$Date: 2003/02/09 19:57:24 $

Making Jini Federations and Firewalls Work

Geoffrey Arnold <garnold@jini.org>

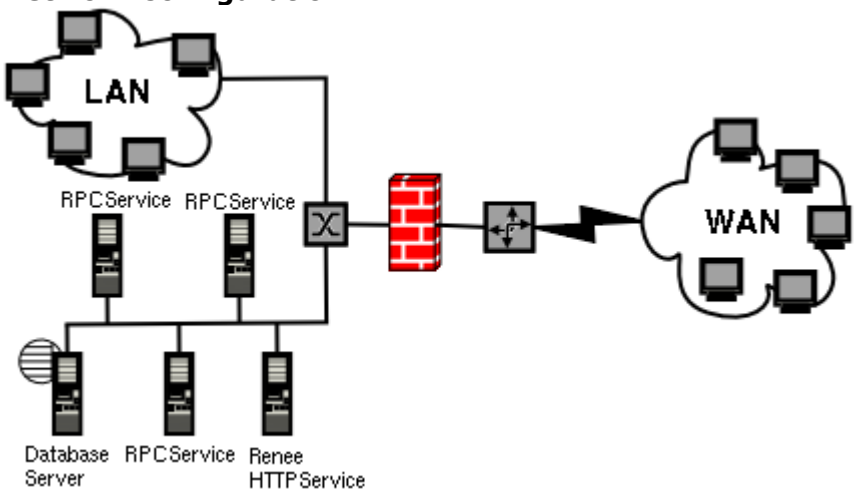
An important part of this complete HowTo project!

Introduction

In the text that follows, I will attempt to explain how we have a Jini federation working over the WAN through a firewall using NAT and split DNS. It should be noted that our target application is remote devices accessing Jini services through a firewall. Realizing that not all network setups are the same, some of the solutions presented will not apply to everyone, and as such this article should not be viewed as a tutorial but rather an informational article for the Community.

This document is broken up into the following sections:

Network Configuration



Our network configuration is somewhat typical of a SMI network setup (see diagram above). Our T1 line is connected to a router acting as a bridge. The router is connected directly to a Linux firewall with NAT support, which segments the connection into a LAN subnet and a DMZ subnet. It should be noted that access to the DMZ is still controlled by the firewall, and as such the DMZ is not completely accessible to the outside world. The firewall has been allocated a block of external IP addresses, and acts as the primary DNS and DHCP server for our internal network. External name resolution is provided by our ISP and administered through a less than adequate control panel. The table below describes how we have configured our network for "split DNS" (split DNS is similar to bind9 views -- thanks for pointing this out Kenji). Names and addresses have been changed to protect the innocent:

DNS Entry	External Resolution	Internal Resolution
app.jini.org	4.3.2.1	10.0.0.1
rpc1.jini.org	4.3.2.2	10.0.0.2
rpc2.jini.org	4.3.2.3	10.0.0.3
rpc3.jini.org	4.3.2.4	10.0.0.4

We have a federation of services running on a number of servers in the DMZ. One machine

(app.jini.org) houses the Jini lookup service (Renee), and an Activatable, Administrable, multi-directory HTTPService for dispensing codebases and related files (simple web pages, JNLP files, etc.). A separate server acts as a dedicated database server (not accessible to the outside world), while the other servers (rpc1.jini.org, rpc2.jini.org, rpc3.jini.org) in the DMZ each run an instance of our RPCService (load-balanced by Renee) and a legacy RPC application. Client applications are deployed using Java Web Start via the WAN (and LAN).

Our firewall allocates tcp ports 8000-8010 for Jini use (service exporting, callbacks) on all servers in the DMZ, as well as tcp port 80 (HTTPService on app.jini.org), tcp/udp port 4160 (Renee on app.jini.org), and tcp port 1098 (RMID on all servers in the DMZ). In the near future we plan to add a JavaSpaces service to the federation following similar port allocation semantics. The table below shows how NAT translates this scheme:

DNS Entry	Ports Forwarded
app.jini.org	tcp 80 (HTTPService) tcp 1098 (RMID) tcp/udp 4160 (Renee) tcp 8000 (Renee) tcp 8001 (HTTPService) **tcp 8000-8010 (open)**
rpc1.jini.org	tcp 1098 (RMID) tcp 8002 (RPCService) **tcp 8000-8010 (open)**
rpc2.jini.org	tcp 1098 (RMID) tcp 8002 (RPCService) **tcp 8000-8010 (open)**
rpc3.jini.org	tcp 1098 (RMID) tcp 8002 (RPCService) **tcp 8000-8010 (open)**

It is important to note that resolving addresses by name is key to making the federation work, due to the fact that services must communicate with each other on our internal network and with clients on the external network. Using IP addresses in stub references will not work both internally and externally. Therefore, because stubs default to using IP addresses in hostnames, it is important to specify the **java.rmi.server.hostname** system property in addition to the **java.rmi.server.codebase** property when starting the RMI daemon and Jini services. Therefore, when a stub references app.jini.org in our split DNS configuration, the name resolves to 10.0.0.1 internally and 4.3.2.1 on the WAN. The property can easily be inherited by all Activatable services on a given host by using the **-C-Djava.rmi.server.hostname** parameter on the RMID command line. The following is a sample script that demonstrates setting the property when starting the RMI daemon and the lookup service:

```
#!/bin/sh
#
. config.sh

rm -rf $JINI_HOME/log/rmid
$JAVA_HOME/bin/rmid \
  -J-Djava.security.policy=$JINI_HOME/policy/rmid.policy \
  -J-Djava.rmi.server.hostname=`hostname --fqdn` \
  -C-Djava.rmi.server.hostname=`hostname --fqdn` \
  -log $JINI_HOME/log/rmid &

rm -rf $JINI_HOME/log/renee
$JAVA_HOME/bin/java \
  -Djava.security.policy=$JINI_HOME/policy/renee.policy \
  -jar $JINI_HOME/lib/renee.jar \
  http://$HTTP_HOST:$HTTP_PORT/renee-dl.jar \
  $JINI_HOME/policy/renee.policy \
  $JINI_HOME/log/renee \
  public \
  -Dorg.jini.renee.port=8000 &
```

Intelligent Port Allocation

Specifying the port on which to export a Remote object is a trivial task. Both Activatable and UnicastRemoteObject objects have parameters in their constructors and ::exportObject() methods that allow you to indicate the port on which the object should be exported. See the following Java API documentation for more details:

http://java.sun.com/j2se/1.4/docs/api/java/rmi/activation/Activatable.html
http://java.sun.com/j2se/1.4/docs/api/java/rmi/server/UnicastRemoteObject.html

One can specify ports for RemoveEventListeners following the same conventions, however subclassing the following object automates the task of selecting a port within a given range:

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;

public abstract class RangeListener implements RemoteEventListener {

  public final static int MIN_PORT = 8000;
  public final static int MAX_PORT = 8010;


  public abstract void notify(RemoteEvent event)
    throws RemoteException;

  public int exportObject(int minPort,int maxPort)
    throws RemoteException {

    for (int port = minPort; port <= maxPort; port++) {
      try {
        UnicastRemoteObject.exportObject(this,port);
        return port;
      }
      catch (RemoteException e) {
        if (port == maxPort)
          throw new RemoteException("port range exceeded");
      }
    }

    throw new RemoteException("unknown error when exporting object");
  }

  public int exportObject()
    throws RemoteException {

    return this.exportObject(MIN_PORT,MAX_PORT);
  }
}
```

Client Deployment with Java Web Start

Java Web Start, Sun's contributed implementation of the JNLP specification, is by far the most simple and intuitive way to deploy Jini client applications across a WAN. JNLP allows for centralized administration of Jini applications, and allows us to better deploy updates and avoid application versioning conflicts.

Because of the security permissions needed by Jini clients, it is a requirement that Jini applications deployed by Java Web Start request "all-permissions" in their JNLP configuration file. This, in turn, requires that every archive that makes up an application codebase be signed. The Java Web Start Developers Guide has an excellent tutorial on self-signing Java archives for development purposes.

JNLP configuration files are quite simple to write. The following is an example of our JNLP configuration file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP File Sample Jini Application -->
<jnlp spec="1.0+"
      codebase="http://app.jini.org/application-core.jar"
      href="http://app.jini.org/application.jnlp">
  <information>
    <title>Sample Jini Application</title>
    <vendor>Data-Basics, Inc.</vendor>
    <homepage href="http://www.databasics.com"/>
    <!-- description></description -->
    <!-- description kind="short"></description -->
    <!-- icon href=""/ -->
    <!-- icon kind="splash" href=""/ -->
    <!-- offline-allowed/ -->
  </information>
  <security>
    <all-permissions/>
  </security>
  <resources>
    <j2se version="1.4+"/>
    <property name="org.jini.app.jlus.host" value="app.jini.org"/>
    <property name="com.sun.jini.reggie.proxy.debug" value="true"/>
    <property name="net.jini.discovery.debug" value="true"/>
    <jar href="http://app.jini.org/application-core.jar"/>
    <jar href="http://app.jini.org/application-lib.jar"/>
    <jar href="http://app.jini.org/application-ext.jar"/>
  </resources>
  <application-desc main-class="org.jini.app.Application"/>
</jnlp>
```

As shown in the JNLP file above, the application codebase has been split into three Java archives (application-core.jar, application-lib.jar, application-ext.jar). This allows us to modify the core application (application-core.jar) without requiring clients to re-download large static libraries (application-lib.jar) or addon classes (application-ext.jar).

Handling Dual Firewalls

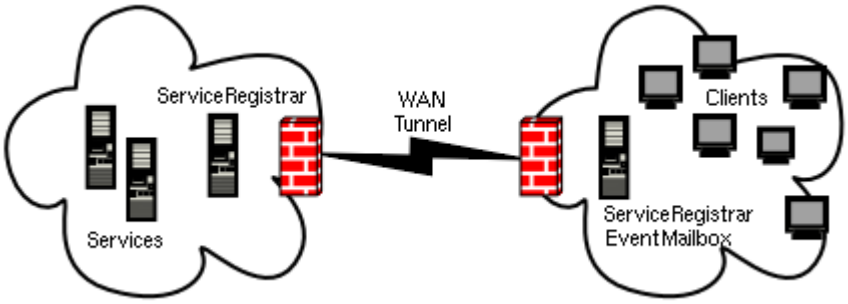
What happens when your Jini clients are also protected by firewalls? Fortunately, the requirements for Jini clients sitting behind firewalls are basically the same as those for Jini services, specifically:

- Clients must have static IP addresses that map to domain names
- Split DNS must be setup to recognize the internal/external IP addresses for the above domains
- The firewall must be setup to accept incoming requests for a given port range
- NAT must be setup to forward the ports within the range to all clients behind the firewall
- The **java.rmi.server.hostname** property must be specified for all clients who are listeners for remote events

Now suppose that you have an infinite (unknown) number of clients attempting to access a remote service over the WAN from behind a firewall. It is unrealistic to assume that the above requirements can be met, especially because of the strict DNS requirements. However, adding an EventMailbox service inside your client LAN can greatly help the situation by requiring only the EventMailbox host to have a DNS entry and a static internal/external IP address. The EventMailbox can either register with the remote lookup service, or with a lookup service running on the client LAN. All clients attempting to register RemoteEventListeners with services across the WAN should first request a listener from the EventMailbox, and then register the returned MailboxRegistration listener with the remote service. The client can then enable delivery from the EventMailbox by registering its own listener with the service, thereby bypassing the need to assign domain names and static IP addresses with split DNS to individual clients.

Want to make this setup even better? Add a lookup service tunnel between the ServiceRegistrar on your LAN and the ServiceRegistrar on the remote network. You can then add robustness to your federation by having multiple lookup services on each network, and using multicast

discovery in your clients. This configuration is depicted below:



Limitations

The setup described above does have some limitations. First, we do not have redundancy in our Jini lookup service, due to the fact that the JNLP file specifies the location of the registrar. However, due to the fact that clients only are required to contacted the registrar upon startup, the service should never become overloaded. Second, each client is limited to a finite number of event listeners, as specified by the RangeListener superclass. This limitation can be overcome by utilizing smarter RemoteEventListeners that can differentiate between RemoteEvents through the use of the *handback* object contained within the event.

$Id: index.html,v 1.7 2003/02/09 19:57:24 garnold Exp $

An important part of this complete HowTo project!