Enhancing Jini for Use Across Non-Multicastable Networks

Ahmed Al-Theneyan[ab] *Piyush Mehrotra*[c] *Mohammad Zubair*[ab]

[a]Computer Science Department, Old Dominion University, Norfolk VA 23529 USA
{theneyan, zubair}@cs.odu.edu
[b]ICASE, MS 132C, NASA Langley Research Center, Hampton VA 23681 USA
{theneyan, zubair}@icase.edu
[c]NAS Division, M/S T27A-1, NASA Ames Research Center, Moffett Field, CA 94035USA
pmehrotra@arc.nasa.gov

Abstract: Distributed heterogeneous systems are being increasingly used to execute a variety of large size simulation and computational problems. Resource management is one of the most important issues in building such systems. Recently, Sun introduced the Jini connection technology for building plug-and-play networks of resources. Jini relies on multicasting across the network for its internal protocols. However, in a distributed environment, such as the one under consideration here, multicasting may not be supported across the subnets for various reasons. In this paper we describe enhancements to Jini required to use it for building a middleware resource management system in a distributed environment that does not support multicasting. In particular, we introduce a lightweight service, called the Tunneling Service, which tunnels multicast messages across the subnets. We have implemented our mechanism and used it to successfully tunnel messages between subnets in a single domain and also between different domains. In this paper we describe our design choices and our implementation of the system.

Keywords: Distributed heterogeneous systems, Jini, Multicasting, Resource management, Tunneling.

1. Introduction

Distributed heterogeneous systems are being increasingly used to execute a variety of large size simulation and computational problems. Such systems require flexible, platform independent and scalable management of the heterogeneous collection of the shared resources [2, 4]. For example, ARCADE is a research project in which we are developing a Web-based framework to provide support for a team of discipline experts collaborating to design, execute and monitor multidisciplinary applications on a distributed heterogeneous network of workstations and parallel machines [3]. In such an environment, the Resource Manager (RM) is required to manage a distributed set of shared resources not necessarily on the same network, which comprises the execution environment and provides information about these resources to the client/user upon request. Typically, these resources could be in multiple domains, which may be distributed geographically and collectively from the underlying computing environment. The resources are varied in nature ranging from compute engines, to data servers, to specialized instruments. The RM has to be flexible enough to not only handle such heterogeneity but also provide information about these resources. The environment is generally dynamic in which resources randomly join and leave. The RM should be able to handle such dynamic behavior without human intervention.

Recently, Sun Microsystems introduced the Jini connection technology which can be used to build dynamic, flexible and easily administered distributed systems. It is based on the idea of federating groups of clients and the resources required by those clients. Jini allows system resources, both hardware and software, to dynamically join and leave the federation [1, 5, 10]. These features of Jini makes it appropriate for building the infrastructure of the ARCADE Resource Manager.

Jini's internal protocols rely on multicasting for discovering and joining lookup services. This becomes an issue when deploying Jini across non-multicastable networks. In this paper, we describe the enhancements that we have made to Jini in order to support systems like ARCADE Resource Manager that need to work with resources in different domains. In particular, we have introduced a lightweight service called the *Tunneling Service* for tunneling multicast message across subnets. In the paper, we describe our design decisions and the implementation of this service.

The rest of the paper is structured as follow: Section 2 gives a short overview of Jini while Section 3 explains the usage of Jini in ARCADE and the problems that Jini suffers while running across non-multicastable subnets. Section 4 explains in detail the approach that we have designed to overcome these problems. Section 5 describes another possible approach and compares the two alternatives. Finally, Section 6 summarizes the paper and discusses some future work.

2. Overview of Jini

Jini is a connection technology introduced by Sun Microsystems that can be used to build a flexible network of resources and services to be shared by a group of clients. Built on top of Java, Jini provides simple mechanisms for resources to join together in a federation with no human intervention. The resources can provide services to clients on the network. The Jini connection technology, based on Java, provides the necessary protocols for services to register themselves with lookup services and for clients to then discover these services. Additional features make the system resilient to failures such as removal of resources, network outages, etc. The whole technology can be segmented into three categories: infrastructure, programming model and services.

The *infrastructure* includes lookup services that serve as a repository of services and uses RMI (Remote Method Invocation), which defines the mechanism of communication between the members. The *programming model* includes interfaces such as discovery, lookup, leasing, remote events and transactions which ease the task of building distributed systems [1, 10]. A *service* is a central concept within Jini. It is essentially an entity that can be used by a person, program or another service to perform a required task.

The runtime infrastructure supports the *discovery and join* protocol that enables services to discover and register with lookup services. Discovery is the process by which a service locates lookup services on the network and obtains references to them. Join is the process by which a resource registers the services it offers with lookup services. In particular, the resource may post with the lookup service, objects representing the services they provide including any code required to use the services.

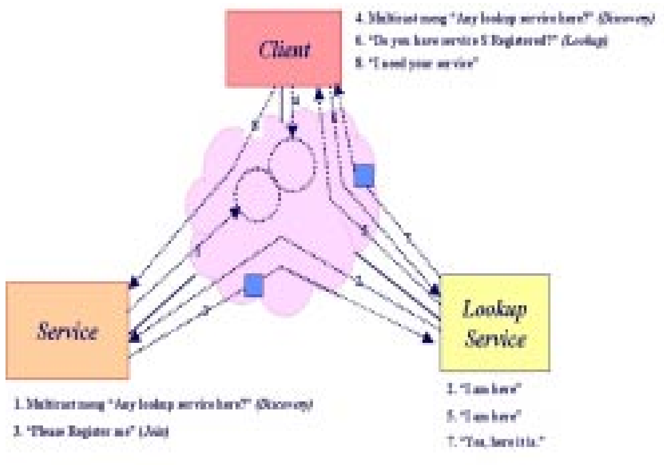
Client
4. Multicast msg "Any lookup service here?" (Discovery)
6. "Do you have service S Registered?" (Lookup)
8. "I need your service"
Service
1. Multicast msg "Any lookup service here?" (Discovery)
3. "Please Register me" (Join)
Lookup Service
2. "I am here"
5. "I am here"
7. "Yes, here it is."

Figure 1: Sequence of steps required to use Jini Technology.

On the other hand, clients use the same protocol to locate and contact services. The discovery protocol is used to locate lookup services. Once an appropriate lookup service has been found, the client can query it to find the reference to the service that it requires. A client may then download the posted object and utilize it to directly use the service. Figure 1 shows a simplified version of the sequence of steps that take place for a service to discover and join a lookup service and for a client to use the lookup service to locate and interact with the service that it is seeking.

Thus, the initial phase for both clients and services is the discovery of a lookup service (LS). As described below, Jini has three discovery protocols: the *multicast request* protocol, the *unicast discovery* protocol and the *multicast announcement* protocol. We refer to clients and services, which are attempting to discover other services as *discovering entities*.

The Multicast Request Protocol

Discovering entities use this protocol to locate all the nearby Lookup Services (LSs). In this protocol, each LS establishes a multicast request server at a well-known multicast end point (224.0.1.85/4160), where it can receive incoming multicast request messages from discovering entities and respond back to them, if necessary, using a direct unicast connection.

The Unicast Discovery Protocol

Discovering entities use this protocol to communicate with a known LS. This LS maybe local, non-local, beyond multicast radius, or one with which a long-term relationship has been established [1, 5, 10]. Using this protocol, the discovering entity sends a unicast discovery message to a specific LS.

The Multicast Announcement Protocol

The Multicast Announcement Protocol is used by LSs to announce their existence. It can be used in two situations. When a new is LS started, it might need to announce its availability and services it provides to clients. Also, when network failure happens and clients lose the connection to an LS, a multicast announcement message can be used to make the clients aware of the availability of the LS [5, 10].

In this protocol, each discovering entity establishes a multicast announcement server at a well-known multicast end point (224.0.1.84/4160), where it can receive incoming multicast announcement messages from LSs.

3. Jini for Resource Management

We have a built a simple middleware resource manager in ARCADE using the Jini connection technology. We have defined our own Java-objects to represent the resources, workstations, and parallel machines in particular. When the ARCADE environment comes online, it starts an ARCADE Resource lookup service on a designated workstation. As the resources in the environment come online a Resource Controller is started up on the resource. This controller then discovers and joins the ARCADE Resource lookup service and uploads its service object. The service object contains some static information, e.g., the type of machine, the speed of the CPU, memory size, etc. It also contains dynamic information, such as the current load. At this point the object is designed such that the dynamic information is loaded into the service object by the resource itself. We are looking at the possibility of the client to pull information about the resources dynamically.

Our experience with Jini technology has brought out some problems in using the technology for resource monitoring. These include scalability, security and the lack of range queries. Here we concentrate on one specific problem: using Jini across networks that do not support multicasting.

The Problem

The brief description in the last section makes it clear that Jini's discovery protocol uses multicast traffic for both the request and announcement messages. This works fine in environments which support multicast. However, some routers on the Internet do not support routing of multicast packets for a variety of reasons. Also, some organizations are not willing to open their firewalls to multicast so as to avoid security problems [8]. Similarly, a local area network divided into subnets, may disable multicast traffic across the subnets to avoid unnecessary traffic that may result in performance degradation. This blocking of multicast traffic across subnets prohibits the use of Jini in such an environment.

One method for working around this problem is to use a tunneling mechanism where the multicast traffic is encapsulated in a unicast packet and is then transferred through unicast routers and non-collaborative firewalls. This method has been used in several projects. For example, MRoutd [9] has been used to achieve tunneling in the MBone [9]. However, there are many problems in the approach taken by the MRoutd implementation, such as the lack of platform independence, wastage of available bandwidth due to the transfer of a lot of control information and the fact that it forwards all the multicast traffic interfaces [8]. Other projects, such as mTunnel [8, 9] and liveGate [6], were designed to overcome some of these problems but there are several reasons for building our own tunneling mechanism and not using some of those existing ones. Having decided to use Jini, we would like to take advantage of the open source code of Jini and embed our mechanism within Jini. Using a pure Jini approach allows us to leverage the capabilities of Jini while activating tunneling in the background without the aid of any of members of the federation.

Also, unlike other tunneling mechanisms, in our environment we do not need to tunnel some of the control information such as the multicast address group and port to which the message is supposed to be delivered. This is because in the context of Jini, our needs are very specific: we need to tunnel only the multicast request and announcement messages that have predetermined multicast endpoints. Providing the right proxies, as explained in the next section, can easily satisfy these requirements.

4. Our Approach

As described in the last section, Jini need to be extended before it can be deployed across non-multicastable networks. To solve this problem, we have introduced a lightweight *Tunneling Service (TS)* with the aim of embedding it within Jini's specifications without affecting Jini's existing functionality.

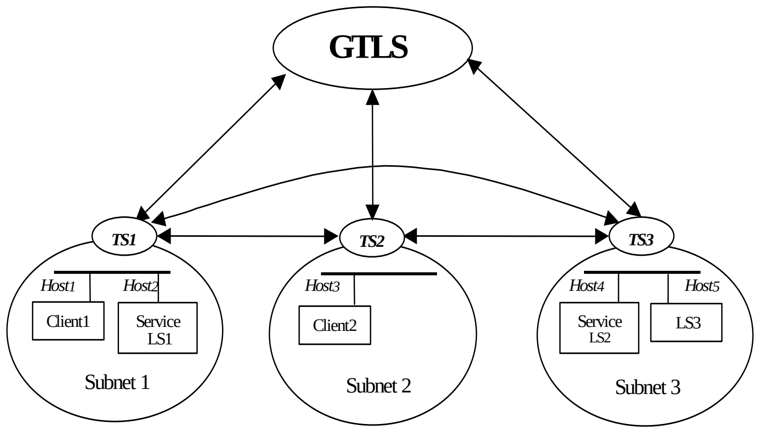

Figure 2: Different non-multicastable subnets connected by the TS

Our approach, as illustrated in Figure 2, involves establishing a *tunneling service* end point, *TS*, at each subnet. Each TS provides a window between its subnet and the rest of the world. The TSs are implemented as Jini services and thus have to register with a known *Global Tunneling Lookup Service (GTLS)* dedicated to maintains the list of TSs in the environment. The GTLS is implemented as a lookup service that can be started at any subnet of the federation. TSs will collaborate with each other in order to tunnel all the multicastable messages across the subnets. Thus, we name this approach the *Collaboration* approach so we can distinguish with the alternative approach the *Central Directory Service (CDS)* scheme, which we discuss in the next section.

Given such architecture, the scenario is as follows. Each TS is going to establish the appropriate multicast endpoints and listen for incoming multicast requests and announcements from within its subnet and will then tunnel the messages out to all the other TSs. Also, each TS is going to listen for incoming tunneled multicast requests and announcements from other subnets and will multicast them locally. Any connection that needs to be setup between the clients, services and the lookup services use the unicast protocol even if they have to cross subnet boundaries.

The underlying aim of our implementation is to make enhancements to Jini that are compatible with the Jini functionality. Thus, we would like the tunneling service to be active in the background without making any changes as far as possible to the behavior of the clients, services and the lookup services. Also, we would like the implementation to work without any modification even if the underlying network supports multicasting and the tunneling service is not required. In the next few subsections we describe the implementation of the Global Tunneling Lookup Service and the Tunneling Service.

4.1 Global Tunneling Lookup Service (GTLS)

In order for the system to work properly, each of the TSs needs to know about all the other TSs in the environment. Thus, we need a central repository that keeps track of all the currently active TSs. Jini provides the functionality required for just such a repository. Hence, we implemented this repository as a lookup service called the Global Tunneling Lookup Service. Using the distributed events interface of Jini, every TS can get notified when a new TS joins or leaves the system. In our implementation, since each TS relies on the unicast discovery protocol in all its interactions with the GTLS, it needs to know the IP address and the port where the GTLS is running.

4.2 Tunneling Service (TS)

The Tunneling Service is the central concept in our solution. This service can be patched into the runtime infrastructure of Jini as a new service just like any other standalone service. A TS has to be started on each subnet that is taking part in the larger system. The system administrator can do this. On the other hand, if suitably modified, the first Jini client, service or LS to start in a subnet could check to see if a TS is already running in the subnet. If not, it can start one. The tunneling service consists of four major parts: the *core tunneling* subsystem which is published at the GTLS as a proxy; the *listener* which keeps track of local multicast requests and announcements and uses other TSs' proxies for tunneling messages; the *notifier* which keeps track of all the other active TSs; and the *wrapper* which implements the infrastructure necessary for the TS to be a Jini service.

The Core Tunneling Subsystem: The core tunneling subsystem is the proxy to the service that is posted with the GTLS by the wrapper. The TSs need to contact the GTLS and download each other's proxies in order to achieve tunneling amongst them. The proxy consists of two methods: one for the incoming tunneled request messages and the other for the incoming tunneled announcement messages. Incoming tunneled requests from other TSs are multicast across the local subnet so that the local LSs can respond appropriately. Similarly, incoming tunneled announcements from other TSs are multicast for the discovering entities in the local subnet. As we explain in the next subsection, a special flag is used to avoid problem of the TS acting on the messages that it has itself tunneled.

The Listener: This is the part of the service that is in charge of catching the necessary multicast traffic, the multicast requests and the multicast announcements from within the local subnet. It listens for incoming multicast requests from any discovering entity in its subnet, at the same multicast request endpoint as any other LS (224.0.1.85/4160). Similarly, it listens for incoming multicast announcements from any LS in its subnet at the same multicast announcement endpoint as any other discovering entity (224.0.1.84/4160). When it receives a request or announcement message, it tunnels it to all the other TSs using their references and proxies that it holds.

The Notifier: This part has been implemented using one of the most useful mechanisms of Jini, the notification mechanism. When a TS starts up, it sends an inquiry to the GTLS about all the currently registered TSs. Then it uses the remote events model supported by Jini to request that the GTLS notify it whenever a new TS registers or leaves the environment.

The Wrapper: The wrapper is the main segment of the service. It publishes the TS's proxy in the GTLS and renews its lease as and when necessary. Also, it launches the assistant subsystems, the Listener and the Notifier, and keeps track of them. If more functionality, such as the encryption of the data for security reasons, or the detection of unnecessary TSs is needed, this can be added as subsystems of the wrapper.

4.3 Jini Modifications

We would have preferred to implement our system without making any modifications to Jini. However, to overcome some of the obstacles of tunneling, we have had to modify the format of the outgoing request and announcement messages. Note that only the message formats need to be modified, the behavior of the rest of Jini remains intact and does not need to be changed.

First is the loopback problem. Consider the situation in which a TS multicasts a message in its local subnet that it has received from an external subnet. The issue is that the *listener* which is listening for all multicast messages within the subnet will also receive this message and will have no way of knowing that the message actually originated from outside the subnet. Thus, we need to distinguish tunneled messages, both requests and announcements, from those originating from within the subnet. We do this by adding a new field, *Tunneled Flag,* in the message's header to show whether this message has been tunneled. This field, added to both kinds of messages as shown in Figure 3, will be set to zero when the message is initially multicast. The TS which then re-multicasts the message in another subnet will set the flag to one so that it can be ignored when it is received by its own listener.

Protocol Version	Port	Tunneled Flag	Host	Group Len	Group1	..	Heard Len	Heard1	..

The new outgoing request message

Protocol Version	Host	Port	LS ID	Tunneled Flag	Group Len	Group1	..

The new outgoing announcement message

Figure 3: New format of Jini messages (shaded fields have been added)

The second problem is the host address of the sender in a tunneled request message. When responding to a request message from a discovering entity, an LS uses the port number included in the message. However, it obtains the IP address of the sender by inquiring for the source of the multicast message. This works well within a subnet where the multicast message is originating from the discovering entity itself. However, in the case of a tunneled request, the IP address is going to represent the TS's host and not the host of the original discovering entity. To overcome this problem, we have added the IP address of the host of the sending entity in the header of the request message, as shown at the top of Figure 3. We don't need to add it in the announcement message since it is already contains the host IP address.

4.4 Current Status

The mechanisms described above have been implemented using the Java Development Kit (JDK) 1.2 and the current Jini reference implementation 1.0 with the modifications that we have described in the previous subsection. We have used the implementation to tunnel messages across subnets in a single domain (*cs.odu.edu*) and also across two separate domains (*cs.odu.edu* and *icase.edu*).

5. Alternative approach to the GTLS

Instead of using the Jini Lookup Service, the repository of TSs in the system, can be implemented as a standalone server that we call the *Central Directory Service (CDS)*. In this scenario, the CDS would not only keep track of the currently active TSs in the system but also act as the conduit for multicasting messages to other subnet. In this approach, the TSs do not need to keep track of other TSs in the system. Using this alternative, each TS listens for incoming multicast requests and announcements in its subnet and sends it to the CDS. It is then the CDSs responsibility to broadcast it to all the other TSs in the system. Each TS still has to listen for incoming requests and announcements tunneled via the CDS and has to multicast them on its subnet.

The Central Directory Service approach has some advantages over the Collaboration approach. In this scheme, the TSs are lighter in weight since they do not have to keep track of the currently active TSs. However, the CDS is a potential communication bottleneck since all messages have to go through it. The Collaboration scheme is a purely Jini approach which leverages off Jini technology in using the mechanism for storing proxies in the GTLS and also the event notification interface for keeping track of active TSs. On the other hand, the CDS scheme allows us to add more functionality to the central repository. For example, in a system such as ARCADE [3], we might need to do some load balancing of executing codes or manage user's remote directories. We cannot add support for such features if we use the Jini LS for our central repository. We will be implementing the alternate approach in the near future in order to compare both the schemes.

6. Conclusion

This paper presents the Tunneling Service, a new service that we have introduced to allow Jini to be used smoothly across non-multicastable subnets. We have described the design and implementation of this service.

There are some other issues that we have not addressed and in particular new features can be added to the system. For example, tunneled data can be encrypted when transported across subnets so we can make sure that only the intended TSs can read it [8]. In addition, tunneling can be done on demand, i.e., we can have a TS only where we need it. Thus, the first Jini client, service or LS that starts in the subnet can start the TS dynamically. Sometimes, we might have more than one TS running at the same network and not aware of each other. Mechanisms like the ones used by mTunnel [7,8] can be added in order to detect unnecessary TSs. A TS might send a multicastable test messages periodically to a specific group address and port and wait for a response. TSs within the same network, if any, exchange messages to identify the redundant TSs. Also, scalability seems to be an issue as Jini is targeted at the workgroup level (about 1000 services). We can achieve high scalability by building a hierarchy of federations. We will be examining our design and adding features as necessary in the near future.

Acknowledgment

This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the authors were in residence at the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681.

References

1. K. Arnold, B. Osullivan, R. Scheifler, J. Waldo and A. Wollrath, The Jini Specification, Addison-Wesley, 1999.
2. K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith and S. Tuecke, A Resource Management Architecture for Metacomputing Systems, Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, 1998.
3. Z. Chen, K. Maly, P. Mehrotra and M. Zubair, ARCADE: A Web-Java Based Framework for Distributed Computing, Proceedings of the WebNet 99, October 1999.
4. I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt and A. Roy, A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation, Proceedings of the International Workshop on Quality of Service, 1999.
5. W. Keith, Core Jini, Prentice Hall, 1999.
6. Live Networks Inc., The "liveGate" Multicast Tunneling Server, <http://www.lvn.com/liveGate/ >.
7. P. Parnes, K. Synnes, and D. Schefstrom, Lightweight Application Level Multicast Tunneling using mTunnel, Computer Communication, 1998.
8. P. Parnes, K. Synnes, and D. Schefstrom, mTunnel: A Multicast Tunneling System With A User Based Quality-Of-Service Model, European Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services, 1997.
9. K. Savetz, N. Randall and Y. Lepage, MBONE: Multicasting Tomorrow's Internet, IDG, 1996.
10. B. Venners, A Talk on Jini/JavaSpaces, <http://www.artima.com/javaseminars/modules/Jini/index.html>.