The Jini Proxy

Esmond Pitt and Neil Belford

White Paper

1. Purpose

1.1 INTRODUCTION

This White Paper describes the Jini Proxy, a standard approach to enabling and controlling Jini RMI traffic through network firewalls, allowing Jini applications to traverse the Internet in a controlled manner. The Jini Proxy solves the present limitations of the multicast Discovery protocols and bypasses the limitations and performance issues of the present RMI/HTTP tunnelling approach.

The Jini Proxy is based on the RMI Proxy developed by this company, for which a separate White Paper is available. This document focusses on the Jini-specific extensions not covered in that paper.

By overcoming and avoiding the limitations inherent in the present Jini Lookup and Discovery implementations and in the RMI/HTTP tunnelling solution, the Jini Proxy makes deployment of Jini applications over the Internet not only genuinely feasible but actually attractive to developers and network administrators.

1.2 AUTHORS

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

1.3 AUTHOR AFFILIATION

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

1.4 PRODUCT

The 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.

Java is a programming system introduced by Sun Microsystems in 1995.

Jini is a networking architecture based the communications and code-mobility features of Java.

RMI—Remote Method Invocation—is a major component of Java, enabling Java programs in different computers to communicate easily and naturally. Java RMI is the foundation of the Jini networking architecture.

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.

2.1 FIREWALLS

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 certain 'well-known' ones, and impose access controls on the non-blocked protocols. 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 firewall*s, which are software programs. This is illustrated in Figure 1 on page 2.

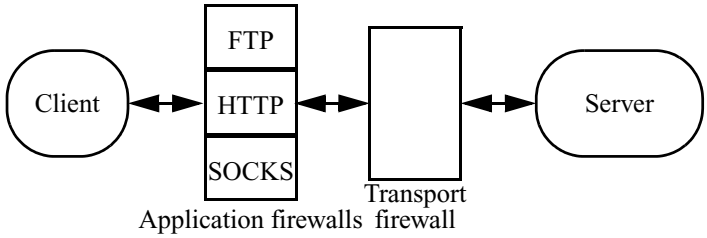

Figure 1. Application and Transport firewalls

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

2.2 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.

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 RMI Proxy White Paper, which in turn agrees with various external documents.

3. Jini and Firewalls—Current Practice

Jini services and clients initially use multicast IP to discover one or more 'Lookup Services', and then use point-to-point TCP/IP to lookup further services in the discovered lookup services. How services and clients communicate with each other once mutually discovered and looked-up is left unspecified by Jini, although geneally RMI/JRMP is found the simplest solution.

This scheme has several flaws in the presence of firewalls.

1. RMI/JRMP does not propagate through firewalls unless explicitly enabled at the firewalls and bridges or routers concerned.
2. IP multicasts are not propagated through firewalls unless explicitly enabled at the firewalls and bridges or routers concerned.
3. IP multicasts from a multi-homed hosts aren't necessarily sent out or received via all its network interfaces.

The effects of all this are as follows:

1. Jini services and clients cannot expect to communicate with each other via RMI through firewalls.
2. A standard Jini Lookup Service running in a multihomed host is only visible in the single subnet connected to that network interface.
3. Similarly, a standard DiscoveryService running in a multi-homed host will only discover Lookup Services accessible via one of its network interfaces.

In other words, Jini services and clients (a) won't be able to find each other through firewalls, and (b) even if they could, they couldn't expect to reliably communicate via RMI.

The Jini Proxy solves the first problem, and the RMI Proxy engine solves the second.

4. The Jini Proxy

In the following sections we describe the objectives of the Jini Proxy and its characteristics.

4.1 OBJECTIVES

The objectives of the Jini Proxy are as follows:

1. Provide the existing facilities of the RMI Proxy in the Jini environment, allowing Jini clients and services to be separated by firewalls.
2. Require minimal Java coding changes in Jini clients and services.

4.2 CHARACTERISTICS

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

The Jini Proxy is able to:

- control read- and write-access to its internal Jini Lookup Service, according to the identity of the calling host, the service interfaces concerned, and the codebase associated with them
- 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 Jini Proxy merely appears as an additional Lookup Service in the Jini environment.

Using the Jini Proxy requires *no* code changes to Jini clients or services, and no change to their operating environments such as special system properties or extra codebases or JAR files. The reason for this is that the Jini Proxy operates correctly as a compliant Jini Lookup Service via *all* its network interfaces (unlike present implemenations of *reggie*). It is therefore visible to (a) all neighbouring Jini services, who will register with it, and (b) all neighbouring Jini clients, who will look it up. This only requires that Jini clients and services be coded so as to use all available Lookup Services which support the required Jini groups, rather than just using, say, one.

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

The Jini Proxy is a 100% Java solution.

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
- execute: execute a remote method

- register: register a service in the Jini Proxy Lookup Service
- lookup : lookup a service in the Jini Proxy Lookup Service
- notify: receive a notification about a state change in the Jini Proxy Lookup Service.

5. Architecture

The Jini Proxy consists of the following major components:

1. the RMI Proxy engine, which executes within designated proxy machines.
2. A Jini Lookup Service called by RMI clients and servers which executes in the same JVM as the RMI Proxy engine.

5.1 SUMMARY

It should be clear that the combination of a multi-homed Jini Lookup Service integrated with access control and stub delegation completely solves the problems of client- and server-side firewalls.

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, and therefore by most Jini service calls—certainly by the ones of interest.)

Each access check is executed in the access-control context of the remote object concerned. This allows Jini/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 register action, so that it can be restricted if necessary to hosts within the enclave protected by the Jini Proxy. Permitted hosts can register their own remote objects, with the assurance that the rest of the Internet is unable to pre-empt or replace those registrations. These actions require a FirewallPermission whose *target* matches the server host and whose *action* matches register.*package.class*, evaluated for each remote interface *package.class* being registered. See also the execute permission described below.

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 clients may lookup remote objects in the proxy lookup service, requiring a FirewallPermission whose target matches the client host and whose action matches lookup.*package.classname*.
3. Which clients may receive notification callbacks about state changes in the proxy lookup service, requiring a FirewallPermission whose target matches the client host and whose action matches notify.*package.classname*. Note that this set will normally be similar to the set for the lookup permission, but the notify permission has been separated from the lookup permission to allow for greater flexibility.
4. 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 execute permission is also used for many standard Jini actions such as lease renewal, DiscoveryAdmin actions (service attribute changes), other administrative services, &c, as shown in Table 1.

The design of the FirewallPermission class allows intersections of codebases, hosts, and remote interfaces to be formed, so that, for example, only clients from certain Internet domains can access certain remote interfaces loaded from certain codebases. This provides an extreme degree of flexibility in configuring the security policy of the Jini Proyx.

6.3 CODE MOBILITY

The Jini Proxy runs under the control of a Java security manager, so the Code Mobility property of Java RMI is preserved by the Jini Proxy. Classes annotated with a codebase property by the sender can be dynamically loaded by the receiver as required. The Jini Proxy does not interfere with or add codebase annotations, apart from annotating its own classes with a codebase for reggie-dl.jar, an action which is under the user's control.

The Jini 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 Jini Proxy. The same rule should be observed by the user's security policy file for the Jini Proxy: no standard Java permissions should be granted to 'foreign' codebases, only FirewallPermissions. This ensures

that code dynamically loaded into the Jini Proxy, e.g. as callbacks, cannot cause a security breach.

7. Specification

The following Java specifications define the Jini Proxy, in addition to those which define the RMI Proxy, given in the RMI Proxy White Paper.

7.1 PERMISSIONS

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

```
package com.rmiproxy;

/**
* Permission class for the Jini Proxy.
*
* Targets include:
* access.package.class
* execute.package.class.method
* lookup.package.class
* notify.package.class
* register.package.class
*
* Actions: as per target in java.net.SocketPermission
*/
public class FirewallPermission extends java.security.BasicPermission {}
```

The operation of these permissions is defined in Table 1.

Table 1. Permissions in the Jini Proxy

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. This permission is also implicitly evaluated for all parameters and results of the Jini Proxy Lookup Service itself.
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. Not evaluated for actions of the Jini Proxy Lookup Service itself except as noted below. There are several specific execute permissions which affect access to services of the Jini Proxy Lookup Service itself. Without these permissions the corresponding operation fails. These are listed in the entries following.

(Sheet 1 of 2)

Table 1. Permissions in the Jini Proxy (continued)

Target	Description
execute.net.jini.admin.Administrable.getAdmin	Allows access to an administrative object for the Jini Proxy Lookup Service. The caller must also have access permission for all remote interfaces implemented by the administrative object returned by this method. The interfaces are not specified here, but the administrative object is the same as returned by *reggie*.
execute.net.jini.admin.JoinAdmin.* execute.net.jini.lookup.DiscoveryAdmin.*	Allow execution of the methods of the JoinAdmin and DiscoveryAdmin interfaces. For space reasons these permissions are shown using the wildcard syntax, but they may be granted individually if required.
execute.net.jini.core.lease.Lease.* execute.net.jini.core.lease.LeaseMap.*	Specific lease renewal and cancellation permissions are required. The reason for this is that remote LeaseRenewalServices may be used by services and clients, and may execute in different protection domains from clients and services: consideration must be given as to whether to grant these permissions to such domains.
execute.net.jini.core.lookup.ServiceRegistrar.getEntryClasses execute.net.jini.core.lookup.ServiceRegistrar.getFieldValues execute.net.jini.core.lookup.ServiceRegistrar.getEntryClasses execute.net.jini.core.lookup.ServiceRegistrar.getGroups execute.net.jini.core.lookup.ServiceRegistrar.getServiceTypes execute.net.jini.core.lookup.ServiceRegistrar.notify	The ServiceRegistrar methods are listed individually. The register and lookup methods have their own permissions and do not require an execute permission as well. ServiceRegistrar.getServiceID is not a remote method and so has no access control. ServiceRegistrar.getLocator is not genuinely a remote method so the same remark applies.
lookup.package.class	Permission for the host to lookup a Jini service. The permission is evaluated for each remote interface implemented by the result(s) which match the ServiceTemplate supplied.
notify.package.class	Permission for the host to receive notifications about registration changes concerning a Jini service. The permission is evaluated for each remote interface implemented by the result(s) which match the ServiceTemplate supplied, *at the time of executing the callback*. The permission is evaluated against the host address of the client which registered the notification callback. If permission to receive notifications about those interfaces has not been granted, the callback is simply suppressed, because any associated AccessException is incurred by the Jini Proxy Lookup Service, not by the client.
register.package.class	Permission for the host to register a remote object; evaluated for each remote interface implemented by the service being registered.

(Sheet 2 of 2)

With this system of permissions, an access control *régime* can be established along three independent axes: codebase, interface name, and client host address.

7.2 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.

7.3 EXAMPLE POLICY FILE SEGMENT

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

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

  /* Allow the entire Internet to access the demo classes */
  permission com.rmiproxy.security.FirewallPermission "access.com.rmiproxy.test.*", "*";

  /* Allow the entire Internet to lookup, lease, and receive notifications */
  permission com.rmiproxy.security.FirewallPermission "lookup.*", "*";
  permission com.rmiproxy.security.FirewallPermission "execute.net.jini.core.lease.*",
  "*";
  permission com.rmiproxy.security.FirewallPermission "notify.*", "*";


  /* Allow the entire Internet to execute the demo */
  permission com.rmiproxy.security.FirewallPermission "execute.com.rmiproxy.test.*",
  "*";

  /* Allow the entire Internet to execute ServiceRegistar methods */
  permission com.rmiproxy.security.FirewallPermission
  "execute.net.jini.lookup.ServiceRegistrar.*", "*";

  /* Allow only hosts in rmiproxy.com to register services */
  permission com.rmiproxy.security.FirewallPermission "register.*", "*.rmiproxy.com";

  /* Allow only hosts in rmiproxy.com to administrate the Jini Proxy Lookup Service */
  permission com.rmiproxy.security.FirewallPermission "execute.net.jini.admin.*",
  "*.rmiproxy.com";

  }
```

NOTE: this is not a complete policy file for the Jini Proxy. A complete policy file would include all the contents of policy.reggie, and specifically would include appropriate net.jini.discovery.DiscoveryPermission and java.net.SocketPermission entries.

8. Additional features

The Jini/RMI Proxy exhibits the following 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 Jini/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 Jini/RMI Proxy.

9. Multi-homing & Multicasting Issues

9.1 MULTIHOMING

A 'multi-homed' host has more than one network interface controller (NIC) and therefore more than one IP address.

Java RMI supports multi-homed proxy hosts via the Java RMI java.rmi.server.hostname property. The design of Java RMI assumes that either:

1. the 'default' network interface of a multi-homed proxy host is accessible to all clients, or
2. the value of the java.rmi.server.hostname property yields a hostname or IP address which is accessible to all clients.

This remark applies to any RMI JVM, not just the JVM in which a Jini/RMI Proxy is executing, but it has particular application to the Jini/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, i.e. to permit access to the default NIC or the one designated by java.rmi.server.hostname from both within and without.

(There is a third possibility: setting java.rmi.server.hostname to a hostname which resolves to different, accessible IP addresses in each subnet to which the host is directly connected (i.e. via each NIC). This is such a complicated solution that it is not further considered here.)

9.2 MULTICASTING

§3.3 relates to several further issues connected with multicasting.

1. A multicast group is an IP subnet, which therefore has a route and a local gateway, which, in many implementations is by default only *one* of the local network interfaces.[1] In these implementations (basically the Berkeley-derived ones), a specific route to the multicast subnet must be established for each interface other than the default.
2. There are multi-homed multicasting implementation problems within Jini 1.1: specifically an incorrect ordering of multicasting control events

1. See Stevens, W.R., *Unix Network Programming*, Prentice Hall PTR, 1998, §19.5 p. 497.

within both the LookupDiscoveryManager and the *reggie* mplementation of the Lookup Service. The latter fault is critical to this discussion. We have incorporated fixes to this problem in the version of *reggie* we supply to support the Jini Proxy, and we are advised that the same fixes will be folded into a future release of Jini.

3. By design, routers won't propagate group-membership events unless they are aware of a neighbouring group member. Routers become aware of neighbouring multicast group members via the Internet Group Management Protocol (IGMP): this is itself carried via multicast, and so is subject to the same issues as in the previous items

Without attention to all the above issues, a router neighbouring a non-default interface on a multi-homed host won't necessarily become aware of its group memberships, and therefore its group memberships won't be effective beyond the router.

10. Limitations of the Jini Proxy

10.1 ACTIVATION

The Jini 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 it is the explicit intention of the MarshalledObject class to render its contents opaque to intermediate Java programs. This is a security breach; however such stubs will not work across firewalls if configured properly, so it is really more a usability breach than a security risk.

10.3 SECURITY EXTENSION

The Jini Proxy will be integrated with the RMI Security Extension proposed in JSR 76, now the Jini 'Davis' project, when available.

10.4 RMI/IIOP

RMI/IIOP support is planned for a separate extension to the RMI Proxy, which will co-operate with the Jini Proxy when available.

11. JDK Versioning Issues

This section defines the interoperability of the Jini/RMI Proxy with various configurations of JDK at service and client.

1. The Jini Proxy itself requires Java 2 JDK 1.4. Beta versions running on JDK 1.3 will probably be made available.
2. Clients and servers using the Jini Proxy can be of any Java version compatible with JDK 1.4 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.[2]
3. The Jini Proxy does not contribute to any Java-level or application-level protocol or class versioning issues between clients and servers, as it appears to 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 Jini Proxy is itself a security solution. See the corresponding section of the RMI Proxy White Paper for a full discussion.

13. Implementation considerations

The Jini Proxy uses the same IP multicast addresses and port numbers as standard Jini, i.e. unicast port 4160 and the multicast groups 224.0.1.84 and 224.0.1.85. For RMI purposes it uses the RMI Registry port number 1099. These are the IP addresses and ports which must be 'open' to the proxy host as far as the transport firewall configuration is concerned.

The conditions specified in §9.1 must be met in the JVM running the Jini Proxy. The same condition that applies to the RMI Proxy.

For the reasons given in §9.2, you must ensure that the proxy host has a route to the multicast subnet (224/8, or 224.0.0.0 in the old subnet notation) via *all* of its network interfaces. For Windows platforms this appears to be the default behaviour; for Unix-based platforms it most definitely is *not*.

2. 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.