Listener Concept and Java RMI

Helge Janicke, Niels-Peter de Witt, Karsten Wolke

November 20, 2002

Abstract

This paper shows possible implementations of the well known Observer Design Pattern (or Listener Concept) in a distributed RMI (Remote Method Invocation) based application. Overviewing the Listener Concept for non distributed programs, different distributed solutions are compared and explained in detail.

Contents

1 Introduction **3**

2 The Listener Concept **3**

2.1 Example for a value change . . . 3
2.2 Definition of the Listener interface . . . 4
2.3 Implementation of the Server class . . . 5
2.4 Implementation of the Listener interface . . . 6
2.4.1 Directly implementing the interface . . . 6
2.4.2 Implementing the interface as an inner class . . . 6
2.4.3 Other possibilities . . . 6

3 Overview about RMI Applications **7**

3.1 Design of a distributed application using RMI . . . 7
3.2 Example of an RMI application . . . 8
3.2.1 Design and implementation of the components . . . 8
3.2.2 Compile sources and generating stubs . . . 13
3.2.3 Distribute classes . . . 14
3.2.4 Start the example application . . . 15

4 Listener Concept in RMI **16**

4.1 Structure of the distributed Listener Concept . . . 16
4.2 Applying the Listener Concept . . . 17
4.3 Different attempts for a remote Listener implementation . . . 18
4.3.1 Similarities to section 2.1 - nothing remote . . . 19
4.3.2 Both Client and Listener are exported remote objects . . . 20

A Build a JAR file for the interfaces **23**

B Installation of a security Manager **24**

C Source code of the examples **28**

C.1 The Modifier . . . 28
C.2 The Server . . . 29
C.3 The Client . . . 32
C.3.1 Example 1 . . . 33
C.3.2 Example 2 . . . 35
C.3.3 Example 3 . . . 36
C.3.4 Example 4 . . . 38
C.3.5 Example 5 . . . 39
C.3.6 Example 6 . . . 41
C.3.7 Example 7 . . . 42
C.3.8 Example 8 . . . 44
C.3.9 Example 9 . . . 45
C.3.10 Example 10 . . . 46
C.3.11 Example 11 . . . 47
C.3.12 Example 12 . . . 48

1 Introduction

This paper describes how to implement the Listener Concept for an environment that uses RMI. The idea of the Listener Concept is that a part of a program reacts according to an event (e.g. data has changed or a button has been pressed). The part of the program, which is creating such an event, is notifying everyone who has signed interest in this event.
In a distributed environment the one who creates an event and the one who wants to be notified on this event are not necessarily on the same machine - this means the message has to be sent over the network. RMI can be used for a distributed environment. Its framework makes the usage of objects on different machines easy to handle. However the implementation of the Listener Concept in RMI is slightly different to a not distributed implementation.
What the Listener Concept is in general and what different kinds of implementations are possible is described in the section *The Listener Concept*. Why the normal approach of implementing a Listener does not work with RMI, and which implementation are possibile is described in the section *Listener and RMI*. The advantages and disadvantages of each variant are also explained and why some attempts do not work. The section *RMI* gives the reader a overview about the RMI-architecture, and describes how to use RMI with an example.
It is recommended to read through the sections *The Listener Concept* and *RMI* first, if you are not familiar with these topics, before starting with the section *Listener and RMI*.

2 The Listener Concept

The Listener Concept is one of the most popular design patterns [3]. It is used, whenever a part of a program wants to be informed of a particular event. The JAVA programing language makes extensive use of this pattern to implement user or software interaction within graphical user interfaces. This section describes the Listener Concept and gives some small examples of how an implementation can look like. Understanding the Listener Concept is mandatory for the discussion how to implement it in a distributed environment.

2.1 Example for a value change

In an object orientated language attributes and methods are combined within one object. Objects send messages to other objects to inform them to execute. The sender of a message always needs to address the recipient explicitly - this is done by a reference to it. The Listener Concept is a mechanism that handles the references for the objects that are interested in events - so called Listeners. Here is a small example:

Step 1:
All Listeners that are interested in the event, sign their interest in the `Server` object. The `Server` keeps an ordered list of all Listeners that have signed in.

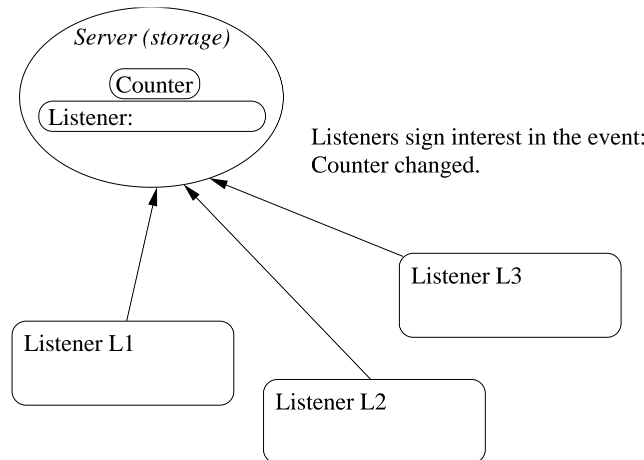

Figure 1: Listeners sign interest

Step 2:
If the value has changed, the `Server` object calls all Listeners stored in the list [1]. Every Listener is now reacting on the event.

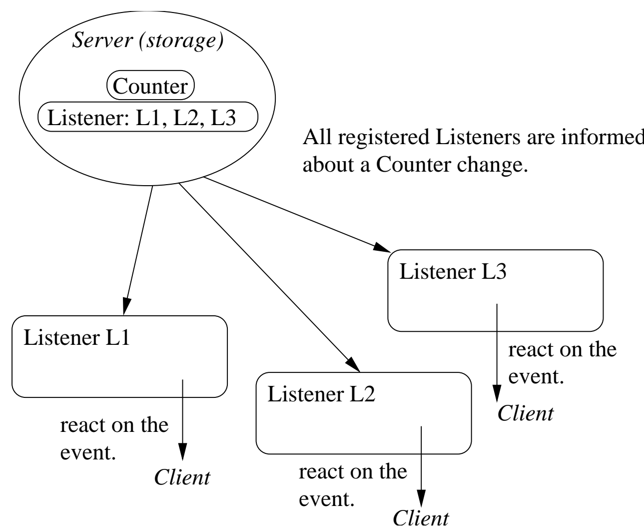

Figure 2: Listeners are invoked

The `Server` class (representing a storage) counts the amount of items that it stores. This counter can be increased or decreased. Whenever the counter changes all `ICounterListeners` that signed interest (Step 1) will be notified of the new value (Step 2). This is depiced in figures 1 and 2.

2.2 Definition of the Listener interface

Listeners are defined by a Listener interface, which specifies at least one method that is called whenever a particular event occurs. Usually the method is passed a so called event object, which describes the source and constraints of the event. The event object is in this example replaced by the new counter value.

[1] The use of ascending or descending order is discussed later

The Listener interface:

```
public interface ICounterListener {
    public void valueChanged(int newCounter);
}
```

2.3 Implementation of the Server class

The `Server` class provides methods to increase and decrease the counter. It also provides a method to register an `ICounterListener`.

```
import java.util.LinkedList;

/** The Server class represents a storage, which informs registered
 *  ICounterListeners of the changes in the amount.
 */
public class Server {

    private int counter = 0;
    private LinkedList listeners = new LinkedList();

    /** Increases counter, informs registered listeners. */
    public void increase() {
        counter++;
        fireCounterChanged();
    }

    /** Decreases counter, informs registered listeners. */
    public void decrease() {
        counter--;
        fireCounterChanged();
    }

    /** Adds an ICounterListener to this Server.
     *  @param listener The listener that signs interest.
     */
    public void addICounterListener(ICounterListener listener) {
        listeners.add(listener);
    }

    /** Calls all Listeners in order they have signed interest. */
    private void fireCounterChanged() {
        for (int i=0; i<listeners.size(); i++) {
            ICounterListener l= (ICounterListener) listeners.get(i);
            l.valueChanged(counter);
        }
    }
}
```

In this implementation the Listeners are invoked in the order they have sign interest (ascending). It is also possible to use the reverse order. This has the advantage when an event object is used. A Listener may consume an event (by setting a flag in the event object) which will prevent ”older” Listeners from being called. In this way it is possible to override behaviour of Listener invocations. Which order is appropriate for the concrete problem has to be decided from case to case. However it definitely has to be documented.

2.4 Implementation of the Listener interface

There are different ways how to implement a Listener interface. This paper concentrates on two, which are used when discussing solutions for the Listener Concept in a distributed environment. The object that is interested in the change is represented by the `Client` class. To denote that it was informed of the value change, it prints the new value on the console.

2.4.1 Directly implementing the interface

Directly implementing the interface is a convenient way, if the `Client` class implements only one (or not too many) Listener interfaces. The advantage is that all other (even private) methods and attributes are accessible. Disadvantages are that there is only one instance of the listener and though it is only possible to distinguish between event sources by explicitly comparing them in the valueChanged() method - if an event object is provided.

```
public class Client implements ICounterListener {
    public Client(Server server) {
        server.addICounterListener(this);
    }
    public void valueChanged(int newValue) {
        System.out.println("Client: New value is " + newValue);
    }
}
```

2.4.2 Implementing the interface as an inner class

Implementing the interface as an inner class seems to be a more difficult way to achieve the same goal - compared with 2.4.1 because the advantages and disadvantages remain the same in a not distributed application. Section 4.3.2 will explain why and in what cases the inner class implementation is preferable.

```
public class Client {
    public Client(Server server) {
        server.addICounterListener(new Listener());
    }
    private void print(int value) {
        System.out.println("Client: New value is " + value);
    }
    public class Listener implements ICounterListener {
        public void valueChanged(int newValue) {
            print(newValue);
        }
    }
}
```

2.4.3 Other possibilities

Another convenient way to implement listeners is via an anonymous class definition. In this case the interface will be implemented in-line:

```
public class Client {
    public Client(Server server) {
        server.addICounterListener(new ICounterListener() {
            public void valueChanged(int newValue) {
                print(newValue);
            }
        });
    }
    private void print(int value) {
```

```
        System.out.println("Client: New value is " + value);
    }
    public class Listener implements ICounterListener {
        public void valueChanged(int newValue) {
            print(newValue);
        }
    }
}
```

This is preferable, if the Listener implementation is only used once. There are no difficulties to decide where the event was fired - because it is coded directly for each `Server`. Like in the other examples it has access to all members of the enclosing class.
Furthermore it is also possible to implement an own class for each Listener implementation.

3 Overview about RMI Applications

Java RMI (Remote Method Invocation) is part of the Java 2 Platform. This framework enables communication between Java objects on different virtual and/or physical machines. RMI applications are usually developed in 2 different parts: a client and a server. A typical server application creates objects (so called remote objects) and gives the client the possibility to invoke these objects. There are two typical mechanisms to get a remote reference.

Binding The server registers a remote object in the RMI-registry which is called naming or binding the object. The client can lookup this object in the same RMI-registry.

Reference A Method can pass or return a remote object reference.

The details of the communication are handled by RMI. The programmer can use remote references like local ones. The naming and lookup are methods of the `java.rmi.Naming` class. RMI provides an easy way to implement a distributed application.
Figure 3 shows, the communication structure. The server calls the RMI-registry and binds an object's reference to a name. It is now available for other virtual machines to look up. The client looks up a remote object reference by its name in the RMI-registry. The lookup returns the reference and the client is now able to invoke methods via this reference.

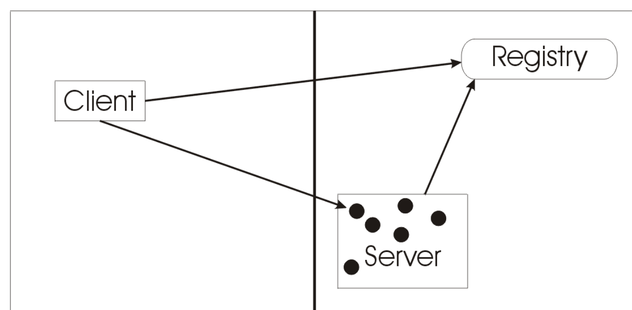

Figure 3: RMI communication structure

RMI also provides a mechanism for loading class byte code, as described in Appendix B.

3.1 Design of a distributed application using RMI

To design a distributed application using RMI it has to be decided which components are remotely accessible and which only locally. For each class, whose instances are remotely accessible a remote interface is to define. This interface declares the remote accessible methods. Remote interfaces have two typical characteristics:

- A remote interface must extend the interface java.rmi.Remote. Any object that implements this interface becomes a remote object.
- Each method of the interface declares java.rmi.RemoteException in its throws clause. In case of a failure (e.g. connection failure) the object throws this exception. To read more detailed infomation about the exception please read the Java API [8].

All remote classes must implement at least one remote interface. These classes may implement other methods which are only available locally. After implementation and compiling stubs and skeletons will be generated. RMI uses a remote Objects stub class as a proxy to simulate a real reference on clients. The Skeleton is on the server side. Since jdk 1.2 the skeleton is not any more used because an additional stub protocol was introduced. Classes and interface are compiled with the command `javac` and the stubs and skeletons with the command `rmic`. After compiling the necessary classes are deployed to the client and server machine. To execute the remote appliation first the RMI-registry is started then the server and client application. Section 3.2 gives an example of an RMI application.

3.2 Example of an RMI application

This illustrates in detail how an RMI application can be written. The example is divided into 4 major parts:

- Design and implementation of components for a distributed application.
- Compile sources and generating stubs.
- Distribute classes.
- Start the application.

3.2.1 Design and implementation of the components

The design and implementation of components for a distributed application is the biggest part of the work and can be divided in:

- Writing an RMI server
  - Designing remote interfaces
  - Implementing remote interfaces
- Creating a client program

This section will implement the following example, shown in UML (Unified Modelling Language) below.

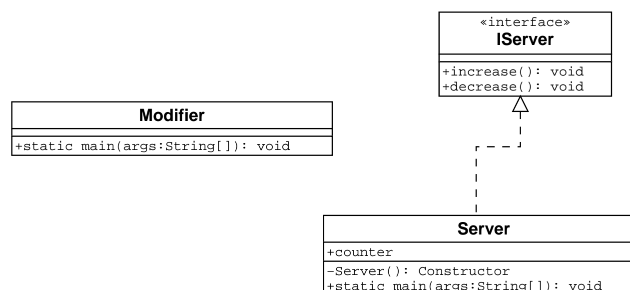

Figure 4: little example

The `Modifier` (client in this case) and the `Server` application can be on different machines. The `Server` will be created as a remote object and the `Modifier` has access to this remote

object. After implementing the RMI environment, the `Modifier` has access to all methods declared in the `IServer` remote interface. In this case it means the `Modifier` can invoke the `increase()` and `decrease()` method. When the application is running the `Modifier` can invoke these methods on the `Server` and therefore change data on the `Server`. Every time the `Modifier` changes data on the `Server`, the `Server` will print a message.

Designing a remote interface
The example `IServer` interface defines the remote accessible part of the `Server`. Following is the code of this interface.

```
package example;

public interface IServer extends java.rmi.Remote {

  public void increase() throws java.rmi.RemoteException;

  public void decrease() throws java.rmi.RemoteException;

}
```

By extending the interface `java.rmi.Remote`, this interface becomes itself a remote interface. Any object that implements this interface becomes a remote object. Every method in this interface is a remote method. Therefore these methods must be defined as being capable of throwing a `java.rmi.RemoteException`. This exception will be thrown by the RMI system during a method call, to indicate communication failures or to wrap any other Exception. Any code that calls remote methods has to handle this exception by either catching it or declaring it in its throws clause. If methods with parameters are used these parameters need to be exportet remote Objects or they need to implement java.io.Serializable. RMI uses the object serialisation mechanism to transport objects by value between Java virtual machines (JVM). This means every time a non or non exported remote Object is used as a parameter for a remote method, a copy of this parameter will be send. Implementing `Serializable` allows the class to convert into a self-describing byte stream that can be used to reconstruct an exact copy of the serialized object. Further information about serialization can be found in the Java API [9].

Implementing a remote interface
An object that implements a remote interface becomes a remote object. The implementation should at least:

- Implement a remote interfaces.
- Define the constructor for the remote object.
- Provide an implementation for each method, declared in the remote interfaces

The `Server` has to create and register/bind the remote object(s). This can be done in the main method of the object itself, which should:

- Create and install a security manager if necessary [2].
- Create one or more instances of a remote object.
- Register at least one of these remote objects in the RMI-registry.

[2]Security manager are discussed in B

The complete implementation of the `Server` class follows. It implements the remote interface `IServer` and also includes a main method for binding the remote object.

```
package example;

import java.rmi.*;

public class Server extends java.rmi.server.UnicastRemoteObject
                    implements IServer {

  int counter;

  private Server() throws RemoteException {
  }

  public void increase() throws RemoteException {
    counter++;
    System.out.println("Increasing value to "+counter);
  }

  public void decrease() throws RemoteException {
    counter--;
    System.out.println("Decreasing value to "+counter);
  }


  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        Naming.rebind(args[0], new Server());
        System.out.println("Server started");
      } else {
        System.out.println("expected argument: location and name for binding Server");
        System.out.println("example: rmi://localhost:1099/Server");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

The implementation of the `Server` class is declared as

```
public class Server extends java.rmi.server.UnicastRemoteObject
implements IServer
```

This declaration means that the class implements the remote interface `IServer` and therefore defines a remote object. It also extends the `java.rmi.UnicastRemoteObject`. `UnicastRemoteObject` is a class, that can be used as a superclass for remote object implementation. By default a class extends `java.lang.Object` and therefore inherits some methods. The superclass `UnicastRemoteObject` supplies implemention for many `java.lang.Object` methods(`equals()`, `hashCode()`, `toString()` ...) so that they are decently for a remote object. `UnicastRemoteObject` also includes some constructors and static methods to handle communication. A remote object does not have to extend from UnicastRemoteObject, but any implementation that does not extend this superclass must supply the appropriate implementation of the `java.lang.Object` methods and export the remote object itself. Further

details can be found in the Java API [10].

The `Server` class implements only one remote interface. The implementation includes methods which are not declared in the remote interface. This methods can only be accessed locally. The first of these methods is the constructor and the second one the main method, that creates a `Server` object and binds itself to the RMI-registry. Every implementation has to declare a constructor which throws a `RemoteException`. Because of this `RemoteException`, it is necessary to overwrite the standard constructor. The export step will be done by RMI which means that after this step is complete the `Server` remote object is ready to accept incoming calls from clients. RMI uses an anonymous port given by the operating system to realise this task. Before a caller can invoke methods from a remote object the caller must have the reference to this object. The caller can get this reference as a return value from an other method or by lookup via the `java.rmi.Naming` class which uses the RMI-registry. The RMI-registry is a database, which bind a specific name to a remote object. Once a remote object is registered with the RMI-registry on the local host, callers on any other host or virtual machines can look up the remote object reference by name. The callers get the reference back and can direct invoke methods on this objects. For the programmer it looks like a local object reference. The communication to the RMI-registry will be done by the `java.rmi.Naming` class. There are methods to bind, rebind, lookup and a few more interesting methods. More detailed information can be found in the Java API [7]. To register an object, the static method `Naming.bind(name : String, obj : Remote)` will be used. The first parameter is an URL-formatted `java.lang.String` representing the location and the name of the remote object. The format of this String is `"//host:port/objectname"`. It is not necessary to specify all subparameter. By default the host is localhost and the port is 1099. The host name is the IP address or the hostname in the network. RMI uses the standard port 1099. The objectname has to be an unique name in the registration because this is the key for the object reference.
In the used example the URL-String will be given as an argument from the command line. For the used example use `"//localhost:1099/Server"` as argument; the host as `localhost`, the default port `1099` and `Server` as the name for the `Server` remote object. If the `Server` is started it prints out a message that it is ready. If the `Server` is startet without an argument, it prints a help and in case of any failure the stack trace.
Note that for security, an application can only bind, rebind and unbind remote objects references on the RMI-registry on its host.

Passing Objects in RMI
Arguments and return values from remote methods can be of almost every type, including local objects, remote objects and primitive Types. More specific: Any entity that can be passed with a wrapper class like primitive types, or an entity that is an exported remote object, or a serializable objects, which means that it implements the java.io.Serializable interface. The rules how parameters and return values are passed are as follows:

- Remote objects are passed by reference. A remote object reference is a stub, which is a client side proxy. It implements the complete set of remote interfaces that the remote object implements. Passing an object by reference means that any changes made from the remote method are reflected in the original remote object. When passing a remote object, only the methods that are declared in the remote interfaces are available from the remote method.

- Local objects are passed by copy, using object serialisation. By default all fields are copied, except those that are marked static or transient.

Creating a client program The code for the client is very easy. The following code shows the `Modifier` class, which is a little bit bigger than necessary to show how RMI works, but this example will be used later in this paper again.

```
package example;
```

```
import java.rmi.*;
import java.io.*;

public class Modifier {

  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        IServer server = (IServer) Naming.lookup(args[0]);
        System.out.println("Modifier started");
        System.out.println("+ to increase\n- to decrease\nexit to exit the program\n");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String input;
        while ((input = in.readLine()) != null && !input.equalsIgnoreCase("exit")) {
                 if (input.equals("+")) {
            server.increase();
          } else if (input.equals("-")) {
            server.decrease();
          } else {
            System.out.println("+ to increase\n- to decrease\nexit to exit the program\n")
          }
        }


      } else {
        System.out.println("expected argument: location and name for looking up the Server
        System.out.println("example: rmi://localhost:1099/Server");
      }

    } catch (Exception exc) {
      // don't care about what Exception it is
      exc.printStackTrace();
    }
  }


}
```

This application give a user the possibility to change the value for a remote object interactively. First the client looks up the remote object via `Naming.lookup(name : String)`. The name parameter has the same syntax as the parameter in the bind method in the `Server` application. For example : `"//localhost:1099/Server"`. The name has to be the same name as the `Server` used to bind the object. The lookup returns the reference of the remote object. If this lookup succeeds, a message will be printed that the `Modifier` is running. Note that the `Server` application has to run before starting the client application. In the other case the `Client` will not find the remote `Server` object and a stack trace will be printed with the error message. After a correct start the client is able to invoke the remote methods `increase()` and `decrease()`. The user can press `+` followed by the Enter key to invoke the `increase()` method and the `-` to invoke the `decrease()` method. The next diagram shows the communication between `Modifier` (client) and `Server`.

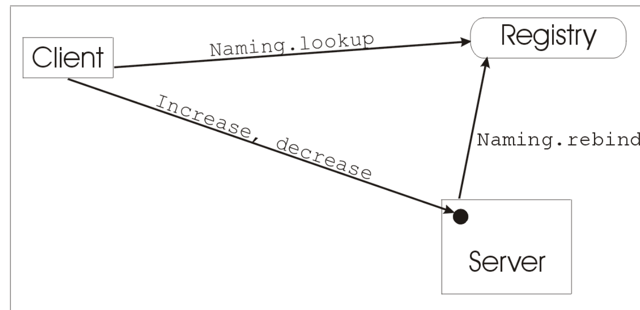

Figure 5: ClientServerCommunication

3.2.2 Compile sources and generating stubs

In a real world scenario in which services are deployed, a developer would create a JAR (Java Archive) file that contains the necessarily interfaces for the server classes. Next, a developer, may be the same, will write the implementation of one of this services (declared by their interfaces). Developer of client programs use the interfaces and independently develop their specific client application. The next steps shows how to compile java classes and interfaces and to create stubs and skeletons.
To compile java code the `javac` compiler given by the java Software Development Kit (SDK) is used. It is recommend to use a SDK version 1.2 or later. The following steps show the correct syntax for MS Windows/Win32 and UNIX/LINUX operating systems. All classes and interfaces are implemented as shown in this paper and saved in the `c:\user\example` directory for Win32 systems and in `/home/user/example` on an UNIX system. This description does not use JAR files. Appendix A shows an example can be find to create a JAR File for the `IServer` interface. More information about JAR files can be found at [1].

The example has only one interface (`IServer`) that describe a remote object or in other words a service.

To compile it under Win32 systems:

```
cd c:\user
javac example\IServer.java
```

To compile it under UNIX systems:

```
cd /home/user
javac example/IServer.java
```

The compilation created an `IServer.class` file, which can be distributed to all developers of server and client application so that they can make use of this interface.

Build the server classes
The developer of the server places the `IServer.class` interface in the following directory. `c:\user\server\example` for Win32 Systems and `/home/user/server/example` for UNIX Systems. The developer of the server also places all his implemented classes in this directory. In case of this example only `Server.java`

Compile server classes and generating stubs under Win32

```
cd c:\user\server\
javac example\Server.java
rmic example.Server
```

Compile server classes and generating stubs under UNIX

```
cd /home/user/server/
javac example/Server.java
rmic example.Server
```

Now there are 3 files more in the directory. The `Server.class` file generated by `javac` and `Server_Skel.class`, `Server_Stub.class` generated by `rmic`. The files which `rmic` generated will be used by the client application. This classes should be made accessible, that other developer can download them.

Build the client classes
The developer of a client has the `IServer` interface from the `Server` and all stubs from the implementing `Server`. In the example all this files with the client application `Modifier.class` are in the following directory.
`c:\user\client\example` for Win32 Systems
`/home/user/client/example` for UNIX Systems.

Compile client classes under Win32

```
cd c:\user\client\
javac example\Modifier.java
```

Compile client classes under UNIX

```
cd /home/user/client/
javac example/Modifier.java
```

Now all necessary class files are generated and can be executed.

3.2.3 Distribute classes

Before executing the application, the necessary files for the client and server have to move to corresponding host machines. This means the client needs the stub classes for all used remte objects. It is even possible to download the stub classes during runtime as mentioned in section 3. To do this it is necessary to install a security manager as mentioned in section 3.2.4 and in the appendix B. The shown implementation of the example doesn't have a security manager. This means the client and server has its neccesary classes. In the example the client needs the `Modifier` class, the `Server_Stub` class and the `IServer`. The server needs the `Server` class, the `Server_Stub` class and the `IServer_Stub`.

We assume that all files are on one computer in the directories shown in section 3.2.2. This means: The RMI application runs only on one machine, but it does not make a difference in the behaviour.

for Win32 Systems:

`c:\user\server\example` for server files
`c:\user\client\example` for client files

for UNIX Systems:

`/home/user/server/example` for server files
`/home/user/client/example` for client files

3.2.4 Start the example application

As mentioned an RMI application has often a security manager installed. The security manager is necessary to load class files from a different virtual machine at runtime. Often this are the stub files. For security reasons the permissions should be restricted which results in the necessity to install a security manager. Appendix B describes how to install a security manager for this little example. However, when the class files ae already distributed to the different machines, it is not necessary to install a security manager.

Start the server
The `Server` application needs an RMI-registry to register itself. This means that an RMI-registry has to be started on the server. The next commands show how the `Server` and the RMI-registry can be started on Win32 and UNIX Systems.

Start RMI-registry and the Server under Win32
If the `start` command is not existent, there is another command `javaw`. However if both commands fail, it is possible to start the RMI-registry without these help only that a new console is required.

```
cd c:\user\server\
start rmiregistry
java example.Server rmi://localhost:1099/Server
```

Start RMI-registry and the Server under UNIX/LINUX

```
cd /home/user/server/
rmiregistry &
java example.Server rmi://localhost:1099/Server
```

The first part of the argument given to the `Server` is optional. It is possible to write the parameter with or without `rmi:` This means the following command does the same:

```
java example.Server //localhost:1099/Server
```

Start the client
The client is a application that depends on a running `Server`. If the `Server` object could not be found an exception stack trace will be printed.
start client under Win32

```
cd c:\user\client\
java example.Modifier rmi://localhost:1099/Server
```

start client under UNIX/LINUX

```
cd /home/user/client/
java example.Modifier rmi://localhost:1099/Server
```

Test the RMI application
Every time the `Modifier` calls an increase/decrease command on the `Server`, the `Server` prints the new value on its own console. After 4 increase commands the consoles should look like the following.

```
java example.Modifier rmi://localhost:1099/Server

Modifier started
+ to increase
- to decrease
exit to exit the program

+
+
+
+
```

Figure 6: littleExampleClientOutput

```
java example.Server //localhost:1099/Server

Server started
Increasing value to 1
Increasing value to 2
Increasing value to 3
Increasing value to 4
```

Figure 7: littleExampleServerOutput

Further Reading
More information can be found in JGurus short course [2] and the RMI specification [4].

4 Listener Concept in RMI

Trying to use the well known Listener Concept in a distributed (RMI) application caused problems. This section describes what we discovered and discusses possible solutions. In the first subsection the structure of the distributed application is explained. The second subsection elucidates why the Listener Concept described in section 2 cannot be used. The further sections show different solutions, their advantages and errors in reasoning.

4.1 Structure of the distributed Listener Concept

The distributed structure is similar to the examples used in the previous sections. Additional to the RMI example 3.2 the `Server` class now has a method to register `ICounterListener` objects. The implementation of the `Server` methods to notify Listeners is taken from example 2.1. Figure 8 shows the new structure, where figure 9 shows the UML diagram of the used `Server` class:

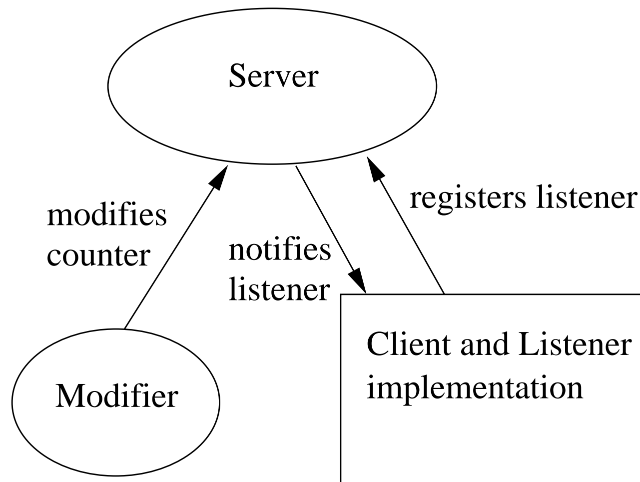

Figure 8: structureRMIListener

The client and Listener implementation is in this diagram a black box. The following sections discuss different implementations of this black box.

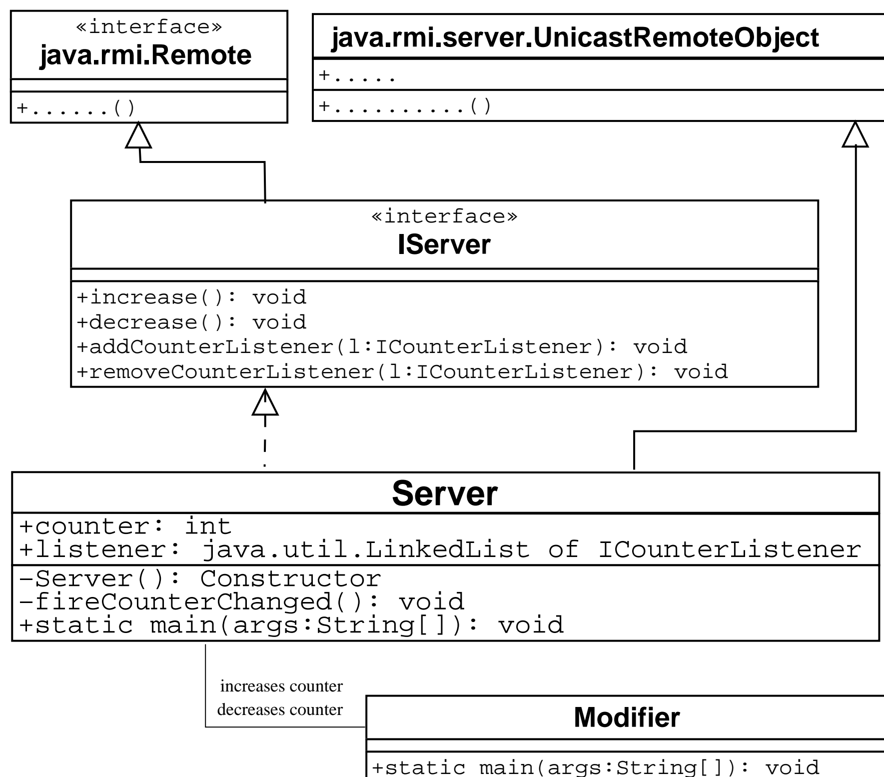

Figure 9: ListenerConceptServerModifier

4.2 Applying the Listener Concept

As mentioned in 3.2.1 parameters to remote method invocations which are not exported remote objects are passed as a copy and not by reference. Therefore the Listener Concept, which was introduced in section 2 cannot be applied directly. If the Listener implementation is not remote, a copy of it is made when calling the `Servers` `addICounterListener()` method. This copy now resides on the server. The weak point is, that when the `Server`

informs the Listener of the new counter value, only the copy is informed. This results in the `Client` prints the message: *"Client: New value is 1"* on the servers console.
This is obviously not the intention of the Listener Concept. The goal is to inform the original `Client` object of the value change. How this can be accomplished is shown in the next section.

4.3 Different attempts for a remote Listener implementation

As a result of the previous section it is necessary to have either (or both) the `Client` implementation or the `ICounterListener` implementation being exported remote objects. This leads to a variety of combinations. Table 1 lists different possibilities.

The Listener as an inner class of the Client

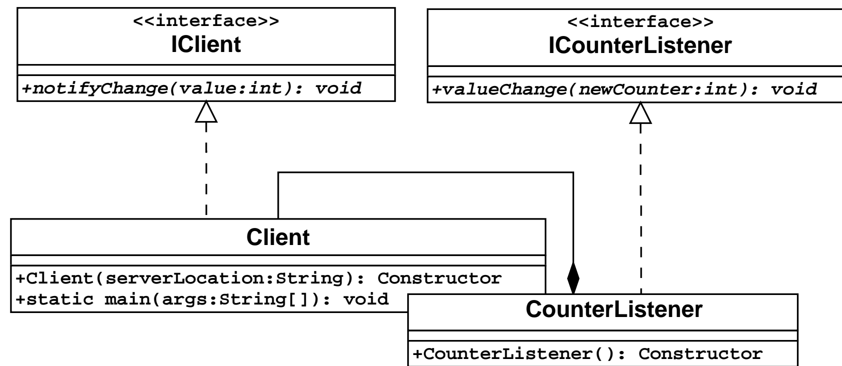

Figure 10: ListenerConceptClientInnerListener

1. non remote Listener with remote enclosing Client class
2. non remote Listener with non remote enclosing Client class
3. remote Listener with remote enclosing Client class
4. remote Listener with non remote enclosing Client class

A standalone Listener with a reference to a Client

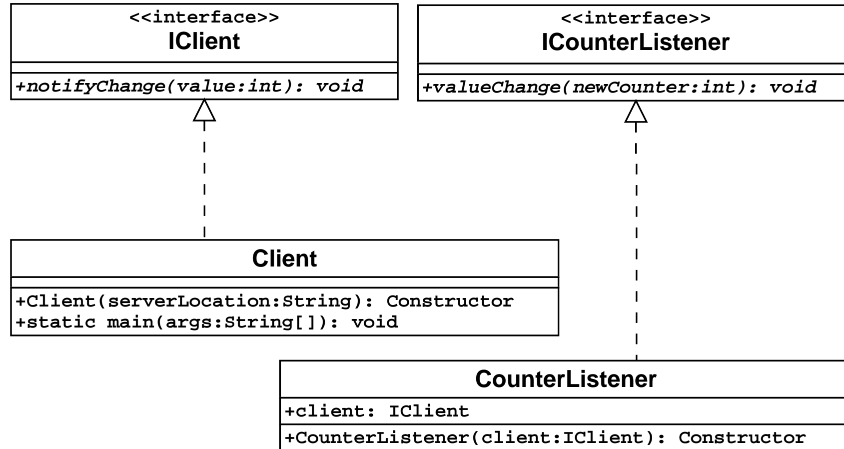

Figure 11: ListenerConceptClientStandaloneListener

5.	non	remote Listener with reference to a		remote Client class
6.	non	remote Listener with reference to a	non	remote Client class
7.		remote Listener with reference to a		remote Client class
8.		remote Listener with reference to a	non	remote Client class

The Client implementing the Listener interface directly

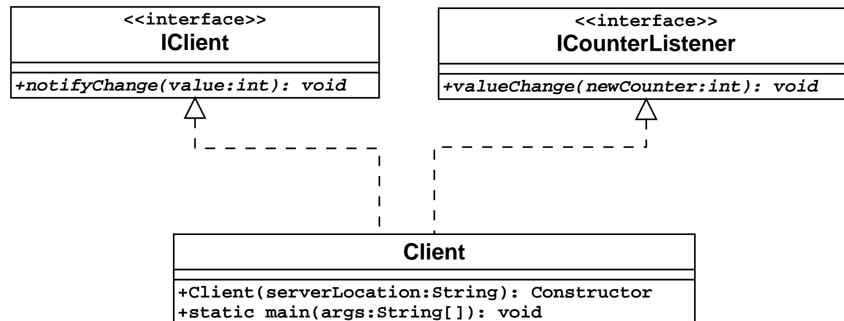

Figure 12: ListenerConceptClientImplementsListenerInterface

9.	non	remote Client class implementing a	non	remote Listener interface
10.	non	remote Client class implementing a		remote Listener interface
11.		remote Client class implementing a	non	remote Listener interface
12.		remote Client class implementing a		remote Listener interface

Table 1: possible solutions

Appendix C shows the implementation for each of these examples.

4.3.1 Similarities to section 2.1 - nothing remote

According to the conclusion of the previous section, solutions 2, 6 and 9 can be sorted out, because none of these implementations are remote. Note that these solutions are correct for non remote usage.
Running solution 10 shows the same behaviour as in 2, 6 and 9. But here the explanation is

a bit trickier. Implementing the remote interface ICounterListener makes the Client a remote object. This would lead to the assumption that it is passed by reference. This is not the case, because it is never exported to the RMI system. Usually remote objects export themselves during construction time by extending java.rmi.server.UnicastRemoteObject. If the export is made explicitly this configuration should behave in the correct way. But making the Client an exported remote object will change the solution 10 into solution 12 - which, as described later, functions perfectly well.
The next section describes the opposite attempt. Instead of having neither the Client nor the Listener remote we now have a closer look what happens if both are exported remote objects.

4.3.2 Both Client and Listener are exported remote objects

All this three implementations (3, 7 and 12) have satisfying results. Whenever the Modifier changes the counter on the Server, a message is sent to the Client. Seeing that each of the solutions behave correctly the question, which is the best one, is at hand.

Solution 3: remote Listener with remote enclosing Client *class*
The inner CounterListener implementation can access all (even private and non remote) members of its enclosing class. This is an advantage, because it can modify attributes which are not directly accessible over the network or from other classes. An other advantage is that when the CounterListener is added to the Server not the (probably) large stub of the Client is transferred, but only the small CounterListener stub. This saves network resources. The disadvantage of this solution is the tricky generation of Listener stubs. Running rmic Client will only produce a stub for the Client class. To generate the CounterListener stub the rmic Client$CounterListener command has to be called because the rmic compiler uses the reflection mechanism to identify remote interfaces. To accomplish this task, it loads the class definition and analyses it. The class definition of an inner class is, by convention, stored in a file with the name *"EnclosingClass$InnerClass"*.

As mentioned above, only the CounterListener stub is transferred to the remote Server. Noticing this fact, their is no difference between solution 3 and 4. It is not relevant for the Listener Concept if the Client is remote or not, because the original Listener object always stays on the Client objects virtual machine.

Solution 7: remote CounterListener *with a reference to a remote* Client
This combination consists of two different remote objects. The CounterListener itself is an exported remote object, as is the Client. It is necessary for the Listener to keep a reference to the Client object, so it is able to invoke it's methods. Even if this scenario is working we do not consider it best. To notify the Client of the new counter value two remote calls are made:

1. Server invokes Listener
2. Listener invokes Client

Furthermore the CounterListener is only able to call methods of the Client which are exported (defined in the Clients remote interface). This is a hard restriction. If the Client is itself a remote object, the Server could have invoked it directly by calling one of its remote methods. Seeing this, there is no need to place a remote CounterListener class in between.

On the other hand if the Client is an exported remote object and the CounterListener not. A copy of the Listener is made on the server side. But even this copy holds a valid reference to the remote Client. This is solution 5, which is preferable to solution 7, because it spares one remote call - but still lacks the ability to call non remote methods. A possible appliance of solution 5 is when the Client is located on a very slow machine (e.g. a ThinClient) and the server is very fast. If the data needs to be prepared for the client, and this preparation takes a lot of resources it might be usefull to have it executed on the

server, before passing the result to the client.

Last but not least, if the `Client` is not remote, but the `CounterListener` - which is solution 8. In this case it is mandatory that both `Client` object and the `CounterListener` object reside in the same virtual machine. This is a good solution, because in case the `Client` was designed to be a non remote object, it is easy to add a remote Listener into the existing structure. The only disadvantage is, as in all standalone Listener solutions, the `CounterListener` can only access public or at most package private methods of the `Client`.

Solution 12: remote `Client` *class implementing a remote* `ICounterListener` *interface*
In this case the `Client` implements two remote interfaces. This is a convinient way of implementing a Listener, and follows the same rules as the Listener implementation shown in section 2 - but remote. The advantage is that again all members of the `Client` are accessible from the Listener (it is the `Client` itself). The disadvantage is that a huge stub is transferred to the server, which uses only the exported methods of the `ICounterListener` interface. This is unnecessary network traffic. Thinking of the remote calls might bring up the idea that for this scenario are again two remote calls made to invoke the `print()` method. This is not the case. The exported Listener methods transfer the message to the original `Client` object using the client stub on the server side. When the Listeners `valueChanged()` method is now invoked on the original `Client` object it does not do a second remote call to call `print()`, because it is working locally on the object - not accessing the methods via a stub. So only one remote call is made.
When solution 10 is modified to export itself as mentioned in 4.3.1 and the `Client` methods are only available locally, the disadvantage that the huge stub is transferred vanishes. In this case the stub only contains the exported Listener methods and their remote invocation will call the local methods of the `Client`. If the `Client` does not need to be remote except of the Listener part, this is the implementation we recommend.

This leaves solution 1 and 11 for further examination
Both of these examples are causing exceptions. For solution 11 an `IllegalArgumentException` is thrown when starting the client. When examining the stack trace as Figure shown in 13 the first method which is located in the example classes is the `Server` stub `addICounterListener()` method.

```
java example11.Client rmi://localhost:1099/Server rmi://localhost:1099/Client

java.lang.IllegalArgumentException: argument type mismatch
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:324)
  at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:261)
  at sun.rmi.transport.Transport$1.run(Transport.java:148)
  at java.security.AccessController.doPrivileged(Native Method)
  at sun.rmi.transport.Transport.serviceCall(Transport.java:144)
  at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:460)
  at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:701)
  at java.lang.Thread.run(Thread.java:536)
  at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(StreamRemoteCall.java:247)
  at sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:223)
  at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:133)
  at example11.Server_Stub.addCounterListener(Unknown Source)
  at example11.Client.<init>(Client.java:22)
  at example11.Client.main(Client.java:42)
```

Figure 13: stacktrace11

The argument passed to this remote method is the `Client` stub. When creating the stub,

only methods which are defined in remote interfaces are generated by the `rmic` compiler. The `ICounterListener` implementation is not remote, therefore the stub does not contain methods to call it - it is not an instance of `ICounterListener`. Because the RMI structure passes the arguments itself it does not throw a ClassCastException as expected, but an IllegalArgumentException.
The explanation of solution 1 is trickier than solution 11. Here the Listener is a non remote inner class implementation of a remote enclosing `Client` class. Again the start of the client fails when adding the Listener via the `Server` stub.

```
java example1.Client rmi://localhost:1099/Server rmi://localhost:1099/Client

java.lang.ClassCastException
  at java.io.ObjectStreamClass$FieldReflector.setObjFieldValues(ObjectStreamClass.java:1813)
  at java.io.ObjectStreamClass.setObjFieldValues(ObjectStreamClass.java:1047)
  at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:1836)
  at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1756)
  at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1636)
  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1264)
  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:322)
  at sun.rmi.server.UnicastRef.unmarshalValue(UnicastRef.java:297)
  at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:246)
  at sun.rmi.transport.Transport$1.run(Transport.java:148)
  at java.security.AccessController.doPrivileged(Native Method)
  at sun.rmi.transport.Transport.serviceCall(Transport.java:144)
  at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:460)
  at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:701)
  at java.lang.Thread.run(Thread.java:536)
  at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(StreamRemoteCall.java:247)
  at sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:223)
  at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:133)
  at example1.Server_Stub.addCounterListener(Unknown Source)
  at example1.Client.<init>(Client.java:12)
  at example1.Client.main(Client.java:24)
```

Figure 14: stacktrace1

The reason for this `ClassCastException` is not obvious. The inner class is serialized and copied to the server. If the inner class is serialized, the `Client` has to be copied or referenced as well, the inner class can access all members of its enclosing class. Copying the enclosing class is not possible, even if it is serializable. It has been exported to the rmi system before and therefore has to reside on the JVM it has been exported to. The Listener should call the `Client` via a reference which is realised via stub calls. The Listener has no reference to a `Client` stub (not even an implicit one) it is not possible to call the original `Client`. When the Listener is deserialized on the server side, the field descriptors in the object stream does not match the contents. That is why the `ClassCastException` is thrown. Unfortunately the stack trace does not give any information about the expected and found class type. We investigated this further, by modifying the `ObjectStreamClass.FieldReflector`-L1813 to provide this information in the message attribute of the `ClassCastException`. The reason for this `ClassCastException` is deep in the serialization mechanism. It wants to cast a `Client_Stub` to a `Client`. As mentioned, a stub has only knowledge about its implementing remote interfaces. Consequential it can not cast the `Client_Stub` to a `Client`.

A Build a JAR file for the interfaces

The example used in section 3.2 has only one interface `IServer.class` that describes a remote object or in other words a service. To compile the file and to build a JAR file use the following commands:
Under Win32 systems:

```
cd c:\user
javac example\IServer.java
jar cvj example.jar example\*.class
```

Under UNIX systems:

```
cd /home/user
javac example/IServer.java
jar cvj example.jar example/*.class
```

The `jar` command displays the following output (dependent to the `-v` option):

```
added manifest
adding: example/Iserver.class (in=233)(out=167)(deflated 28 %)
```

This JAR file can be distributed to all developers of server and client application so that they can use this interface. To use the interfaces, they have to import the package or write the full name every time they use it. When compiling or executing the program that uses this interface the `CLASSPATH` variable has to be extended by the JAR file. More information about JAR files can be found at [1].

B Installation of a security Manager

RMI provides a mechanism to download class byte code. This means during runtime stub files or other class files can be downloaded. For safety, a security manager has to be installed. This manager is required wherever downloaded code will be executed. To install a security Manager 2 lines of code have to be added and a policy file with the permissions is required. The additional code has to be placed direct before any remote object handling.

A security manager needs a policy file. By convention this file has to be called `java.policy` and will be given as a parameter when starting the application. An example for a policy file:

```
grant {
    permission java.net.SocketPermission "*:1024-65535","connect,accept";
    permission java.net.SocketPermission "*:80", "connect";
};
```

Figure 15: general policy file

This is the code for a general policy file. It means that the downloaded code can use all ports between 1024-65535 and additional the port 80 (HTTP). It is even possible to specify permission on files and give them special permissions to this files. For example a permission can look like:

```
permission java.io.FilePermission "d:\\user\\Server\\example\\-", "read";
```

This statement give the downloaded code only the permission to read in this directory. The policy file will be given as an argument to the java virtual machine. Node: Every class which uses a security manager needs a policy file.
To extend the example used in section 3.2 the `Modifier` class has to change. The `Modifier` class is the only class where it makes sense to download code. This class does not need the `Server_Stub` for compiling, it needs the stub only during runtime for a `Server` remote object. This means when the `Modifier` looks up the remote `Server` object, it's JVM will try to download the `Server_Stub` class file and all necessary other classes which are used by the stub. The changed Modifier:

```
package example_sec;

import java.rmi.*;
import java.io.*;

public class Modifier {

  public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    try {
      if (args.length > 0) {
        IServer server = (IServer) Naming.lookup(args[0]);
        System.out.println("Modifier started");
        System.out.println("+ to increase\n- to decrease\n" +
                           "exit to exit the program\n");
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String input;
        while ((input = in.readLine()) != null && !input.equalsIgnoreCase("exit")) {
                  if (input.equals("+")) {
```

```
            server.increase();
          } else if (input.equals("-")) {
            server.decrease();
          } else {
            System.out.println("+ to increase\n- to decrease\n" +
                               "exit to exit the program\n");
          }
        }
      } else {
        System.out.println("expected argument: location and " +
                           "name for looking up the Server");
        System.out.println("example: rmi://localhost:1099/Server");
      }
    } catch (Exception exc) {
      // don't care about what Exception it is
      exc.printStackTrace();
    }
  }
}
```

RMI extends the class loading mechanism with loading classes from FTP and HTTP servers. When the client gets an object where the class byte code is not in the `CLASSPATH`, it will automatic download them via the `RMIClassLoader` class. This means that the FTP or HTTP server has to provide all necessary class files.

Extend the example with a HTTP server and start application

It is not necessary to build up a heavyweight web server to accomplish this downloading of class files. A simple HTTP server that provides all of the functionality needed to make classes available for downloading in RMI via HTTP can be found at ftp://ftp.javasoft.com/pub/jdk1.1/rmi/class-server.zip
The following services are started sequentially to start the application successfully.

- **start HTTP server**
- **start RMI-registry**
- **start** `Server`
- **start** `Client`

Before executing the application, the necessary files for the client and server have to move to corresponding host machines as mentioned in section 3.2.3. To compile the class files and to generate stubs please follow the instructions in section 3.2.2. The server gets the same classes as in the example without a security manager. But this time the client gets only the `Modifier` class and the `IServer` class and not the `Server_Stub`. In this example the HTTP server, `Server` and `Client` could run on a separate machine. The following steps describe how to run this application on one particular machine. It uses the simple HTTP server mentioned in the previous section. Again, there is no difference in the behavior when executing an RMI application on one machine or on different once.
The example has the following directory structure.

for Win32 Systems:
`c:\user\webserver\examples\classServer` for HTTP server files
`c:\user\server\example_sec` for server files
`c:\user\client\example_sec` for client files
`c:\user\public\example_sec` for `Server_Stub`

for UNIX Systems:
`/home/user/webserver/examples/classServer` for HTTP server files

`/home/user/server/example_sec` for server files
`/home/user/client/example_sec` for client files
`/home/user/public/example_sec` for `Server_Stub`

Start HTTP server
Download the simple HTTP server and extract the zip file. The following commands describe how to compile and execute the server. The first argument given to the `ClassFileServer` is the port where the server runs on and the second argument is the classpath where the server locates class files. More information about the http server can be found in the `readme.txt` file locatet in the archive file.

Commands under under Win32

```
cd c:\user\webserver
javac examples\classServer\*.java
java examples.classServer.ClassFileServer 2001 /home/user/public
```

Commands under UNIX/LINUX

```
cd /home/user/webserver/
javac examples/classServer/*.java
java examples.classServer.ClassFileServer 2001 /home/user/public
```

Start RMI-registry and the Server
Before start the RMI-registry, the `CLASSPATH` environment variable should not include the path to any classes, including the stubs for remote objects, that should be downloaded to clients of the remote objects. If the RMI-registry can find the classes in the `CLASSPATH`, it will not load stub classes from the server's code base, specified by the `java.rmi.server.codebase` property when server application is started. Therefore, the client will not get the right code base from the RMI-registry and will throw a `ClassNotFoundException`.
The `java` command to start the `Server` needs a few properties.
The `java.rmi.server.codebase` property specifies the location, a code base URL, of classes originating from this server so that class information for objects sent to other virtual machines will include the location of the class so that a receiver can load it.
The `java.rmi.server.hostname` property indicates the host name of your server.

Commands under Win32
If the `start` command is not existent, there is another command `javaw`. However if both commands fail, it is possible to start the RMI-registry without these help only that a new console is required.

```
cd c:\user
start rmiregistry
cd c:\user\server\
java -Djava.rmi.server.codebase=http://hope0:2001/
     -Djava.rmi.server.hostname=hope0
     example_sec.Server //hope0:1099/Server
```

Commands under UNIX/LINUX

```
cd /home/user
rmiregistry &
cd /user/home/server/
java -Djava.rmi.server.codebase=http://hope0:2001/
     -Djava.rmi.server.hostname=hope0
     example_sec.Server //hope0:1099/Server
```

Start the client
The client is an application that depends on a running `Server`. If the `Server` object could

not be found an exception stack trace will be printed. If there is no http server or the server has not the necessary files in its `CLASSPATH` the client will throw a `ClassNotFoundException`.
The `java` command to start the `Modifier` needs also a property.
The `java.security.policy` property, used to specify the policy file that contains the permissions for the downloaded classes. This example uses the general policy file shown in Figure 15.
Commands under Win32

```
cd c:\user\client\
java -Djava.security.policy=example_sec\java.policy
     example_sec.Modifier //hope0:1099/Server
```

Commands under UNIX/LINUX

```
cd /home/user/client/
java -Djava.security.policy=example_sec\java.policy
     example_sec.Modifier //hope0:1099/Server
```

For further information, syntax and possibilities please read the Java API [6, 5].

C Source code of the examples

The 3 subsections show the source codes of the different examples. The first subsection shows the Modifier which is used for every example the same. The second subsection describes the different Server and the last subsection shows the different source codes of the Clients with the Listeners. Each example has it's own package named `exampleXY`. Where `XY` is the number of the example.

C.1 The Modifier

```
package exampleXY;

import java.rmi.*;
import java.io.*;

public class Modifier {

  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        IServer server = (IServer) Naming.lookup(args[0]);
        System.out.println("Modifier started");
        System.out.println("+ to increase\n- to decrease\n" +
                           "exit to exit the program\n");
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String input;
        while ((input = in.readLine()) != null && !input.equalsIgnoreCase("exit")) {
                 if (input.equals("+")) {
            server.increase();
          } else if (input.equals("-")) {
            server.decrease();
          } else {
            System.out.println("+ to increase\n- to decrease\n" +
                               "exit to exit the program\n");
          }
        }
      } else {
        System.out.println("expected argument: location and " +
                           "name for looking up the Server");
        System.out.println("example: rmi://localhost:1099/Server");
      }
    } catch (Exception exc) {
      // don't care about what Exception it is
      exc.printStackTrace();
    }
  }
}
```

C.2 The Server

The IServer interface

```
package exampleXY;

public interface IServer extends java.rmi.Remote {

  public void increase() throws java.rmi.RemoteException;

  public void decrease() throws java.rmi.RemoteException;

  public void addCounterListener(ICounterListener l)
  throws java.rmi.RemoteException;

  public void removeCounterListener(ICounterListener l)
  throws java.rmi.RemoteException;
}
```

The Server class

There are two different versions of the Server. Both implement the IServer interface but the difference is how they handle the registered `ICounterListeners`. All methods which handles with remote objects or which are remote accessible have to throw a RemoteException as mentioned in 3. The first source code of the Server class handle with remote ICounterListener objects and the second with non remote ICounterListener.

Server which handle with remote ICounterListener

```
package exampleXY;

import java.rmi.*;

public class Server extends java.rmi.server.UnicastRemoteObject
                    implements IServer {

  java.util.LinkedList listener = new java.util.LinkedList();

  int counter;

  private Server() throws RemoteException {
  }

  public void increase() throws RemoteException {
    counter++;
    System.out.println("Increasing value to "+counter);
    fireCounterChanged();
  }

  public void decrease() throws RemoteException {
    counter--;
    System.out.println("Decreasing value to "+counter);
    fireCounterChanged();
  }

  private void fireCounterChanged() throws RemoteException {
    for (int i=0; i< listener.size(); i++) {
      ((ICounterListener) listener.get(i)).valueChanged(counter);
    }
  }
```

```
  public void addCounterListener(ICounterListener l) throws RemoteException {
    listener.add(l);
  }

  public void removeCounterListener(ICounterListener l) throws RemoteException {
    listener.remove(l);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        Naming.rebind(args[0], new Server());
        System.out.println("Server started");
      } else {
        System.out.println("expected argument: location and "+
                           "name for binding Server");
        System.out.println("example: rmi://localhost:1099/Server");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

Server which handles non remote ICounterLister

```
package exampleXY;

import java.rmi.*;

public class Server extends java.rmi.server.UnicastRemoteObject
                    implements IServer {

  java.util.LinkedList listener = new java.util.LinkedList();

  int counter;

  private Server() throws RemoteException {
  }

  public void increase() throws RemoteException {
    counter++;
    System.out.println("Increasing value to "+counter);
    fireCounterChanged();
  }

  public void decrease() throws RemoteException {
    counter--;
    System.out.println("Decreasing value to "+counter);
    fireCounterChanged();
  }

  private void fireCounterChanged() {
    for (int i=0; i< listener.size(); i++) {
      ((ICounterListener) listener.get(i)).valueChanged(counter);
    }
  }
```

```
  public void addCounterListener(ICounterListener l) throws RemoteException {
    listener.add(l);
  }

  public void removeCounterListener(ICounterListener l) throws RemoteException {
    listener.remove(l);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        Naming.rebind(args[0], new Server());
        System.out.println("Server started");
      } else {
        System.out.println("expected argument: location and " +
                           "name for binding Server");
        System.out.println("example: rmi://localhost:1099/Server");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

C.3 The Client

All examples use a Modifier and a Server. The Server is dependent on the `ICounterListener` interface which is used by the example. There are two different source codes of the ICounterListener interface:

- *remote ICounterListener interface which requires the Server that handles remote ICounterListener.*

```
package exampleXY;

public interface ICounterListener extends java.rmi.Remote {

  public void valueChanged(int newCounter) throws java.rmi.RemoteException;
}
```

- *non remote ICounterListener interface which required the Server that handle with non remote ICounterListener.*

```
package exampleXY;

public interface ICounterListener {

  public void valueChanged(int newCounter);
}
```

C.3.1 Example 1

A non remote Listener with remote enclosing Client class
Uses the non remote ICounterListener interface.

The remote IClient interface

```
package example1;

public interface IClient extends java.rmi.Remote {

  public void notifyChange(int value) throws java.rmi.RemoteException;

}
```

The Client class

```
package example1;

import java.rmi.*;

public class Client extends java.rmi.server.UnicastRemoteObject
                    implements IClient{

  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(new CounterListener());
  }

  public void notifyChange(int value) throws RemoteException {
    System.out.println("Client notified new value "+value);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 1) {
        Naming.rebind(args[1], new Client(args[0]));
        System.out.println("Client started");
      } else {
        System.out.println("expected two arguments:");
        System.out.println("1) location and name for looking up Server");
        System.out.println("1) example: rmi://localhost:1099/Server");
        System.out.println("2) location and name for binding Client");
        System.out.println("2) example: rmi://localhost:1099/Client");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
  public class CounterListener  implements ICounterListener,java.io.Serializable {

    public void valueChanged(int value) {
      try {
        notifyChange(value);
      } catch (Exception exc) {
        exc.printStackTrace();
      }
    }
  }
```

```
}
```

C.3.2 Example 2

A non remote Listener with non remote enclosing Client class
Uses the non remote ICounterListener interface.

The Client class

```
package example2;

import java.rmi.*;

public class Client implements java.io.Serializable {

  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(new CounterListener());
  }

  public void notifyChange(int value) {
    System.out.println("Client notified new value "+value);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        new Client(args[0]);
        System.out.println("Client started");
        while(true) {
          Thread.currentThread().sleep(100);
        }
      } else {
        System.out.println("expected two arguments:");
        System.out.println("1) location and name for looking up Server");
        System.out.println("1) example: rmi://localhost:1099/Server");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }


  public class CounterListener implements ICounterListener,java.io.Serializable {

    public CounterListener() {
    }

    public void valueChanged(int value) {
      notifyChange(value);
    }
  }


}
```

C.3.3 Example 3

A remote Listener with remote enclosing Client class
Uses the remote ICounterListener interface.

The remote IClient interface

```
package example3;

public interface IClient extends java.rmi.Remote {

  public void notifyChange(int value)
  throws java.rmi.RemoteException;
}
```

The Client class

```
package example3;

import java.rmi.*;

public class Client extends java.rmi.server.UnicastRemoteObject
                    implements IClient {

  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(new CounterListener());
  }

  public void notifyChange(int value)
  throws RemoteException {
    System.out.println("Client notified new value "+value);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 1) {
        Naming.rebind(args[1], new Client(args[0]));
        System.out.println("Client started");
      } else {
        System.out.println("expected two arguments:");
        System.out.println("1) location and name for looking up Server");
        System.out.println("1) example: rmi://localhost:1099/Server");
        System.out.println("2) location and name for binding Client");
        System.out.println("2) example: rmi://localhost:1099/Client");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }

  }


  public class CounterListener extends java.rmi.server.UnicastRemoteObject
                               implements ICounterListener {

    public CounterListener() throws java.rmi.RemoteException {
    }
```

```
    public void valueChanged(int value) throws java.rmi.RemoteException {
      notifyChange(value);
    }
  }


}
```

C.3.4 Example 4

A remote Listener with non remote enclosing Client class
Uses the remote ICounterListener interface.

The Client class

```
package example4;

import java.rmi.*;

public class Client implements java.io.Serializable  {


  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(new CounterListener());
  }

  public void notifyChange(int value) {
    System.out.println("Client notified new value "+value);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        new Client(args[0]);
        System.out.println("Client started");
        while(true) {
          Thread.currentThread().sleep(100);
        }
      } else {
        System.out.println("expected two arguments:");
        System.out.println("1) location and name for looking up Server");
        System.out.println("1) example: rmi://localhost:1099/Server");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }


  public class CounterListener extends java.rmi.server.UnicastRemoteObject
                               implements ICounterListener {

    public CounterListener() throws java.rmi.RemoteException {
    }

    public void valueChanged(int value) throws java.rmi.RemoteException {
      notifyChange(value);
    }
  }


}
```

C.3.5 Example 5

Standalone non remote Listener with reference to remote Client class
Uses the non remote ICounterListener interface.

The CounterListener class

```
package example5;

public class CounterListener implements ICounterListener, java.io.Serializable {

  IClient client;

  public CounterListener(IClient client) throws java.rmi.RemoteException {
    this.client = client;
  }

  public void valueChanged(int value) {
    try {
      client.notifyChange(value);
    } catch (java.rmi.RemoteException exc) {
      exc.printStackTrace();
    }
  }
}
```

The remote IClient interface

```
package example5;

public interface IClient extends java.rmi.Remote {

  public void notifyChange(int value)
  throws java.rmi.RemoteException;
}
```

The Client class

```
package example5;

import java.rmi.*;

public class Client extends java.rmi.server.UnicastRemoteObject
                    implements IClient {

  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(new CounterListener(this));
  }

  public void notifyChange(int value) throws RemoteException {
    System.out.println("Client notified new value "+value);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 1) {
        Naming.rebind(args[1], new Client(args[0]));
        System.out.println("Client started");
```

```
      } else {
        System.out.println("expected two arguments:");
        System.out.println("1) location and name for looking up Server");
        System.out.println("1) example: rmi://localhost:1099/Server");
        System.out.println("2) location and name for binding Client");
        System.out.println("2) example: rmi://localhost:1099/Client");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

C.3.6 Example 6

Standalone non remote Listener with reference to non remote Client class
Uses the non remote ICounterListener interface.

The CounterListener class

```
package example6;

public class CounterListener implements ICounterListener,
                                        java.io.Serializable {

  Client client;

  public CounterListener(Client client) throws java.rmi.RemoteException {
    this.client = client;
  }

  public void valueChanged(int value) {
    client.notifyChange(value);
  }
}
```

The Client class

```
package example6;

import java.rmi.*;

public class Client implements java.io.Serializable {

  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(new CounterListener(this));
  }

  public void notifyChange(int value) {
    System.out.println("Client notified new value "+value);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        new Client(args[0]);
        System.out.println("Client started");
        while(true) {
          Thread.currentThread().sleep(100);
        }
      } else {
        System.out.println("expected two arguments:");
        System.out.println("1) location and name for looking up Server");
        System.out.println("1) example: rmi://localhost:1099/Server");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

C.3.7 Example 7

Standalone remote Listener with reference to remote Client class
Uses the remote ICounterListener interface.

The CounterListener class

```
package example7;

public class CounterListener extends java.rmi.server.UnicastRemoteObject
                             implements ICounterListener {

  IClient client;

  public CounterListener(IClient client) throws java.rmi.RemoteException {
    this.client = client;
  }

  public void valueChanged(int value) throws java.rmi.RemoteException {
    client.notifyChange(value);
  }
}
```

The remote IClient interface

```
package example7;

public interface IClient extends java.rmi.Remote {

  public void notifyChange(int value)
  throws java.rmi.RemoteException;
}
```

The Client class

```
package example7;

import java.rmi.*;

public class Client extends java.rmi.server.UnicastRemoteObject
                    implements IClient {

  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(new CounterListener(this));
  }

  public void notifyChange(int value) throws RemoteException {
    System.out.println("Client notified new value "+value);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 1) {
        Naming.rebind(args[1], new Client(args[0]));
        System.out.println("Client started");
      } else {
        System.out.println("expected two arguments:");
        System.out.println("1) location and name for looking up Server");
```

```
        System.out.println("1) example: rmi://localhost:1099/Server");
        System.out.println("2) location and name for binding Client");
        System.out.println("2) example: rmi://localhost:1099/Client");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

C.3.8 Example 8

Standalone remote Listener with reference to non remote Client class
Uses the remote ICounterListener interface.

The CounterListener class

```
package example8;

public class CounterListener extends java.rmi.server.UnicastRemoteObject
                             implements ICounterListener {

  Client client;

  public CounterListener(Client client) throws java.rmi.RemoteException {
    this.client = client;
  }

  public void valueChanged(int value) throws java.rmi.RemoteException {
    client.notifyChange(value);
  }
}
```

The Client class

```
package example8;

import java.rmi.*;

public class Client implements java.io.Serializable {

  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(new CounterListener(this));
  }

  public void notifyChange(int value) {
    System.out.println("Client notified new value "+value);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        new Client(args[0]);
        System.out.println("Client started");
        while(true) {
          Thread.currentThread().sleep(100);
        }
      } else {
        System.out.println("expected two arguments:");
        System.out.println("1) location and name for looking up Server");
        System.out.println("1) example: rmi://localhost:1099/Server");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

C.3.9 Example 9

A non remote Client class implementing a non remote Listener interface
Uses the non remote ICounterListener interface.

The Client class

```
package example9;

import java.rmi.*;

public class Client implements ICounterListener, java.io.Serializable {

  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(this);
  }

  public void notifyChange(int value) {
    System.out.println("Client notified new value "+value);
  }

  public void valueChanged(int value) {
      notifyChange(value);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        new Client(args[0]);
        System.out.println("Client started - quit with CTRL-C");
        while(true) {
            Thread.currentThread().sleep(100);
        }
      } else {
        System.out.println("expected argument:");
        System.out.println("1) location and name for looking up Server");
        System.out.println("1) example: rmi://localhost:1099/Server");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

C.3.10 Example 10

A non remote Client class implementing a remote Listener interface
Uses the remote ICounterListener interface.

The Client class

```
package example10;

import java.rmi.*;

public class Client implements ICounterListener, java.io.Serializable {

  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(this);
  }

  public void notifyChange(int value) {
    System.out.println("Client notified new value "+value);
  }

  public void valueChanged(int value) throws RemoteException {
    notifyChange(value);
  }

  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        new Client(args[0]);
        System.out.println("Client started - quit with CTRL-C");
        while(true) {
            Thread.currentThread().sleep(100);
        }
      } else {
        System.out.println("expected argument:");
        System.out.println("1) location and name for looking up Server");
        System.out.println("1) example: rmi://localhost:1099/Server");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

C.3.11 Example 11

A remote Client class implementing a non remote Listener interface
Uses the non remote ICounterListener interface.

The remote IClient interface

```
package example11;

public interface IClient extends java.rmi.Remote {

  public void notifyChange(int value) throws java.rmi.RemoteException;
}
```

The Client class

```
package example11;

import java.rmi.*;

public class Client extends java.rmi.server.UnicastRemoteObject
                    implements IClient,
                               ICounterListener,
                               java.io.Serializable {

  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(this);
  }

  public void notifyChange(int value) throws RemoteException {
    System.out.println("Client notified new value "+value);
  }

  public void valueChanged(int value) {
    try {
        notifyChange(value);
    } catch (RemoteException exc) {
        exc.printStackTrace();
    }
  }

  public static void main(String[] args) {
    try {
      if (args.length > 1) {
        Naming.rebind(args[1], new Client(args[0]));
        System.out.println("Client started");
      } else {
        System.out.println("expected two arguments:");
        System.out.println("1) location and name for looking up Server");
        System.out.println("1) example: rmi://localhost:1099/Server");
        System.out.println("2) location and name for binding Client");
        System.out.println("2) example: rmi://localhost:1099/Client");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

C.3.12 Example 12

A remote Client class implementing a remote Listener interface
Uses the remote ICounterListener interface.

The remote IClient interface

```
package example12;

public interface IClient extends java.rmi.Remote {

  public void notifyChange(int value)
  throws java.rmi.RemoteException;
}
```

The Client class

```
package example12;

import java.rmi.*;

public class Client extends java.rmi.server.UnicastRemoteObject
                    implements IClient,
                               ICounterListener {

  public Client(String serverLocation) throws Exception {
    IServer server = (IServer) Naming.lookup(serverLocation);
    server.addCounterListener(this);
  }

  public void notifyChange(int value) throws RemoteException {
    System.out.println("Client notified new value "+value);
  }

  public void valueChanged(int value) throws RemoteException {
    try {
        notifyChange(value);
    } catch (RemoteException exc) {
        exc.printStackTrace();
    }
  }

  public static void main(String[] args) {
    try {
      if (args.length > 1) {
        Naming.rebind(args[1], new Client(args[0]));
        System.out.println("Client started");
      } else {
        System.out.println("expected two arguments:");
        System.out.println("1) location and name for looking up Server");
        System.out.println("1) example: rmi://localhost:1099/Server");
        System.out.println("2) location and name for binding Client");
        System.out.println("2) example: rmi://localhost:1099/Client");
      }
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

List of Figures

1 Listeners sign interest 4
2 Listeners are invoked 4
3 RMI communication structure 7
4 little example 8
5 ClientServerCommunication 13
6 littleExampleClientOutput 16
7 littleExampleServerOutput 16
8 structureRMIListener 17
9 ListenerConceptServerModifier 17
10 ListenerConceptClientInnerListener 18
11 ListenerConceptClientStandaloneListener 19
12 ListenerConceptClientImplementsListenerInterface 19
13 stacktrace11 21
14 stacktrace1 22
15 general policy file 24

List of Tables

1 possible solutions 19

References

[1] Sun Microsystems by Alan Sommerer. *Java API Specification, The Java Archive (JAR) File Format*. eee, September 1998. 3.2.2, A

[2] Sun Microsystems by jGuru. *Fundamentals of RMI*. http://developer.java.sun.com/-developer/onlineTraining/rmi/, February 2000. 3.2.4

[3] R. Johnson E. Gamma, R. Helm and J. Vlissides. *Design Patterns, Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995. 2

[4] Sun Microsystems. *Java RMI Specification*. http://java.sun.com/products/jdk/1.2/-docs/guide/rmi/spec/rmiTOC.doc.html, 1998. 3.2.4

[5] Sun Microsystems. *Java RMI Specification, Dynamic Class Loading*. http://java.sun.-com/products/jdk/1.2/docs/guide/rmi/spec/rmi-arch.doc4.html, 1998. B

[6] Sun Microsystems. *Java API Specification, Class SecurityManager*. http://java.sun.-com/j2se/1.4.1/docs/api/java/lang/SecurityManager.html , 2002. B

[7] Sun Microsystems. *Java API Specification, Naming*. //java.sun.com/j2se/1.4.1/-docs/api/java/rmi/Naming.html , 2002. 3.2.1

[8] Sun Microsystems. *Java API Specification, RemoteException*. http://java.sun.com/-j2se/1.4.1/docs/api/java/rmi/RemoteException.html , 2002. 3.1

[9] Sun Microsystems. *Java API Specification, Serializable*. http://java.sun.com/j2se/-1.4.1/docs/api/java/io/Serializable.html , 2002. 3.2.1

[10] Sun Microsystems. *Java API Specification, UnicastRemoteObject*. p://java.sun.com/-j2se/1.4.1/docs/api/java/rmi/server/UnicastRemoteObject.html , 2002. 3.2.1