The RMI Proxy

Esmond Pitt and Neil Belford

WHITE PAPER

1 Preface

1.1 INTRODUCTION

This White Paper describes the RMI Proxy, a standard approach to enabling and controlling Java RMI/JRMP traffic through network firewalls, allowing RMI applications to traverse the Internet in a controlled manner without the limitations and performance issues of the present RMI/HTTP tunnelling approach.

The RMI Proxy is a hitherto-missing "piece of the puzzle" of the Sun Java/RMI offering. By overcoming and avoiding the problems inherent in the present RMI/HTTP tunnelling solution, the RMI Proxy makes deployment of RMI applications over the Internet not only genuinely feasible but actually attractive to developers and network administrators.

1.2 AUTHORS

Esmond Pitt, BA MIEEE FACS: *pitte@acm.org*
Neil Belford, MSC MIEEE: *n.belford@computer.org*

1.3 AUTHOR AFFILIATION

Telekinesis Pty Ltd, Box 82, 85 Grattan St, Carlton 3053, Victoria, Australia

1.4 PRODUCT

The RMI Proxy product described in this White Paper is available from Telekinesis on commercial terms.

2 Definition of Terms

We begin this discussion by defining a number of terms.

2.1 INTERNET

It is not exactly necessary to define the Internet here. However, for the purposes of this discussion it is important to note that it provides a number of "well-known" application-level services:

- DNS—Domain Name Service
- Telnet—remote terminal sessions
- FTP—File Transfer Protocol
- SMTP—Simple Mail Transfer Protocol
- HTTP—HyperText Transfer Protocol, used for the World Wide Web (WWW)

All these services are provided over a lower-level service called TCP—Transmission Control Protocol, which in turn is implemented over an even lower-level protocol called IP—Internet Protocol.

The Internet also supports arbitrary application protocols defined by, and only understood by, both ends of any custom computer software; generally, these are provided over TCP mentioned above.

2.2 JAVA

Java is a programming system introduced by Sun Microsystems in 1995. Java is the *lingua franca* of the WWW. The familiar *browser applets* are implemented in Java.

2.3 RMI

RMI—Remote Method Invocation—is a major component of Java, enabling Java programs in different computers to communicate easily and naturally. Java RMI is a critical component of the Enterprise Java Bean architecture underlying the Java Enterprise Edition, and the foundation of the Jini network technology.

2.4 JRMP

The data protocol by which RMI objects communicate is known as JRMP—Java Remote Method Protocol. RMI is in fact capable of communicating over other protocols, including (at the time of writing) CORBA/IIOP.

2.5 FIREWALLS

Purpose. In order to prevent office-wide local area networks (LANs) becoming part of the global Internet, corporations normally place a *firewall* between their LANs and the Internet proper.

Like a physical firewall, an Internet firewall provides a substantial level of security. It is the function of the firewall to prevent all except authorized communications between the Internet and the inner LAN.

Specifically, firewalls generally block all communications except the well-known ones (DNS, HTTP, FTP, SMTP, and generally excluding Telnet). This means that arbitrary application protocols are purposely blocked by the firewalls.

Firewall configurations are under the control of network administrators. In theory, network administrators can be persuaded to *open* certain *ports* for application-specific purposes, but they are generally rather difficult to convince about this (as they should be), as there are major security considerations involved.

Structure. Firewalls are generally composed of two parts: a *transport firewall*, which is a hardware component, and one or more *application firewalls*, which are software programs. This is illustrated in Figure 1 on page 3.

Generally, a transport firewall is configured to permit outward communications originating from a trusted application firewall; applications such as Web browsers inside the firewall are configured to communicate via the application firewalls or *proxies* rather than directly with the desired outside computer system. This tech-

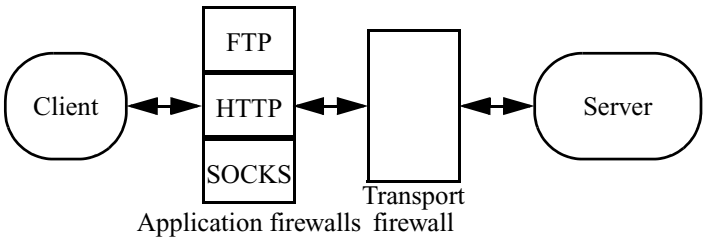

FIGURE 1. Application and Transport firewalls

nique is known as *proxying*. Proxies are generally provided for DNS, HTTP and SMTP, and sometimes for FTP.

2.6 HTTP TUNNELLING

Most application-level protocols which attempt to solve the firewall issue do so by the technique of *HTTP tunnelling*, in which the communications masquerade as HTTP requests and responses. This is rather like enclosing a sealed addressed envelope inside another sealed addressed envelope, with the understanding that the inner envelope is to be posted by the recipient of the outer envelope.

2.7 SOCKS

SOCKS is an authentication protocol use to traverse client-side firewalls. A SOCKS server is sited at the firewall, configured to accept connections only from certain clients within the firewall, and possibly only on certain ports. A SOCKS client establishes an authenticated SOCKS session with the SOCKS server, which establishes the client's real desired connection with some host/port on the Internet.

Notwithstanding certain statements in the RMI Frequently Asked Questions (FAQ) list, there is no such thing as server-side SOCKS.

2.8 CORBA

This is the acronym for Common Object Request Broker Architecture. This is a large framework, into which an application can be written, supporting connection-based communications over networks.

2.9 GIOP

The data protocol by which CORBA objects communicate is known as GIOP—General Inter-ORB Protocol. The specialization of this to TCP/IP is known as IIOP—Internet Inter-ORB Protocol.

2.10 THE GIOP PROXY

An architecture has been proposed for GIOP to enable CORBA applications to communicate across one or more firewalls. One of the components of this architecture is an application firewall (proxy) for GIOP, the *GIOP Proxy*; another component is bi-directional GIOP. This architecture is described in the OMG document CORBA orbos/98-05-04, hereinafter the *GIOP Proxy document*, which has been accepted for inclusion in the CORBA 3.0 specification.

2.11 UPSTREAM AND DOWNSTREAM

In this document, *upstream* points to the client and *downstream* points to the server. This may seem back-to-front. It is deliberately employed here to coincide with the usage in the GIOP Proxy document.

3 RMI and Firewalls—Current Practice

This section describes how RMI-based systems currently deal with Internet firewalls.

3.1 CLIENTS

A typical client-side firewall configuration is shown in Figure 2 on page 4.

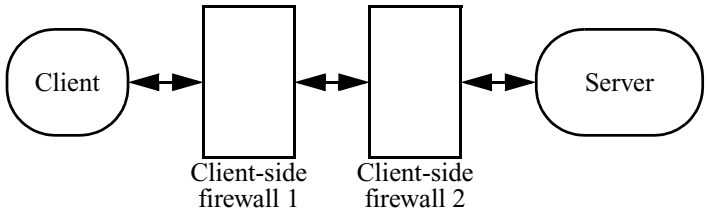

FIGURE 2. Client-side firewalls

Presently RMI clients can traverse client-side firewalls using one of two techniques.

HTTP Tunnelling. RMI clients can employ HTTP tunnelling through one or more client-side HTTP proxy servers, provided the Java system property http.proxyHost is set at the client.

This technique suffers from several problems: some inherent in the technique, and some due to the current RMI specification and implementation.

1. Inefficiencies in the technique, arising (i) from the necessity to pack and unpack RMI-protocol data into and from HTTP-protocol data; (ii) from the redirection step at the ultimate HTTP server into the real RMI server; and (iii) from excess TCP connection formation: each HTTP proxy in the proxy chain must initiate a new TCP connection—to the next proxy or the eventual target—per RMI call, whereas a direct RMI client-server conversation conserves the TCP connection for consecutive RMI calls within a connection expiry period. Sun have stated that RMI/HTTP, when redirected via the *rmi-cgi* script, is up to 10 times as slow as direct RMI. This factor alone casts serious doubt on the viability of the technique.
2. RMI implementations to date employ HTTP tunnelling as a fall-back after a direct connection fails. Within the official API, RMI clients cannot be configured to use only HTTP tunnelling to specific servers; a direct connection must always be attempted first. This presently imposes a 15-second overhead on each new client connection.
3. Certain firewall vendors have chosen not to emit the network error conditions which trigger the RMI/HTTP fallback behaviour. This makes RMI's HTTP tunnelling dependent on the firewall vendor. In other words RMI/HTTP does not work over the entire Internet as it actually exists.
4. Callbacks to servers in the client host cannot be supported.

All in all, the HTTP tunnelling situation is not a happy one.

SOCKS. RMI clients can use SOCKS to get beyond client-side firewalls, SOCKS configuration permitting. This is automatic, provided the Java system properties

socksProxyHost and socksProxyPort are set at the client. At the present time SOCKS is the best available client-side solution, although like RMI/HTTP it does not support callbacks.

3.2 SERVERS

Presently RMI clients cannot traverse server-side firewalls *at all*, as illustrated in Figure 3 on page 5.

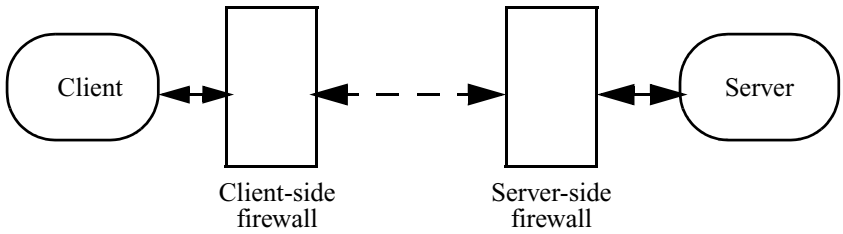

FIGURE 3. Client-side and Server-side firewalls

This means that RMI servers which are intended to be accessible to RMI clients anywhere on the Internet *cannot* be protected by any firewall mechanism: they *must* be logically outside any firewall on the server side (or in a firewall-designated 'DMZ', which amounts to the same thing).

3.3 CALLBACKS

An RMI server passed as a parameter to a remote method call from a client to a server allows the server to *call-back* an object at the client. This object is simply an RMI server located in the client JVM. Therefore, if the client is behind a firewall, the technique cannot work, for the same reasons as in the previous section.

Neither HTTP tunnelling nor SOCKS can solve the callback problem.

Prior to JDK 1.2.2 it was possible to solve this problem, on small scales, by forcing use of RMI's *multiplexing protocol*. This technique re-uses a single physical TCP connection for multiple virtual RMI connections. In JDK 1.2.2, Sun removed RMI's ability to initiate this protocol, which was only intended to support an early Netscape browser using JDK 1.02.[1]

The multiplex solution only applied to RMI/JRMP. The multiplex protocol is layered on top of JRMP, and hence cannot support RMI/IIOP.

4 The RMI Proxy

We have seen that application firewalls—proxies—exist for the HTTP, FTP, DNS, and SMTP application protocols, and that a GIOP Proxy has been accepted for CORBA 3.0.

This paper describes the *RMI Proxy*—an RMI Application Firewall for the RMI/JRMP applicationn protocol.

In thefollowing sections we describe the objectives of the RMI Proxy and its characteristics.

1. RMI servers still respond to the multiplex protocol, to support legacy clients.

4.1 OBJECTIVES

The objectives of the RMI Proxy are as follows:

1. Eliminate the security nightmare of RMI over HTTP for network administrators, by providing a controlled means of importing and exporting RMI services, via an application firewall which supports only the RMI/JRMP protocol. *No other solution to this problem presently exists.*
2. Provide a major reduction in overhead when traversing firewalls as compared to HTTP tunnelling.
3. Allow controlled access to RMI servers behind firewalls (server-side firewalls). *No other solution to this problem presently exists.*
4. Allow controlled access to client-side callbacks behind firewalls. *No other solution to this problem presently exists.*
5. Allow an RMI client behind a firewall controlled access to an RMI server outside it. *The only existing solutions to this problem are SOCKS or HTTP tunnelling.*
6. Preserve all possible properties of RMI; justify the exclusion of any RMI feature.
7. Require minimal Java coding changes in RMI clients and servers.
8. Take maximum advantage of existing Java classes, packages, and configuration and security features.

4.2 CHARACTERISTICS

The RMI Proxy is a Java application and API which permits controlled penetration of firewalls by approved RMI clients and servers.

The RMI Proxy provides an RMI-protocol proxying solution to replace RMI/HTTP tunnelling, with a large degree of access control available to the network administrator.

The RMI Proxy is able to:

- block access by non-JRMP protocols
- control write-access to the RMI Proxy Registry, according to the identity of the calling host
- permit or deny access and execution of remote methods by RMI clients, according to the identity of the client host
- permit or deny code mobility via the RMI codebase facility, in both directions

The RMI Proxy provides a major reduction in overhead over HTTP tunnelling. The duration of TCP connections is identical to that of direct-connected RMI. The overhead imposed by each RMI Proxy in the proxy chain is little more than that of an additional RMI indirection per proxy.

The RMI Proxy fully supports:

- server-side firewalls
- client-side firewalls
- client-side callbacks behind client-side firewalls

The RMI Proxy preserves all features of RMI except (i) Activation,[2] (ii) RMI/IIOP, and (iii) the accuracy of the return value of RemoteServer.getClientHost.[3]

2. Activatable stubs are proxied as unicast stubs.

Using the RMI Proxy requires only (i) changing use of the Naming class to ProxyNaming, and (ii) setting the rmi.proxyHost property at clients, servers, and intermediate proxy hosts.

The RMI Proxy uses the existing Java security-policy mechanism to control its configuration, and uses the existing java.rmi.AccessException to report access errors.

The RMI Proxy is a 100% Java solution.

The RMI Proxy firewall component implements a standard RMI Registry.

4.3 ACCESS CONTROL

Access control is provided by the familiar Java 2 security policy files, which control trans-firewall communications.

Access control is exerted against the intersection of the client hostname/IP address and the action being performed, by a special FirewallPermission class. The usual wildcard facilities are supported. Access control is exerted against the following *actions*:

- access: send or receive an object implementing a remote interface
- bind: bind, rebind, or unbind a name
- execute: execute a remote method
- lookup : lookup a name in the proxy registry

5 *Architecture*

The RMI Proxy consists of the following major components:

1. the RMI Proxy program, which executes within designated proxy machines
2. the ProxyNaming class, a modified Naming class, called by RMI clients and servers

5.1 RMI PROXY

The RMI Proxy consists of the following sub-components:

1. The Proxy Registry: a normal RMI Registry which is subjected to access control rules by the RMIProxy server implementation.
2. An RMI protocol engine which mediates between downstream RMI servers and upstream RMI clients, performing protocol validation and access control in the process.

The system relies on two elementary facts of RMI: (i) a remote reference can only be obtained via a parameter or result of a remote method call, and (ii) the initial remote reference is bootstrapped via a naming service—the RMI Registry.

5.2 PROXY NAMING

The ProxyNaming class exports exactly the same API as the standard RMI Naming class.

5.3 CLIENT-SIDE API

When an RMI client obtains a named reference, it normally uses the Naming.lookup method, quoting an RMI URL of the form:

3. HTTP tunnelling also suffers from the latter problem. The utility of the RemoteServer.getClientHost feature through routers and firewalls is dubious, given the widespread use of Network Address translation (NAT).

rmi://host/name

where host names a host running the RMI Registry, and name is the bound name of a remote object in that registry.

In the RMI Proxy API, the the client uses ProxyNaming instead of Naming. If a client-side RMI Proxy is present, the system property rmi.proxyHost specifies its location as an RMI URL. If this property is set, the client is running in an RMI Proxy environment, so the lookup is delegated to the RMI Proxy, which has a less restricted view of the outside world than the original client. If in turn there is another proxy for that proxy, the lookup is re-delegated, and so on until the final proxy is reached at the outermost client-side firewall, at which point the standard Naming.lookup is executed, as it would have been at the original client if no proxy property was set there.

This lookup either succeeds or fails.

If it *fails*, an RMI exception is propagated all the way back to the original client, and nothing remarkable has occurred—the remote object was not found.

If the lookup *succeeds*, the stub obtained (the downstream stub) is access-checked at the RMI Proxy. This may fail with a java.rmi.AccessException; otherwise a stub which refers to the RMI Proxy host is returned to the client (the upstream stub). The RMI Proxy maintains an association between upstream and downstream stubs, and forwards RMI calls from the one to the other, subject to access control.

From the point of view of the client, the existence of the entire RMI Proxy is invisible. All that the client knows is that it asked a naming service for a remote reference and it obtained one; the remote reference is of the expected type; and the reference works—it communicates with the remote object.

5.4 SERVER-SIDE API

1. An RMI server accessible to the Internet is bound with the ProxyNaming.bind method instead of Naming.bind or Registry.bind, quoting an RMI URL of the form

   rmi://host/name

   where host names a host running the RMI Registry, and name is the bound name of a remote object in that registry.[4]

   By the normal rules of RMI, an RMI server can only be bound to a name in the RMI Registry by a process running in the same host machine as the Registry. The RMI Proxy modifies this constraint, subject to access control at the proxy.

   The system property rmi.proxyHost names an accessible host where an RMI Proxy is running. If this property is set, the JVM is running in an RMI Proxy environment, so the ProxyNaming.bind operation is executed in *both* the local registry *and* in the RMIProxy. If in turn there is another proxy for that proxy, the bind is re-delegated, and so on until the final proxy is reached. In the original host and in each proxy including the last, the standard Naming.bind is also executed by ProxyNaming.bind . This causes the binding to occur in all the registries executing in the proxy host chain.

   The server is now bound in the local registry and the registries in the proxy host chain. A lookup operation to any of the proxy registries returns an upstream stub referring to the RMI Proxy, which forwards to the downstream stub after access control, exactly as in the client-side case (discussed above under "Client-side API" on page 7), eventually reaching the original RMI server.

4. As in Naming.bind, the hostname part of the URL can effectively only be *localhost* or another name for the local host.

2. Instead of calling Naming.unbind or Registry.unbind, a proxied RMI server is unbound by ProxyNaming.unbind. This action first calls Naming.unbind and then is delegated to the proxy chain, causing unbinding from all the registries in the proxy-chain.

In this way , any RMI server bound to a registry behind one of more firewalls, and any RMI server reachable from the initial server, is made accessible from the Internet, by being made visible at the nearest RMI Proxy, and so on recursively.

An RMI client can now lookup an RMI server at the outermost server-side proxy, and obtain a downstream stub which ultimately refers, via a delegation-chain of RMI Proxies, to a server inside the firewall: this server would not have been directly accessible to the client.

5.5 MULTIPLE PROXIES

If the client is behind one or more firewalls, its own client-side RMI Proxies will do their own delegations, so that the client receives a stub it can use. The final upstream stub points to the innermost client-side proxy, which points to the next, which eventually points to the outermost server-side proxy, which points inwards to the next, which ultimately points to the downstream server, as illustrated in Figure 4 on page 9.

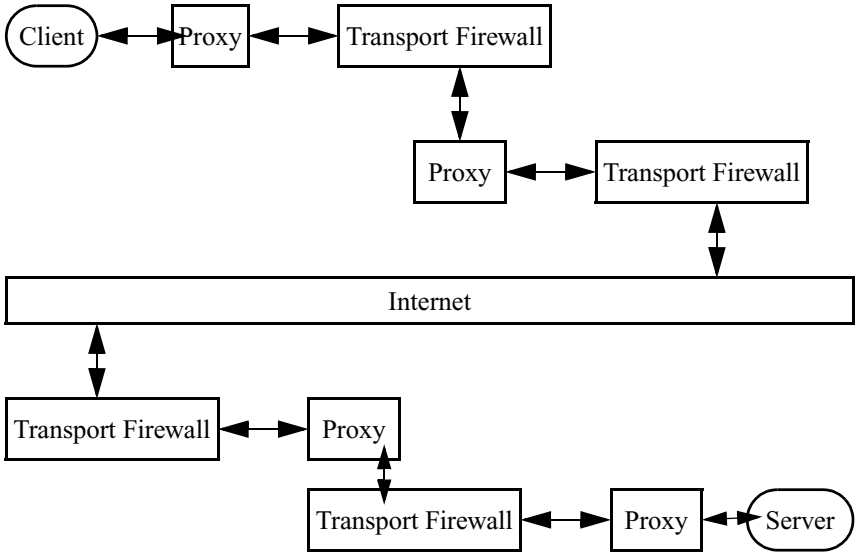

FIGURE 4. Multiple client-side and server-side proxies

5.6 DELEGATION

If no access failure occurs, an incoming RMI call from an upstream stub is *delegated* to the downstream stub. The result of, or exception thrown by, the downstream execution of the method are propagated upstream.

Further delegations between newly arrived downstream stubs and new upstream stubs may occur.. This process is a simple recursion on the original stub delegation process. In this way, over time, the complete remote object graph accessed by the client becomes replicated by delegation stubs in the upstream client, and by upstream/downstream stub associations in the RMI Proxy.

It is most important to understand this process. It is the foundation of the architecture. It is impossible for a remote stub to pass through the Access Controller without having an upstream proxy generated to replace it (except by subterfuge, inside a MarshalledObject. We took the view that these are intended to be opaque, and we treat them that way).

5.7 DGC

The RMI Proxy *believes all* RMI DGC (Distributed Garbage Collection) messages, and releases its own downstream stubs when DGC tells it that the corresponding upstream stubs are unreferenced. While this can lead to premature unexporting of the final downstream server, potentially causing problems for clients leading to a NoSuchObjectException being thrown, the behaviour is justified on two grounds: (a) it is essential, to avoid resource leaks, and (b) any RMI server is free to behave likewise: indeed, by default any RMI server does behave likewise unless local references to it are held. Therefore any RMI client is *already* exposed to the problem.

5.8 JRMP

By its construction, the RMI Proxy only accepts and transmits the JRMP protocol. It produces no output at one side unless the input at the other side is valid JRMP.

5.9 SUMMARY

It should be clear that the combination of Naming.bind delegation, ProxyNaming.lookup, access control, and stub delegation completely solves the problems of client- and server-side firewalls.

5.10 JNDI PROVIDER

A JNDI provider is also supplied which implements the proxy registry scheme, i.e. which stands in the same relation to the existing JNDI/RMI Registry provider as ProxyNaming does to Naming.

6 *Access Control*

Access control is enforced via a Java 2 policy file, which can be maintained by the *policytool*, at several levels. Every remote method call arriving from the upstream stub is access-checked against the local access-control policy, throwing a java.rmi.AccessException on any access-control failure. (As AccessException extends RemoteException, it can be thrown by any RMI call.)

Each access check is executed in the access-control context of the remote object concerned. This allows RMI Proxy security policy files to be partitioned into protection domains for different codebases.

6.1 CONNECTION CONTROL

Coarse-grained control over who can connect to the proxy at all is enforced at the level of hostname/IP address, by standard SocketPermissions with action=accept. These are evaluated by Java itself during the underlying ServerSocket.accept, in the RMI Proxy's own protection domain. In this way, effective access to the RMI Proxy for *any* purpose can be controlled per client host-address (in the global protection domain).

Similarly, coarse-grained control over which hosts the RMI Proxy can connect *to*, on behalf of clients, is provided by standard SocketPermissions with action=connect. These are evaluated by Java itself during construction of the underlying Socket, in the global protection domain. In this way, effective access *beyond* the RMI Proxy for *any* purpose can be controlled per client host-address in the global protection domain.

6.2 ACTION CONTROL

The RMI Proxy introduces a new FirewallPermission class. This serves two purposes: a server-side purpose and a client-side purpose.

On the server side, the FirewallPermission controls which hosts can perform the bind, rebind, and unbind actions, so that they can be restricted to hosts within the enclave protected by the RMI Proxy. Permitted hosts can register bindings for their own remote objects, with the assurance that the rest of the Internet is unable to preempt or replace those bindings. These actions require a FirewallPermission whose *target* matches the server host and whose *action* matches bind.*name*,where *name* is the name being bound, rebound, or unbound.

On the client side, the FirewallPermission controls:

1. which clients may access which remote interfaces, requiring a FirewallPermission whose target matches the client host and whose action matches access.*package.class*, as above. In this permission, "accessing" a remote interface means sending or receiving it via a parameter or result of a remote call.
2. which client may execute remote methods via the proxy, requiring a FirewallPermission whose target matches the client host and whose action matches execute.*package.classname.methodname*.

The FirewallPermission class allows intersections of codebases, hosts, remote interfaces, and bound names to be formed, so that, for example, only clients from certain Internet domains can access certain remote interfaces from certain codebases.

6.3 CODE MOBILITY

The Code Mobility property of Java RMI is preserved by the RMI Proxy: classes annotated with a codebase property by the sender can be dynamically loaded by the receiver as required. The RMI Proxy does not interfere with or add codebase annotations. The RMI Proxy runs under the control of a Java security manager, so code mobility is supported through it. The RMI Proxy does not grant any standard Java permissions to mobile code passing through it, so mobile classes with side-effects on serialization or deserialization may fail to pass through the RMI Proxy.

7 Specification

This section lists the Java specifications necessary to define the RMI Proxy.

These classes are all defined in the com.rmiproxy package.

7.1 PROXY NAMING

```
package com.rmiproxy;

// virtually extends java.rmi.Naming, or replaces it
public final class ProxyNaming
{
    // Same methods as java.rmi.Naming
}
```

7.2 PERMISSIONS

The permission class below supports Java 2 security policy file entries in the usual way.

```
package com.rmiproxy;
```

```
// Permission class for this package
// Targets include:
// access.package.class
// bind.name
// execute.package.class.method
// lookup.name
// Actions in this class as per target in java.net.SocketPermission
public class FirewallPermission extends java.security.BasicPermission {}
```

The operation of these permissions is defined in Table 1 on page 12. With this

TABLE 1. RMI Proxy Permissions

Target	Description
access.package.class	Permission for the host to access (send or receive) the specified package.class; evaluated for each remote interface implemented by the remote object being sent or received.
bind.name	Permission for the host to bind, rebind, or unbind a name to a remote object.
execute.package.class.method	Permission for the host to execute a remote method, evaluated for each remote interface which specifies the method which is implemented by the remote object concerned.
lookup.name	Permission for the host to lookup the name.

system of permissions, a four-way access control *régime* can be established: by codebase, interface name, bound name, and client host address.

7.3 PROTECTION DOMAINS

All permissions are evaluated in the *access control context* of the remote object concerned. This is a complex topic, but in effect it means that permissions for different codebases can be partitioned into different protection domains in the security policy file, as shown in the following example.

7.4 EXAMPLE POLICY FILE SEGMENT

```
/* RMI Proxy permissions for the RMI Proxy Beta demonstration codebase */

grant codebase “http://codebase.rmi.proxy.com/beta/demo” {

    /* Allow the entire Internet to access the demo classes*/
    permission com.rmiproxy.security.FirewallPermission ”access.com.rmi-
proxy.test.*”, “*”;

    /* Allow the entire Internet to lookup the registry*/
    permission com.rmiproxy.security.FirewallPermission ”lookup.*”, “*”;

    /* Allow the entire Internet to execute the demo*/
    permission com.rmiproxy.security.FirewallPermission ”execute.com.rmi-
proxy.test.*”, “*”;

    /* Allow only rmiproxy.com to bind/unbind the registry*/
    permission com.rmiproxy.security.FirewallPermission ”bind.*”, “*.rmi-
proxy.com”;

    }
```

7.5 SYSTEM PROPERTIES

rmi.proxyHost. This is the URL of the nearest RMI Proxy host for the current JVM, e.g. rmi://proxyhost. Corresponds to http.proxyHost and http.proxyPort for RMI.

rmi.nonProxyHosts. This is a "|"-separated list of zero or more URL for hosts to be contacted directly by the current JVM, rather than via the rmi.proxyHost. Corresponds to http.nonProxyHosts for RMI, and has the same format.

8 Additional features

1. The security mechanism obeys the cardinal security principle that nothing is permitted unless it is specifically enabled.
2. Only genuine RMI/JRMP protocol conversations can penetrate the proxy. Anything else, say a Telnet conversation, will lead to an RMI protocol error, which causes a RemoteException to be propagated to the client. (What a Telnet client may make of a RemoteException is another matter, but the client won't be able to establish a Telnet session with the target host via the RMI Proxy, because the proxy will only reject and never forward). Only RMI can induce output at the other side of an RMI Proxy.
3. The RMI Proxy executes under the control of a Java Security Manager, making it subject to all the existing rules of Java and the Java security policy at the proxy.
4. For the same reason, RMI code mobility ("codebase") through the proxy can be supported in both directions, either direction, or neither direction, depending on the security policy at the proxy, which can be controlled to the granularity of individual class/codebase pairs.
5. The existing Java security-policy mechanism is used to handle access control within the RMI Proxy. No new external subsystem is required.

9 Multi-homing

A 'multi-homed' host has more than one network interface controller (NIC) and therefore more than one IP address. The RMI Proxy supports multi-homed proxy hosts via the Java RMI java.rmi.server.hostname property.

This is not an ideal solution, because the property is static, global, and read only once in the lifetime of a JVM.

We have identified a better solution, whereby the local address of an accepted socket is remembered for the associated client, and used to define the local endpoint of any RMI stub originating at the proxy and returned to that client.

Our implementation of this solution is subject to enhancements or corrections in the JDK, or the RMI Custom Reference proposal JSR-078 which was proposed for JDK 1.4 but which has recently been vetoed for reconsideration by the Java Community Process.

Pending that implementation, the java.rmi.server.hostname property of a multi-homed host must be set to an NIC whose IP address is accessible to all intended clients. This remark applies to any RMI JVM, not just the JVM in which an RMI Proxy is executing, but it has particular application to the RMI Proxy because it is always expected to be subject to the control of an immediately adjacent firewall. In this case, the expected clients are probably both within and without the firewall, and the firewall must be configured appropriately

10 Limitations of the RMI Proxy

10.1 ACTIVATION

The RMI Proxy supports RMI Activation in a limited way.

The present implementation rewrites Activatable stubs as normal unicast (UnicastRemoteObject) stubs. The effect of this is that the "activatability" of the object is lost at the first proxy in the chain outwards from the server; all upstream clients, including other proxies in the server- and client-side chains and the eventual real client, are returned non-activatable stubs.

10.2 CONCEALED STUBS

The present implementation makes no proxying arrangements for "concealed" stubs—stubs wrapped inside MarshalledObjects—on the grounds that this is the explicit intention of the MarshalledObject class to render its contents opaque to intermediate Java programs.

10.3 SOCKET FACTORIES

Socket factories are not currently supported, mainly for reasons associated with standardization of port numbers. Essentially, each new socket factory introduced into a new RMI JVM requires a new port number. We do plan to support RMI/SSL via a standard port number, subject to resolution of several related issues.

When available, the RMI Proxy will be integrated with the RMI Security Extension proposed for JDK 1.4.

10.4 GETCLIENTHOST

The result returned by RemoteServer.getClientHost is the InetAddress of the nearest RMI Proxy in the proxy chain, never that of the real RMI client.[5] For this reason, the client host validated by the FirewallPermission may be the next RMI Proxy rather than the true client.

10.5 RMI/IIOP

The present implementation does not support RMI/IIOP stubs. This is a minor implementation issue whose resolution merely requires a small change in the javax.rmi.* stub classes.

10.6 INDIRECT REMOTE STUBS

The present implementation correctly processes remote stubs which are *reachable* from parameters or return values (i.e. present in the serialized object graph other than at the root). This requires us to ship the RMI Proxy with a small change in the sun.rmi.MarshalOutputStream class. This is required only at the RMI Proxy itself, not in the client/server API.

11 JDK Versioning Issues

1. The RMI Proxy itself requires Java 2 JDK 1.3 or later.
2. Clients and servers using the RMI Proxy can be of any Java version compatible with JDK 1.3 under serialization and which can use the RMI 1.1 or 1.2 stub protocols. This includes JDK 1.1, JDK 1.2.x, JDK 1.3, and as many future JDK versions as may comply with this specification.[6]
3. The RMI Proxy does not contribute to any Java-level or application-level protocol or class versioning issues between clients and servers, as it appears to

5. RMI/HTTP suffers from the same problem, for the same reason.

be completely transparent to both. It faithfully transmits whatever protocols, classes, and data it receives. It does not interpose itself as a codebase.

12 Security implications

The RMI Proxy is itself a security solution. The following are the only ways we have identified in which it relaxes existing Java security or compromises computer or network security generally (beyond its actual purpose, which is to provide access-controlled penetration of otherwise blocked firewalls).

1. The ProxyNaming class modifies the normal RMI rule that an RMI server can only be bound into an RMI Registry by a process running on the same host as the Registry. The ProxyNaming class, as an essential feature of the architecture, implements binding to a remote registry running as part of an RMI Proxy, subject to access control sat the proxy.
2. The result of the RemoteServer.getClientHost method is the InetAddress of the nearest RMI Proxy,[7] not that of the real upstream client. This behaviour is also exhibited by the present HTTP tunnelling technique. RMI servers which use client-host data to authenticate and authorize the client will be misled as to who the client really is and may permit security breaches to occur. Better authentication techniques than RemoteServer.getClientHost already exist (e.g. SSL) and are planned (e.g. the proposed RMI Security Extension).

13 Implementation considerations

1. The proxy registry uses the standard RMI Registry port number 1099.
2.

6. To support JDK 1.1, skeletons must be available for downloading via the same codebase as the corresponding stubs. Alternatively, stubs and skeletons can be located via the CLASSPATH of the RMI Proxy itself, although this configuration is insecure and not recommended.
7. RMI servers with clients using HTTP tunnelling already suffer from this problem.