ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Networking in Java

Stanislav Chachkov

Department of Computer Science
Software Engineering Laboratory
April 2001

Useful Links

➥ Main Java Site - http://www.javasoft.com

➥ EPFL Java Site (docs and tutorial!) - http://sunwww:1995.epfl.ch/Java/

➥ Java RMI Guide — http://sunwww:1995.epfl.ch/Java/jdk1.3/docs/guide/rmi/

➥ Java Object Serialization Guide — http://sunwww.epfl.ch:1995/Java/jdk1.3/docs/guide/serialization/

➥ Java Tutorial on RMI — http://sunwww:1995.epfl.ch/for_EPFL/javatut_28Feb2001/

Sockets: `java.net` package

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

- Client:

```
try{
    Socket s = new Socket("www.epfl.ch", 8000);
    s.getOutputStream().write(request);
    reply = s.getInputStream().read();
    s.close();
}catch(IOException ioe){...}
```

- Server:

```
try{
    ServerSocket ss = new ServerSocket(8000);
    Socket s = ss.accept();
    request = s.getInputStream().read(); //do something ...
    s.getOutputStream().write(reply);
    s.close();
}catch(IOException ioe){...}
```


Another scenario for `ServerSocket`

The Server: maximum availability for multiple clients

```
try{
    ServerSocket ss = new ServerSocket(8000);
    while(true)
        new Handler(ss.accept()).start();
}catch(IOException ioe){...}
```

... and the Handler: one Handler thread for each client

```
class Handler extends Thread{
    Socket myConnection;
    Handler(Socket s){ myConnection = s; }

    public void run(){
        //..... serve the client, for example:
        request = s.getInputStream().read();//then do something ...
        s.getOutputStream().write(reply);
        s.close();
    }
}
```


The Need for a Protocol

What is the meaning of exchanged messages?

For example:
Server receives `request="G123"` and returns a `reply="EPFL"`.

That can mean:
`request`="give me record #123 from your database #G"
`reply` ="OK, the record you asked is "EPFL""

- Protocol gives a meaning to the messages.
- An implementation of a protocol "encodes" and "decodes" the messages.

Remote Method Invocation

Java "Protocol" + Network = RMI

- Java "Protocol": an object invokes a method of another object, waits for it to execute and receives the return value or an exception.
- RMI: a client object invokes a method (via a **remote reference**) of a server object. RMI encodes, sends, receives, decodes and executes the invocation on the server object. The result of the invocation is returned in the same way to the client.
- RMI calls are like normal calls: synchronous (i.e. they block the caller until the method returns from execution)
- RMI calls are different from normal calls: they can fail because of transmission problems.

The Remote Interface

Each Remote Object must implement a Remote Interface:

```
import java.rmi.*;

public interface Database extends java.rmi.Remote{

    public String getRecord(int index)
                        throws RemoteException;
    public void addRecord(int index, String record)
                        throws RemoteException;
    public int getSize()
                        throws RemoteException;
}
```


The Remote Interface must:

1. be declared **`public`**
2. extend **`java.rmi.Remote`** interface
3. have **`java.rmi.RemoteException`** in signature of each method (+ other exceptions)
4. use remote interfaces of remote objects in signatures of its methods, not implementation classes!

The Implementation Object

The Implementation Object lives on the Server.
This object can be accessed from network using RMI protocol.

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

public class DatabaseImpl extends UnicastRemoteObject
                        implements Database {

    public DatabaseImpl()throws RemoteException{ ... }

    public String getRecord(int index){ ... }
    public String addRecord(int index, String record)
    { ... }
    public int getSize() { ... }
}
```


The Implementation object must:

1. implement a remote interface
2. define the constructor
3. provide implementation of methods declared in remote interface (obvious)

Extending `UnicastRemoteObject` adds some communication facilities to the implementation class:

- Define that default socket-based transport of RMI will be used
- The object will run “all the time”

Notes on Constructor of Implementation Object

To start listening for incoming requests from clients, an implementation object must be exported :

`export` operation throws an `RemoteException`

- The default constructor of `UnicastRemoteObject` *exports* the object automatically

```
    public DatabaseImpl()throws RemoteException{
        super();
    }
```

- The other possibility (not a sub-class of `UnicastRemoteObject`):

```
    public class DBImpl_2 implements Database /*no extends*/{
    public DBImpl_2()throws RemoteException{
        UnicastRemoteObject.exportObject(this);
    }
```


Implementation of remote methods

```
public String getRecord(int index){
    String record = ...;
    return record;
}
```

- Arguments and return values of a remote method can be of any type
- Non-remote arguments/return values are serialized (i.e. a copy is sent)

NB: Non-remote arguments/return values must implement `Serializable` interface!

- For remote arguments/return values a remote reference is sent (i.e. no copy is made).

Creation of a Remote object on the Server

1. When a remote object is created it is automatically exported — ready to accept incoming requests:

```
DatabaseImpl db1 = new DatabaseImpl();
```

2. To be "visible" to the clients, remote object must be *registered* in RMI registry:

```
Naming.rebind( "DB_ONE", db1);
```

First argument of rebind is the name assigned to the remote object. Syntax:

```
full:                              "//host:port/name"
assuming default rmiregistry port: "//host/name"
assuming localhost and default port: "name"
```

For security purposes `host` can only be the local host.
Obviously, to execute a `rebind` or a `lookup` RMIRegistry must be started!

Remote Method Invocation

How does the client invoke the method of a remote object?

1. Obtain a reference to the remote object

```
try{
    Database db =
     (Database)Naming.lookup("//www.epfl.ch/DB_ONE");
```

2. Invoke a method as usual

```
    String reply = db.getRecord(123);
}catch(Exception e){e.printStackTrace();}
```


Generating Stubs

Run the `rmic` tool on a compiled **implementation** class (not on remote interface):

```
rmic DatabaseImpl
```

or for classes in a package:

```
rmic rmicourse.DatabaseImpl
```

Use `-v1.2` flag of `rmic` to generate less code! (not compatible with RMI v1.1)

Run RMI Registry

The RMI Registry is a name server that allows clients to get a reference to a remote object.

To start the RMI Registry from a shell:

```
rmiregistry or
rmiregistry 8000
```

To start the RMI Registry programmatically:

```
java.rmi.registry.LocateRegistry.createRegistry(8000);
```

The default RMI Registry port is #1099
Each time the remote interface is modified the RMI Registry must be restarted!

RMI Checklist

1. Remote interface (extends java.rmi.Remote)

2. Implementation object (implements 1, extends UnicastRemoteObject)

3. javac

4. rmic

5. start rmiregistry

6. rebind

7. lookup

For Advanced Users

The Java RMI is highly customizable:

- use `java.rmi.activation.Activatable` instead of `UnicastRemoteObject` if you want a remote object that can be activated (created) when a client requests it.
- use `RMISocketFactory` if you want a network protocol other than TCP/IP.
- install a `RMISecurityManager` if the client and the server do not share the same CLASSPATH and
- specify a URL from where remote stubs and other classes are loaded by the client and the server.

Serialization

Remember: non-remote arguments/return values of a remote method must implement `Serializable` interface.

```
public class Bidule implements java.io.Serializable{
    private String name = “EPFL”;//////// YES
    private transient Image hugeBitmap;// NO
    private static int nextBidule= 0;/// NO
}
```

For further information please read the serialization guide

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Networking in Java

Stanislav Chachkov

Department of Computer Science
Software Engineering Laboratory
April 2001

Useful Links

➥ Main Java Site - http://www.javasoft.com

➥ EPFL Java Site (docs and tutorial!) - http://sunwww:1995.epfl.ch/Java/

➥ Java RMI Guide — http://sunwww:1995.epfl.ch/Java/jdk1.3/docs/guide/rmi/

➥ Java Object Serialization Guide — http://sunwww.epfl.ch:1995/Java/jdk1.3/docs/guide/serialization/

➥ Java Tutorial on RMI — http://sunwww:1995.epfl.ch/for_EPFL/javatut_28Feb2001/

Sockets: `java.net` package

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

- Client:

```
try{
    Socket s = new Socket("www.epfl.ch", 8000);
    s.getOutputStream().write(request);
    reply = s.getInputStream().read();
    s.close();
}catch(IOException ioe){...}
```

- Server:

```
try{
    ServerSocket ss = new ServerSocket(8000);
    Socket s = ss.accept();
    request = s.getInputStream().read(); //do something ...
    s.getOutputStream().write(reply);
    s.close();
}catch(IOException ioe){...}
```


Another scenario for `ServerSocket`

The Server: maximum availability for multiple clients

```
try{
    ServerSocket ss = new ServerSocket(8000);
    while(true)
        new Handler(ss.accept()).start();
}catch(IOException ioe){...}
```

... and the Handler: one Handler thread for each client

```
class Handler extends Thread{
    Socket myConnection;
    Handler(Socket s){ myConnection = s; }

    public void run(){
        //..... serve the client, for example:
        request = s.getInputStream().read();//then do something ...
        s.getOutputStream().write(reply);
        s.close();
    }
}
```


The Need for a Protocol

What is the meaning of exchanged messages?

For example:
Server receives `request="G123"` and returns a `reply="EPFL"`.

That can mean:
`request`="give me record #123 from your database #G"
`reply` ="OK, the record you asked is "EPFL""

- Protocol gives a meaning to the messages.
- An implementation of a protocol "encodes" and "decodes" the messages.

Remote Method Invocation

Java "Protocol" + Network = RMI

- Java "Protocol": an object invokes a method of another object, waits for it to execute and receives the return value or an exception.
- RMI: a client object invokes a method (via a **remote reference**) of a server object. RMI encodes, sends, receives, decodes and executes the invocation on the server object. The result of the invocation is returned in the same way to the client.
- RMI calls are like normal calls: synchronous (i.e. they block the caller until the method returns from execution)
- RMI calls are different from normal calls: they can fail because of transmission problems.

The Remote Interface

Each Remote Object must implement a Remote Interface:

```
import java.rmi.*;

public interface Database extends java.rmi.Remote{

    public String getRecord(int index)
                        throws RemoteException;
    public void addRecord(int index, String record)
                        throws RemoteException;
    public int getSize()
                        throws RemoteException;
}
```


The Remote Interface must:

1. be declared **`public`**
2. extend **`java.rmi.Remote`** interface
3. have **`java.rmi.RemoteException`** in signature of each method (+ other exceptions)
4. use remote interfaces of remote objects in signatures of its methods, not implementation classes!

The Implementation Object

The Implementation Object lives on the Server.
This object can be accessed from network using RMI protocol.

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

public class DatabaseImpl extends UnicastRemoteObject
                        implements Database {

    public DatabaseImpl()throws RemoteException{ ... }

    public String getRecord(int index){ ... }
    public String addRecord(int index, String record)
    { ... }
    public int getSize() { ... }
}
```


The Implementation object must:

1. implement a remote interface
2. define the constructor
3. provide implementation of methods declared in remote interface (obvious)

Extending `UnicastRemoteObject` adds some communication facilities to the implementation class:

- Define that default socket-based transport of RMI will be used
- The object will run “all the time”

Notes on Constructor of Implementation Object

To start listening for incoming requests from clients, an implementation object must be exported :

`export` operation throws an `RemoteException`

- The default constructor of `UnicastRemoteObject` *exports* the object automatically

```
    public DatabaseImpl()throws RemoteException{
        super();
    }
```

- The other possibility (not a sub-class of `UnicastRemoteObject`):

```
    public class DBImpl_2 implements Database /*no extends*/{
    public DBImpl_2()throws RemoteException{
        UnicastRemoteObject.exportObject(this);
    }
```


Implementation of remote methods

```
public String getRecord(int index){
    String record = ...;
    return record;
}
```

- Arguments and return values of a remote method can be of any type
- Non-remote arguments/return values are serialized (i.e. a copy is sent)

NB: Non-remote arguments/return values must implement `Serializable` interface!

- For remote arguments/return values a remote reference is sent (i.e. no copy is made).

Creation of a Remote object on the Server

1. When a remote object is created it is automatically exported — ready to accept incoming requests:

```
DatabaseImpl db1 = new DatabaseImpl();
```

2. To be "visible" to the clients, remote object must be *registered* in RMI registry:

```
Naming.rebind( "DB_ONE", db1);
```

First argument of rebind is the name assigned to the remote object. Syntax:

```
full:                              "//host:port/name"
assuming default rmiregistry port: "//host/name"
assuming localhost and default port: "name"
```

For security purposes `host` can only be the local host.
Obviously, to execute a `rebind` or a `lookup` RMIRegistry must be started!

Remote Method Invocation

How does the client invoke the method of a remote object?

1. Obtain a reference to the remote object

```
try{
    Database db =
     (Database)Naming.lookup("//www.epfl.ch/DB_ONE");
```

2. Invoke a method as usual

```
    String reply = db.getRecord(123);
}catch(Exception e){e.printStackTrace();}
```


Generating Stubs

Run the `rmic` tool on a compiled **implementation** class (not on remote interface):

```
rmic DatabaseImpl
```

or for classes in a package:

```
rmic rmicourse.DatabaseImpl
```

Use `-v1.2` flag of `rmic` to generate less code! (not compatible with RMI v1.1)

Run RMI Registry

The RMI Registry is a name server that allows clients to get a reference to a remote object.

To start the RMI Registry from a shell:

```
rmiregistry or
rmiregistry 8000
```

To start the RMI Registry programmatically:

```
java.rmi.registry.LocateRegistry.createRegistry(8000);
```

The default RMI Registry port is #1099
Each time the remote interface is modified the RMI Registry must be restarted!

RMI Checklist

1. Remote interface (extends java.rmi.Remote)

2. Implementation object (implements 1, extends UnicastRemoteObject)

3. javac

4. rmic

5. start rmiregistry

6. rebind

7. lookup

For Advanced Users

The Java RMI is highly customizable:

- use `java.rmi.activation.Activatable` instead of `UnicastRemoteObject` if you want a remote object that can be activated (created) when a client requests it.
- use `RMISocketFactory` if you want a network protocol other than TCP/IP.
- install a `RMISecurityManager` if the client and the server do not share the same CLASSPATH and
- specify a URL from where remote stubs and other classes are loaded by the client and the server.

Serialization

Remember: non-remote arguments/return values of a remote method must implement `Serializable` interface.

```
public class Bidule implements java.io.Serializable{
    private String name = “EPFL”;//////// YES
    private transient Image hugeBitmap;// NO
    private static int nextBidule= 0;/// NO
}
```

For further information please read the serialization guide

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Networking in Java

Stanislav Chachkov

Department of Computer Science
Software Engineering Laboratory
April 2001

Useful Links

➥ Main Java Site - http://www.javasoft.com

➥ EPFL Java Site (docs and tutorial!) - http://sunwww:1995.epfl.ch/Java/

➥ Java RMI Guide — http://sunwww:1995.epfl.ch/Java/jdk1.3/docs/guide/rmi/

➥ Java Object Serialization Guide — http://sunwww.epfl.ch:1995/Java/jdk1.3/docs/guide/serialization/

➥ Java Tutorial on RMI — http://sunwww:1995.epfl.ch/for_EPFL/javatut_28Feb2001/

Sockets: `java.net` package

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

- Client:

```
try{
    Socket s = new Socket("www.epfl.ch", 8000);
    s.getOutputStream().write(request);
    reply = s.getInputStream().read();
    s.close();
}catch(IOException ioe){...}
```

- Server:

```
try{
    ServerSocket ss = new ServerSocket(8000);
    Socket s = ss.accept();
    request = s.getInputStream().read(); //do something ...
    s.getOutputStream().write(reply);
    s.close();
}catch(IOException ioe){...}
```


Another scenario for `ServerSocket`

The Server: maximum availability for multiple clients

```
try{
    ServerSocket ss = new ServerSocket(8000);
    while(true)
        new Handler(ss.accept()).start();
}catch(IOException ioe){...}
```

... and the Handler: one Handler thread for each client

```
class Handler extends Thread{
    Socket myConnection;
    Handler(Socket s){ myConnection = s; }

    public void run(){
        //..... serve the client, for example:
        request = s.getInputStream().read();//then do something ...
        s.getOutputStream().write(reply);
        s.close();
    }
}
```


The Need for a Protocol

What is the meaning of exchanged messages?

For example:
Server receives `request="G123"` and returns a `reply="EPFL"`.

That can mean:
`request`="give me record #123 from your database #G"
`reply` ="OK, the record you asked is "EPFL""

- Protocol gives a meaning to the messages.
- An implementation of a protocol "encodes" and "decodes" the messages.

Remote Method Invocation

Java "Protocol" + Network = RMI

- Java "Protocol": an object invokes a method of another object, waits for it to execute and receives the return value or an exception.
- RMI: a client object invokes a method (via a **remote reference**) of a server object. RMI encodes, sends, receives, decodes and executes the invocation on the server object. The result of the invocation is returned in the same way to the client.
- RMI calls are like normal calls: synchronous (i.e. they block the caller until the method returns from execution)
- RMI calls are different from normal calls: they can fail because of transmission problems.

The Remote Interface

Each Remote Object must implement a Remote Interface:

```
import java.rmi.*;

public interface Database extends java.rmi.Remote{

    public String getRecord(int index)
                        throws RemoteException;
    public void addRecord(int index, String record)
                        throws RemoteException;
    public int getSize()
                        throws RemoteException;
}
```


The Remote Interface must:

1. be declared **`public`**
2. extend **`java.rmi.Remote`** interface
3. have **`java.rmi.RemoteException`** in signature of each method (+ other exceptions)
4. use remote interfaces of remote objects in signatures of its methods, not implementation classes!

The Implementation Object

The Implementation Object lives on the Server.
This object can be accessed from network using RMI protocol.

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

public class DatabaseImpl extends UnicastRemoteObject
                        implements Database {

    public DatabaseImpl()throws RemoteException{ ... }

    public String getRecord(int index){ ... }
    public String addRecord(int index, String record)
    { ... }
    public int getSize() { ... }
}
```


The Implementation object must:

1. implement a remote interface
2. define the constructor
3. provide implementation of methods declared in remote interface (obvious)

Extending `UnicastRemoteObject` adds some communication facilities to the implementation class:

- Define that default socket-based transport of RMI will be used
- The object will run “all the time”

Notes on Constructor of Implementation Object

To start listening for incoming requests from clients, an implementation object must be exported :

`export` operation throws an `RemoteException`

- The default constructor of `UnicastRemoteObject` *exports* the object automatically

```
    public DatabaseImpl()throws RemoteException{
        super();
    }
```

- The other possibility (not a sub-class of `UnicastRemoteObject`):

```
    public class DBImpl_2 implements Database /*no extends*/{
    public DBImpl_2()throws RemoteException{
        UnicastRemoteObject.exportObject(this);
    }
```


Implementation of remote methods

```
public String getRecord(int index){
    String record = ...;
    return record;
}
```

- Arguments and return values of a remote method can be of any type
- Non-remote arguments/return values are serialized (i.e. a copy is sent)

NB: Non-remote arguments/return values must implement `Serializable` interface!

- For remote arguments/return values a remote reference is sent (i.e. no copy is made).

Creation of a Remote object on the Server

1. When a remote object is created it is automatically exported — ready to accept incoming requests:

```
DatabaseImpl db1 = new DatabaseImpl();
```

2. To be "visible" to the clients, remote object must be *registered* in RMI registry:

```
Naming.rebind( "DB_ONE", db1);
```

First argument of rebind is the name assigned to the remote object. Syntax:

```
full:                              "//host:port/name"
assuming default rmiregistry port: "//host/name"
assuming localhost and default port: "name"
```

For security purposes `host` can only be the local host.
Obviously, to execute a `rebind` or a `lookup` RMIRegistry must be started!

Remote Method Invocation

How does the client invoke the method of a remote object?

1. Obtain a reference to the remote object

```
try{
    Database db =
     (Database)Naming.lookup("//www.epfl.ch/DB_ONE");
```

2. Invoke a method as usual

```
    String reply = db.getRecord(123);
}catch(Exception e){e.printStackTrace();}
```


Generating Stubs

Run the `rmic` tool on a compiled **implementation** class (not on remote interface):

```
rmic DatabaseImpl
```

or for classes in a package:

```
rmic rmicourse.DatabaseImpl
```

Use `-v1.2` flag of `rmic` to generate less code! (not compatible with RMI v1.1)

Run RMI Registry

The RMI Registry is a name server that allows clients to get a reference to a remote object.

To start the RMI Registry from a shell:

```
rmiregistry or
rmiregistry 8000
```

To start the RMI Registry programmatically:

```
java.rmi.registry.LocateRegistry.createRegistry(8000);
```

The default RMI Registry port is #1099
Each time the remote interface is modified the RMI Registry must be restarted!

RMI Checklist

1. Remote interface (extends java.rmi.Remote)

2. Implementation object (implements 1, extends UnicastRemoteObject)

3. javac

4. rmic

5. start rmiregistry

6. rebind

7. lookup

For Advanced Users

The Java RMI is highly customizable:

- use `java.rmi.activation.Activatable` instead of `UnicastRemoteObject` if you want a remote object that can be activated (created) when a client requests it.
- use `RMISocketFactory` if you want a network protocol other than TCP/IP.
- install a `RMISecurityManager` if the client and the server do not share the same CLASSPATH and
- specify a URL from where remote stubs and other classes are loaded by the client and the server.

Serialization

Remember: non-remote arguments/return values of a remote method must implement `Serializable` interface.

```
public class Bidule implements java.io.Serializable{
    private String name = “EPFL”;//////// YES
    private transient Image hugeBitmap;// NO
    private static int nextBidule= 0;/// NO
}
```

For further information please read the serialization guide

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Networking in Java

Stanislav Chachkov

Department of Computer Science
Software Engineering Laboratory
April 2001

Useful Links

➥ Main Java Site - http://www.javasoft.com

➥ EPFL Java Site (docs and tutorial!) - http://sunwww:1995.epfl.ch/Java/

➥ Java RMI Guide — http://sunwww:1995.epfl.ch/Java/jdk1.3/docs/guide/rmi/

➥ Java Object Serialization Guide — http://sunwww.epfl.ch:1995/Java/jdk1.3/docs/guide/serialization/

➥ Java Tutorial on RMI — http://sunwww:1995.epfl.ch/for_EPFL/javatut_28Feb2001/

Sockets: `java.net` package

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

- Client:

```
try{
    Socket s = new Socket("www.epfl.ch", 8000);
    s.getOutputStream().write(request);
    reply = s.getInputStream().read();
    s.close();
}catch(IOException ioe){...}
```

- Server:

```
try{
    ServerSocket ss = new ServerSocket(8000);
    Socket s = ss.accept();
    request = s.getInputStream().read(); //do something ...
    s.getOutputStream().write(reply);
    s.close();
}catch(IOException ioe){...}
```


Another scenario for `ServerSocket`

The Server: maximum availability for multiple clients

```
try{
    ServerSocket ss = new ServerSocket(8000);
    while(true)
        new Handler(ss.accept()).start();
}catch(IOException ioe){...}
```

... and the Handler: one Handler thread for each client

```
class Handler extends Thread{
    Socket myConnection;
    Handler(Socket s){ myConnection = s; }

    public void run(){
        //..... serve the client, for example:
        request = s.getInputStream().read();//then do something ...
        s.getOutputStream().write(reply);
        s.close();
    }
}
```


The Need for a Protocol

What is the meaning of exchanged messages?

For example:
Server receives `request="G123"` and returns a `reply="EPFL"`.

That can mean:
`request`="give me record #123 from your database #G"
`reply` ="OK, the record you asked is "EPFL""

- Protocol gives a meaning to the messages.
- An implementation of a protocol "encodes" and "decodes" the messages.

Remote Method Invocation

Java "Protocol" + Network = RMI

- Java "Protocol": an object invokes a method of another object, waits for it to execute and receives the return value or an exception.
- RMI: a client object invokes a method (via a **remote reference**) of a server object. RMI encodes, sends, receives, decodes and executes the invocation on the server object. The result of the invocation is returned in the same way to the client.
- RMI calls are like normal calls: synchronous (i.e. they block the caller until the method returns from execution)
- RMI calls are different from normal calls: they can fail because of transmission problems.

The Remote Interface

Each Remote Object must implement a Remote Interface:

```
import java.rmi.*;

public interface Database extends java.rmi.Remote{

    public String getRecord(int index)
                        throws RemoteException;
    public void addRecord(int index, String record)
                        throws RemoteException;
    public int getSize()
                        throws RemoteException;
}
```


The Remote Interface must:

1. be declared **`public`**
2. extend **`java.rmi.Remote`** interface
3. have **`java.rmi.RemoteException`** in signature of each method (+ other exceptions)
4. use remote interfaces of remote objects in signatures of its methods, not implementation classes!

The Implementation Object

The Implementation Object lives on the Server.
This object can be accessed from network using RMI protocol.

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

public class DatabaseImpl extends UnicastRemoteObject
                        implements Database {

    public DatabaseImpl()throws RemoteException{ ... }

    public String getRecord(int index){ ... }
    public String addRecord(int index, String record)
    { ... }
    public int getSize() { ... }
}
```


The Implementation object must:

1. implement a remote interface
2. define the constructor
3. provide implementation of methods declared in remote interface (obvious)

Extending `UnicastRemoteObject` adds some communication facilities to the implementation class:

- Define that default socket-based transport of RMI will be used
- The object will run “all the time”

Notes on Constructor of Implementation Object

To start listening for incoming requests from clients, an implementation object must be exported :

`export` operation throws an `RemoteException`

- The default constructor of `UnicastRemoteObject` *exports* the object automatically

```
    public DatabaseImpl()throws RemoteException{
        super();
    }
```

- The other possibility (not a sub-class of `UnicastRemoteObject`):

```
    public class DBImpl_2 implements Database /*no extends*/{
    public DBImpl_2()throws RemoteException{
        UnicastRemoteObject.exportObject(this);
    }
```


Implementation of remote methods

```
public String getRecord(int index){
    String record = ...;
    return record;
}
```

- Arguments and return values of a remote method can be of any type
- Non-remote arguments/return values are serialized (i.e. a copy is sent)

NB: Non-remote arguments/return values must implement `Serializable` interface!

- For remote arguments/return values a remote reference is sent (i.e. no copy is made).

Creation of a Remote object on the Server

1. When a remote object is created it is automatically exported — ready to accept incoming requests:

```
DatabaseImpl db1 = new DatabaseImpl();
```

2. To be "visible" to the clients, remote object must be *registered* in RMI registry:

```
Naming.rebind( "DB_ONE", db1);
```

First argument of rebind is the name assigned to the remote object. Syntax:

```
full:                              "//host:port/name"
assuming default rmiregistry port: "//host/name"
assuming localhost and default port: "name"
```

For security purposes `host` can only be the local host.
Obviously, to execute a `rebind` or a `lookup` RMIRegistry must be started!

Remote Method Invocation

How does the client invoke the method of a remote object?

1. Obtain a reference to the remote object

```
try{
    Database db =
     (Database)Naming.lookup("//www.epfl.ch/DB_ONE");
```

2. Invoke a method as usual

```
    String reply = db.getRecord(123);
}catch(Exception e){e.printStackTrace();}
```


Generating Stubs

Run the `rmic` tool on a compiled **implementation** class (not on remote interface):

```
rmic DatabaseImpl
```

or for classes in a package:

```
rmic rmicourse.DatabaseImpl
```

Use `-v1.2` flag of `rmic` to generate less code! (not compatible with RMI v1.1)

Run RMI Registry

The RMI Registry is a name server that allows clients to get a reference to a remote object.

To start the RMI Registry from a shell:

```
rmiregistry or
rmiregistry 8000
```

To start the RMI Registry programmatically:

```
java.rmi.registry.LocateRegistry.createRegistry(8000);
```

The default RMI Registry port is #1099
Each time the remote interface is modified the RMI Registry must be restarted!

RMI Checklist

1. Remote interface (extends java.rmi.Remote)

2. Implementation object (implements 1, extends UnicastRemoteObject)

3. javac

4. rmic

5. start rmiregistry

6. rebind

7. lookup

For Advanced Users

The Java RMI is highly customizable:

- use `java.rmi.activation.Activatable` instead of `UnicastRemoteObject` if you want a remote object that can be activated (created) when a client requests it.
- use `RMISocketFactory` if you want a network protocol other than TCP/IP.
- install a `RMISecurityManager` if the client and the server do not share the same CLASSPATH and
- specify a URL from where remote stubs and other classes are loaded by the client and the server.

Serialization

Remember: non-remote arguments/return values of a remote method must implement `Serializable` interface.

```
public class Bidule implements java.io.Serializable{
    private String name = “EPFL”;//////// YES
    private transient Image hugeBitmap;// NO
    private static int nextBidule= 0;/// NO
}
```

For further information please read the serialization guide

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Networking in Java

Stanislav Chachkov

Department of Computer Science
Software Engineering Laboratory
April 2001

Useful Links

➥ Main Java Site - http://www.javasoft.com

➥ EPFL Java Site (docs and tutorial!) - http://sunwww:1995.epfl.ch/Java/

➥ Java RMI Guide — http://sunwww:1995.epfl.ch/Java/jdk1.3/docs/guide/rmi/

➥ Java Object Serialization Guide — http://sunwww.epfl.ch:1995/Java/jdk1.3/docs/guide/serialization/

➥ Java Tutorial on RMI — http://sunwww:1995.epfl.ch/for_EPFL/javatut_28Feb2001/

Sockets: `java.net` package

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

- Client:

```
try{
    Socket s = new Socket("www.epfl.ch", 8000);
    s.getOutputStream().write(request);
    reply = s.getInputStream().read();
    s.close();
}catch(IOException ioe){...}
```

- Server:

```
try{
    ServerSocket ss = new ServerSocket(8000);
    Socket s = ss.accept();
    request = s.getInputStream().read(); //do something ...
    s.getOutputStream().write(reply);
    s.close();
}catch(IOException ioe){...}
```


Another scenario for `ServerSocket`

The Server: maximum availability for multiple clients

```
try{
    ServerSocket ss = new ServerSocket(8000);
    while(true)
        new Handler(ss.accept()).start();
}catch(IOException ioe){...}
```

... and the Handler: one Handler thread for each client

```
class Handler extends Thread{
    Socket myConnection;
    Handler(Socket s){ myConnection = s; }

    public void run(){
        //..... serve the client, for example:
        request = s.getInputStream().read();//then do something ...
        s.getOutputStream().write(reply);
        s.close();
    }
}
```


The Need for a Protocol

What is the meaning of exchanged messages?

For example:
Server receives `request="G123"` and returns a `reply="EPFL"`.

That can mean:
`request`="give me record #123 from your database #G"
`reply` ="OK, the record you asked is "EPFL""

- Protocol gives a meaning to the messages.
- An implementation of a protocol "encodes" and "decodes" the messages.

Remote Method Invocation

Java "Protocol" + Network = RMI

- Java "Protocol": an object invokes a method of another object, waits for it to execute and receives the return value or an exception.
- RMI: a client object invokes a method (via a **remote reference**) of a server object. RMI encodes, sends, receives, decodes and executes the invocation on the server object. The result of the invocation is returned in the same way to the client.
- RMI calls are like normal calls: synchronous (i.e. they block the caller until the method returns from execution)
- RMI calls are different from normal calls: they can fail because of transmission problems.

The Remote Interface

Each Remote Object must implement a Remote Interface:

```
import java.rmi.*;

public interface Database extends java.rmi.Remote{

    public String getRecord(int index)
                        throws RemoteException;
    public void addRecord(int index, String record)
                        throws RemoteException;
    public int getSize()
                        throws RemoteException;
}
```


The Remote Interface must:

1. be declared **`public`**
2. extend **`java.rmi.Remote`** interface
3. have **`java.rmi.RemoteException`** in signature of each method (+ other exceptions)
4. use remote interfaces of remote objects in signatures of its methods, not implementation classes!

The Implementation Object

The Implementation Object lives on the Server.
This object can be accessed from network using RMI protocol.

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

public class DatabaseImpl extends UnicastRemoteObject
                        implements Database {

    public DatabaseImpl()throws RemoteException{ ... }

    public String getRecord(int index){ ... }
    public String addRecord(int index, String record)
    { ... }
    public int getSize() { ... }
}
```


The Implementation object must:

1. implement a remote interface
2. define the constructor
3. provide implementation of methods declared in remote interface (obvious)

Extending `UnicastRemoteObject` adds some communication facilities to the implementation class:

- Define that default socket-based transport of RMI will be used
- The object will run “all the time”

Notes on Constructor of Implementation Object

To start listening for incoming requests from clients, an implementation object must be exported :

`export` operation throws an `RemoteException`

- The default constructor of `UnicastRemoteObject` *exports* the object automatically

```
    public DatabaseImpl()throws RemoteException{
        super();
    }
```

- The other possibility (not a sub-class of `UnicastRemoteObject`):

```
    public class DBImpl_2 implements Database /*no extends*/{
    public DBImpl_2()throws RemoteException{
        UnicastRemoteObject.exportObject(this);
    }
```


Implementation of remote methods

```
public String getRecord(int index){
    String record = ...;
    return record;
}
```

- Arguments and return values of a remote method can be of any type
- Non-remote arguments/return values are serialized (i.e. a copy is sent)

NB: Non-remote arguments/return values must implement `Serializable` interface!

- For remote arguments/return values a remote reference is sent (i.e. no copy is made).

Creation of a Remote object on the Server

1. When a remote object is created it is automatically exported — ready to accept incoming requests:

```
DatabaseImpl db1 = new DatabaseImpl();
```

2. To be "visible" to the clients, remote object must be *registered* in RMI registry:

```
Naming.rebind( "DB_ONE", db1);
```

First argument of rebind is the name assigned to the remote object. Syntax:

```
full:                              "//host:port/name"
assuming default rmiregistry port: "//host/name"
assuming localhost and default port: "name"
```

For security purposes `host` can only be the local host.
Obviously, to execute a `rebind` or a `lookup` RMIRegistry must be started!

Remote Method Invocation

How does the client invoke the method of a remote object?

1. Obtain a reference to the remote object

```
try{
    Database db =
     (Database)Naming.lookup("//www.epfl.ch/DB_ONE");
```

2. Invoke a method as usual

```
    String reply = db.getRecord(123);
}catch(Exception e){e.printStackTrace();}
```


Generating Stubs

Run the `rmic` tool on a compiled **implementation** class (not on remote interface):

```
rmic DatabaseImpl
```

or for classes in a package:

```
rmic rmicourse.DatabaseImpl
```

Use `-v1.2` flag of `rmic` to generate less code! (not compatible with RMI v1.1)

Run RMI Registry

The RMI Registry is a name server that allows clients to get a reference to a remote object.

To start the RMI Registry from a shell:

```
rmiregistry or
rmiregistry 8000
```

To start the RMI Registry programmatically:

```
java.rmi.registry.LocateRegistry.createRegistry(8000);
```

The default RMI Registry port is #1099
Each time the remote interface is modified the RMI Registry must be restarted!

RMI Checklist

1. Remote interface (extends java.rmi.Remote)

2. Implementation object (implements 1, extends UnicastRemoteObject)

3. javac

4. rmic

5. start rmiregistry

6. rebind

7. lookup

For Advanced Users

The Java RMI is highly customizable:

- use `java.rmi.activation.Activatable` instead of `UnicastRemoteObject` if you want a remote object that can be activated (created) when a client requests it.
- use `RMISocketFactory` if you want a network protocol other than TCP/IP.
- install a `RMISecurityManager` if the client and the server do not share the same CLASSPATH and
- specify a URL from where remote stubs and other classes are loaded by the client and the server.

Serialization

Remember: non-remote arguments/return values of a remote method must implement `Serializable` interface.

```
public class Bidule implements java.io.Serializable{
    private String name = “EPFL”;//////// YES
    private transient Image hugeBitmap;// NO
    private static int nextBidule= 0;/// NO
}
```

For further information please read the serialization guide

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Networking in Java

Stanislav Chachkov

Department of Computer Science
Software Engineering Laboratory
April 2001

Useful Links

➥ Main Java Site - http://www.javasoft.com

➥ EPFL Java Site (docs and tutorial!) - http://sunwww:1995.epfl.ch/Java/

➥ Java RMI Guide — http://sunwww:1995.epfl.ch/Java/jdk1.3/docs/guide/rmi/

➥ Java Object Serialization Guide — http://sunwww.epfl.ch:1995/Java/jdk1.3/docs/guide/serialization/

➥ Java Tutorial on RMI — http://sunwww:1995.epfl.ch/for_EPFL/javatut_28Feb2001/

Sockets: `java.net` package

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

- Client:

```
try{
    Socket s = new Socket("www.epfl.ch", 8000);
    s.getOutputStream().write(request);
    reply = s.getInputStream().read();
    s.close();
}catch(IOException ioe){...}
```

- Server:

```
try{
    ServerSocket ss = new ServerSocket(8000);
    Socket s = ss.accept();
    request = s.getInputStream().read(); //do something ...
    s.getOutputStream().write(reply);
    s.close();
}catch(IOException ioe){...}
```


Another scenario for `ServerSocket`

The Server: maximum availability for multiple clients

```
try{
    ServerSocket ss = new ServerSocket(8000);
    while(true)
        new Handler(ss.accept()).start();
}catch(IOException ioe){...}
```

... and the Handler: one Handler thread for each client

```
class Handler extends Thread{
    Socket myConnection;
    Handler(Socket s){ myConnection = s; }

    public void run(){
        //..... serve the client, for example:
        request = s.getInputStream().read();//then do something ...
        s.getOutputStream().write(reply);
        s.close();
    }
}
```


The Need for a Protocol

What is the meaning of exchanged messages?

For example:
Server receives `request="G123"` and returns a `reply="EPFL"`.

That can mean:
`request`="give me record #123 from your database #G"
`reply` ="OK, the record you asked is "EPFL""

- Protocol gives a meaning to the messages.
- An implementation of a protocol "encodes" and "decodes" the messages.

Remote Method Invocation

Java "Protocol" + Network = RMI

- Java "Protocol": an object invokes a method of another object, waits for it to execute and receives the return value or an exception.
- RMI: a client object invokes a method (via a **remote reference**) of a server object. RMI encodes, sends, receives, decodes and executes the invocation on the server object. The result of the invocation is returned in the same way to the client.
- RMI calls are like normal calls: synchronous (i.e. they block the caller until the method returns from execution)
- RMI calls are different from normal calls: they can fail because of transmission problems.

The Remote Interface

Each Remote Object must implement a Remote Interface:

```
import java.rmi.*;

public interface Database extends java.rmi.Remote{

    public String getRecord(int index)
                        throws RemoteException;
    public void addRecord(int index, String record)
                        throws RemoteException;
    public int getSize()
                        throws RemoteException;
}
```


The Remote Interface must:

1. be declared **`public`**
2. extend **`java.rmi.Remote`** interface
3. have **`java.rmi.RemoteException`** in signature of each method (+ other exceptions)
4. use remote interfaces of remote objects in signatures of its methods, not implementation classes!

The Implementation Object

The Implementation Object lives on the Server.
This object can be accessed from network using RMI protocol.

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

public class DatabaseImpl extends UnicastRemoteObject
                        implements Database {

    public DatabaseImpl()throws RemoteException{ ... }

    public String getRecord(int index){ ... }
    public String addRecord(int index, String record)
    { ... }
    public int getSize() { ... }
}
```


The Implementation object must:

1. implement a remote interface
2. define the constructor
3. provide implementation of methods declared in remote interface (obvious)

Extending `UnicastRemoteObject` adds some communication facilities to the implementation class:

- Define that default socket-based transport of RMI will be used
- The object will run “all the time”

Notes on Constructor of Implementation Object

To start listening for incoming requests from clients, an implementation object must be exported :

`export` operation throws an `RemoteException`

- The default constructor of `UnicastRemoteObject` *exports* the object automatically

```
    public DatabaseImpl()throws RemoteException{
        super();
    }
```

- The other possibility (not a sub-class of `UnicastRemoteObject`):

```
    public class DBImpl_2 implements Database /*no extends*/{
    public DBImpl_2()throws RemoteException{
        UnicastRemoteObject.exportObject(this);
    }
```


Implementation of remote methods

```
public String getRecord(int index){
    String record = ...;
    return record;
}
```

- Arguments and return values of a remote method can be of any type
- Non-remote arguments/return values are serialized (i.e. a copy is sent)

NB: Non-remote arguments/return values must implement `Serializable` interface!

- For remote arguments/return values a remote reference is sent (i.e. no copy is made).

Creation of a Remote object on the Server

1. When a remote object is created it is automatically exported — ready to accept incoming requests:

```
DatabaseImpl db1 = new DatabaseImpl();
```

2. To be "visible" to the clients, remote object must be *registered* in RMI registry:

```
Naming.rebind( "DB_ONE", db1);
```

First argument of rebind is the name assigned to the remote object. Syntax:

```
full:                              "//host:port/name"
assuming default rmiregistry port: "//host/name"
assuming localhost and default port: "name"
```

For security purposes `host` can only be the local host.
Obviously, to execute a `rebind` or a `lookup` RMIRegistry must be started!

Remote Method Invocation

How does the client invoke the method of a remote object?

1. Obtain a reference to the remote object

```
try{
    Database db =
     (Database)Naming.lookup("//www.epfl.ch/DB_ONE");
```

2. Invoke a method as usual

```
    String reply = db.getRecord(123);
}catch(Exception e){e.printStackTrace();}
```


Generating Stubs

Run the `rmic` tool on a compiled **implementation** class (not on remote interface):

```
rmic DatabaseImpl
```

or for classes in a package:

```
rmic rmicourse.DatabaseImpl
```

Use `-v1.2` flag of `rmic` to generate less code! (not compatible with RMI v1.1)

Run RMI Registry

The RMI Registry is a name server that allows clients to get a reference to a remote object.

To start the RMI Registry from a shell:

```
rmiregistry or
rmiregistry 8000
```

To start the RMI Registry programmatically:

```
java.rmi.registry.LocateRegistry.createRegistry(8000);
```

The default RMI Registry port is #1099
Each time the remote interface is modified the RMI Registry must be restarted!

RMI Checklist

1. Remote interface (extends java.rmi.Remote)

2. Implementation object (implements 1, extends UnicastRemoteObject)

3. javac

4. rmic

5. start rmiregistry

6. rebind

7. lookup

For Advanced Users

The Java RMI is highly customizable:

- use `java.rmi.activation.Activatable` instead of `UnicastRemoteObject` if you want a remote object that can be activated (created) when a client requests it.
- use `RMISocketFactory` if you want a network protocol other than TCP/IP.
- install a `RMISecurityManager` if the client and the server do not share the same CLASSPATH and
- specify a URL from where remote stubs and other classes are loaded by the client and the server.

Serialization

Remember: non-remote arguments/return values of a remote method must implement `Serializable` interface.

```
public class Bidule implements java.io.Serializable{
    private String name = “EPFL”;//////// YES
    private transient Image hugeBitmap;// NO
    private static int nextBidule= 0;/// NO
}
```

For further information please read the serialization guide

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Networking in Java

Stanislav Chachkov

Department of Computer Science
Software Engineering Laboratory
April 2001

Useful Links

➥ Main Java Site - http://www.javasoft.com

➥ EPFL Java Site (docs and tutorial!) - http://sunwww:1995.epfl.ch/Java/

➥ Java RMI Guide — http://sunwww:1995.epfl.ch/Java/jdk1.3/docs/guide/rmi/

➥ Java Object Serialization Guide — http://sunwww.epfl.ch:1995/Java/jdk1.3/docs/guide/serialization/

➥ Java Tutorial on RMI — http://sunwww:1995.epfl.ch/for_EPFL/javatut_28Feb2001/

Sockets: `java.net` package

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

- Client:

```
try{
    Socket s = new Socket("www.epfl.ch", 8000);
    s.getOutputStream().write(request);
    reply = s.getInputStream().read();
    s.close();
}catch(IOException ioe){...}
```

- Server:

```
try{
    ServerSocket ss = new ServerSocket(8000);
    Socket s = ss.accept();
    request = s.getInputStream().read(); //do something ...
    s.getOutputStream().write(reply);
    s.close();
}catch(IOException ioe){...}
```


Another scenario for `ServerSocket`

The Server: maximum availability for multiple clients

```
try{
    ServerSocket ss = new ServerSocket(8000);
    while(true)
        new Handler(ss.accept()).start();
}catch(IOException ioe){...}
```

... and the Handler: one Handler thread for each client

```
class Handler extends Thread{
    Socket myConnection;
    Handler(Socket s){ myConnection = s; }

    public void run(){
        //..... serve the client, for example:
        request = s.getInputStream().read();//then do something ...
        s.getOutputStream().write(reply);
        s.close();
    }
}
```


The Need for a Protocol

What is the meaning of exchanged messages?

For example:
Server receives `request="G123"` and returns a `reply="EPFL"`.

That can mean:
`request`="give me record #123 from your database #G"
`reply` ="OK, the record you asked is "EPFL""

- Protocol gives a meaning to the messages.
- An implementation of a protocol "encodes" and "decodes" the messages.

Remote Method Invocation

Java "Protocol" + Network = RMI

- Java "Protocol": an object invokes a method of another object, waits for it to execute and receives the return value or an exception.
- RMI: a client object invokes a method (via a **remote reference**) of a server object. RMI encodes, sends, receives, decodes and executes the invocation on the server object. The result of the invocation is returned in the same way to the client.
- RMI calls are like normal calls: synchronous (i.e. they block the caller until the method returns from execution)
- RMI calls are different from normal calls: they can fail because of transmission problems.

The Remote Interface

Each Remote Object must implement a Remote Interface:

```
import java.rmi.*;

public interface Database extends java.rmi.Remote{

    public String getRecord(int index)
                        throws RemoteException;
    public void addRecord(int index, String record)
                        throws RemoteException;
    public int getSize()
                        throws RemoteException;
}
```


The Remote Interface must:

1. be declared **`public`**
2. extend **`java.rmi.Remote`** interface
3. have **`java.rmi.RemoteException`** in signature of each method (+ other exceptions)
4. use remote interfaces of remote objects in signatures of its methods, not implementation classes!

The Implementation Object

The Implementation Object lives on the Server.
This object can be accessed from network using RMI protocol.

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

public class DatabaseImpl extends UnicastRemoteObject
                        implements Database {

    public DatabaseImpl()throws RemoteException{ ... }

    public String getRecord(int index){ ... }
    public String addRecord(int index, String record)
    { ... }
    public int getSize() { ... }
}
```


The Implementation object must:

1. implement a remote interface
2. define the constructor
3. provide implementation of methods declared in remote interface (obvious)

Extending `UnicastRemoteObject` adds some communication facilities to the implementation class:

- Define that default socket-based transport of RMI will be used
- The object will run “all the time”

Notes on Constructor of Implementation Object

To start listening for incoming requests from clients, an implementation object must be exported :

`export` operation throws an `RemoteException`

- The default constructor of `UnicastRemoteObject` *exports* the object automatically

```
    public DatabaseImpl()throws RemoteException{
        super();
    }
```

- The other possibility (not a sub-class of `UnicastRemoteObject`):

```
    public class DBImpl_2 implements Database /*no extends*/{
    public DBImpl_2()throws RemoteException{
        UnicastRemoteObject.exportObject(this);
    }
```


Implementation of remote methods

```
public String getRecord(int index){
    String record = ...;
    return record;
}
```

- Arguments and return values of a remote method can be of any type
- Non-remote arguments/return values are serialized (i.e. a copy is sent)

NB: Non-remote arguments/return values must implement `Serializable` interface!

- For remote arguments/return values a remote reference is sent (i.e. no copy is made).

Creation of a Remote object on the Server

1. When a remote object is created it is automatically exported — ready to accept incoming requests:

```
DatabaseImpl db1 = new DatabaseImpl();
```

2. To be "visible" to the clients, remote object must be *registered* in RMI registry:

```
Naming.rebind( "DB_ONE", db1);
```

First argument of rebind is the name assigned to the remote object. Syntax:

```
full:                              "//host:port/name"
assuming default rmiregistry port: "//host/name"
assuming localhost and default port: "name"
```

For security purposes `host` can only be the local host.
Obviously, to execute a `rebind` or a `lookup` RMIRegistry must be started!

Remote Method Invocation

How does the client invoke the method of a remote object?

1. Obtain a reference to the remote object

```
try{
    Database db =
     (Database)Naming.lookup("//www.epfl.ch/DB_ONE");
```

2. Invoke a method as usual

```
    String reply = db.getRecord(123);
}catch(Exception e){e.printStackTrace();}
```


Generating Stubs

Run the `rmic` tool on a compiled **implementation** class (not on remote interface):

```
rmic DatabaseImpl
```

or for classes in a package:

```
rmic rmicourse.DatabaseImpl
```

Use `-v1.2` flag of `rmic` to generate less code! (not compatible with RMI v1.1)

Run RMI Registry

The RMI Registry is a name server that allows clients to get a reference to a remote object.

To start the RMI Registry from a shell:

```
rmiregistry or
rmiregistry 8000
```

To start the RMI Registry programmatically:

```
java.rmi.registry.LocateRegistry.createRegistry(8000);
```

The default RMI Registry port is #1099
Each time the remote interface is modified the RMI Registry must be restarted!

RMI Checklist

1. Remote interface (extends java.rmi.Remote)

2. Implementation object (implements 1, extends UnicastRemoteObject)

3. javac

4. rmic

5. start rmiregistry

6. rebind

7. lookup

For Advanced Users

The Java RMI is highly customizable:

- use `java.rmi.activation.Activatable` instead of `UnicastRemoteObject` if you want a remote object that can be activated (created) when a client requests it.
- use `RMISocketFactory` if you want a network protocol other than TCP/IP.
- install a `RMISecurityManager` if the client and the server do not share the same CLASSPATH and
- specify a URL from where remote stubs and other classes are loaded by the client and the server.

Serialization

Remember: non-remote arguments/return values of a remote method must implement `Serializable` interface.

```
public class Bidule implements java.io.Serializable{
    private String name = “EPFL”;//////// YES
    private transient Image hugeBitmap;// NO
    private static int nextBidule= 0;/// NO
}
```

For further information please read the serialization guide

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Networking in Java

Stanislav Chachkov

Department of Computer Science
Software Engineering Laboratory
April 2001

Useful Links

➥ Main Java Site - http://www.javasoft.com

➥ EPFL Java Site (docs and tutorial!) - http://sunwww:1995.epfl.ch/Java/

➥ Java RMI Guide — http://sunwww:1995.epfl.ch/Java/jdk1.3/docs/guide/rmi/

➥ Java Object Serialization Guide — http://sunwww.epfl.ch:1995/Java/jdk1.3/docs/guide/serialization/

➥ Java Tutorial on RMI — http://sunwww:1995.epfl.ch/for_EPFL/javatut_28Feb2001/

Sockets: `java.net` package

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

- Client:

```
try{
    Socket s = new Socket("www.epfl.ch", 8000);
    s.getOutputStream().write(request);
    reply = s.getInputStream().read();
    s.close();
}catch(IOException ioe){...}
```

- Server:

```
try{
    ServerSocket ss = new ServerSocket(8000);
    Socket s = ss.accept();
    request = s.getInputStream().read(); //do something ...
    s.getOutputStream().write(reply);
    s.close();
}catch(IOException ioe){...}
```


Another scenario for `ServerSocket`

The Server: maximum availability for multiple clients

```
try{
    ServerSocket ss = new ServerSocket(8000);
    while(true)
        new Handler(ss.accept()).start();
}catch(IOException ioe){...}
```

... and the Handler: one Handler thread for each client

```
class Handler extends Thread{
    Socket myConnection;
    Handler(Socket s){ myConnection = s; }

    public void run(){
        //..... serve the client, for example:
        request = s.getInputStream().read();//then do something ...
        s.getOutputStream().write(reply);
        s.close();
    }
}
```


The Need for a Protocol

What is the meaning of exchanged messages?

For example:
Server receives `request="G123"` and returns a `reply="EPFL"`.

That can mean:
`request`="give me record #123 from your database #G"
`reply` ="OK, the record you asked is "EPFL""

- Protocol gives a meaning to the messages.
- An implementation of a protocol "encodes" and "decodes" the messages.

Remote Method Invocation

Java "Protocol" + Network = RMI

- Java "Protocol": an object invokes a method of another object, waits for it to execute and receives the return value or an exception.
- RMI: a client object invokes a method (via a **remote reference**) of a server object. RMI encodes, sends, receives, decodes and executes the invocation on the server object. The result of the invocation is returned in the same way to the client.
- RMI calls are like normal calls: synchronous (i.e. they block the caller until the method returns from execution)
- RMI calls are different from normal calls: they can fail because of transmission problems.

The Remote Interface

Each Remote Object must implement a Remote Interface:

```
import java.rmi.*;

public interface Database extends java.rmi.Remote{

    public String getRecord(int index)
                        throws RemoteException;
    public void addRecord(int index, String record)
                        throws RemoteException;
    public int getSize()
                        throws RemoteException;
}
```


The Remote Interface must:

1. be declared **`public`**
2. extend **`java.rmi.Remote`** interface
3. have **`java.rmi.RemoteException`** in signature of each method (+ other exceptions)
4. use remote interfaces of remote objects in signatures of its methods, not implementation classes!

The Implementation Object

The Implementation Object lives on the Server.
This object can be accessed from network using RMI protocol.

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

public class DatabaseImpl extends UnicastRemoteObject
                        implements Database {

    public DatabaseImpl()throws RemoteException{ ... }

    public String getRecord(int index){ ... }
    public String addRecord(int index, String record)
    { ... }
    public int getSize() { ... }
}
```


The Implementation object must:

1. implement a remote interface
2. define the constructor
3. provide implementation of methods declared in remote interface (obvious)

Extending `UnicastRemoteObject` adds some communication facilities to the implementation class:

- Define that default socket-based transport of RMI will be used
- The object will run “all the time”

Notes on Constructor of Implementation Object

To start listening for incoming requests from clients, an implementation object must be exported :

`export` operation throws an `RemoteException`

- The default constructor of `UnicastRemoteObject` *exports* the object automatically

```
    public DatabaseImpl()throws RemoteException{
        super();
    }
```

- The other possibility (not a sub-class of `UnicastRemoteObject`):

```
    public class DBImpl_2 implements Database /*no extends*/{
    public DBImpl_2()throws RemoteException{
        UnicastRemoteObject.exportObject(this);
    }
```


Implementation of remote methods

```
public String getRecord(int index){
    String record = ...;
    return record;
}
```

- Arguments and return values of a remote method can be of any type
- Non-remote arguments/return values are serialized (i.e. a copy is sent)

NB: Non-remote arguments/return values must implement `Serializable` interface!

- For remote arguments/return values a remote reference is sent (i.e. no copy is made).

Creation of a Remote object on the Server

1. When a remote object is created it is automatically exported — ready to accept incoming requests:

```
DatabaseImpl db1 = new DatabaseImpl();
```

2. To be "visible" to the clients, remote object must be *registered* in RMI registry:

```
Naming.rebind( "DB_ONE", db1);
```

First argument of rebind is the name assigned to the remote object. Syntax:

```
full:                              "//host:port/name"
assuming default rmiregistry port: "//host/name"
assuming localhost and default port: "name"
```

For security purposes `host` can only be the local host.
Obviously, to execute a `rebind` or a `lookup` RMIRegistry must be started!

Remote Method Invocation

How does the client invoke the method of a remote object?

1. Obtain a reference to the remote object

```
try{
    Database db =
     (Database)Naming.lookup("//www.epfl.ch/DB_ONE");
```

2. Invoke a method as usual

```
    String reply = db.getRecord(123);
}catch(Exception e){e.printStackTrace();}
```


Generating Stubs

Run the `rmic` tool on a compiled **implementation** class (not on remote interface):

```
rmic DatabaseImpl
```

or for classes in a package:

```
rmic rmicourse.DatabaseImpl
```

Use `-v1.2` flag of `rmic` to generate less code! (not compatible with RMI v1.1)

Run RMI Registry

The RMI Registry is a name server that allows clients to get a reference to a remote object.

To start the RMI Registry from a shell:

```
rmiregistry or
rmiregistry 8000
```

To start the RMI Registry programmatically:

```
java.rmi.registry.LocateRegistry.createRegistry(8000);
```

The default RMI Registry port is #1099
Each time the remote interface is modified the RMI Registry must be restarted!

RMI Checklist

1. Remote interface (extends java.rmi.Remote)

2. Implementation object (implements 1, extends UnicastRemoteObject)

3. javac

4. rmic

5. start rmiregistry

6. rebind

7. lookup

For Advanced Users

The Java RMI is highly customizable:

- use `java.rmi.activation.Activatable` instead of `UnicastRemoteObject` if you want a remote object that can be activated (created) when a client requests it.
- use `RMISocketFactory` if you want a network protocol other than TCP/IP.
- install a `RMISecurityManager` if the client and the server do not share the same CLASSPATH and
- specify a URL from where remote stubs and other classes are loaded by the client and the server.

Serialization

Remember: non-remote arguments/return values of a remote method must implement `Serializable` interface.

```
public class Bidule implements java.io.Serializable{
    private String name = “EPFL”;//////// YES
    private transient Image hugeBitmap;// NO
    private static int nextBidule= 0;/// NO
}
```

For further information please read the serialization guide

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Networking in Java

Stanislav Chachkov

Department of Computer Science
Software Engineering Laboratory
April 2001

Useful Links

➥ Main Java Site - http://www.javasoft.com

➥ EPFL Java Site (docs and tutorial!) - http://sunwww:1995.epfl.ch/Java/

➥ Java RMI Guide — http://sunwww:1995.epfl.ch/Java/jdk1.3/docs/guide/rmi/

➥ Java Object Serialization Guide — http://sunwww.epfl.ch:1995/Java/jdk1.3/docs/guide/serialization/

➥ Java Tutorial on RMI — http://sunwww:1995.epfl.ch/for_EPFL/javatut_28Feb2001/

Sockets: `java.net` package

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

- Client:

```
try{
    Socket s = new Socket("www.epfl.ch", 8000);
    s.getOutputStream().write(request);
    reply = s.getInputStream().read();
    s.close();
}catch(IOException ioe){...}
```

- Server:

```
try{
    ServerSocket ss = new ServerSocket(8000);
    Socket s = ss.accept();
    request = s.getInputStream().read(); //do something ...
    s.getOutputStream().write(reply);
    s.close();
}catch(IOException ioe){...}
```


Another scenario for `ServerSocket`

The Server: maximum availability for multiple clients

```
try{
    ServerSocket ss = new ServerSocket(8000);
    while(true)
        new Handler(ss.accept()).start();
}catch(IOException ioe){...}
```

... and the Handler: one Handler thread for each client

```
class Handler extends Thread{
    Socket myConnection;
    Handler(Socket s){ myConnection = s; }

    public void run(){
        //..... serve the client, for example:
        request = s.getInputStream().read();//then do something ...
        s.getOutputStream().write(reply);
        s.close();
    }
}
```


The Need for a Protocol

What is the meaning of exchanged messages?

For example:
Server receives `request="G123"` and returns a `reply="EPFL"`.

That can mean:
`request`="give me record #123 from your database #G"
`reply` ="OK, the record you asked is "EPFL""

- Protocol gives a meaning to the messages.
- An implementation of a protocol "encodes" and "decodes" the messages.

Remote Method Invocation

Java "Protocol" + Network = RMI

- Java "Protocol": an object invokes a method of another object, waits for it to execute and receives the return value or an exception.
- RMI: a client object invokes a method (via a **remote reference**) of a server object. RMI encodes, sends, receives, decodes and executes the invocation on the server object. The result of the invocation is returned in the same way to the client.
- RMI calls are like normal calls: synchronous (i.e. they block the caller until the method returns from execution)
- RMI calls are different from normal calls: they can fail because of transmission problems.

The Remote Interface

Each Remote Object must implement a Remote Interface:

```
import java.rmi.*;

public interface Database extends java.rmi.Remote{

    public String getRecord(int index)
                        throws RemoteException;
    public void addRecord(int index, String record)
                        throws RemoteException;
    public int getSize()
                        throws RemoteException;
}
```


The Remote Interface must:

1. be declared **`public`**
2. extend **`java.rmi.Remote`** interface
3. have **`java.rmi.RemoteException`** in signature of each method (+ other exceptions)
4. use remote interfaces of remote objects in signatures of its methods, not implementation classes!

The Implementation Object

The Implementation Object lives on the Server.
This object can be accessed from network using RMI protocol.

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

public class DatabaseImpl extends UnicastRemoteObject
                        implements Database {

    public DatabaseImpl()throws RemoteException{ ... }

    public String getRecord(int index){ ... }
    public String addRecord(int index, String record)
    { ... }
    public int getSize() { ... }
}
```


The Implementation object must:

1. implement a remote interface
2. define the constructor
3. provide implementation of methods declared in remote interface (obvious)

Extending `UnicastRemoteObject` adds some communication facilities to the implementation class:

- Define that default socket-based transport of RMI will be used
- The object will run “all the time”

Notes on Constructor of Implementation Object

To start listening for incoming requests from clients, an implementation object must be exported :

`export` operation throws an `RemoteException`

- The default constructor of `UnicastRemoteObject` *exports* the object automatically

```
    public DatabaseImpl()throws RemoteException{
        super();
    }
```

- The other possibility (not a sub-class of `UnicastRemoteObject`):

```
    public class DBImpl_2 implements Database /*no extends*/{
    public DBImpl_2()throws RemoteException{
        UnicastRemoteObject.exportObject(this);
    }
```


Implementation of remote methods

```
public String getRecord(int index){
    String record = ...;
    return record;
}
```

- Arguments and return values of a remote method can be of any type
- Non-remote arguments/return values are serialized (i.e. a copy is sent)

NB: Non-remote arguments/return values must implement `Serializable` interface!

- For remote arguments/return values a remote reference is sent (i.e. no copy is made).

Creation of a Remote object on the Server

1. When a remote object is created it is automatically exported — ready to accept incoming requests:

```
DatabaseImpl db1 = new DatabaseImpl();
```

2. To be "visible" to the clients, remote object must be *registered* in RMI registry:

```
Naming.rebind( "DB_ONE", db1);
```

First argument of rebind is the name assigned to the remote object. Syntax:

```
full:                              "//host:port/name"
assuming default rmiregistry port: "//host/name"
assuming localhost and default port: "name"
```

For security purposes `host` can only be the local host.
Obviously, to execute a `rebind` or a `lookup` RMIRegistry must be started!

Remote Method Invocation

How does the client invoke the method of a remote object?

1. Obtain a reference to the remote object

```
try{
    Database db =
     (Database)Naming.lookup("//www.epfl.ch/DB_ONE");
```

2. Invoke a method as usual

```
    String reply = db.getRecord(123);
}catch(Exception e){e.printStackTrace();}
```


Generating Stubs

Run the `rmic` tool on a compiled **implementation** class (not on remote interface):

```
rmic DatabaseImpl
```

or for classes in a package:

```
rmic rmicourse.DatabaseImpl
```

Use `-v1.2` flag of `rmic` to generate less code! (not compatible with RMI v1.1)

Run RMI Registry

The RMI Registry is a name server that allows clients to get a reference to a remote object.

To start the RMI Registry from a shell:

```
rmiregistry or
rmiregistry 8000
```

To start the RMI Registry programmatically:

```
java.rmi.registry.LocateRegistry.createRegistry(8000);
```

The default RMI Registry port is #1099
Each time the remote interface is modified the RMI Registry must be restarted!

RMI Checklist

1. Remote interface (extends java.rmi.Remote)

2. Implementation object (implements 1, extends UnicastRemoteObject)

3. javac

4. rmic

5. start rmiregistry

6. rebind

7. lookup

For Advanced Users

The Java RMI is highly customizable:

- use `java.rmi.activation.Activatable` instead of `UnicastRemoteObject` if you want a remote object that can be activated (created) when a client requests it.
- use `RMISocketFactory` if you want a network protocol other than TCP/IP.
- install a `RMISecurityManager` if the client and the server do not share the same CLASSPATH and
- specify a URL from where remote stubs and other classes are loaded by the client and the server.

Serialization

Remember: non-remote arguments/return values of a remote method must implement `Serializable` interface.

```
public class Bidule implements java.io.Serializable{
    private String name = “EPFL”;//////// YES
    private transient Image hugeBitmap;// NO
    private static int nextBidule= 0;/// NO
}
```

For further information please read the serialization guide

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Networking in Java

Stanislav Chachkov

Department of Computer Science
Software Engineering Laboratory
April 2001

Useful Links

➥ Main Java Site - http://www.javasoft.com

➥ EPFL Java Site (docs and tutorial!) - http://sunwww:1995.epfl.ch/Java/

➥ Java RMI Guide — http://sunwww:1995.epfl.ch/Java/jdk1.3/docs/guide/rmi/

➥ Java Object Serialization Guide — http://sunwww.epfl.ch:1995/Java/jdk1.3/docs/guide/serialization/

➥ Java Tutorial on RMI — http://sunwww:1995.epfl.ch/for_EPFL/javatut_28Feb2001/

Sockets: `java.net` package

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

- Client:

```
try{
    Socket s = new Socket("www.epfl.ch", 8000);
    s.getOutputStream().write(request);
    reply = s.getInputStream().read();
    s.close();
}catch(IOException ioe){...}
```

- Server:

```
try{
    ServerSocket ss = new ServerSocket(8000);
    Socket s = ss.accept();
    request = s.getInputStream().read(); //do something ...
    s.getOutputStream().write(reply);
    s.close();
}catch(IOException ioe){...}
```


Another scenario for `ServerSocket`

The Server: maximum availability for multiple clients

```
try{
    ServerSocket ss = new ServerSocket(8000);
    while(true)
        new Handler(ss.accept()).start();
}catch(IOException ioe){...}
```

... and the Handler: one Handler thread for each client

```
class Handler extends Thread{
    Socket myConnection;
    Handler(Socket s){ myConnection = s; }

    public void run(){
        //..... serve the client, for example:
        request = s.getInputStream().read();//then do something ...
        s.getOutputStream().write(reply);
        s.close();
    }
}
```


The Need for a Protocol

What is the meaning of exchanged messages?

For example:
Server receives `request="G123"` and returns a `reply="EPFL"`.

That can mean:
`request`="give me record #123 from your database #G"
`reply` ="OK, the record you asked is "EPFL""

- Protocol gives a meaning to the messages.
- An implementation of a protocol "encodes" and "decodes" the messages.

Remote Method Invocation

Java "Protocol" + Network = RMI

- Java "Protocol": an object invokes a method of another object, waits for it to execute and receives the return value or an exception.
- RMI: a client object invokes a method (via a **remote reference**) of a server object. RMI encodes, sends, receives, decodes and executes the invocation on the server object. The result of the invocation is returned in the same way to the client.
- RMI calls are like normal calls: synchronous (i.e. they block the caller until the method returns from execution)
- RMI calls are different from normal calls: they can fail because of transmission problems.

The Remote Interface

Each Remote Object must implement a Remote Interface:

```
import java.rmi.*;

public interface Database extends java.rmi.Remote{

    public String getRecord(int index)
                        throws RemoteException;
    public void addRecord(int index, String record)
                        throws RemoteException;
    public int getSize()
                        throws RemoteException;
}
```


The Remote Interface must:

1. be declared **`public`**
2. extend **`java.rmi.Remote`** interface
3. have **`java.rmi.RemoteException`** in signature of each method (+ other exceptions)
4. use remote interfaces of remote objects in signatures of its methods, not implementation classes!

The Implementation Object

The Implementation Object lives on the Server.
This object can be accessed from network using RMI protocol.

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

public class DatabaseImpl extends UnicastRemoteObject
                        implements Database {

    public DatabaseImpl()throws RemoteException{ ... }

    public String getRecord(int index){ ... }
    public String addRecord(int index, String record)
    { ... }
    public int getSize() { ... }
}
```


The Implementation object must:

1. implement a remote interface
2. define the constructor
3. provide implementation of methods declared in remote interface (obvious)

Extending `UnicastRemoteObject` adds some communication facilities to the implementation class:

- Define that default socket-based transport of RMI will be used
- The object will run “all the time”

Notes on Constructor of Implementation Object

To start listening for incoming requests from clients, an implementation object must be exported :

`export` operation throws an `RemoteException`

- The default constructor of `UnicastRemoteObject` *exports* the object automatically

```
    public DatabaseImpl()throws RemoteException{
        super();
    }
```

- The other possibility (not a sub-class of `UnicastRemoteObject`):

```
    public class DBImpl_2 implements Database /*no extends*/{
    public DBImpl_2()throws RemoteException{
        UnicastRemoteObject.exportObject(this);
    }
```


Implementation of remote methods

```
public String getRecord(int index){
    String record = ...;
    return record;
}
```

- Arguments and return values of a remote method can be of any type
- Non-remote arguments/return values are serialized (i.e. a copy is sent)

NB: Non-remote arguments/return values must implement `Serializable` interface!

- For remote arguments/return values a remote reference is sent (i.e. no copy is made).

Creation of a Remote object on the Server

1. When a remote object is created it is automatically exported — ready to accept incoming requests:

```
DatabaseImpl db1 = new DatabaseImpl();
```

2. To be "visible" to the clients, remote object must be *registered* in RMI registry:

```
Naming.rebind( "DB_ONE", db1);
```

First argument of rebind is the name assigned to the remote object. Syntax:

```
full:                              "//host:port/name"
assuming default rmiregistry port: "//host/name"
assuming localhost and default port: "name"
```

For security purposes `host` can only be the local host.
Obviously, to execute a `rebind` or a `lookup` RMIRegistry must be started!

Remote Method Invocation

How does the client invoke the method of a remote object?

1. Obtain a reference to the remote object

```
try{
    Database db =
     (Database)Naming.lookup("//www.epfl.ch/DB_ONE");
```

2. Invoke a method as usual

```
    String reply = db.getRecord(123);
}catch(Exception e){e.printStackTrace();}
```


Generating Stubs

Run the `rmic` tool on a compiled **implementation** class (not on remote interface):

```
rmic DatabaseImpl
```

or for classes in a package:

```
rmic rmicourse.DatabaseImpl
```

Use `-v1.2` flag of `rmic` to generate less code! (not compatible with RMI v1.1)

Run RMI Registry

The RMI Registry is a name server that allows clients to get a reference to a remote object.

To start the RMI Registry from a shell:

```
rmiregistry or
rmiregistry 8000
```

To start the RMI Registry programmatically:

```
java.rmi.registry.LocateRegistry.createRegistry(8000);
```

The default RMI Registry port is #1099
Each time the remote interface is modified the RMI Registry must be restarted!

RMI Checklist

1. Remote interface (extends java.rmi.Remote)

2. Implementation object (implements 1, extends UnicastRemoteObject)

3. javac

4. rmic

5. start rmiregistry

6. rebind

7. lookup

For Advanced Users

The Java RMI is highly customizable:

- use `java.rmi.activation.Activatable` instead of `UnicastRemoteObject` if you want a remote object that can be activated (created) when a client requests it.
- use `RMISocketFactory` if you want a network protocol other than TCP/IP.
- install a `RMISecurityManager` if the client and the server do not share the same CLASSPATH and
- specify a URL from where remote stubs and other classes are loaded by the client and the server.

Serialization

Remember: non-remote arguments/return values of a remote method must implement `Serializable` interface.

```
public class Bidule implements java.io.Serializable{
    private String name = “EPFL”;//////// YES
    private transient Image hugeBitmap;// NO
    private static int nextBidule= 0;/// NO
}
```

For further information please read the serialization guide