THE RMI PROXY USER GUIDE

1 Introduction

Java RMI allows Java programs executing within different Java Virtual Machines to communicate using the method-call syntax of Java. This works well as long as Internet firewalls don't get in the way.

If there is a firewall between the client and the server, up until now you have had two choices:

- you can configure the client to use SOCKS to traverse its firewall
- by default, RMI falls back to a technique called *HTTP tunnelling*, which encapsulates the RMI protocol inside HTTP requests and responses

Both techniques assume that any firewalls in the configuration are at the client side, and that the RMI server is not behind a firewall.

1.1 Firewalls

There are two kinds of firewalls, normally found in combination:

1. TCP/IP or transport firewalls, which merely control access to TCP and UDP ports by number, and control the propagation of other IP protocols.

2. Application firewalls, which are aware of an application protocol such as HTTP, FTP, GIOP, RMI, etc, and control the traffic through them based on the actual protocol content.

The RMI Proxy is an application firewall or *application proxy*. It does for RMI what an HTTP proxy does for HTTP—and more. Provided that any associated TCP/IP firewall permits access to the designated RMI Proxy ports, the RMI Proxy controls traffic through it, providing two critical security functions:

- it enforces the RMI protocol: any other protocol (such as HTTP, Telnet, &c) is rejected
- it provides access control, down to the class and method level, over remote objects being accessed

The RMI Proxy provides a *Proxy Registry*, which mirrors registries inside the firewall, subject to its own access control.

Using the RMI Proxy, RMI Servers can now be located inside server-side firewall enclaves.

1.2 How RMI Proxying works

An RMI server behind a firewall uses the nearest RMI Proxy to have its RMI Registry binding *propagated* to a designated RMI Proxy associated with that firewall. Depending on the number of intervening transport firewalls between the server and the Internet, this binding may be further propagated along a *chain* of RMI Proxy hosts. The outermost server-side RMI Proxy effectively makes the remote object available to 'external' clients.

For an RMI client, a lookup operation on that remote object must be directed to the visible server-side proxy, not the actual server. If the client itself is behind a transport firewall, the lookup is *delegated* to its designated RMI Proxy, which may in turn lead to a sequence of delegations along a chain of client-side proxies. The outermost client-side proxy is able to communicate via standard RMI with the outermost server-side proxy.

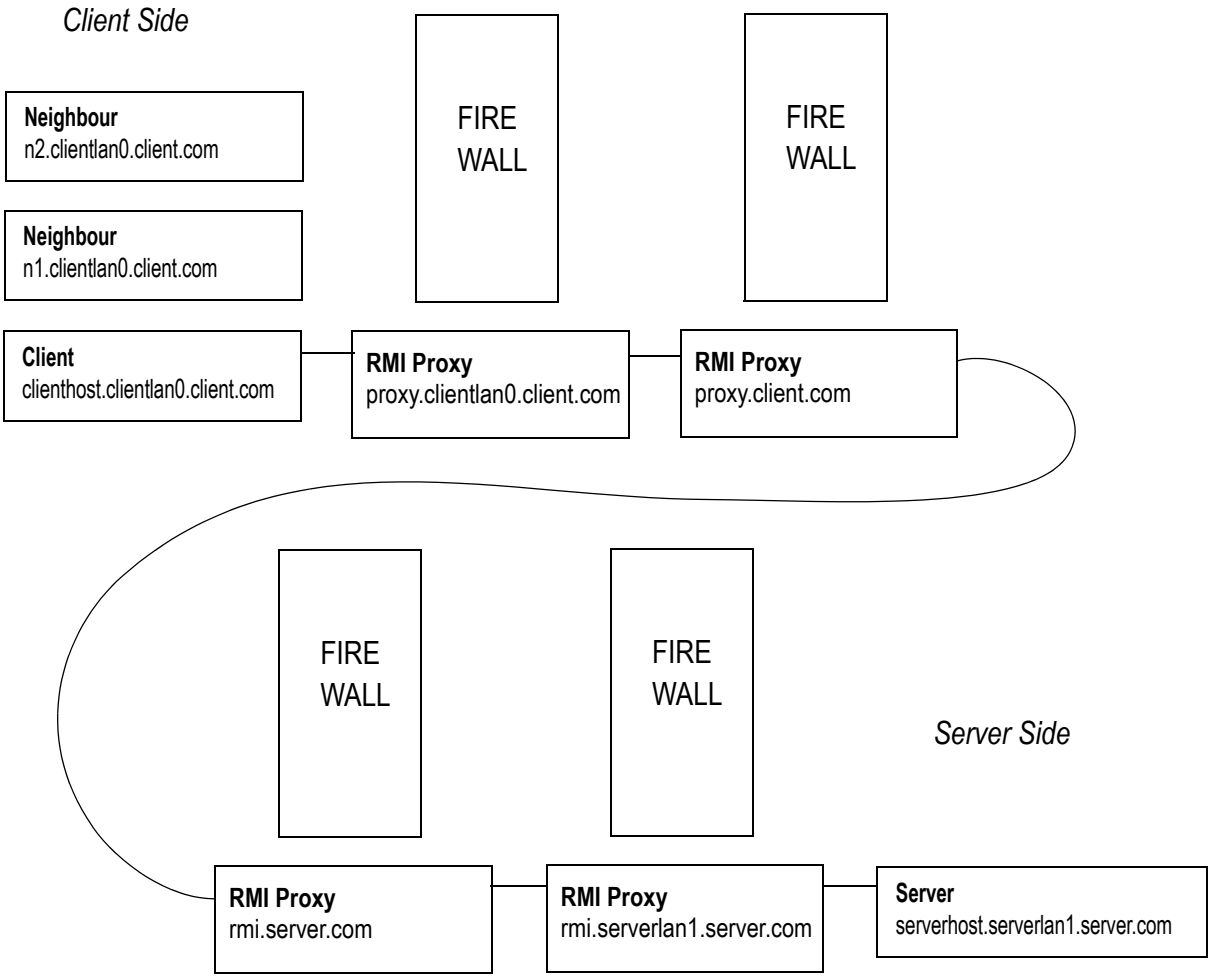

Figure 1: Sample RMI network setup

The RMI Proxies mediate between downstream RMI servers and upstream RMI clients. They perform RMI *protocol validation* and forward RMI calls from one to the other, ensuring that remote

stubs passed along the chain are appropriate for their recipient. They can also perform *access control* in the process, checking that remote method calls are permitted at each step in the chain.

1.3 Related documentation

The RMI Proxy—FAQ
RMI Proxy API Specification
The RMI Proxy White Paper
The Java Platform Standard Edition Home Page
The Java RMI Home Page
The Java Security Home Page

2 RMI Proxy: product & API

The RMI Proxy consists of the following components:

- the API, supplied as RMIProxy.jar, which must be made available to clients and servers via their CLASSPATH
- the Proxy proper, supplied as ProxyBoot.jar and ProxyMain.jar: an executable program which runs in designated RMI Proxy hosts
- The RMI Proxy - FAQ
- this Guide
- the RMI Proxy API Specification
- The RMI Proxy White Paper

2.1 Package com.rmiproxy

The package com.rmiproxy provides an API to support point-to-point proxying for RMI clients and servers.

Table 1: com.rmiproxy package

Interfaces	Purpose
Firewall	Defines constants
Version	Defines the current firewall version and copyright notice
Classes	
Configuration	Holds configuration details
ProxyNaming	Main access point to the RMI Proxy
Exceptions	

Table 1: com.rmiproxy package (continued)

FirewallException	RMI Proxy-specific exception class

Developers need only really concern themselves with the class ProxyNaming. This class has the same API as java.rmi.Naming, and should be used in its place.

The following features are not included in the API (but are planned for a future release):

- Support for end-to-end (pass-through) proxying
- Support for transparent and secure (SSL/TLS) communications via either point-to-point or end-to-end proxying

2.2 Package com.rmiproxy.jndi

The package com.rmiproxy.jndi implements a JNDI provider for the RMI Proxy Registry. It corresponds precisely to the RMI Registry JNDI provider, except that it calls ProxyNaming instead of Naming. It contains a RegistryContext class and a RegistryContextFactory class.

2.3 JDK versions

- The RMI Proxy itself requires Java 2 JDK 1.3 or 1.3.1. It should work correctly under 1.4.x although this has not been tested.
- Clients and servers using the RMI Proxy can be of any Java version which is compatible with JDK 1.3 under serialization and which can use the RMI 1.1 or 1.2 stub protocols. This includes JDKs 1.0, 1.1.x, 1.2.x, 1.3.x, 1.4.x, and as many future JDK versions as may comply with this specification
- JDK 1.1 is supported provided that JDK 1.1 skeletons for RMI servers are available via the same codebase as the corresponding stubs. If any skeleton is not found, the corresponding service will only be available via the RMI Proxy using the 1.2 protocol; if the stub was a 1.1 stub (generated by JDK 1.1, or subject to *rmic*'s *-v1.1* switch in JDK 1.2 or later), the service won't be available at all.

3 Setting up RMI Proxying

The following basic steps are required to establish RMI Proxying:

- Modify client and server code to use the com.rmiproxy.ProxyNaming class
- Set the system property rmi.proxyHost correctly at the client, server and each RMI Proxy host
- Configure the security policy file correctly at the client and server
- Install, customize and run the RMIProxy program on proxy hosts.

For the sake of illustration in the following sections, assume we have the network setup as in Figure 1, and assume we are dealing with the following Remote interface:

```
public interface RemoteEcho extends Remote
{
   Object echo(Object object) throws RemoteException;
}
```

3.1 Use ProxyNaming instead of Naming

3.1.1 At the server

If your server is behind one or more firewalls, you must use the com.rmiproxy.ProxyNaming class instead of the java.rmi.Naming class.

Example code:

```
ProxyNaming.bind("rmi://localhost/" + RemoteEcho.class.getName());
```

If your server is not behind any firewalls, no coding changes are required. (If you want to defer this decision until runtime, use ProxyNaming: it functions identically to Naming if no RMI Proxy is configured.)

3.1.2 At the client

If your client is behind one or more firewalls, you must use the com.rmiproxy.ProxyNaming class instead of the java.rmi.Naming class. Also, you must ensure that the name supplied to ProxyNaming methods specifies the remote host as being the outermost server-side RMI Proxy rather than the server itself, if the server is behind a firewall.

Example code:

```
RemoteEcho echoObject =
(RemoteEcho)ProxyNaming.lookup("rmi://rmi.server.com/" +
                RemoteEcho.class.getName());
```

If your client is not behind any firewalls, no coding changes are required. (Again, if you want to defer this decision until runtime, use ProxyNaming: it functions identically to Naming if no RMI Proxy is configured.)

3.2 Configure the property rmi.proxyHost

The system property rmi.proxyHost must be set correctly at the client, the server and each intermediate RMI Proxy. It must specify the URL of the RMI Proxy (if any) for this host.

For our sample network setup:

Table 2: Sample rmi.proxyHost Settings

Client clienthost.clientlan0.client.com	rmi.proxyHost=rmi://proxy.clientlan0.client.com

Table 2: Sample rmi.proxyHost Settings

RMI Proxy proxy.clientlan0.client.com	rmi.proxyHost=rmi://proxy.client.com
RMI Proxy proxy.client.com	Property not set
Server serverhost.serverlan1.server.com	rmi.proxyHost=rmi://rmi.serverlan1.server.com
RMI Proxy rmi.serverlan1.server.com	rmi.proxyHost=rmi://rmi.server.com
RMI Proxy rmi.server.com	Property not set

Note that the outermost RMI proxies on the client and server sides do not set the property value at all.

The property can be set:

- In a configuration file called *rmiproxy.cfg*, located in the same directory that the client/server/proxy is being run from
- By a command line option: e.g. -Drmi.proxyHost=rmi://proxy

The system property rmi.nonProxyHosts can also be set. This must specify a "|"-separated list of URLs of hosts that need *not* be accessed via the RMI proxy. For example, clienthost.lan1.client.com may specify N1 and N2 as non-proxy hosts, since they are reachable on the local network without any firewalls intervening, so any RMI servers on N1 or N2 can be accessed in the standard way.

Thus a sample *rmiproxy.cfg* file for Client clienthost.clientlan0.client.com would look like:

```
rmi.proxyHost=rmi://proxy.clientlan0.client.com
// (or: rmi://proxy if in the same subdomain)
rmi.nonProxyHosts=rmi://N1.clientlan0.client.com|rmi://N2.clientlan0.client.com
// (or: rmi://N1|rmi://N2 if in the same subdomain)
```

If your client or server is not behind any firewalls, do not set its rmi.proxyHost property.

3.3 Configure multi-homing

If your RMI Proxy host is ‘multi-homed’, i.e. if it has more than one network interface and therefore more than one IP address, you need to inform the RMI Proxy. This is a common condition in well-configured proxy setups.

Because of restrictions in Java RMI, you must choose an IP address for your multi-homed proxy which is accessible to all clients both inside and outside the firewall, and configure the firewall

accordingly. The RMI Proxy is still free to use whichever network interface it likes according to the rules of TCP/IP, but the RMI stubs it exports will connect only via the chosen IP address.

To enable multi-homing in the RMI Proxy, set the java.rmi.server.hostname property at the RMI Proxy to the IP address chosen above, by specifying it in the RMI Proxy command line, as follows:

```
-Djava.rmi.server.hostname=ipAddress
```

You may also need to set this property at any clients or servers which use RMI Proxy host, if they are also multi-homed, if the IP addresses are in different subnets. This is not an RMI Proxy issue, it is a Java RMI issue common to all multi-homed RMI servers and clients.

3.4 Configuring security policy for clients and servers

RMI clients and servers require a security policy file if a security manager is present. This is *not* the file *rmiproxy.policy* described below, it is an application-specific file. To use the RMI Proxy, clients and servers must have the following PropertyPermission, SocketPermission, and FilePermission entries:

```
/* Permit the client or server to read the rmi.proxyHosts &c properties. */
permission java.util.PropertyPermission "rmi.*", "read";

/* Permit the client or server to connect to, and accept connections from,
   the RMI Proxy.
   REPLACE proxyhost WITH the host.domain of the nearest RMI Proxy,
   or a wildcard setting */
permission java.net.SocketPermission "proxyhost:1099", "accept, connect";

/* Permit the proxy to read its configuration files &c.
   Note that no write access is granted.*/
permission java.io.FilePermission ".${/}-", "read";
```

Install, Configure & Run the RMIProxy program

3.5 Installation

On a proxy host machine, install the ProxyBoot.jar, ProxyMain.jar, and RMIProxy.policy files in a directory of your choosing. This host must also have an installation of Java 2 Standard Edition JDK 1.3 or later. JDK 1.3 is available from the Java Platform Standard Edition home page.

3.6 Configuration—firewall configuration

The firewall controlling the RMI Proxy host must permit it to:

- accept *incoming and outgoing* TCP connections via port 1099 (the port allocated by the IANA to the Java RMI Registry)

- create inward connections to any port via any local port
- create outward connections to port 1099 via any local port
- create whatever inward or outward connections, via any local port, are required for codebase purposes, e.g. port 80 on your codebase hosts.

3.7 Configuration—security policy for the RMI Proxy

Before running the RMI Proxy program, its security policy file must be customized. The RMI Proxy's security is completely specified in its own policy file, which must be present at each RMI proxy host. A default policy file called *rmiproxy.policy* is shipped with the RMI Proxy. Ensure it is present in the directory that the proxy will be run from. (You can also define any number of your own policy files with different names, and then configure the proxy to run using a selected file with the -Djava.security.policy command line option.)

The security policy file consists of:

- one or more *global permissions blocks*, of which one specifies all the permissions required for the RMI Proxy to execute at all
- zero or more *codebase-specific blocks*.

Any of these can specify permissions to be granted. In the case of code-base-specofic blocks, they specify permissions granted to the associated codebase. (A *codebase* is a Java code source, typically an HTTP URL, representing a distinct application. Usually a codebase is a Java JAR file, which may or may not be *signed* by the supplier in the X.509 sense.)

3.7.1 RMI Proxy global permissions

The global permissions must include the standard contents of ${JAVA_HOME}/lib/security/java.policy *plus* at least the following:

```
grant {

/* Required to allow the proxy to read the rmi.proxyHosts &c properties.
   Servers & clients will also need this permission if running under a security manager.*/
permission java.util.PropertyPermission "rmi.*", "read";

/* Required by the proxy for internal reasons.*/
permission java.lang.RuntimePermission "accessClassInPackage.sun.rmi.server";

/* Required by the proxy for internal reasons.*/
permission java.lang.RuntimePermission "accessClassInPackage.sun.rmi.transport";

/* Required by the proxy for internal reasons.*/
permission java.lang.RuntimePermission "getProtectionDomain";
```

```
/* Permit the proxy to connect toanywhere on port 80 (HTTP) for codebase purposes.
   Revise this as required. */
permission java.net.SocketPermission "*:80", "connect";

/* Permit the proxy to connect to, and accept connections from,
   anywhere on port 1099, the RMI Proxy port.
   Change this wildcard setting as appropriate. */
permission java.net.SocketPermission "*:1099", "accept, connect";

/* Permit the proxy to listen to port 1099. */
permission java.net.SocketPermission "localhost:1099", "listen";

/* Permit the proxy to connect to, and accept connections on
   any port inside its domain.
   Change the domain setting as appropriate. */
permission java.net.SocketPermission "*.rmiproxy.com", "accept, connect";

/* Permit the proxy to read its configuration files &c.
   Note that no write access is required or granted.*/
permission java.io.FilePermission ".${/}-", "read";

/* Permit the proxy to use its own remote interfaces.
   DO NOT change the domain settings! */
permission com.rmiproxy.security.FirewallPermission "access.com.rmiproxy.*", "*";
permission com.rmiproxy.security.FirewallPermission "bind.com.rmiproxy.*", "*";
permission com.rmiproxy.security.FirewallPermission "execute.com.rmiproxy.*", "*";
permission com.rmiproxy.security.FirewallPermission "lookup.com.rmiproxy.*", "*";

};
```

3.7.2 RMI Proxy client permissions

The RMI Proxy client permissions control access to remote classes, methods, and registry bindings. These are described in the section ‘Security in the RMI Proxy environment’**,** where an example policy file is also provided. Client permissions can be placed into further global grant blocks, or into code-base-specific grant blocks. The former is simpler; the latter is more powerful, as it allows you to assign different sets of client permissions to different codebases.

3.8 Execution of the proxy program

Run the RMI Proxy with the following command-line:

```
java -Xbootclasspath/p:ProxyBoot.jar -jar RMIProxy.jar
```

or, if this is an ‘inner’ proxy which has a ‘next outer’ proxy:

```
java -Xbootclasspath/p:ProxyBoot.jar -jar RMIProxy.jar -Drmi.proxyHost=URL
```

where URL is of the form rmi://host,[1] and specifies the host of the next outer RMI Proxy. Alternatively, the system property rmi.proxyHost can be defined in a configuration file called *rmiproxy.cfg* placed in the current directory.

The -Djava.security.policy command line option can also be used to specify the name of the security policy file to be used, if different from *rmiproxy.policy*:

```
java -Xbootclasspath/p:ProxyBoot.jar -jar RMIProxy.jar -Drmi.proxyHost=URL
-Djava.security.policy=policy
```

If this is an 'inner' proxy and has an HTTP proxy, you should also specify -Dhttp.proxyHost=host, and -Dhttp.proxyPort=port if non-standard (other than 80), where *host* and *port* define the HTTP proxy host and port. This allows access to external codebases.

4 Security in the RMI Proxy environment

4.1 General

The first line of security for the RMI Proxy is the firewall and its configuration. The second line of security is the RMI Proxy security configuration. By default, nothing is permitted, satisfying the primary rule of security.

Nothing but the RMI protocol (JRMP) can penetrate the RMI Proxy. (Have your net-admin try it with Telnet, FTP, or HTTP, watching the other side of the proxy.)

Regardless of what the firewall permits, the RMI Proxy cannot:

- listen at any local port
- accept a connection from any non-local port
- connect to any non-local port

unless a matching SocketPermission is granted in its security policy file. (This is nothing special in the RMI Proxy, it's just how java.net.SocketPermission works for any Java 2 program which runs under a security manager.)

No RMI call through the RMI Proxy via any port (even when permitted by a SocketPermission) can proceed unless a matching FirewallPermission is granted in its security policy file. FirewallPermissions are granted to tuples of {codebase, action, interface/method name, client IP address}, so extremely fine-grained control is possible.

4.2 Permissions

The RMI Proxy provides fine-grained access control over remote classes, remote methods, and registry bindings, intersected with client host address and codebase URL. This takes the form of zero or

1. The rmi:// portion cannot be omitted in the current release.

more com.rmiproxy.security.FirewallPermission entries. Each such entry associates a requesting *hostname* (or IP address) with an *action*, for various actions executed by the proxy. If the host doesn't have the required permission to execute the action, it incurs an RMI AccessException and nothing propagates through the proxy.

4.2.1 Accessing remote interfaces

In order for a remote class to be *accessible* through an RMI Proxy (as a parameter or result of a remote method invocation), the proxy must have an access permission matching the client host and the class name.

Example: to access the remote class com.rmiproxy.test.RemoteEcho being sent by or returned to a client, the proxy must have a permission which matches:

```
com.rmiproxy.security.FirewallPermission
"clienthost","access.com.rmiproxy.test.RemoteEcho";
```

The RMI Proxy must set up internal proxying for each new remote object passing through it in either direction, so this action is access-controlled. Note that 'access' can occur in either direction:

- from the client to beyond the proxy in the case of a remote method parameter
- from beyond the proxy to the client in the case of a remote result.

The permission is checked during every remote method call.

4.2.2 Executing remote methods

In order for a client to *execute* a remote method via an RMI Proxy, the proxy must have an execute permission matching the client host and the relevant class and method name, expressed as class.method.

Example: for a client to execute the remote method com.rmiproxy.test.RemoteEcho.echo, the proxy must have a permission which matches:

```
rmiproxy.security.FirewallPermission
"execute.com.rmiproxy.test.RemoteEcho.echo","clienthost";
```

4.2.3 Lookup

In order for a client to *lookup* a name in the RMI Proxy Registry, the proxy must have a lookup permission matching the client host and the name being looked up.

Example: for a client to lookup the name 'EchoServer' in the proxy registry, the proxy must have a permission which matches:

```
com.rmiproxy.security.FirewallPermission "lookup.EchoServer","clienthost";
```

4.2.4 Bindings

In order for a requesting host to *bind*, *rebind*, or *unbind* a name into the RMI Proxy Registry, the proxy must have a bind permission matching the requesting host and the name being bound.

Example: To bind , rebind, or unbind an instance of com.rmiproxy.test.RemoteEcho as 'Echo-Server', the proxy must have a permission which matches:

```
com.rmiproxy.security.FirewallPermission "bind.EchoServer","requestingHost";
```

It must also have an access permission for each of the associated remote interfaces associated with the object being bound or rebound.

4.2.5 Wildcards in permissions

In all cases, the hostname part of the permission (the *target*) may *begin* with ".*", or consist entirely of "*", in the usual wildcard sense.

In all cases, the *action* part of the permission may *end* with ".*", or consist entirely of "*", in the usual wildcard way.

In this way, access to entire packages of classes can be granted to entire subdomains of the Internet.

4.3 Codebase

The RMI Proxy permissions can in turn can be partitioned into sets for distinct codebases, or they can be global permissions in a "grant {" block. The RMI Proxy always checks a permission in the access-control context of the class concerned, so it is possible to entirely segregate proxy permissions for one codebase from those for another.

4.4 Hostnames in permissions

In the nature of things, an RMI Proxy which has its own downstream proxy between itself and the Internet receives requests from the downstream proxy.

Similarly, an RMI Proxy at the junction with the internet may receive requests from other RMI proxies at the junction with the Internet, proxying different domains.

In both circumstances, the hostname associated with the request is that of the requesting proxy, not that of the original client. This must be taken into account when configuring hostnames into permissions in the RMI Proxy Security Policy. The proxy doesn't see the ultimate client's hostname, it sees the hostname of the RMI Proxy if any nearest to itself in the proxy chain between itself and the ultimate client.

In bind permissions, the 'client' requesting the action may be an RMI server. Don't get confused by this. In the context of RMI Proxy permissions, 'client' means the host which is requesting the action, not the ultimate role of that host in RMI interactions.

4.5 Example RMI Proxy policy file

This example shows parts of a policy file for the RMI Proxy. The required global block described under Section 4.1 is omitted from this example. The example relates to the RMI Proxy Test Echo server.

```
grant codebase "http://codebase.rmiproxy.com/beta" {

   /* Grant *.rmiproxy.com permission for all bindings in the proxy */
   permission com.rmiproxy.security.FirewallPermission "bind.*","*.rmiproxy.com";

   /* Grant the entire Internet permission to lookup this proxy registry */
   permission com.rmiproxy.security.FirewallPermission "lookup","*";

   /* Grant the entire Internet permission to access interfaces
   from com.rmiproxy.test.* */
   permission com.rmiproxy.security.FirewallPermission "access.com.rmiproxy.test.*","*";

   /* Grant the entire Internet permission to execute remote methods in
   com.rmiproxy.test.* */
   permission com.rmiproxy.security.FirewallPermission
"execute.com.rmiproxy.test.*","*";

}
```

For further information on Java Security see The Java Security Home Page.

5 Code Example

A simple remote echo server:

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

import com.rmiproxy.*;

public interface RemoteEcho extends Remote
{
   Object echo(Object object) throws RemoteException;
}

// Remote Server implementation:
class RemoteEchoServer extends UnicastRemoteObject implements RemoteEcho
{
```

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

    public Object echo(Object object) throws RemoteException
     {
        return object;
    }

     public static void main(String[] args) throws Exception
    {
        RemoteEchoServer server = new RemoteEchoServer();
        ProxyNaming.rebind(RemoteEcho.class.getName(), server);
    }
}

// Client implementation:
class RemoteEchoClient
{
    public static void main(String[] args) throws Exception
    {
        String rmihost = "rmi://rmi.server.com/";
        RemoteEcho echoServer = (RemoteEcho)ProxyNaming.lookup
            (rmihost + RemoteEcho.class.getName());
        System.out.println(echoServer.echo("o che bon eccho"));
    }
}
```