Communication (Part II)

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)
Remote Procedure Call (RPC)
Many distributed systems have been based on explicit message exchange between processes.
Many distributed systems have been based on explicit message exchange between processes.

However, the send and receive methods do not hide communication at all, which is important to achieve access transparency in distributed systems.
Many distributed systems have been based on explicit message exchange between processes.

However, the send and receive methods do not hide communication at all, which is important to achieve access transparency in distributed systems.

Proposed solution: to allow programs to call procedures located on other machines: Remote Procedure Call (RPC).
Local Procedure Call

- a: Parameter passing in a local procedure call: the stack before the call to `read(fd, buf, bytes)`.

- b: The stack while the called procedure is active.
Remote Procedure Call (RPC)

- Principle of **RPC** between a **client** and **server** program.
RPC abstracts procedure calls between processes on networked systems.
RPC abstracts procedure calls between processes on networked systems.

Stubs: client-side proxy for the actual procedure on the server.
Basics of RPCs

- RPC abstracts procedure calls between processes on networked systems.

- Stubs: client-side proxy for the actual procedure on the server.

- The client-side stub
  - Locates the server
  - Marshalls the parameters
Basics of RPCs

- **RPC abstracts procedure calls** between processes on networked systems.

- **Stubs**: *client-side* proxy for the **actual procedure** on the **server**.

- The **client-side stub**
  - Locates the server
  - Marshalls the parameters

- The **server-side stub**
  - Receives the message from client-side stub
  - Unpacks the marshalled parameters
  - Performs the procedure on the server
Steps of a RPC (1/2)

1. **Client procedure calls client stub.**
2. **Stub builds message, and calls local OS.**
3. **OS sends message to remote OS.**
4. **Remote OS gives message to server stub.**
5. **Server stub unpacks parameters and calls server.**

![Diagram showing the steps of a RPC](image)
Steps of a RPC (2/2)

6. **Server** makes local call and returns **result** to **stub**.

7. **Stub** builds **message**, and calls **OS**.

8. **OS** sends **message** to client’s **OS**.

9. **Client’s OS** gives message to **stub**.

10. **Client stub unpacks** result and returns to the **client**.

Implementation of add

```
Client machine

Client process

k = add(i,j)

proc:  "add"
int:   val(i)
int:   val(j)

Server machine

Implementation of add

Server process

k = add(i,j)

proc:  "add"
int:   val(i)
int:   val(j)
```

Amir H. Payberah (Tehran Polytechnic)
Parameter marshaling: there is more than just wrapping parameters into a message.
Parameter marshaling: there is more than just wrapping parameters into a message.

Client and server machines may have different data representations (think of byte ordering).
Parameter Passing (1/2)

- **Parameter marshaling**: there is more than just wrapping parameters into a message.

- Client and server machines may have different data representations (think of byte ordering).

- Wrapping a parameter means transforming a value into a sequence of bytes.
Parameter marshaling: there is more than just wrapping parameters into a message.

Client and server machines may have different data representations (think of byte ordering).

Wrapping a parameter means transforming a value into a sequence of bytes.

Client and server have to agree on the same encoding:
  • How are basic data values represented (integers, floats, characters)
  • How are complex data values represented (arrays, unions)

Client and server need to properly interpret messages, transforming them into machine-dependent representations.
Some assumptions:

- **Copy in/copy out** semantics: while procedure is executed, **nothing can be assumed** about **parameter values**.
- **All** data that is to be operated on is passed by parameters. Excludes passing **references to (global) data**.
Some assumptions:

- **Copy in/copy out** semantics: while procedure is executed, **nothing can be assumed** about parameter values.
- **All** data that is to be operated on is passed by parameters. Excludes passing references to (global) data.

**Conclusion:** **full access transparency cannot** be realized.
Some assumptions:

- **Copy in/copy out** semantics: while procedure is executed, **nothing can be assumed** about **parameter values**.
- **All** data that is to be operated on is passed by parameters. Excludes passing **references to (global) data**.

**Conclusion**: **full access transparency cannot** be realized.

**Observation**: a **remote reference** mechanism **enhances access transparency**:

- Remote reference offers **unified access** to remote data.
- Remote references can be **passed as parameter** in RPCs.
Try to get rid of the strict request-reply behavior, but let the client continue without waiting for an answer from the server.
Remote Method Invocation (RMI)
Introduction

- Remote Method Invocation (RMI)
- \( \text{RMI} = \text{RPC} + \text{object oriented} \)
- RPC in C and RMI in Java
Steps of a RMI (1/3)

1. The server program controls the remote objects.
2. The server registers an interface with a naming service: makes the interface accessible by clients.
Steps of a RMI (1/3)

1. The server program controls the remote objects.
2. The server registers an interface with a naming service: makes the interface accessible by clients.
3. The interface contains the signatures for those methods of the object that the server wishes to make publicly available.
Steps of a RMI (1/3)

1. The server program controls the remote objects.
2. The server registers an interface with a naming service: makes the interface accessible by clients.
3. The interface contains the signatures for those methods of the object that the server wishes to make publicly available.
4. Clients use the naming service to obtain a reference to this interface: called a stub.
The stub is a local surrogate for the remote object.

On the server system, there is another surrogate called a skeleton.
5 The **stub** is a **local surrogate** for the **remote object**.

6 On the **server system**, there is another **surrogate** called a **skeleton**.

7 When the **client** program **invokes a method** of the remote object, it appears to the client as though the method is being invoked **directly on the object**.

8 What is actually happening, however, is that an **equivalent method** is being called in the **stub**.
9. The **stub forwards** the call and any parameters to the **skeleton** on the **remote machine**.
9. The **stub forwards** the **call and any parameters** to the **skeleton** on the remote machine.

10. Only **primitive types** and those **reference types** that implement the **Serializable** interface may be used as parameters.
9. The stub forwards the call and any parameters to the skeleton on the remote machine.

10. Only primitive types and those reference types that implement the Serializable interface may be used as parameters.

11. Upon receipt of the byte stream, the skeleton converts this stream into the original method call and associated parameters.
Steps of a RMI (3/3)

9. The **stub forwards** the **call and any parameters** to the **skeleton** on the **remote machine**.

10. Only **primitive types** and those **reference types** that implement the **Serializable** interface may be used as parameters.

11. Upon receipt of the byte stream, the **skeleton converts** this stream into the **original method call and associated parameters**.

12. The **skeleton calls** the implementation of the **method** on the server.
Setting up a RMI connection four steps:

1. Create the interface.
2. Define a class that implements this interface.
3. Create the server process.
4. Create the client process.
Setting up a RMI Connection (1/4)

- Create the interface.

- This interface should extend interface Remote.

```java
import java.rmi.*;

public interface Hello extends Remote {
    public String getGreeting() throws RemoteException;
}
```
Define a class that implements this interface.

The implementation class must extend class `RemoteObject` or one of `RemoteObject`'s subclasses.
- E.g., `UnicastRemoteObject` that supports TCP point-to-point communication.

We must provide a constructor for our implementation object.

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

public class HelloImpl extends UnicastRemoteObject implements Hello {
    public HelloImpl() throws RemoteException { ...
    public String getGreeting() throws RemoteException {
        return ("Hello there!");
    }
}
```
Create the **server** process.

The server creates **object(s)** of the above implementation class and registers them with a **naming service** called the **registry**.

Establishes a **connection** between the **object’s name** and its reference, by using method **rebind** that takes two arguments:

1. a **string** that holds the **name** of the **remote object** as a **URL** with protocol **rmi**.
2. a **reference** to the **remote object**.

Clients will then be able to use the remote object’s name to retrieve a reference to that object via the **registry**.
import java.rmi.*;
public class HelloServer {
    private static final String HOST = "localhost";
    public static void main(String[] args) throws Exception {
        //Create a reference to an implementation object...
        HelloImpl temp = new HelloImpl();

        //Create the string URL holding the object’s name...
        String rmiObjectName = "rmi://" + HOST + "/Hello";

        //’Bind’ the object reference to the name...
        Naming.rebind(rmiObjectName, temp);

        System.out.println("Binding complete...
"};
}
Create the client process.

The client obtains a reference to the remote object from the registry, by calling method `lookup`.

```java
import java.rmi.*;
public class HelloClient {
    private static final String HOST = "localhost";
    public static void main(String[] args) {
        try {
            // Obtain a reference to the object from the registry
            Hello greeting = (Hello)Naming.lookup("rmi:///" + HOST + "/Hello");

            // Use the above reference to invoke the remote object's method...
            System.out.println("Message received: " + greeting.getGreeting());
        } catch(Exception ex) { ... } }
}
```
Compiling and running a RMI application consists of four steps:

1. Compile all files with `javac`.
2. Start the RMI registry: `rmiregistry`.
3. Run the server.
4. Run the client.
Summary
Summary

- Send and receive methods do not hide provide access transparency.
- Remote Procedure Call (RPC) - in C
- Client stub and server stub (skeleton)
- Parameter passing: marshaling
- Remote Method Invocation (RMI) - in Java
Reading

- Chapter 4 of the Distributed Systems: Principles and Paradigms.

- Chapter 5 of An Introduction to Network Programming with Java.
Questions?