



Using Custom Socket Factories with Java™ RMI

This tutorial shows you steps to follow to implement and use custom socket factories with Java™ Remote Method Invocation (Java RMI). Custom socket factories can be used to control how remote method invocations are communicated at the network level. For example, they can be used to set socket options, control address binding, control connection establishment (such as to require authentication), and to control data encoding (such as to add encryption or compression).

When a remote object is exported, such as with the constructors or `exportObject` methods of `java.rmi.server.UnicastRemoteObject` or `java.rmi.activation.Activatable`, then it is possible to specify a custom client socket factory (a `java.rmi.server.RMIClientSocketFactory` instance) and a custom server socket factory (a `java.rmi.server.RMIServerSocketFactory` instance) to be used when communicating remote invocations for that remote object.

A client socket factory controls the creation of sockets used to initiate remote invocations and thus can be used to control how connections are established and the type of socket to use. A server socket factory controls the creation of server sockets used to receive remote invocations and thus can be used to control how incoming connections are listened for and accepted as well as the type of sockets to use for incoming connections.

The remote stub for a remote object contains the client socket factory, if any, that the remote object was exported with, and thus a client socket factory must be serializable, and its code may be downloaded by clients just like the code for stub classes or any other serializable object passed over a remote invocation.

There is also a `LocateRegistry.createRegistry` method for exporting a registry with custom socket factories and a `LocateRegistry.getRegistry` method for obtaining the stub for a registry with a custom client socket factory.

(Note that there is also a global socket factory for Java RMI, which can be set with the `setSocketFactory` method of `java.rmi.server.RMISocketFactory`. This global socket factory is used for creating sockets when a remote stub does not contain a custom client socket factory and for creating server sockets when a remote object was not exported with a custom server socket factory.)

This tutorial has three parts:

- [Implementing a Custom Socket Factory.](#)
- [Using a Custom Socket Factory in an Application.](#)
- [Compiling and Running the Application.](#)

The source code for this tutorial is available in the following formats:

- [sockets.zip](#)
- [sockets.tar](#)
- [sockets.tar.Z](#)

Many users are interested in secure communication between Java RMI clients and servers, such as with mutual authentication and encryption. Custom socket factories provide a hook for doing this. For more information, see [Using Java RMI with SSL](#).

Implementing a Custom Socket Factory

Below are three steps for implementing a pair of custom socket factory classes:

1. [Implement a custom `ServerSocket` and `Socket`.](#)
2. [Implement a custom `RMIClientSocketFactory`.](#)
3. [Implement a custom `RMIServerSocketFactory`.](#)

Step 1:

Implement a custom `ServerSocket` and `Socket`

The type of socket to use is an application-specific decision. If your server sends or receives sensitive data, you might want a socket that encrypts the data.

In this example, the custom socket factory creates sockets that perform simple XOR encryption and decryption. Note that this kind of encryption is easily decrypted by a knowledgeable cryptanalyst and is only used here to keep the example simple.

The custom XOR socket implementation includes the following sources. XOR sockets use special input and output stream implementations to handle XOR-ing the data written to or read from the socket.

- [XorInputStream.java](#)
- [XorOutputStream.java](#)
- [XorServerSocket.java](#)
- [XorSocket.java](#)

Step 2:

Implement a custom `RMIClientSocketFactory`

The client-side socket factory, `XorClientSocketFactory`, implements the `java.rmi.server.RMIClientSocketFactory` interface. The client socket factory needs to implement the `createSocket` method to return the appropriate client socket instance, an `XorSocket`.

The client socket factory must implement the `java.io.Serializable` interface so that instances can be serialized to clients as part of remote stubs. It is also essential to implement the `equals` and `hashCode` methods so that the Java RMI implementation will correctly share resources among remote stub instances with equivalent client socket factories.

```
package examples.rmisocfac;

import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class XorClientSocketFactory
    implements RMIClientSocketFactory, Serializable {
    private byte pattern;

    public XorClientSocketFactory(byte pattern) {
        this.pattern = pattern;
    }

    public Socket createSocket(String host, int port)
        throws IOException
    {
        return new XorSocket(host, port, pattern);
    }

    public int hashCode() {
        return (int) pattern;
    }

    public boolean equals(Object obj) {
        return (getClass() == obj.getClass() &&
            pattern == ((XorClientSocketFactory) obj).pattern);
    }
}
```

Step 3:

Implement a custom `RMI ServerSocketFactory`

The server-side socket factory, `XorServerSocketFactory`, implements the `java.rmi.server.RMI ServerSocketFactory` interface. The server socket factory needs to implement the `createServerSocket` method to return the appropriate server socket instance, an `XorServerSocket`.

The server socket factory does not need to implement the `Serializable` interface because server socket factory instances are not contained in remote stubs. It is still essential for the server socket factory to implement the `equals` and `hashCode` methods so that the Java RMI implementation will correctly share resources among remote objects exported with equivalent socket factories.

```
package examples.rmisocfac;

import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class XorServerSocketFactory
    implements RMIServerSocketFactory {
    private byte pattern;

    public XorServerSocketFactory(byte pattern) {
        this.pattern = pattern;
    }
}
```

```

public ServerSocket createServerSocket(int port)
    throws IOException
{
    return new XorServerSocket(port, pattern);
}

public int hashCode() {
    return (int) pattern;
}

public boolean equals(Object obj) {
    return (getClass() == obj.getClass() &&
        pattern == ((XorServerSocketFactory) obj).pattern);
}
}

```

Using a Custom Socket Factory in an Application

There are only two more steps to complete when using a custom socket factory for a remote object:

1. Write a server application that creates a remote object and exports it to use your custom `RMIClientSocketFactory` and `RMI ServerSocketFactory` implementations. Store a reference to the remote object's stub in a Java RMI registry so that clients can look it up.
2. Write a client application that looks up the stub for the remote object and invokes a remote method. The custom socket factories do not need to be referenced in the client application. The client-side socket factory will be downloaded to the client when the client looks up the stub for the remote object.

Step 1:

Write a server application

If communication with a remote object requires the use of custom sockets, you need to specify which custom socket factories to use when you export the remote object. When your application exports the remote object specifying custom socket factories, the Java RMI runtime will use the corresponding custom `RMI ServerSocketFactory` to create a server socket to accept incoming calls to the remote object, and it will create a stub that contains the corresponding custom `RMIClientSocketFactory`. That client socket factory will be used to create connections when remote invocations are made to the remote object using the stub.

This example is similar to the example in the tutorial [Getting Started Using Java RMI](#), but it uses custom socket factories instead of the default sockets used by the Java RMI implementation.

The application uses the following `Hello` remote interface:

```

package examples.rmisocfac;

public interface Hello extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException;
}

```

The server application creates a remote object implementing the `Hello` remote interface and exports the object to use custom socket factories using the `java.rmi.server.UnicastRemoteObject.exportObject` method that takes the custom socket factories as arguments. Next, it creates a local registry and, in that registry, it binds a reference to the remote object's stub with the name "Hello".

```

package examples.rmisocfac;

import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class HelloImpl implements Hello {

    public HelloImpl() {}

    public String sayHello() {
        return "Hello World!";
    }

    public static void main(String args[]) {

        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }

        byte pattern = (byte) 0xAC;
        try {
            /*
             * Create remote object and export it to use
             * custom socket factories.
             */
            HelloImpl obj = new HelloImpl();
            RMIClientSocketFactory csf = new XorClientSocketFactory(pattern);
            RMI ServerSocketFactory ssf = new XorServerSocketFactory(pattern);
            Hello stub =

```

```

        (Hello) UnicastRemoteObject.exportObject(obj, 0, csf, ssf);
    /*
     * Create a registry and bind stub in registry.
     */
    LocateRegistry.createRegistry(2002);
    Registry registry = LocateRegistry.getRegistry(2002);
    registry.rebind("Hello", stub);
    System.out.println("HelloImpl bound in registry");
} catch (Exception e) {
    System.out.println("HelloImpl exception: " + e.getMessage());
    e.printStackTrace();
}
}
}

```

Step 2: Write a client application

The client application obtains a reference to the registry used by the server application. It then looks up the remote object's stub and invokes its remote method `sayHello`:

```

package examples.rmisocfac;

import java.rmi.*;
import java.rmi.registry.*;

public class HelloClient {

    public static void main(String args[]) {

        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }

        try {
            Registry registry = LocateRegistry.getRegistry(2002);
            Hello obj = (Hello) registry.lookup("Hello");
            String message = obj.sayHello();
            System.out.println(message);

        } catch (Exception e) {
            System.out.println("HelloClient exception: " +
                e.getMessage());
            e.printStackTrace();
        }
    }
}

```

Compiling and Running the Application

There are four steps to compile and run the application:

1. [Compile the remote interface, client, and server classes](#)
2. [Run `rmic` on the implementation class \(optional\)](#)
3. [Start the server](#)
4. [Run the client](#)

Step 1:

Compile the remote interface, client, and server classes

```

javac -d . XorInputStream.java
javac -d . XorOutputStream.java
javac -d . XorSocket.java
javac -d . XorServerSocket.java
javac -d . XorServerSocketFactory.java
javac -d . XorClientSocketFactory.java
javac -d . Hello.java
javac -d . HelloClient.java
javac -d . HelloImpl.java

```

Step 2:

Run `rmic` on the implementation class (optional)

Note: Running `rmic` to pregenerate a stub class for a remote object's class is only required if the remote object needs to support pre-5.0 clients. As of the 5.0 release, if a pregenerated stub class for a remote object's class cannot be loaded when the remote object is exported, the remote object's stub class is generated dynamically.

```
rmic -d . examples.rmisocfac.HelloImpl
```

Step 3:
Start the server

```
java -Djava.security.policy=policy examples.rmisocfac.HelloImpl
```

The server output should look like this:

```
HelloImpl bound in registry
```

Step 4:
Run the client

In another window start the client application making sure that the application classes are in the class path:

```
java -Djava.security.policy=policy examples.rmisocfac.HelloClient
```

The client output should look like this:

```
Hello World!
```

Note: Both the server and client applications use a security policy file that grants permissions only to files in the local class path (the current directory). The server application needs permission to accept connections, and both the server and client applications need permission to make connections. The permission `java.net.SocketPermission` is granted to the specified codebase URL, a "file:" URL relative to the current directory. This permission grants the ability to both accept connections from and make connections to any host on unprivileged ports (that is ports ≥ 1024).

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

Note that if you want to customize the communication to the registry as well, such as to secure registry communication, then that can be done by passing the appropriate custom socket factories to the `LocateRegistry.createRegistry` and `LocateRegistry.getRegistry` invocations.