Java

Remote Method Invocation(RMI)

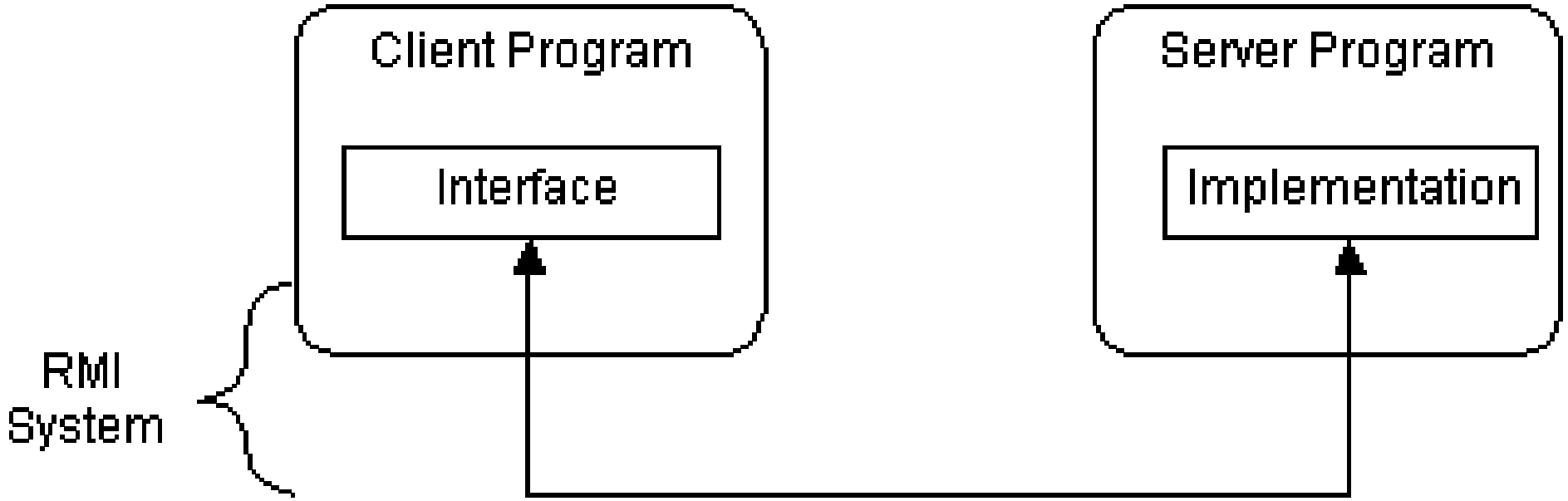
Presented by Bartosz Sakowicz

Overview

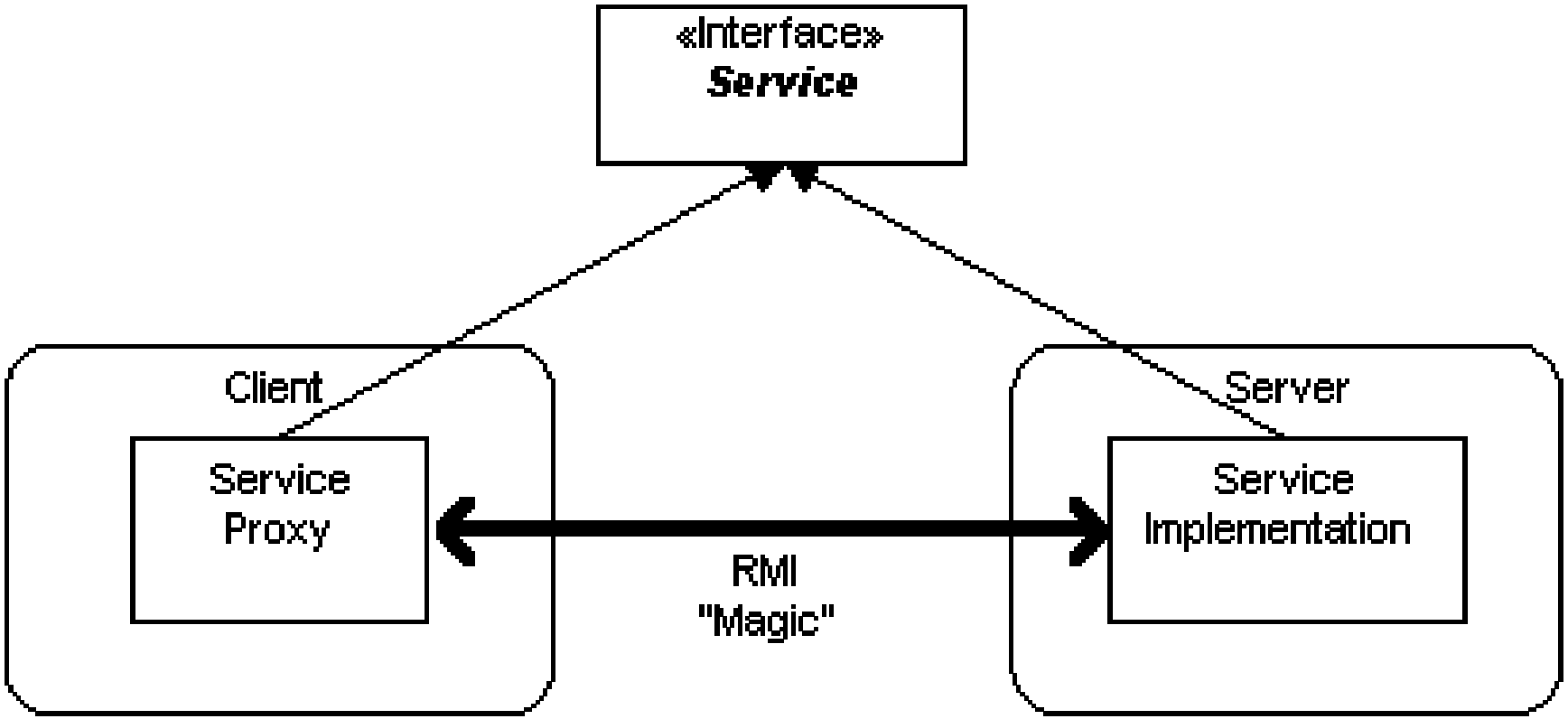
- The Java Remote Method Invocation (RMI) system allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM.
- RMI provides for remote communication between programs written in the Java programming language.
- The RMI architecture is based on one important principle: **the definition of behavior and the implementation of that behavior are separate concepts.**
- RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.

RMI architecture

Interfaces define behavior and classes define implementation:

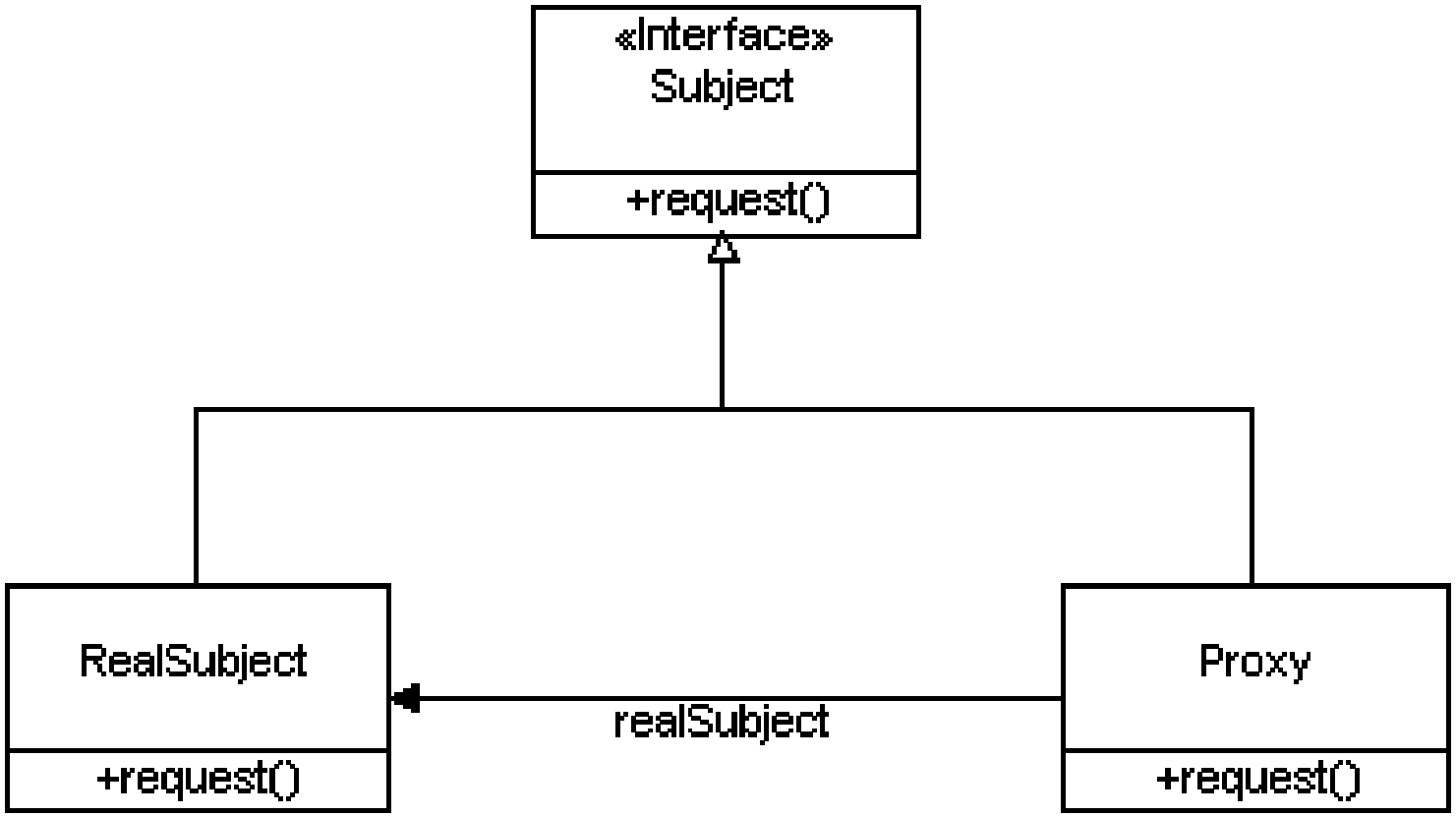


RMI architecture(2)



RMI supports two classes that implement the same interface. The first class is the implementation of the behavior, and it runs on the server. The second class acts as a proxy for the remote service and it runs on the client.

Stub and Skeleton



RMI uses the Proxy design pattern. The stub class plays the role of the proxy, and the remote service implementation class plays the role of the RealSubject. Skeleton is analogous class on the server side (obsolote until java 1.2).

Stub

When a stub’s method is invoked, it does the following:

- initiates a connection with the remote JVM containing the remote object,
- marshals (writes and transmits) the parameters to the remote JVM,
- waits for the result of the method invocation,
- unmarshals (reads) the return value or exception returned, and returns the value to the caller.

Skeleton

When a skeleton receives an incoming method invocation it does the following:

• unmarshals (reads) the parameters for the remote method,

• invokes the method on the actual remote object implementation, and marshals (writes and transmits) the result (return value or exception) to the caller.

In the Java 2 SDK, Standard Edition, v1.2 an additional stub protocol was introduced that eliminates the need for skeletons in Java 2 platform-only environments. Instead, generic code is used to carry out the duties performed by skeletons in JDK1.1.

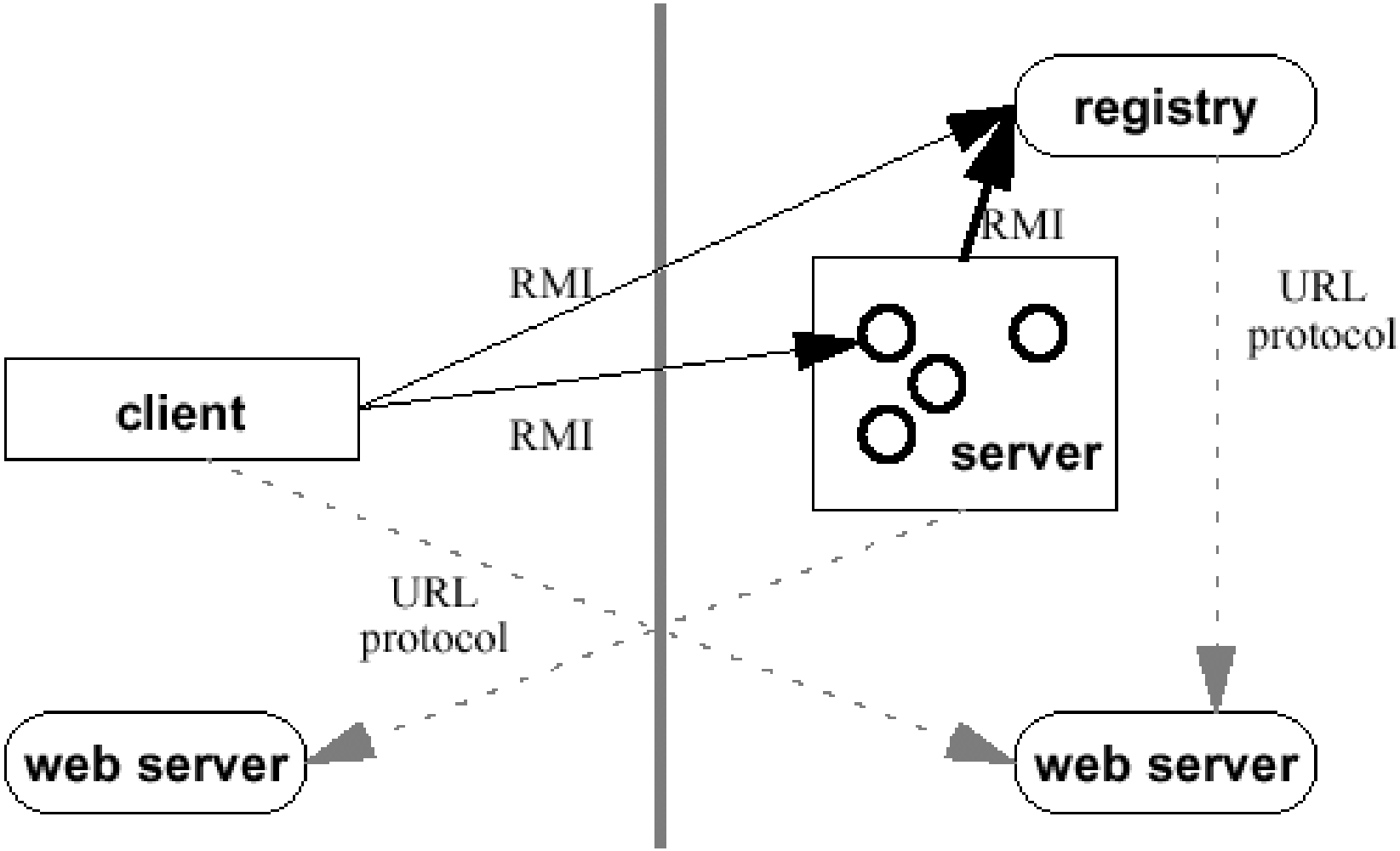
Distributed Applications

Distributed object applications need to:

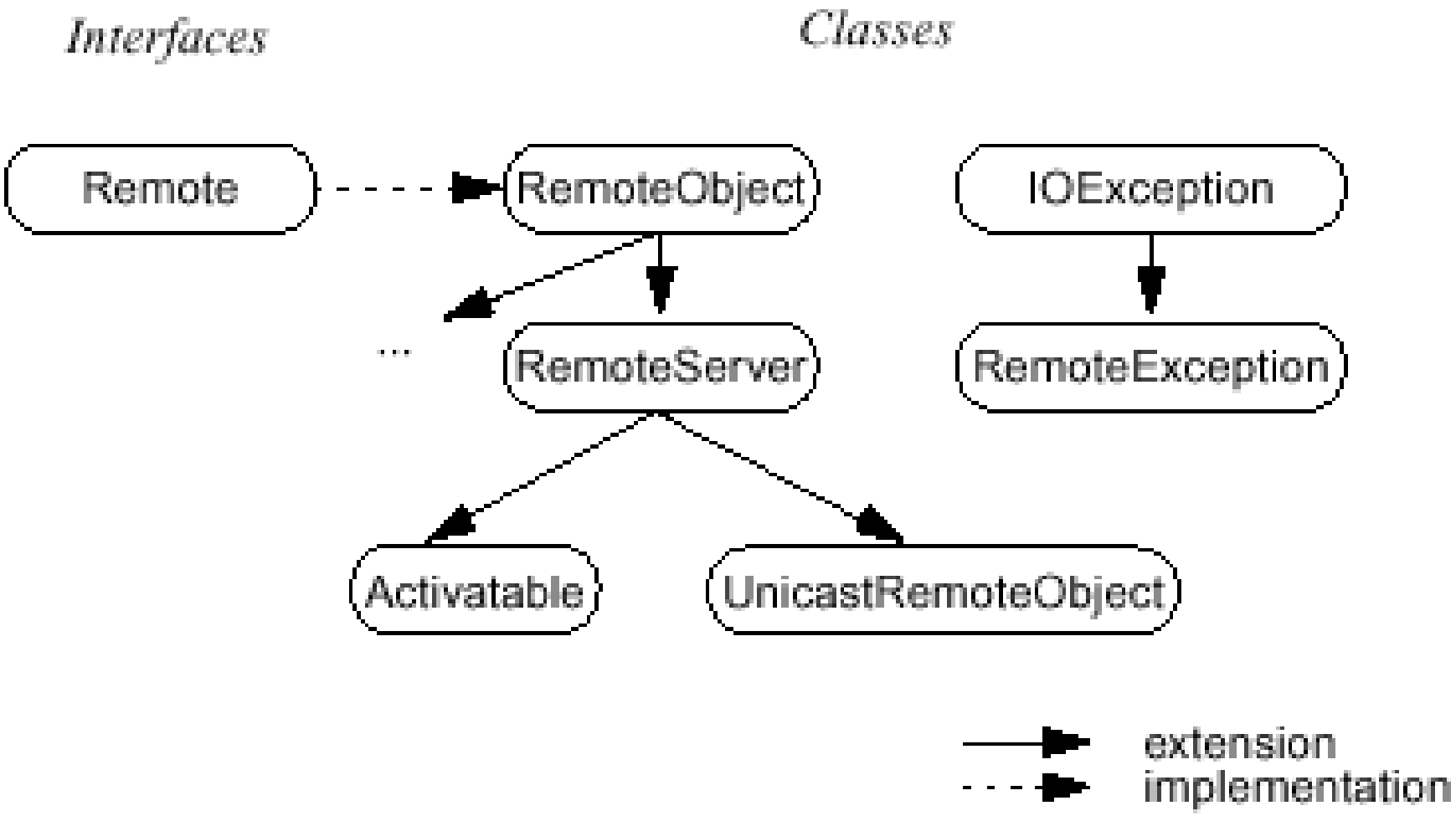
- ***Locate remote objects*: Applications can use one of two mechanisms to obtain references to remote objects. An application can register its remote objects with RMI's simple naming facility, the rmiregistry, or the application can pass and return remote object references as part of its normal operation.
- ***Communicate with remote objects***: Details of communication between remote objects are handled by RMI; to the programmer, remote communication looks like a standard Java method invocation.
- ***Load class bytecodes for objects that are passed around***: Because RMI allows a caller to pass objects to remote objects, RMI provides the necessary mechanisms for loading an object's code, as well as for transmitting its data.

Distributed Applications(2)

RMI can use any URL protocol(supported by Java) to provide communication:



RMI classes and Interfaces



RemoteException

The java.rmi.RemoteException class is the superclass of exceptionsthrown by the RMI runtime during a remote method invocation.

The exception java.rmi.RemoteException is thrown when a remotemethod invocation fails for some reason. Some reasons for remote method invocation failure include:

• Communication failure (the remote server is unreachable or is refusing

connections; the connection is closed by the server, etc.)

• Failure during parameter or return value marshalling or unmarshalling

• Protocol errors

The class RemoteException is a **checked** exception.

UnicastRemoteObject and Activatable

The **java.rmi.server.UnicastRemoteObject** class defines a singleton(unicast) remote object whose references are valid only **while the server process is alive.**

The class **java.rmi.activation.Activatable** is an abstract class that defines an *activatable* remote object that **starts executing when its remote methods are invoked** and can shut itself down when necessary.

Presented by Bartosz Sakowicz

DMCS TUL

Passing parameters

An argument to, or a return value from, a remote object can be any object that is **serializable**. This includes primitive types, remote objects, and non-remote objects that implement the java.io.Serializable interface.

Passing Non-remote Objects

A non-remote object, that is passed as a parameter of a remote method invocation or returned as a result of a remote method invocation, **is passed by *copy***; that is, the object is serialized using the object serialization mechanism of the Java platform.

Passing Remote Objects

When passing an exported remote object as a parameter or return value in a remote method call, the stub for that remote object is passed instead.

Threads in RMI

A method dispatched by the RMI runtime to a remote object implementation **may or may not execute in a separate thread.**

The RMI runtime makes noguarantees with respect to mapping remote object invocations to threads.

Since remote method invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure **its implementation is thread-safe.**

Presented by Bartosz Sakowicz

DMCS TUL

Designing distributed application components

- ***Defining the remote interfaces*:** A remote interface specifies the methods that can be invoked remotely by a client.
- ***Implementing the remote objects*:** Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces (either local or remote) and other methods (which are available only locally).
- ***Implementing the clients*:** Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

Remote interface

1. The remote interface must be **public** (it cannot have “package access,”). Otherwise, a client will get an error when attempting to load a remote object that implements the remote interface.
2. The remote interface must extend the interface **java.rmi.Remote**.
3. Each method in the remote interface must declare **java.rmi.RemoteException** in its **throws** clause in addition to any application-specific exceptions.
4. A remote object passed as an argument or return value (either directly or embedded within a local object) must be declared as the remote interface, not the implementation class.

Remote interface(2)

Example:

```
import java.rmi.*;
 public interface PerfectTimeI extends Remote {
     long getPerfectTime() throws RemoteException;
}
```

Implementing the remote interface

The server must contain a class that extends **UnicastRemoteObject** and implements the remote interface. This class can also have additional methods, but only the methods in the remote interface are available to the client.

You must explicitly define the constructor for the remote object even if you're only defining a default constructor that calls the base-class constructor. You must write it out since it must throw **RemoteException**.

Implementing the remote interface(2)

Main steps with server implementation:

1. Create and install a security manager that supports RMI. The only one available for RMI as part of the Java distribution is **RMISecurityManager**.
2. Create one or more instances of a remote object.
3. Register at least one of the remote objects with the RMI remote object registry.

Implementing the remote interface(3)

Example:

```
import java.rmi.*;

import java.rmi.server.*;

import java.rmi.registry.*;

import java.net.*;

public class PerfectTime extends UnicastRemoteObject
implements PerfectTimeI {

        public long getPerfectTime() throws RemoteException {

                return System.currentTimeMillis(); }

        public PerfectTime() throws RemoteException {  }
```

Implementing the remote interface(4)

```
public static void main(String[] args) throws Exception {
        System.setSecurityManager(
        new RMISecurityManager()); // source of problems
        PerfectTime pt = new PerfectTime();
        Naming.bind( "//localhost:2005/PerfectTime", pt);
        // or rebind
        System.out.println("Ready to do time");
    }
}
```

Setting up the registry

The name of the registry server is **rmiregistry.**To start it in the background process write:

start rmiregistry (Win 32)

rmiregistry & (Unix)

The default port is 1099. If you want it to be at some other port, you add an argument on the command line to specify the port:

rmiregistry 2005 &

Setting up the registry(2)

The name for the service is arbitrary.The important thing is that it's a unique name in the registry that the client knows to look for to procure the remote object. If the name is already in the registry, you'll get an AlreadyBoundException (use **rebind** method).

You aren't forced to start up **rmiregistry** as an external process. If you know that your application is the only one that's going to use the registry, you can start it up inside your program with the line:

LocateRegistry.createRegistry(2005);

Creating stubs and skeletons

All of the Remote interfaces and classes should be compiled using **javac**. Once this has been completed, the stubs and skeletons for the Remote interfaces should be compiled by using the **rmic** stub compiler:

rmic PerfectTime // without extension: ".java" !

The only problem one might encounter with this command is that rmic might not be able to find the files Hello.class and HelloInterface.class even though they are in the same directory where rmic is being executed. If this happens try setting the **CLASSPATH** environment variable to the current directory.

Using the remote objects

Example:

```
import java.rmi.*;
import java.rmi.registry.*;
public class DisplayPerfectTime {
public static void main(String[] args) throws Exception {
        System.setSecurityManager( new RMISecurityManager());
        PerfectTimeI t = (PerfectTimeI)
        Naming.lookup( "//peppy:2005/PerfectTime");
        // cast to interface, not the class !
        System.out.println("Perfect time = " + t.getPerfectTime());
}
```

RMI client-side callbacks

In many architectures, a server may need to make a remote call to a client. Examples include progress feedback, time tick notifications, warnings of problems, etc.

To accomplish this, a client must also act as an RMI server. It may be impractical for a client to extend UnicastRemoteObject. In these cases, a remote object may prepare itself for remote use by calling the static method:

UnicastRemoteObject.exportObject (<remote_object>)

Security problems

One of the most common problems one encounters with RMI is a failure due to security constraints. If the current security policy does not allow to connect to some host and port, then the call throws an exception. This usually causes your program to terminate with a message such as:

java.security.AccessControlException: access denied (java.net.SocketPermission 127.0.0.1:1099 connect,resolve)

Security problems(2)

There are several ways to modify the security policy of a program. The simplest technique is to define a subclass of SecurityManager and to call **System.setSecurityManager** on an object of this subclass. In the definition of this subclass, you should override those check methods for which you want a different policy. For example:

System.setSecurityManager (new RMISecurityManager() {

public void checkConnect (String host, int port) {}

public void checkConnect (String host, int port, Object context) {}

});

Security problems(3)

The second way to grant additional permissions is to specify them in a policy file and then request that they be loaded using options such as the following:

java -Djava.security.manager -Djava.security.policy=policy-file MyClass

Advanced topics

- Firewalls (HTTP) - usage of HTTP servers

  http://<host>:<port>/

  http://<host>:80/cgi-bin/java-rmi?forward=<port>

- Activation protocol - possible to make remote activation
- Distributed Garbage Collecting