Network Programming in Java

Agenda

- Socket-based communication
- Remote method invocation (RMI)

Distributed computations

Today's computing environments are

distributed: computations take place on different network hosts

heterogeneous: the hosts can be running different operating systems

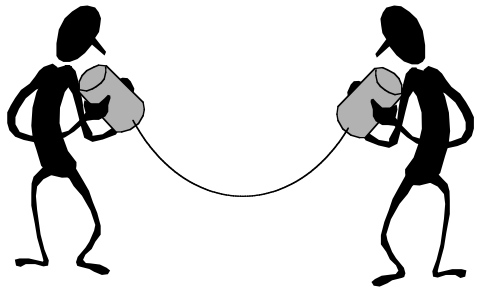
Communication

Java provides two mechanisms for distributed computing:

(1) *Socket-based communication* (`java.net`)
Sockets are the endpoints of two-way connections between two distributed components that communicate with each other.

(2) *Remote method invocation* (RMI) (`java.rmi`)
RMI allows distributed components to be manipulated (almost) as if they were all on the same host.

Socket-based communication



Sockets are the end points of connections between two hosts and can be used to send and receive data.

There are two kinds of sockets: *server sockets* and *client sockets*.

A *server socket* waits for requests from clients.
A *client socket* can be used to send and receive data.

Ports

A server socket listens at a specific *port*.

A port is positive integer less than or equal to 65565.

The port number is necessary to distinguish different server applications running on the same host.

Ports 1 through 1023 are reserved for administrative purposes (e.g. 21 for FTP, 23 for Telnet, 25 for e-mail, and 80 for HTTP).

Server sockets

A server socket is an instance of the `ServerSocket` class and can be created by one of these constructors:

```
ServerSocket(int port)
ServerSocket(int port, int backlog)
```

`port`: port number at which the server will be listening for requests from clients.

`backlog`: the maximum length of the queue of clients waiting to be processed (default is 50).

Server sockets can be created only with Java applications, not applets.

Methods of `ServerSocket`

`Socket accept()`

Waits for a connection request. The thread that executes the method will be blocked until a request is received, at which time the method returns a client socket.

`void close()`

Stops waiting for requests from clients.

Typical use of `ServerSocket`

```
try {
    ServerSocket s = new ServerSocket(port);
    while (true) {
        Socket incoming = s.accept();
        «Handle a client»
        incoming.close();
    }
    s.close();
} catch (IOException e) {
    «Handle exception»
}
```

Client sockets

A client socket is an instance of the `Socket` class and can be obtained in two ways:

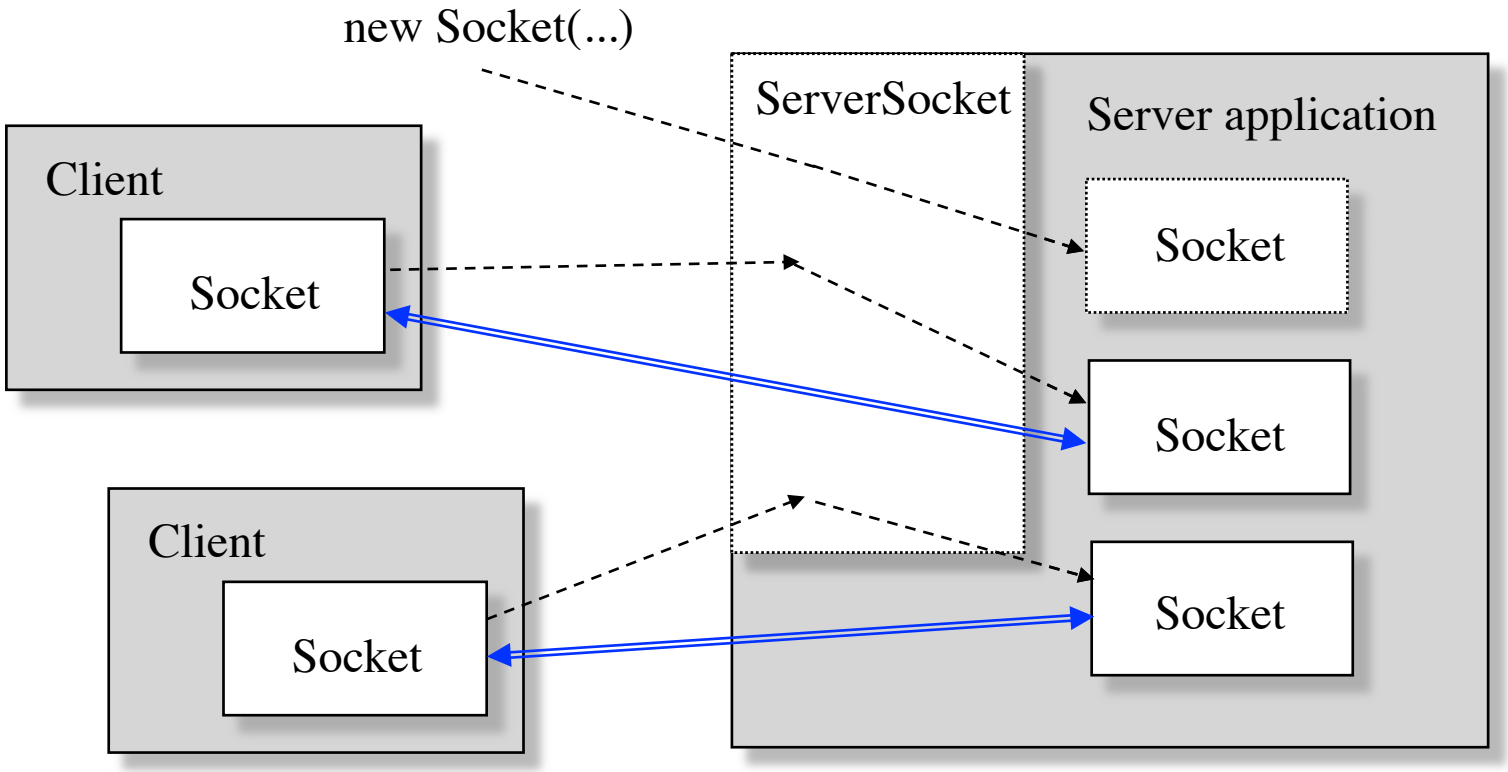
(1) On the server side as return value of the `accept()` method.

(2) On the client side by using the constructor

```
Socket(String host, int port)
```

`host`: the address of the host
`port`: the port number

Clients' communication with a server



Communication is handled on both sides by `Socket` objects.

Methods of `Socket`

`getInputStream()`

Returns an `InputStream` object for receiving data

`getOutputStream()`

Returns and `OutputStream` object for sending data

`close()`

Closes the socket connection

Typical use of `Socket`

```
try {
    Socket socket = new Socket(host, port);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(
            socket.getInputStream()));
    PrintWriter out = new PrintWriter(
        new OutputStreamWriter(
            socket.getOutputStream()));
    «Send and receive data»
    in.close();
    out.close();
    socket.close();
} catch (IOException e) {
    «Handle exception»
}
```

Development of client/server programs

1. Decide if it reasonable to implement a server and one or more matching clients.

2. Design a text based communication protocol

3. Implement the server

4. Test the server with a telnet program

5. Implement and test a Java client

telnet: A terminal emulation program for TCP/IP networks (such as the Internet)

A simple echo server


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

public class EchoServer {
    public static void main(String[] args) {
        try {
            ServerSocket s = new ServerSocket(8008);
            while (true) {
                Socket incoming = s.accept();
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(
                        incoming.getInputStream()));
                PrintWriter out = new PrintWriter(
                    new OutputStreamWriter(
                        incoming.getOutputStream()));
```

continued

```
                out.println("Hello! This is the Java EchoServer.");
                out.println("Enter BYE to exit.");
                out.flush();
                while (true) {
                    String str = in.readLine();
                    if (str == null)
                        break; // client closed connection
                    out.println("Echo: " + str);
                    out.flush();
                    if (str.trim().equals("BYE"))
                        break;
                }
                in.close();
                out.close();
                incoming.close();
            }
        } catch (Exception e) {}
    }
}
```

Testing the server with telnet

```
venus% telnet saturn 8008
Trying 140.192.34.63 ...
Connected to saturn.
Escape character is '^]'.
Hello! This is the Java EchoServer.
Enter BYE to exit.
Hi, this is from venus
Echo: Hi, this is from venus
BYE
Echo: BYE
Connection closed by foreign host.
```

A simple Java client

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

public class EchoClient {
    public static void main(String[] args) {
        try {
            String host =
                args.length > 0 ? args[0] : "localhost";
            Socket socket = new Socket(host, 8008);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(
                    socket.getInputStream()));
            PrintWriter out = new PrintWriter(
                new OutputStreamWriter(
                    socket.getOutputStream()));
```

continued

```
                // send data to the server
                for (int i = 1; i <= 10; i++) {
                    System.out.println("Sending: line " + i);
                    out.println("line " + i);
                    out.flush();
                }
                out.println("BYE");
                out.flush();

                // receive data from the server
                while (true) {
                    String str = in.readLine();
                    if (str == null)
                        break;
                    System.out.println(str);
                }
                in.close();
                out.close();
                socket.close();
            } catch (Exception e) {}
        }
    }
```

Running the Java client

```
venus% java EchoClient saturn
Sending: line 1
Sending: line 2
...
Sending: line 10
Hello! This is Java EchoServer.
Enter BYE to exit.
Echo: line 1
Echo: line 2
...
Echo: line 10
Echo: BYE
```

An echo server that handles multiple clients simultaneously

Use a separate thread for each client.

```
public class MultiEchoServer {
    public static void main(String[] args) {
        try {
            ServerSocket s = new ServerSocket(8009);
            while (true) {
                Socket incoming = s.accept();
                new ClientHandler(incoming).start();
            }
        } catch (Exception e) {}
    }
}
```

ClientHandler

```
public class ClientHandler extends Thread {
    protected Socket incoming;

    public ClientHandler(Socket incoming) {
        this.incoming = incoming;
    }

    public void run() {
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(
                    incoming.getInputStream()));
            PrintWriter out = new PrintWriter(
                new OutputStreamWriter(
                    incoming.getOutputStream()));
```

continued

```
                out.println("Hello! ...");
                out.println("Enter BYE to exit.");
                out.flush();
                while (true) {
                    String str = in.readLine();
                    if (str == null)
                        break;
                    out.println("Echo: " + str);
                    out.flush();
                    if (str.trim().equals("BYE"))
                        break;
                }
                in.close();
                out.close();
                incoming.close();
            } catch (Exception e) {}
        }
    }
```

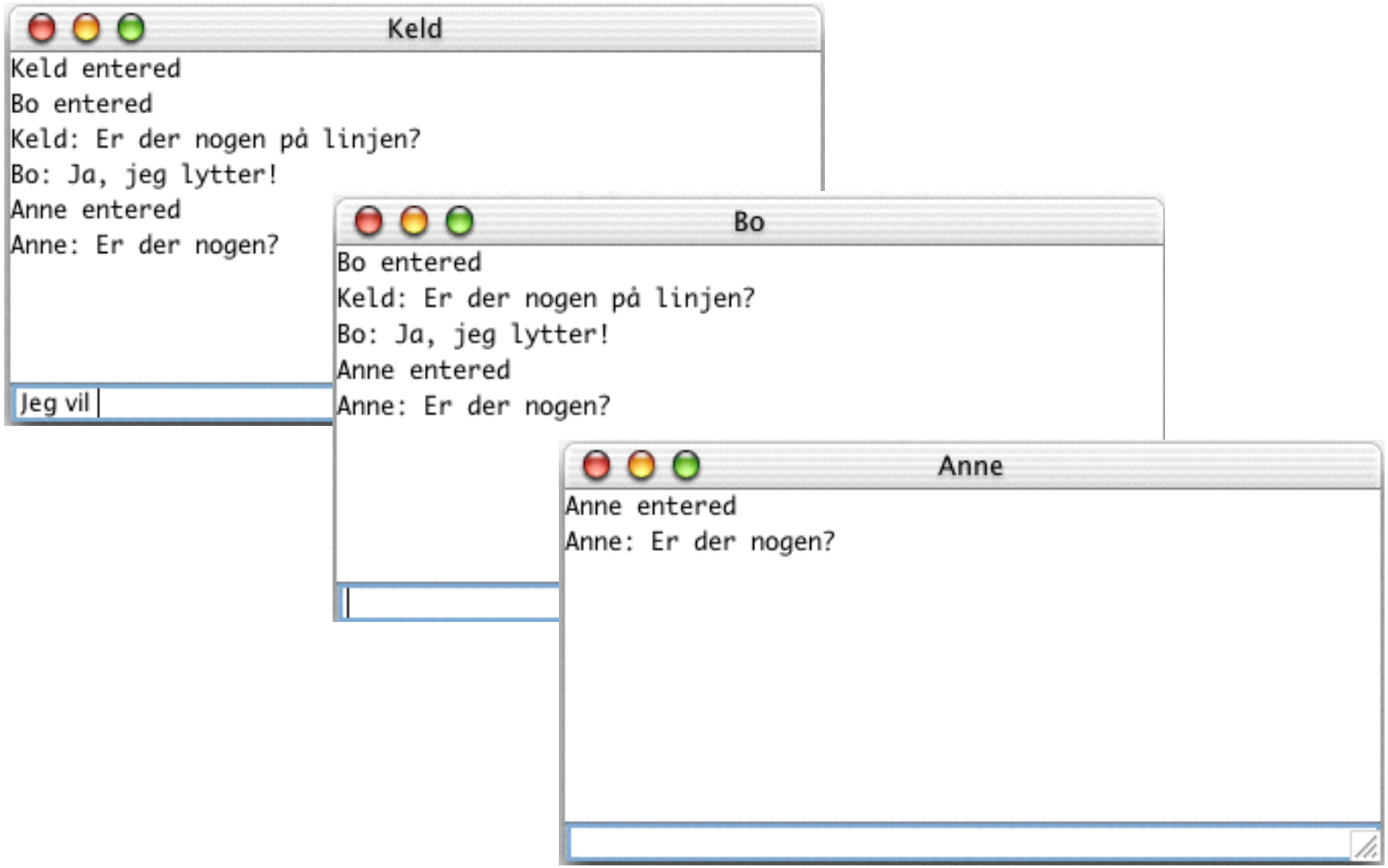
Broadcasting messages to clients

Development of a chat server that

- handles multiple clients simultaneously
- broadcasts a message received from a client to all other active clients.

We need to keep track of active clients.

Chat example



ChatServer

```
public class ChatServer {
    public ChatServer(int port) throws IOException {
        ServerSocket s = new ServerSocket(port);
        while (true)
            new ChatHandler(s.accept()).start();
     }

    public static void main(String[] args)
                            throws IOException {
        if (args.length != 1)
            throw new RuntimeException(
                        "Syntax: java ChatServer <port>");
        new ChatServer(Integer.parseInt(args[0]));
    }
}
```

ChatHandler

```
public class ChatHandler extends Thread {
    Socket socket;
    DataInputStream in;
    DataOutputStream out;
    static Set<ChatHandler> handlers =
        (Set<ChatHandler>) Collections.synchronizedCollection(
                                    new HashSet<ChatHandler>());

    public ChatHandler(Socket socket) throws IOException {
        this.socket = socket;
        in = new DataInputStream(socket.getInputStream());
        out = new DataOutputStream(socket.getOutputStream());
        handlers.add(this);
    }
```

continued

```
public void run() {
    String name = "";
    try {
        name = in.readUTF();
        System.out.println("New client " + name + " from " +
                           socket.getInetAddress());
        broadcast(name + " entered");
        while(true)
            broadcast(name + ": " + in.readUTF());
    } catch (IOException e) {
        System.out.println("-- Connection to user lost.");
    } finally {
        handlers.remove(this);
        try {
            broadcast(name + " left");
            in.close();
            out.close();
            socket.close();
        } catch (IOException e) {}
    }
}
```

continued

```
static void broadcast(String message)
                        throws IOException {
    synchronized (handlers) {
        for (ChatHandler handler : handlers) {
            handler.out.writeUTF(message);
            handler.out.flush();
        }
    }
}
```

Note that the for-loop needs to be `synchronized` because it will be executed by all threads that are handling clients.

ChatClient

```
public class ChatClient {
    String name;
    Socket socket;
    DataInputStream in;
    DataOutputStream out;
    ChatFrame gui;

    public ChatClient(String name, String server, int port) {
        try {
            this.name = name;
            socket = new Socket(server, port);
            in = new DataInputStream(socket.getInputStream());
            out = new DataOutputStream(socket.getOutputStream());
            out.writeUTF(name);
            gui = new ChatFrame(this);
            while (true)
                gui.output.append(in.readUTF() + "\n");
        } catch (IOException e) {}
    }
```

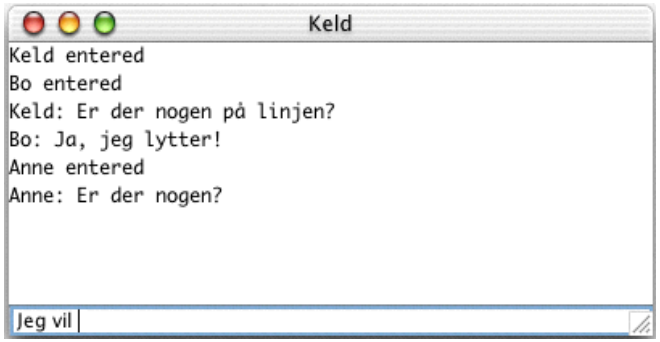
continued

```
    void sendTextToChat(String str) {
        try {
            out.writeUTF(str);
        } catch (IOException e) { e.printStackTrace(); }
    }

    void disconnect() {
        try {
            in.close();
            out.close();
            socket.close();
        } catch (IOException e) { e.printStackTrace(); }
    }

    public static void main(String[] args) throws IOException {
        if (args.length != 3)
            throw new RuntimeException(
                "Syntax: java ChatClient <name> <serverhost> <port>");
        new ChatClient(args[0], args[1], Integer.parseInt(args[2]);
    }
}
```

ChatFrame




```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ChatFrame extends JFrame {
    JTextArea output = new JTextArea();
    JTextField input = new JTextField();

    public ChatFrame(final ChatClient client) {
        super(client.name);
        Container pane = getContentPane();
        pane.setLayout(new BorderLayout());
        pane.add(new JScrollPane(output), BorderLayout.CENTER);
        output.setEditable(false);
        pane.add(input, BorderLayout.SOUTH);
```

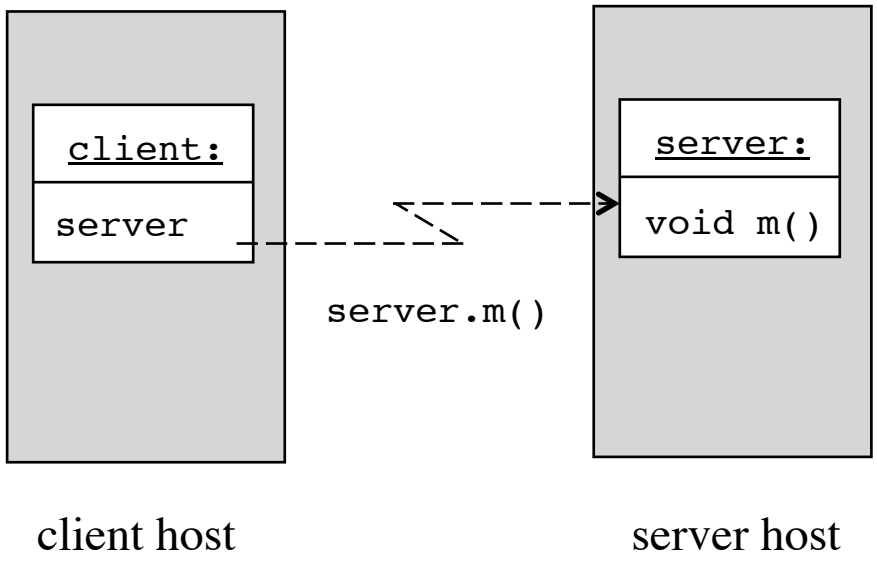
continued

```
        input.addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                if (e.getKeyCode() == KeyEvent.VK_ENTER) {
                    client.sendTextToChat(input.getText());
                    input.setText("");
                }
            }
        });
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                client.disconnect();
                System.exit(0);
            }
        });
        setSize(400, 200);
        setVisible(true);
        input.requestFocus();
    }
}
```

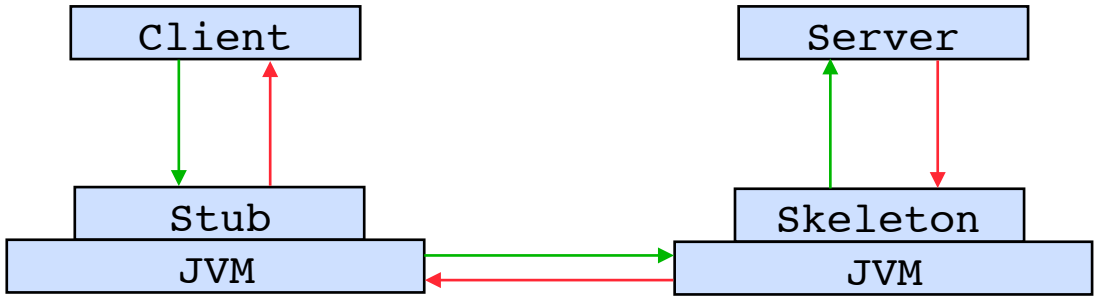
Remote method invocation

RMI

Objects residing on different hosts may be manipulated as if they were on the same host.



The RMI architecture



Server:
An object on the server host that provides services to clients

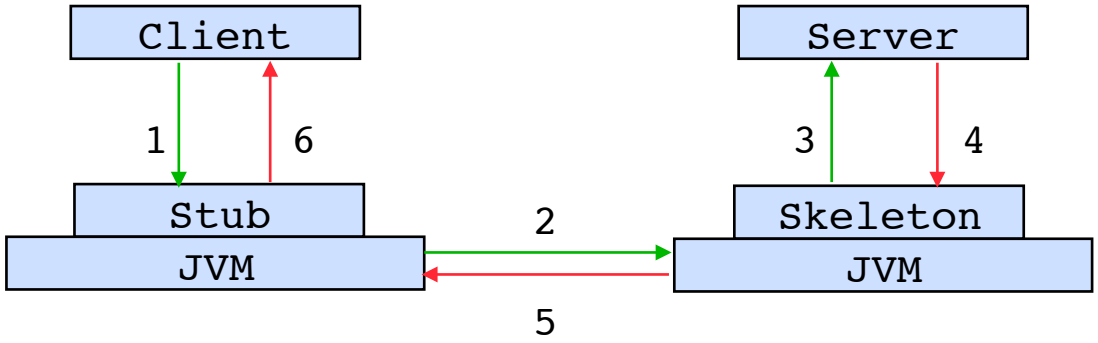
Client:
An object that uses the services provided by the server

Stub:
An object that resides on the same host as the client and serves as a proxy, or surrogate, for the remote server

Skeleton:
An object that resides on the same host as the server, receiving requests from the stubs and dispatching the requests to the server

Service contract:
A Java interface that defines the services provided by the server

Remote method invocation



Invocation of `server.m()` by the client:

1. The method of the stub, `stub.m()`, is invoked
2. The stub marshals the arguments and call information to the skeleton on the server host
3. The skeleton unmarshals the call information and the arguments and invokes the method of the server: `server.m()`
4. The server object executes the method and returns the result to the skeleton
5. The skeleton marshals the result and sends the result back to the stub
6. The stub unmarhals the result and returns the result to the client

RMI programming

`Server`, `Client` and `Service contract` er written by the programmer.

Stubs and skeletons are generated by a RMI compiler (e.g. `rmic`) from the compiled `Server` class.

Passing of arguments

If an argument of a remote method invocation is a local object, the object is serialized, sent to the remote host, and deserialized (that is, a copy of the local object is passed to the remote host).

If an argument of a remote method invocation is a remote object, a remote object reference is passed.

An important question is: How does a client locate the server that will provide the service?

RMI registry

Each RMI server is identified by a URL with the protocol `rmi`.

```
rmi://host:port/name
```

`host`: name or IP address of the host on which the RMI registry is running (if omitted: `localhost`)

`port`: port number of the RMI registry (if omitted: 1099)

`name`: name bound to the RMI server

The server is registered on the server host in a RMI registry. This process is called *binding*:

```
Naming.bind(name, server)
```

Lookup in a RMI registry

A client can locate a remote object by a lookup in the server host's RMI registry:

```
Remote server = Naming.lookup(url)
```

Here `url` is of the form

```
//host:port/name
```

where `host` is the host (remote or local) where the registry is located, and `port` is the port number on which the registry accepts calls. If port is omitted, then the port defaults to 1099.

Operations on a RMI registry

(static methods in `Naming`)

```
static void bind(String name, Remote obj)

static void rebind(String name, Remote obj)

static void unbind(String name)

static Remote lookup(String url)

static String[] list(String url)
```

Application of RMI

1. Define an interface for the remote object.

```
public interface Contract extends Remote {
    public void aService(...) throws RemoteException;
    // other services
}
```

This is the contract between the server and its clients.
The contract interface must extend the `Remote` interface.
The methods in this interface must declare that they throw the `RemoteException` exception.
The types of the arguments and return values must be serializable.

continued

2. Define a service implementation class that implements the contract interface. The class must extend the `UnicastRemoteObject` class.

```
public class ServiceProvider extends UnicastRemoteObject
                                  implements Contract {
    public void aService(...) throws RemoteException {
        // implementation
    }
    // implementation of other services
}
```

continued

3. Create an instance of the server, and register that server to the RMI registry:

```
Contract remoteObj = new ServiceProvider(...);
Naming.rebind(name, remoteObj);
```

4. Generate the stub and skeleton classes, using the RMI compiler.

continued

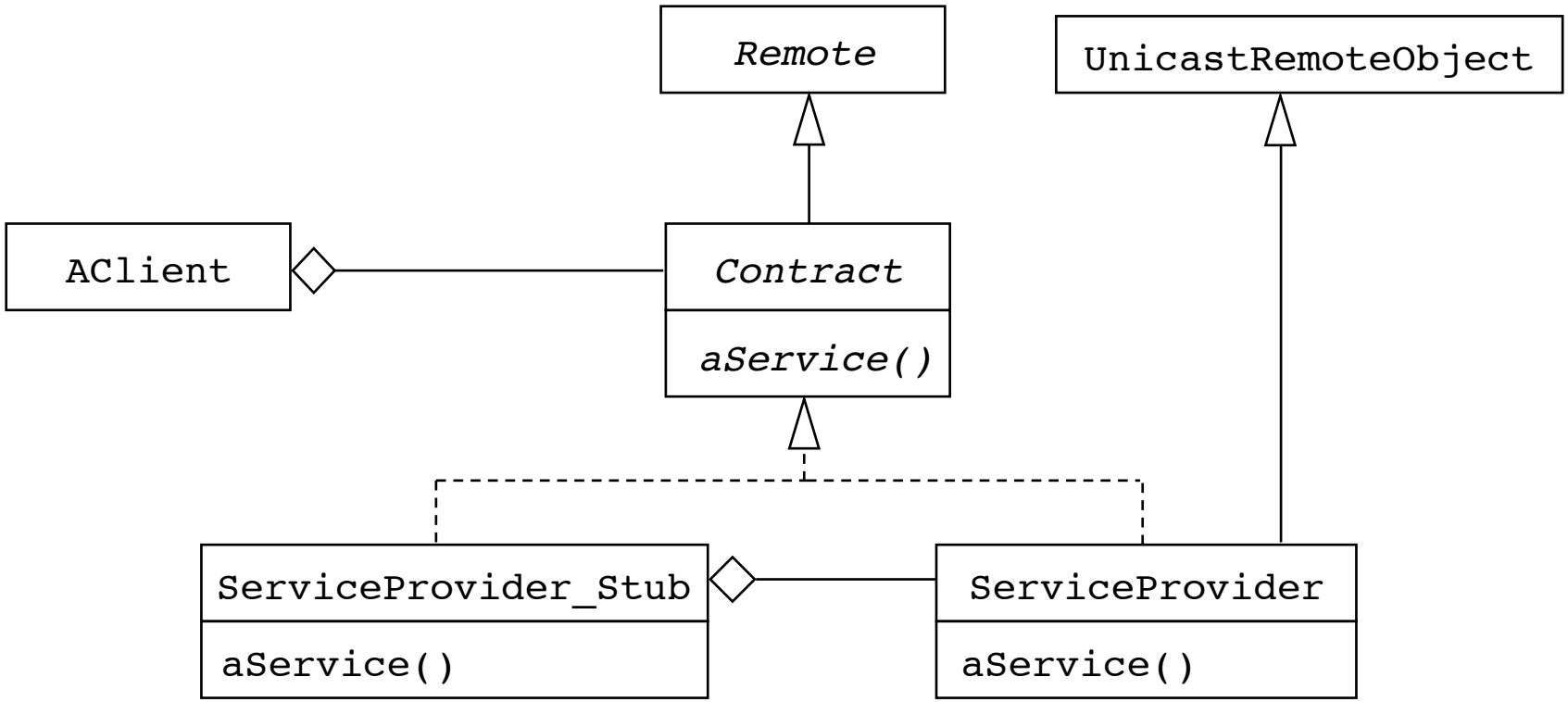
5. Develop a client that uses the service provided by the contract interface.

   It must first locate the remote object that provides the service before the remote methods can be invoked.

```
Contract serverObj = (Contract) Naming.lookup(url);
//...
serverObj.aService(...);
//...
```

continued

Structure of RMI applications



Development of a RMI-based chat system

Remote method invocation on both server and client.

Server:

- login
- logout
- sendMessage

Client:

- receiveLogin
- receiveLogout
- receiveMessage

Contract interfaces

```
public interface ChatServerInterface extends Remote {
    public void login(String name, ChatClientInterface newClient)
                    throws RemoteException;
    public void logout(String name) throws RemoteException;
    public void sendMessage(Message message) throws RemoteException;
}
```

```
public interface ChatClientInterface extends Remote {
    public void receiveLogin(String name) throws RemoteException;
    public void receiveLogout(String name) throws RemoteException;
    public void receiveMessage(Message message) throws RemoteException;
}
```

Message

```
public class Message implements java.io.Serializable {
    public String name, text;

    public Message(String name, String text) {
        this.name = name;
        this.text = text;
    }
}
```

ChatServer

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;

public class ChatServer extends UnicastRemoteObject
                        implements ChatServerInterface {
    Map<String, ChatClientInterface> chatters =
        new HashMap<String, ChatClientInterface>();

    public ChatServer() throws RemoteException {}

    public synchronized void login(String name,
           ChatClientInterface newClient) throws RemoteException {
        chatters.put(name, newClient);
        for (ChatClientInterface client : chatters.values())
            client.receiveLogin(name);
    }
```

continued

```
    public synchronized void logout(String name)
                            throws RemoteException {
        chatters.remove(name);
        for (ChatClientInterface client : chatters.values())
            client.receiveLogout(name);
        System.out.println("client " + name + " logged out");
    }

    public synchronized void sendMessage(Message message)
                            throws RemoteException {
         for (ChatClientInterface client : chatters.values())
             client.receiveMessage(message);
    }

    public static void main(String[] args) {
        try {
            LocateRegistry.createRegistry(1099);
            Naming.rebind("ChatServer", new ChatServer());
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

ChatClient

```
public class ChatClient extends UnicastRemoteObject
                        implements ChatClientInterface {
    String name;
    ChatServerInterface server;
    ChatFrame gui;

    public ChatClient(String name, String url) throws RemoteException {
        this.name = name;
        try {
            server = (ChatServerInterface)
                java.rmi.Naming.lookup("rmi://" + url + "/ChatServer");
            server.login(name, this);
        } catch (Exception e) { e.printStackTrace(); }
        gui = new ChatFrame(this);
    }
```

continued

```
public void receiveLogin(String name) {
    gui.output.append(name + " entered\n");
}

public void receiveLogout(String name) {
    gui.output.append(name + " left\n");
}

public void receiveMessage(Message message) {
    gui.output.append(message.name + ": " + message.text + "\n");
}
```

continued

```
    void sendTextToChat(String text) {
        try {
            server.sendMessage(new Message(name, text));
        } catch (RemoteException e) { e.printStackTrace(); }
    }

    void disconnect() {
        try {
            server.logout(name);
        } catch (Exception e) { e.printStackTrace(); }
    }

    public static void main(String[] args) {
        if (args.length != 2)
            throw new RuntimeException(
                "Usage: java ChatClient <user> <host>");
        try {
            new ChatClient(args[0], args[1]);
        } catch (RemoteException e) { e.printStackTrace(); }

    }
}
```