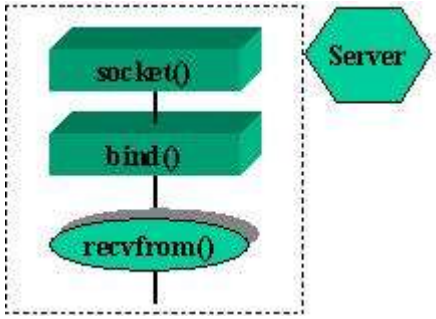
- Các câu lệnh read(), write() có thể được thưc hiện nhiều lần (ký hiệu bằng hình ellipse).
- Kênh ảo sẽ bị xóa khi Server hoặc Client đóng socket bằng lệnh close().

Như vậy toàn bộ tiến trình diễn ra như sau:



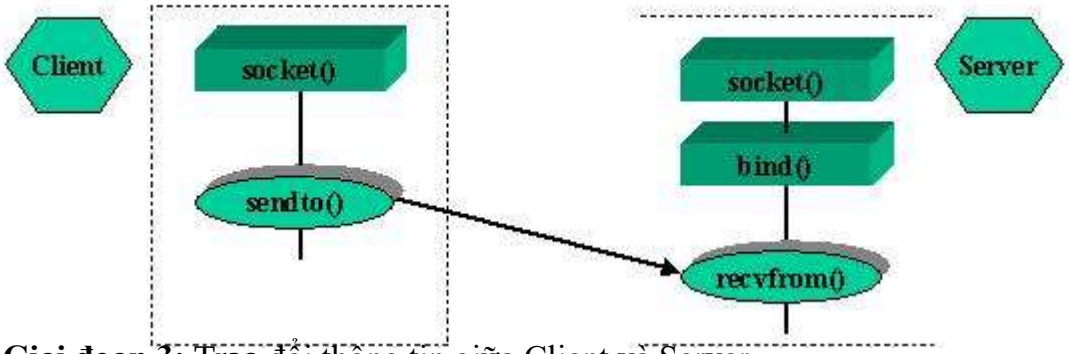
1.2.2. Mô hình Client-Server sử dụng Socket ở chế độ không nối kết (UDP)

Giai đoạn 1: Server tạo Socket - gán số hiệu cổng.

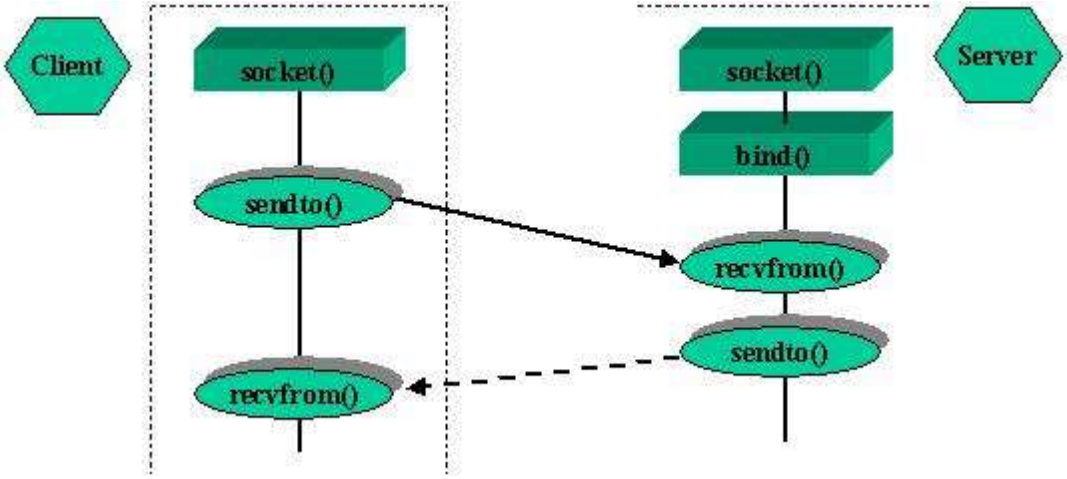


- socket(): Server yêu cầu tạo một socket để có thể sử dụng các dịch vụ của tầng vận chuyển.
- bind(): Server yêu cầu gán số hiệu cổng cho socket..

Giai đoạn 2: Client tạo Socket.



Giai đoạn 3: Trao đổi thông tin giữa Client và Server.



Sau khi tạo Socket xong, Client và Server có thể trao đổi thông tin qua lại với nhau thông qua hai hàm sendto() và recvfrom(). Đơn vị dữ liệu trao đổi giữa Client và Server là các **Datagram Package** (Gói tin thư tín). Protocol của ứng dụng phải định nghĩa khuôn dạng và ý nghĩa của các Datagram Package. Mỗi Datagram Package có chứa thông tin về địa chỉ người gởi và người nhận (IP, Port).

1.3. Socket dưới ngôn ngữ Java

Java hỗ trợ lập trình mạng thông qua các lớp trong gói **java.net**. Một số lớp tiêu biểu được dùng cho lập trình Client-Server sử dụng socket làm phương tiện giao tiếp như:

- InetAddress: Lớp này quản lý địa chỉ Internet bao gồm địa chỉ IP và tên máy tính.
- Socket: Hỗ trợ các phương thức liên quan đến Socket cho chương trình Client ở chế độ có nối kết.
- ServerSocket: Hỗ trợ các phương thức liên quan đến Socket cho chương trình Server ở chế độ có nối kết.
- DatagramSocket: Hỗ trợ các phương thức liên quan đến Socket ở chế độ không nối kết cho cả Client và Server.
- DatagramPacket: Lớp cài đặt gói tin dạng thư tín người dùng (Datagram Packet) trong giao tiếp giữa Client và Server ở chế độ không nối kết.

1.3.1. Xây dựng chương trình Client ở chế độ có nối kết

Các bước tổng quát:

1. Mở một socket nối kết đến server đã biết địa chỉ IP (hay tên miền) và số hiệu cổng.
2. Lấy InputStream và OutputStream gán với Socket.
3. Tham khảo Protocol của dịch vụ để định dạng đúng dữ liệu trao đổi với Server.
4. Trao đổi dữ liệu với Server nhờ vào các InputStream và OutputStream.
5. Đóng Socket trước khi kết thúc chương trình.

1.3.1.1. Lớp java.net.Socket

Lớp Socket hỗ trợ các phương thức cần thiết để xây dựng các chương trình client sử dụng socket ở chế độ có nối kết. Dưới đây là một số phương thức thường dùng để xây dựng Client:

public Socket(String HostName, int PortNumber) throws IOException

Phương thức này dùng để nối kết đến một server có tên là HostName, cổng là PortNumber. Nếu nối kết thành công, một kênh ảo sẽ được hình thành giữa Client và Server.

- HostName: Địa chỉ IP hoặc tên logic theo dạng tên miền.
- PortNumber: có giả trị từ 0 ..65535

Ví dụ: Mở socket và nối kết đến Web Server của khoa Công Nghệ Thông Tin, Đại Học Cần Thơ:

Socket s = new Socket("www.cit.ctu.edu.vn",80);
Hoặc: Socket s = new Socket("203.162.36.149",80);

public InputStream getInputStream()

Phương thức này trả về InputStream nối với Socket. Chương trình Client dùng InputStream này để nhận dữ liệu từ Server gởi về.

Ví dụ: Lấy InputStream của Socket s:

InputStream is = s.getInputStream();

public OutputStream getOutputStream()

Phương thức này trả về OutputStream nối với Socket. Chương trình Client dùng OutputStream này để gởi dữ liệu cho Server.

Ví dụ: Lấy OutputStream của Socket s:

OutputStream os = s.getOutputStream();

public close()

Phương thức này sẽ đóng Socket lại, giải phóng kênh ảo, xóa nối kết giữa Client và Server.

Ví dụ: Đóng Socket

s: s.close();

1.3.1.2. Chương trình TCPEchoClient

Trên hệ thống UNIX, Dịch vụ Echo được thiết kế theo kiến trúc Client-Server sử dụng Socket làm phương tiện giao tiếp. Cổng mặc định dành cho Echo Server là 7, bao gồm cả hai chế độ có nối kết và không nối kết.

Chương trình TCPEchoClient sẽ nối kết đến EchoServer ở chế độ có nối k ết, lần lượt gở i đến Echo Server 10 ký tự từ '0' đến '9', chờ nhận kết quả trả về và hiển thị chúng ra màn hình.

Hãy lưu chương trình sau vào tập tin TCPEchoClient.java

```
import java.io.*;
import java.net.Socket;

public class TCPEchoClient{
   public static void main(String args[]){
      try {
         Socket s = new Socket(args[0],7);            // Nối kết đến Server
         InputStream is = s.getInputStream();         // Lấy InputStream
         OutputStream os = s.getOutputStream(); // Lấy OutputStream
         for (int i='0'; i<='9';i++){   // Gui '0' ->'9' den EchoServer
```

```
            os.write(i);                    // Gởi 1 ký tự sang Server
            int ch = is.read();             // Chờ nhận 1 ký tự từ Server
            System.out.print((char)ch);     // In ký tự nhận được ra màn hình
         }
      }     //try
      catch(IOException ie){
         System.out.println("Loi: Khong tao duoc socket");
      }     //catch
   }   //main
}
```

Biên dịch và thực thi chương trình như sau:

Chương trình này nhận mộ t đối số là địa chỉ IP hay tên miền của máy tính mà ở đó Echo Server đang chạy. Trong hệ thống mạng TCP/IP mỗi máy tính được gán một địa chỉ IP cục bộ là **127.0.0.1** hay có tên là **localhost**. Trong ví dụ trên, chương trình Client nối kết đến Echo Server trên cùng máy với nó.

1.3.2. Xây dựng chương trình Server ở chế độ có nối kết

1.3.2.1. Lớp java.net.ServerSocket

Lớp ServerSocket hỗ trợ các phương thức cần thiết để xây dụng các chương trình Server sử dụ ng socket ở chế độ có nối kết. Dưới đây là một số phương thức thường dùng để xây dựng Server:

public ServerSocket(int PortNumber);

Phương thức này tạo một Socket với số hiệu cổng là PortNumber mà sau đó Server sẽ lắng nghe trên cổng này.

Ví dụ: Tạo socket cho Server với số hiệu cổng là

7: ServerSocket ss = new ServerSocket(7);

public Socket accept()

Phương thức này lắng nghe yêu cầu nối kết của các Client. Đây là mộ t ph ương thức hoạt động ở chế độ nghẽn. Nó sẽ bị nghẽn cho đến khi có một yêu cầu nối kết của client gởi đến.

Khi có yêu cầu nối kết của Client gở i đến, nó sẽ chấp nhận yêu cầu nối kết, trả về một Socket là một đầu của kênh giao tiếp ảo giữa Server và Client yêu cầu nối kết.

Ví dụ: Socket ss chờ nhận yêu cầu nối kết:

Socket s = ss.accept();

Server sau đó sẽ lấy InputStream và OutputStream của Socket mới **s** để giao tiếp với Client.

1.3.2.2. Xây dựng chương trình Server phục vụ tuần tự

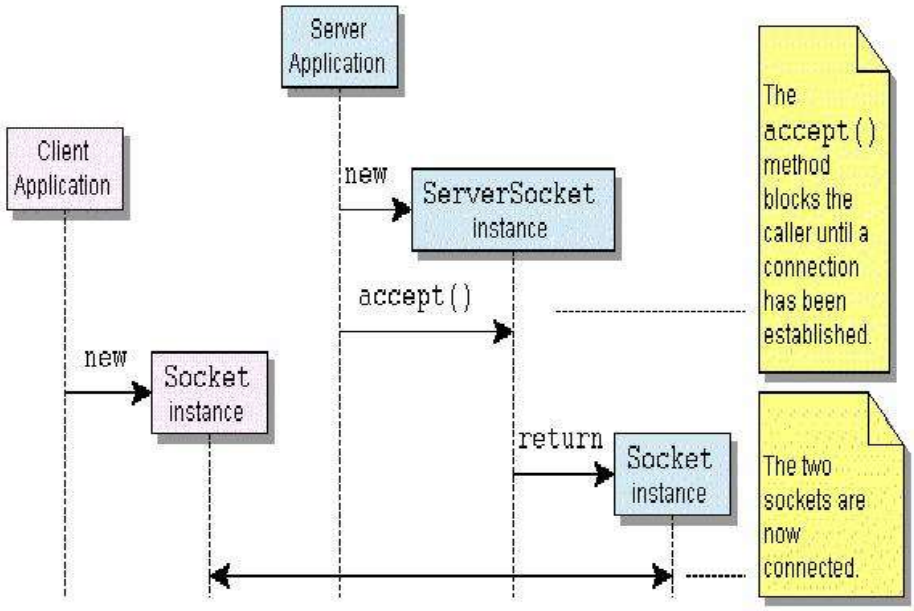
Mộ t Server có thể được cài đặt để phục vụ các Client theo hai cách: phục vụ tuần tự hoặc phục vụ song song.

Trong chế độ phụ c vụ tuần tự, tại một thờ i điểm Server chỉ chấp nhận một yêu cầu nối k ết. Các yêu cầu nối kết của các Client khác đều không được đáp ứng (đưa vào hàng đợi).

Ngược lại trong chế độ phục vụ song song, tại một thời điểm Server chấp nhận nhiều yêu cầu nối kết và phục vụ nhiều Client cùng lúc.

Các bước tổng quát của một Server phục vụ tuần tự

1. Tạo socket và gán số hiệu cổng cho server.
2. Lắng nghe yêu cầu nối kết.
3. Với một yêu cầu nối kết được chấp nhận thực hiện các bước sau:
    - Lấy InputStream và OutputStream gắn với Socket của kênh ảo vừa được hình thành.
    - Lặp lại công việc sau:
        - Chờ nhận các yêu cầu (công việc).
        - Phân tích và thực hiện yêu cầu.
        - Tạo thông điệp trả lời.
        - Gởi thông điệp trả lời về Client.
        - Nếu không còn yêu cầu hoặc Client kết thúc, đóng Socket và quay lại bước 2.



1.3.2.3. Chương trình STCPEchoServer

STCPEchoServer cài đặt một Echo Server phục vụ tuần tự ở chế độ có nối kết. Server lắng nghe trên cổng mặc định số 7.

Hãy lưu chương trình sau vào tập tin STCPEchoServer.java

```
import java.net.*;
import java.io.*;
public class STCPEchoServer {
       public final static int defaultPort = 7;
   public static void main(String[] args) {
       try {
          ServerSocket ss = new ServerSocket(defaultPort);
             while (true) {
                try {
                   Socket s = ss.accept();
                   OutputStream os = s.getOutputStream();
                   InputStream is = s.getInputStream(); int
                   ch=0;
                   while(true) { ch
                      = is.read();
                      if(ch == -1) break;
                      os.write(ch);
                   }
                   s.close();
                }
                catch (IOException e) {
                   System.err.println(" Connection Error: "+e);
                }
             }
       }
       catch (IOException e) {
```

```
            System.err.println(" Server Creation Error:"+e);
        }
    }
}
```

Biên dịch và thực thi chương trình theo cách sau:

Mở một cửa số DOS khác và thực thi chương trình TCPEchoClient ta có kết quả như sau:

Hai chương trình này có thể nằm trên hai máy khác nhau. Trong trường h ợp đ ó khi thực hiện chương trình TCPEchoClient phải chú ý nhập đúng địa chỉ IP của máy tính đang chạy chương trình STCPEchoServer.

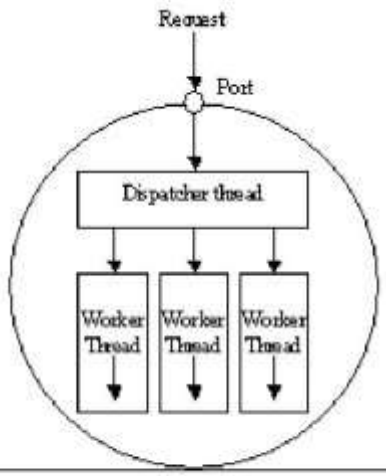
Xem địa chỉ IP của một máy tính Windows bằng lệnh **ipconfig**.

1.3.2.4. Server phục vụ song song

Các bước tổng quát của một Server phục vụ song song

Server phục vụ song song gồm 2 phần thực hiện song song nhau:

- Phần 1: Xử lý các yêu cầu nối kết.
- Phần 2: Xử lý các thông điệp yêu cầu từ khách hàng.



Có cấu trúc như hình sau, trong đ ó **Phần 1** là (Dispatcher Thread), **Phần 2** là các (Worker Thread)

Phần 1: Lặp lại các công việc sau:

- Lắng nghe yêu cầu nối kết của khách

hàng.

- Chấp nhận một yêu cầu nối kết.
  - Tạo kênh giao tiếp ảo mới với khách hàng.
  - Tạo Phần 2 để xử lý các thông điệp yêu cầu của khách hàng.

Phần 2: Lặp lại các công việc sau:

- Chờ nhận thông điệp yêu cầu của khách hàng.
- Phân tích và xử lý yêu cầu.
- Gởi thông điệp trả lời cho khách hàng.

Phần 2 sẽ kết thúc khi kênh ảo bị xóa đi.

V ới mỗi Client, trên Server sẽ có mộ t **Phần 2** để x ử lý yêu cầu của khách hàng. Như vậy tại một thời điểm bất kỳ luôn tồn tại 1 **Phần 1** và 0 hoặc nhiều **Phần 2**.

Do Phần 2 thực thi song song với Phần 1 cho nên nó được thiết kế là một Thread.

1.3.2.5. Chương trình PTCPEchoServer

PTCPEchoServer cài đặt một Echo Server phục vụ song song ở chế độ có nối kết. Server lắng nghe trên cổng mặc định là 7. Chương trình này gồm 2 lớp:

- Lớp TCPEchoServer, cài đặt các chức năng của Phần 1 - xử lý các yêu cầu nối kết của TCPEchoClient.
- Lớp RequestProcessing, là một Thread cài đặt các chức năng của Phần 2 - Xử lý các thông điệp yêu cầu.

Hãy lưu chương trình sau vào tập tin PTCPEchoServer.java

```
import java.net.*;
import java.io.*;
public class PTCPEchoServer {
   public final static int defaultPort = 7; // Cổng mặc
   định public static void main(String[] args) {
      try {
         ServerSocket ss = new ServerSocket(defaultPort); //Tạo socket cho
         server while (true) {
            try {
               Socket s = ss.accept(); // Lắng nghe các yêu cầu nối kết
               RequestProcessing rp = new RequestProcessing(s); // Tạo phần xử lý
               rp.start(); // Khởi động phần xử lý cho Client hiện tại
            }
            catch (IOException e) {
               System.out.println("Connection Error: "+e);
            }
         }
      }
      catch (IOException e) {
         System.err.println("Create Socket Error: "+e);
      }
   }
}
```

```
class RequestProcessing extends Thread {
   Socket channel; //Socket của kênh ảo nối với Client hiện tại
   public RequestProcessing(Socket s){
      channel = s; // Nhận socket của kênh ảo nối với Client
   }
   public void run()
      { try {
         OutputStream os = channel.getOutputStream();
         InputStream is = channel.getInputStream();
         while (true) {
            int n = is.read();
            if (n == -1) break;
            os.write(n);
         }
      }
      catch (IOException e) {   // Nhận ký tự từ Client
                                // Thoát nêu kênh ảo bị xóa
                                // Gởi ký tự nhận được vê Client



         System.err.println("Request Processing Error: "+e);
      }
   }
}
```

Biên dịch và thực thi chương trình như sau:

Sau đó mở thêm 2 của sổ DOS khác để thực thi chương trình TCPEchoClient nối kết tới PTCPEchoServer. Ta sẽ nhận thấy rằng PTCPEchoServer có khả năng phục vụ đồng thời nhiều Client.

1.3.3. Xây dựng chương trình Client - Server ở chế độ không nối kết

Khi sử dụng socket, ta có thể chọn giao thức UDP cho lớp v ận chuyển. UDP viết tắt của User Datagram Protocol, cung cấp cơ chế vận chuyển không bảo đảm và không nối kết trên mạng IP, ngược với giao thức vận chuyển tin cậy, có nối kết TCP.

Cả giao thức TCP và UDP đều phân dữ liệu ra thành các gói tin. Tuy nhiên TCP có thêm vào những tiêu đề (Header) vào trong gói tin để cho phép truy ền lại nhữ ng gói tin thất lạc và tập hợp các gói tin lại theo thứ tự đúng đắn. UDP không cung cấp tính năng
này, nếu một gói tin bị thất lạc hoặc b ị lỗi, nó sẽ không được truyền lại, và thứ tự đến đích của các gói tin cũng không giống như thứ tự lúc nó được gởi đi.

Tuy nhiên, về tốc độ, UDP sẽ truyền nhanh gấp 3 lần TCP. Cho nên chúng thường đượ c dùng trong các ứng dụng đòi hỏi th ờ i gian truyền tải ngắn và không cần tính chính xác cao, ví dụ truyền âm thanh, hình ảnh . . .

Mô hình client - server sử dụng lớp ServerSocket và Socket ở trên sử dụng giao thứ c TCP. Nếu muốn sử dụng mô hình client - server với giao thức UDP, ta sử dụng hai lớp java.net.**DatagramSocket** và java.net.**DatagramPacket**.

DatagramSocket được sử dụng để truyền và nhận các DatagramPacket. Dữ liệu được truy ền đ i là một mảng những byte, chúng được gói vào trong lớp DatagramPacket. Chiều dài của dữ liệu tối đa có thể đưa vào DatagramPacket là khoảng 60.000 byte (phụ thuộc vào dạng đường truyền). Ngoài ra DatagramPacket còn chứa địa chỉ IP và cổng của quá trình gởi và nhận dữ liệu.

Cổng trong giao thức TCP và UDP có th ể trùng nhau. Trên cùng một máy tính, bạn có thể gán cổng 20 cho socket dùng giao thức TCP và cổng 20 cho socket sử dụng giao thức UDP.

1.3.3.1. Lớp DatagramPacket

L ớp này dùng để đóng gói dữ liệu gởi đi. Dưới đây là các phương thức thường sử dụng để thao tác trên dữ liệu truyền / nhận qua DatagramSocket.

public DatagramPacket(byte[] b, int n)

- Là phương thức khởi tạo, cho phép tạo ra một DatagramPacket chứa **n** bytes dữ liệu đầu tiên của mảng **b.** (n phải nhỏ hơn chiều dài của mảng b)
- Phương thức trả về một đối tượng thuộc lớp DatagramePacket

Ví dụ: Tạo DatagramPacket để nhận dữ liệu:

```
byte buff[] = new byte[60000]; // Nơi chứa dữ liệ u nhận được DatagramPacket
inPacket = new Datagrampacket(buff, buff.lenth);
```

public DatagramPacket(byte[] b, int n, InternetAddress ia, int port)

- Phương thức này cho phép tạo một DatagramPacket chứa dữ liệu và cả địa chỉ của máy nhận dữ liệu.
- Phương thức trả về một đối tượng thuộc lớp DatagramePacket

Ví dụ: Tạo DatagramPacket chứa chuỗi "My second UDP Packet", với địa chỉ máy nhận là www.cit.ctu.edu.vn, cổng của quá trình nhận là 19:

```
try {//Địa chỉ Internet của máy nhận
  InetAddress ia = InetAddess.getByName("www.cit.ctu.edu.vn");
  int port = 19; // Cổng của socket nhận
  String s = "My second UDP Packet"; // Dữ liệu gởi đi
 byte[] b = s.getBytes(); // Đổi chuỗi thành mảng bytes //
```

```
   Tạo gói tin gởi đi
    DatagramPacket outPacket = new DatagramPacket(b, b.length, ia, port);
}
catch (UnknownHostException e) {
```

```
    System.err.println(e);
}
```

Các phương thức lấy thông tin trên một DatagramPacket nhận được

Khi nh ận được một DatagramPacket từ một quá trình khác gởi đến, ta có thể lấy thông tin trên DatagramPacket này bằng các phương thức sau:

- public synchronized() InternetAddress getAddress() : Địa chỉ máy gởi
- public synchronized() int getPort() : Cổng của quá trình gởi
- public synchronized() byte[] getData() : Dữ liệu từ gói tin
- public synchronized() int getLength() : Chiều dài của dữ liệu trong gói tin

Các phương thức đặt thông tin cho gói tin gởi

Trước khi gởi một DatagramPacket đi, ta có thể đặt thông tin trên DatagramPacket này bằng các phương thức sau:

- public synchronized() void setAddress(IntermetAddress dis) : Đặt địa chỉ máy nhận.
- public synchronized() void setPort(int port) : Đặt cổng quá trình nhận
- public synchronized() void setData(byte buffer[]) : Đặt dữ liệu gởi
- public synchronized() void setLength(int len) : Đặt chiều dài dữ liệu gởi

1.3.3.2. Lớp DatagramSocket

Lớp này hỗ trợ các phương thức sau để gởi / nhận các DatagramPacket

public DatagramSocket() throws SocketException

- Tạo Socket kiểu không nối kết cho Client. Hệ thống tự động gán số hiệu cổng chưa sử dụng cho socket.

Ví dụ: Tạo một socket không nối kết cho Client:

```
try{
    DatagramSocket ds = new
DatagramSocket(); } catch(SocketException
se) {
    System.out.print("Create DatagramSocket Error: "+se);
}
```

public DatagramSocket(int port) throws SocketException

- Tạo Socket kiểu không nối kết cho Server với số hiệu cổng được xác định trong tham số (port).

Ví dụ: Tạo một socket không nối kết cho Server với số hiệu cổng là 7:

```
try{
    DatagramSocket dp = new DatagramSocket(7);
} catch(SocketException se) {
    System.out.print("Create DatagramSocket Error: "+se);
```

```
}
```

public void send(DatagramPacket dp) throws IOException

- Dùng để gởi một DatagramPacket đi.

Ví dụ: Gởi chuỗi "My second UDP Packet", cho quá trình ở địa chỉ www.cit.ctu.edu.vn, cổng nhận là 19:

```
try {
   DatagramSocket ds = new DatagramSocket(); //Tạo Socket
   //Địa chỉ Internet của máy nhận
   InetAddress ia = InetAddess.getByName("www.cit.ctu.edu.vn");
   int port = 19; //  Cổng của quá trình nhận
   String s = "My second UDP Packet";  // Dữ liệu cần gởi
   byte[] b = s.getBytes();  // Đổi sang mảng bytes
   // Tạo gói tin
   DatagramPacket outPacket = new DatagramPacket(b, b.length, ia, port);
   ds.send(outPacket); // Gởi gói tin đi
  }
  catch (IOException e) {
     System.err.println(e);
  }
```

public synchronized void receive(Datagrampacket dp) throws IOException

- Chờ nhận một DatagramPacket. Quá trình sẽ bị nghẽn cho đến khi có dữ liệu đến.

Ví dụ:

```
try {
   DatagramSocket ds = new DatagramSocket(); //Tạ o Socket
    byte[] b = new byte[60000]; // Nơi chứa dữ liệu nhận được
   DatagramPacket inPacket = new DatagramPacket(b, b.length); // Tạo gói
   tin ds.receive(inPacket); // Chờ nhận gói tin
}
catch (IOException e)
   {
   System.err.println(e)
   ;
}
```

1.3.3.3. Chương trình UDPEchoServer

Chương trình UDPEchoServer cài đặt Echo Server ở ch ế độ không nối k ết, cổ ng mặc định là 7. Chương trình chờ nhận từng gói tin, lấy dữ liệu ra khỏi gói tin nhận được và gởi ngược dữ liệu đó về Client.

Lưu chương trình sau vào tập tin UDPEchoServer.java

```
import
java.net.*;
import java.io.*;
public class UDPEchoServer {
```

```
public final static int port = 7; // Cổng mặc định của
Server public static void main(String[] args) {
   try {
      DatagramSocket ds = new DatagramSocket(port); // T ạo Socket với c ổng
      là 7 byte[] buffer = new byte[6000]; // Vùng đệm chứa dữ liệu cho gói tin
      nhận while(true) { // Tạo gói tin nhận
         DatagramPacket incoming = new
         DatagramPacket(buffer,buffer.length); ds.receive(incoming); // Chờ
         nhận gói tin gởi đến
          // Lấy dữ liệu khỏi gói tin nhận
         String theString = new String(incoming.getData(),0,incoming.getLength());
         // Tạo gói tin gởi chứa dữ liệu vừa nhận được
         DatagramPacket outsending = new DatagramPacket(theString.getBytes(),
         incoming.getLength(),incoming.getAddress(), incoming.getPort());
         ds.send(outsending);
      }
   }
   catch (IOException e)
      {
      System.err.println(e)
      ;
   }
  }
}
```

Biên dịch và thực thi chương trình như sau

1.3.3.4. Chương trình UDPEchoClient

Chương trình này cho phép người sử dụng nhận các chuỗi từ bàn phím, gởi chuỗi sang EchoServer ở chế độ không nối kết ở cổng số 7, chờ nhận và in dữ liệu từ Server gởi về ra màn hình.

Lưu chương trình sau vào tập tin UDPEchoClient.java

```
import java.net.*;
import java.io.*;
public class UDPEchoClient extends Object{
      public final static int serverPort = 7; // Cổng mặc định của Echo Server
   public static void main(String[] args) {
```

```
        try {
          if (args.length ==0) { // Kiểm tra tham số, là địa chỉ của Server
            System.out.print("Syntax: java UDPClient HostName");
          return;
          }
          DatagramSocket ds = new DatagramSocket();   // Tạo DatagramSocket
          InetAddress server = InetAddress.getByName(args[0]);     // Địa chỉ Server
          while(true) {
            InputStreamReader isr = new InputStreamReader(System.in); // Nhập
            BufferedReader br = new BufferedReader(isr);                    // một chuỗi
            String theString = br.readLine();                              // từ bàn phím
            byte[] data = theString.getBytes(); // Đổi chuỗi ra mảng bytes //
            Tạo gói tin gởi
            DatagramPacket dp = new DatagramPacket(data,data.length,server,
            serverPort); ds.send(dp); // Send gói tin sang Echo Server
            byte[] buffer = new byte[6000]; // Vùng đệm cho dữ liệu nhận //
            Gói tin nhận
            DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);
           ds.receive(incoming); // Chờ nhận dữ liệu từ EchoServer gởi về
            // Đổi dữ liệu nhận được dạng mảng bytes ra chuỗi và in ra màn hình
            System.out.println(new String(incoming.getData(), 0, incoming.getLength()));
          }
        }
        catch (IOException e) {
          System.err.println(e);
        }
      }
}
```

Biên dịch và thực thi chương trình như sau:

Chú ý, khi thực hiện chương trình UDPEchoClient phải đưa vào đối số là địa chỉ của máy tính đang thự c thi chương trình UDPEchoServer. Trong ví dụ trên, Server và Client cùng ch ạy trên một máy nên địa ch ỉ củ a UDPEchoServer là ***localhost*** (hay 127.0.0.1). Nếu UDPEchoServer chạy trên máy tính khác thì khi thực thi, ta phải biết được địa chỉ IP của máy tính đó và cung cấp vào đố i số của chương trình. Chẳng hạn, khi UDPEchoServer đang phục vụ trên máy tính ở địa chỉ 172.18.250.211, ta sẽ thực thi UDPEchoClient theo cú pháp sau:

```
java UDPEchoClient 172.18.250.211
```

1.4. Bài tập áp dụng

Chủ đề 1: Client ở chế độ có nối kết

- **Mục đích**:

  Viết các chương trình Client nối kết đến các server theo các Protocol chuẩn.

- **Yêu cầu**

  Sinh viên thực hiện các bài tập sau

  - **Bài 1** : Viết chương trình nhận đối số là một URL. Nối kết đến Web Server trong URL nhận được, lấy trang web về và in ra màn hình theo dạng textfile (html).

Chủ đề 2: Client - Server chế độ có nối kết

- **Mục đích**:
  - Viết các chương trình Client -Server theo chế độ có nối kết.
- **Yêu cầu**

  Sinh viên thực hiện các bài tập sau, với mỗi bài tập hãy thiết kế một Server phục vụ ở chế độ tuần tự và một Server phục vụ ở chế độ song song.

  - **Bài 1**: Viết chương trình theo mô hình Client-Server sử dụng dụng Socket ở chế độ có nối kết. Trong đó :
    + Server làm nhiệm vụ đọc một ký tự số từ '0' đến '9'.
    (Ví dụ : nhận số 0 : trả về "khong" , 1 : trả về "một" ; ... ... 9 : trả về "chín", nếu nhận ký tự khác s ố thì trả về "Không phải số nguyên"). + Client sẽ nhập vào 1 ký tự, gửi qua Server, nhận kết quả trả về từ Server và thể hiện lên màn hình
  - **Bài 2**: Viết chương trình theo mô hình Client-Server sử dụng Socket ở chế độ có nối kết. Trong đó :
    + Server sẽ nhận các yêu cầu là một chuỗi có khuôn dạng như sau:
    "OP Operant1 Operant2\n"
    Trong đó:
    - OP là một ký tự chỉ phép toán muốn thực hiện: '+','-', '*', '/'.
    - Operant1, Operant2 là đối số của phép toán.
    - Các thành phần trên cách nhau bởi 1 ký tự trắng ' '.
    - Kết thúc yêu cầu bằng ký tự xuống dòng '\n'.

    Mỗi khi server nh ận được một thông điệp nó sẽ thực hiện phép toán:

Operant1 OP Operant2 để cho ra kết quả, sau đó đổi kết quả thành chuỗi và gởi về Client.

+ Client cho phép người dùng nhập các phép toán muốn tính theo cách thức thông thường. Ví dụ: 100+200. Client tạo ra thông điệp yêu cầu theo đúng dạng do Server qui định, mô tả về phép toán mu ốn Server thực thi, rồi gởi sang Server, chờ nhận kết quả trả về và in ra màn hình.

Chủ đề 3: Client-Server ở chế độ không nối kết

- **Mục đích**:
  - Viết các chương trình Client -Server theo chế độ không nối kết.
- **Yêu cầu**
  - **Bài 1** : Viết chương trình Talk theo chế độ không nối kết. Cho phép hai người ngồi trên hai máy tính có thể tán gẫu (chat) với nhau.

CHƯƠNG 5
RPC và RMI

Mục đích

Chương này nhằm giới thiệu cách thức xây dựng các ứng dụng phân tán bằng các cơ chế gọi thủ tục từ xa (RPC - Remote Procedure Call và RMI - Remote Method Invocation)

Yêu cầu

Sau khi hoàn tất chương này, bạn có thể:

- Định nghĩa được ứng dụng phân tán là gì.
- Trình bày được kiến trúc của một ứng dụng phân tán xây dựng theo cơ chế gọi thủ tục từ xa (RPC).
- Trình bày được kiến trúc của một ứng dụng phân tán (hay còn gọi Ứng dụng đối tượng phân tán) xây dựng theo cơ chế RMI của Java.
- Trình bày được các cơ chế liên quan khi xây dựng một ứng dụng theo kiểu RMI.
- Trình bày được cơ chế vận hành của một ứng dụng theo kiểu RMI.

- Giải thích được vai trò rmiregistry server.
- Liệt kê được các lớp của java hỗ trợ xây dựng các ứng dụng kiểu RMI.
- Trình bày chi tiết các bước phải qua khi xây dựng một ứng dụng theo kiểu RMI.
- Biên soạn, biên dịch và thực thi thành công chương trình minh họa Hello.
- Phân tích, thiết kế và cài đặt được các chương trình theo cơ chế RMI để giải quyết các vấn đề cụ thể.

1.1. Lời gọi thủ tục xa (RPC- Remote Procedure Call)

1.1.1. Giới thiệu

Lời gọi thủ tục xa là một cơ chế cho phép một chương trình có thể g ọi thực thi một thủ tục (hay hàm) trên mộ t máy tính khác. Trong chương trình lúc này, tồn tại hai loại thủ tục: thủ tục cục bộ và thủ tục ở xa.

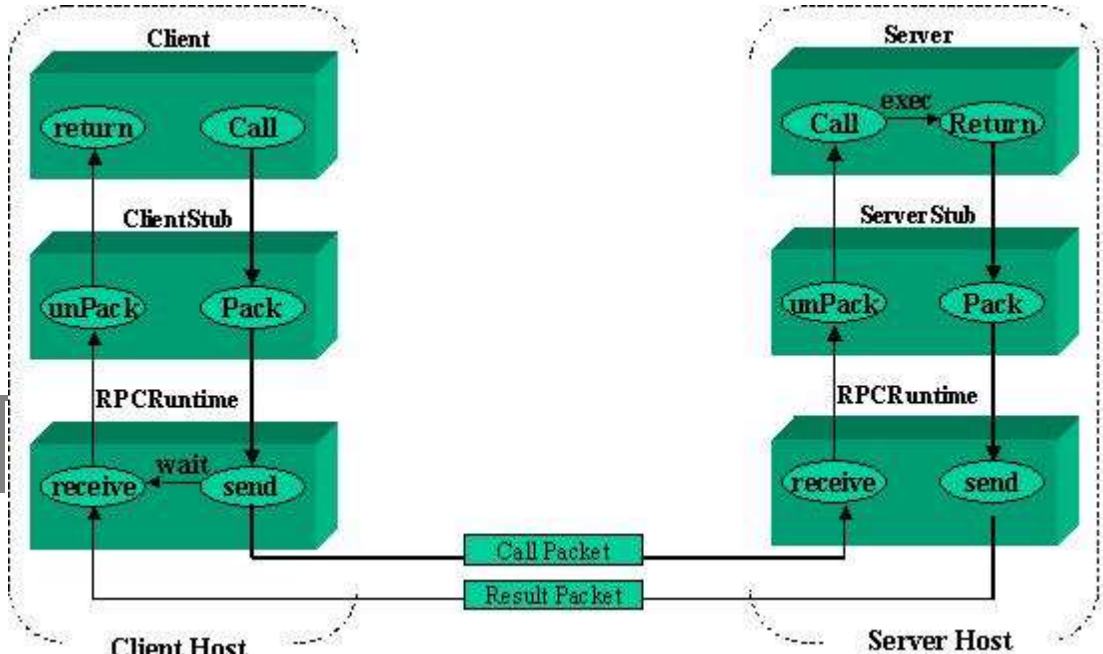
- Thủ tục cục bộ là thủ tục được định nghĩa, cài đặt và thực thi tại máy của chương trình.
- Thủ tục ở xa là thủ tục được định nghĩa, cài đặt và thực thi trên một máy tính khác.

Cú pháp giữa lờ i gọi thủ tục cục bộ và ở xa thì giống nhau. Tuy nhiên, khi một thủ tục ở xa được gọi đến, một thành phần của chương trình gọi là **Stub** sẽ chuyển hướng để kích hoạt một thủ tục tương ứng nằm trên một máy tính khác với máy của chương trình gọi. Đối với người lập trình, việc gọi thủ tục xa và thủ tục cục bộ thì giống nhau về mặt cú pháp. Đây chính là cơ chế cho phép đơn giản hóa việc xây dựng các ứng dụng Client-Server. Trong hệ thống RPC, Server chính là máy tính cung cấp các thủ tục ở xa cho phép các chương trình trên các máy tính khác gọ i thự c hiện. Client chính là các chương trình có thể gọi các thủ tục ở xa trong quá trình tính toán của mình.

Một Client có thể gọ i thủ tục ở xa của nhiều hơn một máy tính. Như vậy sự thực thi của chương trình Client lúc này không còn gói gọn trên mộ t máy tính của Client mà nó trải rộng trên nhiều máy tính khác nhau. Đây chính là mô hình của ứng dụng phân tán (Distributed Application).

1.1.2. Kiến trúc của chương trình Client-Server cài đặt theo cơ chế lời gọi thủ tục xa

Một ứng dụng Client-Server theo cơ chế RPC được xây dựng gồm có sáu phần như sơ đồ dưới đây:



Hình 5.1 Kiến trúc chương trình kiểu RPC

Phần Client là một quá trình người dùng, nơi khởi tạo một lời gọi thủ tục từ xa. Mỗi lời gọi thủ tục ở xa trên phần Client sẽ kích hoạt một thủ tục cục bộ tương ứng nằm trong phần Stub của Client.

Phần ClientStub cung cấp mộ t bộ các hàm cục bộ mà phần Client có th ể gọi. Mỗi một hàm của ClientStub đại diện cho một hàm ở xa được cài đặt và thực thi trên Server.

Mỗi khi một hàm nào đó củ a ClientStub được gọi bởi Client, ClientStub sẽ đóng gói một thông đ iệp để mô tả về th ủ tục ở xa tươ ng ứng mà Client muốn thực thi cùng với các tham số nếu có. Sau đó nó sẽ nhờ hệ thống RPCRuntime cục bộ gởi thông điệp này đến phần Server Stub của Server.

Phần RPCRuntime quản lý việc truy ền thông điệp thông qua mạng giữa máy Client và máy Server. Nó đảm nhận việc truyền lại, báo nhận, chọn đường gói tin và mã hóa thông tin.

RPCRuntime trên máy Client nhận thông điệp yêu cầu từ ClientStub, gởi nó cho RPCRuntime trên máy Server bằng lệnh send(). Sau đó gọi lệnh wait() để chờ kết quả trả về từ Server.

Khi nhận được thông tiệp từ RPCRuntime của Client gởi sang, RPCRuntime bên phía server chuyển thông điệp lên phần ServerStub.

ServerStub mở thông điệp ra xem, xác định hàm ở xa mà Client muốn thực hiện cùng với các tham số của nó. ServerStub gọi một thủ tục tương ứng nằm trên phần Server.

Khi nhận được yêu cầu của ServerStub, Server cho thực thi thủ tục được yêu cầu và gởi kết quả thực thi được cho ServerStub.

ServerStub đóng gói kết quả thực trong một gói tin trả lời, chuyển cho phần RPCRuntime cục bộ để nó gởi sang RPCRuntime của Client .

RPCRuntime bên phía Client chuyển gói tin trả lời nhận được cho phần ClientStub. ClientStub mở thông điệp chứa kết quả thực thi về cho Client tại vị trí phát ra lời gọi thủ tục xa.

Trong các thành phần trên, RPCRuntime được cung cấp b ởi hệ thống. ClientStub và ServerStub có thể tạo ra thủ công (phải lập trình) hay có thể tạo ra bằng các công cụ cung cấp bởi hệ thống.

Cơ chế RPC được hỗ trợ bởi hầu hết các hệ điều hành mạng cũng như các ngôn ngữ lập trình.

1.2. Kích hoạt phương thức xa (RMI- Remote Method Invocation)

1.2.1. Giới thiệu

RMI là một sự cài đặt cơ chế RPC trong ngôn ngữ lập trình h ướng đối tượng Java. Hệ thống RMI cho phép một đối tượng chạy trên một máy ảo Java này có thể kích hoạt một phương thức của một đối tượ ng đang chạy trên một máy ảo Java khác. Đối tượng có phương thức được gọi từ xa gọi là các đối tượng ở xa (Remote Object).

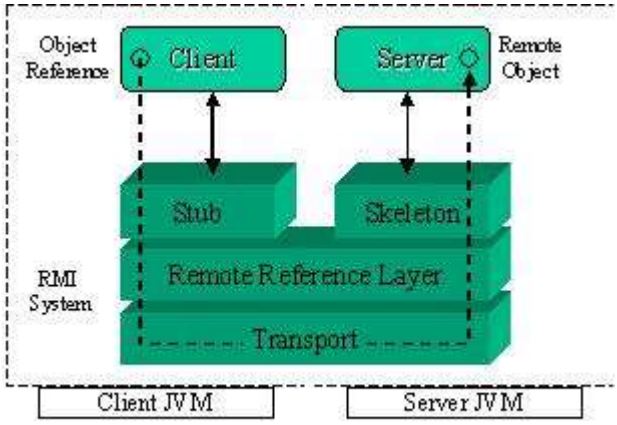
Một ứng dụng RMI thường bao gồm 2 phần phân biệt: Môt chương trình Server và một chương trình Client.

- Chương trình Server tạo một số các Remote Object, tạo các **tham chiếu** (reference) đến chúng và chờ những chương trình Client kích hoạt các phương thức của các Remote Object này.
- Chương trình Client lấy một **tham chiếu** đến một hoặc nhiều Remote Object trên Server và kích hoạt các phương thức từ xa thông qua các tham chiếu.

Một chương trình Client có thể kích hoạt các phương thức ở xa trên một hay nhiều Server. Tức là sự thực thi của chương trình được trải rộng trên nhiều máy tính. Đây chính là đặc điểm củ a các ứng dụng phân tán. Nói cách khác, RMI là cơ chế để xây dựng các ứng dụng phân tán dưới ngôn ngữ Java.

1.2.2. Kiến trúc của chương trình Client-Server theo cơ chế RMI

Kiến trúc một chương trình Client-Server theo cơ chế RMI được mô tả như hình dưới đây:

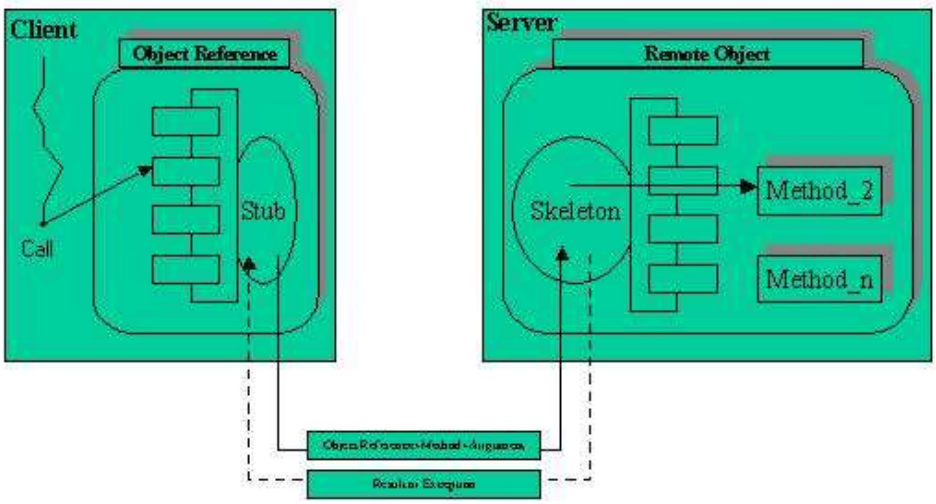


Hình 5.2 - Kiến trúc chương trình kiểu RMI

Trong đó:

- Server là chương trình cung cấp các đối tượng có thể được gọi từ xa.
- Client là chương trình có tham chiếu đến các phương thức của các đối tượng ở xa trên Server.
- Stub chứa các tham chiếu đến các phương thức ở xa trên Server.
- Skeleton đón nhận các tham chiếu từ Stub để kích hoạt phương thức tương ứng trên Server.
- Remote Reference Layer là hệ thống truyền thông của RMI.

Con đường kích hoạt một phương thức ở xa được mô tả như hình dưới đây:



hánh

Hình 5.3 Cơ chế hoạt động của RMI

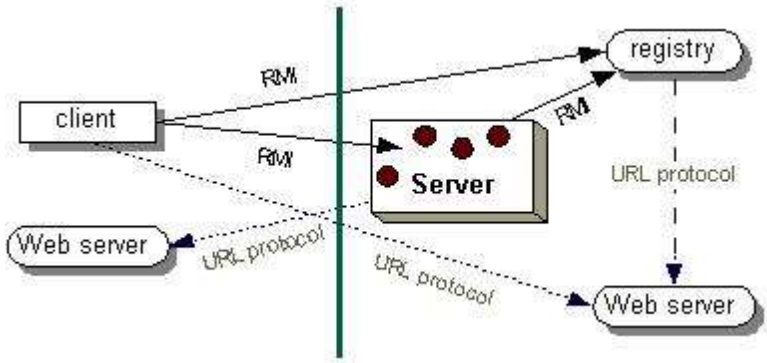
1.2.3. Các cơ chế liên quan trong một ứng dụng đối tượng phân tán

Trong một ứng dụng phân tán cần có các cơ chế sau:

- *Cơ chế định vị đối tượng ở xa (Locate remote objects)*: Cơ chế này xác định cách thức mà chương trình Client có thể lấy được **tham chiếu** (Stub) đến các đối tượng ở xa. Thông thường người ta sử dụng một Dịch vụ danh bạ (Naming Service) lưu giữ các tham khảo đến các đối tượng cho phép gọi từ xa mà Client sau đó có thể tìm kiếm.
- *Cơ chế giao tiếp với các đối tượng ở xa (Communicate with remote objects)*: Chi tiết của cơ chế giao tiếp với các đối tượng ở xa được cài đặt bởi hệ thống RMI.
- *Tải các lớp dạng bytecodes cho các lớp mà nó được chuyển tải qua lại giữa Máy ảo (Load class bytecodes for objects that are passed around)*: Vì RMI cho phép các chương trình gọi phương thức từ xa trao đổi các đối tượng với các phương thức ở xa dưới dạng các tham số hay giá trị trả về của phương thức, nên RMI cần có cơ chế cần thiết để tải mã Bytecodes của các đối tượng từ máy ảo này sang máy ảo khác.

Hình dưới đây mô tả một ứng d ụng phân tán dưới RMI sử dụng dịch vụ danh bạ để lấy các tham khảo của các đối tượng ở xa. Trong đó:

- Server đăng ký tên cho đối tượng có thể được gọi từ xa của mình với Dịch vụ danh bạ (Registry Server).
- Client tìm đối tượng ở xa thông qua tên đã được đăng ký trên Registry Server (looks up) và tiếp đó gọi các phương thức ở xa.



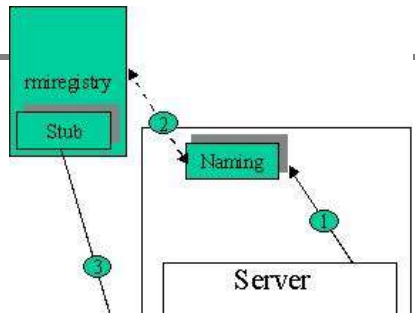
Hình 5.4 Vai trò của dịch vụ tên

- Hình minh họa cũng cho thấy cách thức mà hệ thống RMI sử dụng một WebServer sẵn có để truyền tải mã bytecodes của các lớp qua lại giữa Client và Server.

1.2.4. Cơ chế vận hành của của một ứng dụng Client-Server theo kiểu RMI

Tiến trình vận hành của một ứng dụng Client-Server theo kiểu RMI diễn ra như sau:

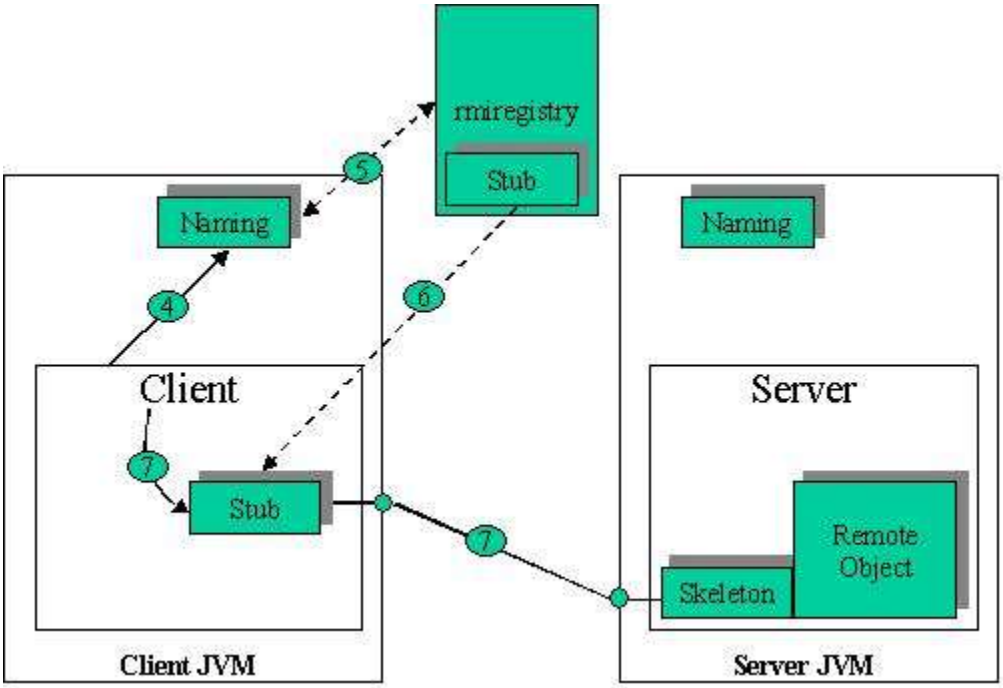
- Bước 1: Server tạo ra các đối tượng



ánh

cho phép gọi từ xa cùng với các Stub và Skeleton của chúng.

- Bước 2: Server sử dụng lớp Naming để đăng ký tên cho một đối tượng từ xa (1).
- Bước 3: Naming đăng ký Stub của đối tượng từ xa với Registry Server (2).
- Bước 4: Registry Server sẵn sàng cung cấp tham thảo đến đối tượng từ xa khi có yêu cầu (3).
- Client yêu cầu Naming định vị đối tượng xa qua tên đã được đăng ký (phương thức lookup) với dịch vụ tên (4).
- Naming tải Stub của đối tượng xa từ dịch vụ tên mà đối tượng xa đã đăng ký về Client (5).
- Cài đặt đối tượng Stub và trả về tham khảo đối tượng xa cho Client (6).
- Client thực thi một lời gọi phương thức xa thông qua đối tượng Stub (7).



1.2.5. Các lớp hỗ trợ chương trình theo kiểu Client-Server trong Java

Java hỗ trợ các lớp cần thiết để cài đặt các ứng dụng Client-Server theo kiểu RMI trong các gói: java.rmi. Trong số đó các lớp thường được dùng là:

- java.rmi.Naming
- java.rmi.RMISecurityManager

- java.rmi.RemoteException;
- java.rmi.server.RemoteObject
- java.rmi.server.RemoteServer
- java.rmi. server.UnicastRemoteObject

1.3. Xây dựng một ứng dụng phân tán với RMI

Xây dựng một ứng dụng phân tán bằng cơ chế RMI gồm các bước sau:

1. Thiết kế và cài đặt các thành phần của ứng dụng.
2. Biên dịch các chương trình nguồn và tạo ra Stub và Skeleton.
3. Tạo các lớp có thể truy xuất từ mạng cần thiết.
4. Khởi tạo ứng dụng

1.3.1. Thiết kế và cài đặt các thành phần của ứng dụng.

Đầu tiên bạn phải xác định lớp nào là lớp cục bộ, lớp nào là lớp được gọi từ xa. Nó bao gồm các bước sau:

- *Định nghĩa các giao diện cho các phương thức ở xa (remote interfaces)*: Một remote interface mô tả các phương thức mà nó có thể được kích hoạt từ xa bởi các Client. Đi cùng với việc định nghĩa Remote Interface là việc xác định các lớp cục bộ làm tham số hay giá trị trả về của các phương thức được gọi từ xa.
- *Cài đặt các đối tượng từ xa (remote objects)*: Các Remote Object phải cài đặt cho một hoặc nhiều Remote Interfaces đã được định nghĩa. Các lớp của Remote Object class cài đặt cho các phương thức được gọi từ xa đã được khai báo trong Remote Interface và có thể định nghĩa và cài đặt cho cả các phương thức được sử dụng cục bộ. Nếu có các lớp làm đối số hay giá trị trả về cho các phương thức được gọi từ xa thì ta cũng định nghĩa và cài đặt chúng.
- *Cài đặt các chương trình Client*: Các chương trình Client có sử dụng các Remote Object có thể được cài đặt ở bất kỳ thời điểm nào sau khi các Remote Interface đã được định nghĩa.

1.3.2. Biên dịch các tập tin nguồn và tạo Stubs và Skeleton

Giai đoạn này gồm 2 bước: Bước thứ nhất là dùng chương trình biên dịch javac để biên dịch các tập tin nguồn như các remote interface, các lớp cài đặt cho các remote interface, lớ p server, lớp client và các lớp liên quan khác. Kế tiếp ta dùng trình biện dịch rmic để tạo ra stub và skeleton cho các đối tượng từ xa từ các lớp cài đặt cho các remote interface.

1.3.3. Tạo các lớp có thể truy xuất từ mạng

Tạo một tập tin chứa tất cả các file có liên quan như các remote interface stub, các lớp hỗ trợ mà chúng cần thiết phải tải về Client và làm cho tập tin này có thể truy cập đến thông qua một Web server.

1.3.4. Thực thi ứng dụng

Thực thi ứng dụng bao gồm việc thực thi rmiregistry server, thực thi server, và thực thi client.

Tóm lại các công việc phải làm là:

- Tạo giao diện (interface) khai báo các phương thức được gọi từ xa của đối tượng.
- Tạo lớp cài đặt (implement) cho giao diện đã được khai báo.
- Viết chương trình Server.
- Viết chương trình Client.
- Dịch các tập tin nguồn theo dạng RMI để tạo ra các lớp tương ứng và stub cho client, skeleton cho server.
- Khởi động dịch vụ registry.
- Thực hiện chương trình Server.
- Thực thi chương trình Client.

1.3.4. Ví dụ minh họa

Trong ví dụ này chúng ta định nghĩa một phương thức String sayHello() được gọi từ xa. Mỗ i khi phương thức này được kích hoạt nó sẽ trả về chuỗi "Hello World" cho Client gọi nó.

Dưới đây là các bước để xây dựng ứng dụng:

Bướ c 01: Tạo giao diện (interface) khai báo các phương thức được gọi từ xa của đối tượng.

- Cú pháp tổng quát:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface InterfaceName extends Remote { ReturnType
   remoteMethodOne() throws RemoteException; ReturnType
   remoteMethodTwo() throws RemoteException;
   . . .
}
```

- Định nghĩa remote interface có tên là HelloItf, có phương thức được gọi từ xa là String sayHello() như sau:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface HelloItf extends Remote {
   String sayHello() throws RemoteException;
}
```

Lưu chương trình này vào tập tin HelloItf.java

Bước 02: Tạo lớp cài đặt (implement) cho giao diện đã được khai báo:

- Cú pháp tổng quát:

```
import java.rmi. server.UnicastRemoteObject;
import java.rmi.RemoteException;
public class RemoteClass extends UnicastRemoteObject
implements InterfaceName {
   public RemoteClass() throws RemoteException {
      super();
      ..... // Implement of Method
   }
   public ReturnType remoteMethodOne() throws RemoteException {
      ..... // Implement of Method
   }
   public ReturnType remoteMethodTwo() throws RemoteException {
      ..... // Definition of Method
   }
}
```

- Định nghĩa lớp cài đặt có tên là Hello cài đặt cho remote interface HelloItf

```
import java.rmi. server.UnicastRemoteObject;
import java.rmi.RemoteException;
public class Hello extends UnicastRemoteObject implements HelloItf {
   public Hello() throws RemoteException {
      super();
   }
   public String sayHello() {
      return "Hello World !";
   }
}
```

Lưu chương trình này vào tập tin Hello.java

Bước 03: Viết chương trình Server:

- Cú pháp tổng quát:

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
public class ServerName {
   public static void main(String args[]) {
      if (System.getSecurityManager() == null) { // Cài đặt cơ chế bảo mật
         System.setSecurityManager(new RMISecurityManager());
      }
      try {
         // Tạo các đối tượng từ xa

RemoteClass remoteObject = new RemoteClass();

// Đăng ký tên cho các đối tượng từ xa
```

```
            Naming.rebind(“RegistryName", remoteObject);
            ...
        }
        catch (Exception e) { System.out.println(”Error:
            . . .” + e);
        }
    }
}
```

- Tạo server có tên HelloServer chứa một đối tượng từ xa obj thuộc lớp cài đặt Hello. Đăng ký tên cho đối tượng obj là HelloObject

```
import java.rmi.Naming;
import java.rmi.RemoteException; import
java.rmi.RMISecurityManager; public class
HelloServer {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
                Hello obj = new Hello();
                Naming.rebind("HelloObject", obj);
                System.out.println("HelloObject is registried");
        }
        catch (Exception e) {
            System.out.println("Error: " + e);
        }
    }
}
```

Lưu chương trình này vào tập tin HelloServer.java

Bước 04: Viết chương trình Client:

- Cú pháp tổng quát:

```
import java.rmi.Naming;
import java.rmi.RemoteException; public
class Client {
    public static void main(String args[]) {
        String remoteObjectURL = "rmi://NameServer/RegistryName”;
        Interfacename object = null;
        try {
            object = (InterfaceName)Naming.lookup(remoteObjectURL);
            object.remoteMethodOne();
        ...
        }
        catch (Exception e) {
                    System.out.println(" Error: ”+ e);
                }
```

```
    }
}
```

- Tạo client có tên là HelloClient, tìm đối tượng HelloObject trên rmiregistry chẳng hạn tại địa chỉ 172.18.211.160. Gọi phương thức sayHello() và in kết quả trả về ra màn hình.

```
import java.rmi.Naming;
import java.rmi.RemoteException;
public class HelloClient {
   public static void main(String args[]) {
      String helloURL = "rmi://172.18.211.160/HelloObject";
      HelloItf object = null;
      try {
         object = (HelloItf)Naming.lookup( helloURL);
         String message = object.sayHello();
         System.out.println(message);
      }
      catch (Exception e) {
         System.out.println("Client Error :" + e);
      }
   }
}
```

Lưu chương trình vào tập tin HelloClient.java

Bước 05: Dịch các tập tin nguồn theo dạng RMI để tạo ra các lớp tương ứng và stub cho client, skeleton cho server:

- Cú pháp tổng quát:

  javac InterfaceName.java RemoteClass.java Server.java Client.java

  (Tạo ra các lớp InterfaceName.class RemoteClass.class Server.class Client.class)

  rmic RemoteClass
  (Tạo ra các lớp cho Skeleton và Stub: RemoteClass_Skel.class RemoteClass_Stub.class)

- Biên dịch các lớp trong Hello:

  javac Hello.java HelloItf.java HelloServer.java HelloClient.java

  rmic Hello.class

Bước 06: Khởi động dịch vụ rmiregistry

- Cú pháp tổng quát:

  start rmiregistry [port]

  Cổng mặc định là 1099.

- Khởi động dịch vụ rmiregistry trên cổng mặc định như sau:

Khi đó rmiregistry server sẽ chạy trên một cửa sổ mới, giữ nguyên cửa sổ này, không đóng nó lại.

Bước 07: Thực hiện chương trình Server

- Cú pháp tổng quát:

  java -Djava.security.policy =UrlOfPolicyFile ServerName

  Trong đó UrlOfPolicyFile là địa chỉ theo dạng URL của tập tin mô tả chính sách về bảo mật mã nguồn của Server (policy file). Nó qui định "ai" (chương trình, máy tính, quá trình trên) sẽ có quyền download các tập tin của nó trong đó có stub. Để đơn giản trong phần này ta cho phép tất cả mọi người đều có quyền download các tập tin của Server. Khi triển khai các ứng dụng thật sự ta phải có các chính sách bảo mật nghiêm ngặt hơn (Tham khảo tài liệu về Security của Java). File policy có dạng như sau:

  ```
  grant {
  // Allow everything for now permission
  java.security.AllPermission; };
  ```

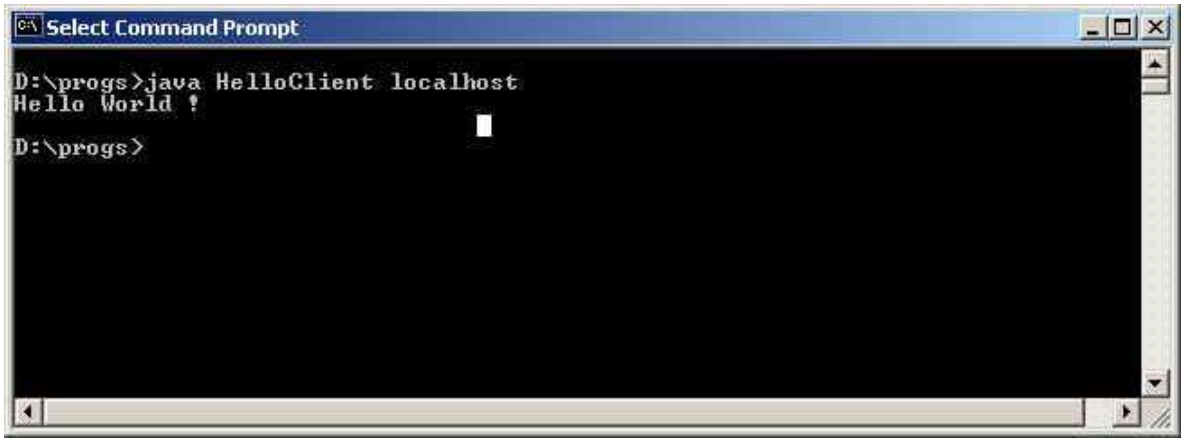
  Lưu nội dung trên vào tập tin có tên **policy.java**

- Thực thi HelloServer với địa tập tin plolicy nằm ở thư mục D:\progs\policy.java

Bước 08: Thực thi chương trình Client:

- Cú pháp tổng quát java ClientName
- Thực thi HelloClient với địa chỉ của rmiregistry đưa vào trong tham số

Để thực thi được chương trình HelloClient cần có hai class nằm cùng thư mục với nó là HelloItf.class và Hello_Stub.class.



1.4. Bài tập áp dụng

- **Mục đích**:

  Xây dựng ứng dụng phân tán theo cơ chế RMI.

- **Yêu cầu**

  Sinh viên thực hiện các bài tập sau:

  - **Bài 1** : Xây dựng một ứng dụng phục vụ việc bán vé máy bay cho các đại lý phân tán ở các tỉnh thành khác nhau. Ứng dụng này có các lớp sau:
    - Lớp chuyến bay: Đại diện cho một chuyến bay
      - Có các thuộc tính: Số hiệu chuyến bay, Ngày giờ bay, Nơi đi, Nơi đến, Thời gian bay, Tổng số ghế, Số lượng ghế đã bán, Số lượng ghế còn trống.
      - Các phương thức trên một chuyến bay: phương thức xem thông tin về chuyến bay, phương thức mua vé,

phương thức trả vé. Để phục vụ cho nhiều đại lý các phương thức trên thuộc loại được gọi từ xa.

- Lớp Server, tạo ra nhiều chuyến bay và duy trì nó để cho phép các đại lý thực hiện các giao dịch trên chuyến bay cụ thể.
- Client là chương trình cho phép mỗi đại lý được quyền xem thông tin về chuyến bay, mua vé, trả vé theo yêu cầu.