Implementing Jini Servers without Object Serialization Support

Tero Hasu
Helsinki University of Technology
Telecommunications Software and Multimedia Laboratory
`tero.hasu@hut.fi`

Abstract

Jini is a technology that provides a framework for discovering and providing services in ad hoc network environments. One of the core ideas of Jini is that services are accessed through a Java object supplied by the provider of a service. To distribute such objects between processes and hosts, they need to be translated into a common transport format; Sun's Jini implementation uses the Java object serialization mechanism for making the translation. However, the serialization functionality is not available on all Java virtual machines, not to mention non-Java runtime environments.

Jini clients cannot be implemented without serialization support, as to use a service they need to be able to deserialize the object via which the service is accessed. Jini servers, on the other hand, do not need to run any code supplied by the client, and serialization capability is not essential on the server. When serialization is unavailable, so is RMI (Remote Method Invocation), as RMI relies on the serialization mechanism for passing objects by value in remote procedure calls. Sun's Jini implementation uses RMI for most of the communication between client and server. Nevertheless, the use of RMI is not required by the Jini specification, and some other remote procedure call mechanism may be used to facilitate communication. In this paper, we explain how to implement Jini servers such that they can be run on a platform without either RMI or object serialization support.

KEYWORDS: Java, Jini, object serialization, RMI, XML-RPC

1 Introduction

Mobile technology is becoming more commonplace, and there is a lot of interest in using it to access services in mobile, ad hoc environments. Jini is a Java-based technology that provides a framework for locating and using services in a dynamic environment, and open, Java-enabled devices are becoming more common. It is possible that Jini will become popular along with the devices, and if that should happen, the availability of a Jini interface or lack thereof might significantly affect the popularity of a service. It might thus become a common requirement to include a Jini interface in both new and existing services, either as the sole access method or as an alternative one.

The Jini implementation currently available from Sun requires either a J2SE (Java 2 Standard Edition) or J2EE (Java 2 Standard Edition) runtime environment. (In the context of this paper, there are no significant differences between the two platforms, and for the rest of this paper, we shall simply use J2SE to refer to both of them.) If a service does not already run under J2SE, porting it to J2SE or running the Jini functionality separately under J2SE is not necessarily the best option. Perhaps the most straightforward way to add Jini support would be to link the server executable against a library that contains the necessary functionality for allowing Jini clients to locate the service, and to acquire the object that contains the Java interface to the service.

Running a large Java runtime may not even be an option when it comes to resource-constrained devices such as PDAs, mobile phones, and appliances. In such cases one could consider running Jini on a separate, more powerful device, but especially when on the move, such a device may not always be accessible. To guarantee the availability of the services offered by a device, it is perhaps best to run Jini functionality on the same device, so that the services offered by the device are discoverable whenever the device is accessible, and only then. To accomplish this with resource-constrained devices, a Jini implementation that runs on a lightweight JRE (Java runtime environment), or even without a JRE of any kind, is required.

In this paper, we look into how to implement Jini-enabled servers such that they do not need to support complex, Java-specific technologies that are only available on the most capable of JVMs. In particular, we shall concentrate on object serialization and RMI, both of which are utilized by Sun's reference implementation of Jini. Both technologies are rather complicated, and implementations tend to be large. For instance, the total size of the RMI-related Java class files contained in the J2SE 1.4.1 runtime is in excess of half a megabyte, and that figure includes neither the dependencies of RMI or any runtime storage requirements. Therefore it is no wonder that, for instance, the popular Connected, Limited Device Configuration (CLDC) of Java 2 Micro Edition (J2ME) does not include either RMI or serialization, although their exclusion may also have been due to security concerns [9]. We believe that when knowing how to implement Jini functionality without the two technologies, it should be perfectly feasible to implement Jini servers that do not even require a Java runtime.

One of the challenges in our work was determining what parts of Jini could be implemented without object serialization and RMI. Early on, we concluded that Jini clients require object deserialization and dynamic class loading support, which ruled them out, and we then focused on servers alone. An advantage that Jini servers have is that they can have their own code executed in a client, and the client is likely to have a powerful JRE. This helps in overcoming situations where essential functionality is unavailable on the server side, and indeed, we believe that it is possible to implement Jini services that fully conform to the Jini specification without having serialization support on the server side.

We also needed to find alternative ways to implement the functionality for which serialization and RMI would have been a natural choice. For instance, we decided to apply the platform independent XML-RPC technology instead of RMI for remote procedure calls. When we found serialization to be essential for certain functionality, we needed to determine how to have the server and client share responsibilities in implementing that functionality. To get confirmation that our solutions work in practice, we constructed a proof-of-concept implementation that utilizes some of those solutions. We shall describe our implementation later on in this paper.

We shall follow this introduction by discussing related work in Section 2. After that, we have Section 3 that gives background information regarding technologies particularly relevant to this paper. In Section 4 we go through the functionality that is required of Jini conformant entities, and in Section 5 we then discuss how to implement some of that functionality without using the Java object serialization mechanism on the server side. In Sections 6 and 7 we describe our prototype implementations, and we then evaluate them in Section 8. Finally, in Section 9, we present our conclusions.

2 Related Work

The component in Jini that allows the registration and discovery of services is called a *lookup service* (LUS). In this paper, we present a LUS implementation that does not use RMI. There are two commercial LUS implementations called JMatos [8] and CMatos [6] by PsiNaptic [5] that likewise do not require RMI functionality from the underlying platform.

JMatos runs on a variety of Java platforms, including Personal Java, J2SE, and J2ME. Presumably, however, not just any J2ME implementation will do, but it must be one that supports server-side sockets, which as of yet are not available on most J2ME-equipped devices currently on the market. Server sockets are required for implementing the Jini discovery functionality, and possibly also for implementing an HTTP server or some other mechanism that allows Java class files to be dynamically loaded from the host that is running the LUS.

CMatos is written in C, and does not require a Java runtime at all; any requirement for executing Java bytecode in the LUS host has been removed [6]. This being so, it should also not be necessary to do any object serialization on the server side, which provides some confirmation to our suspicions to that effect.

JMatos and CMatos are closed source, and there does not appear to be much information available as to exactly how they have been implemented. Thus, although we present an implementation with similar goals in this paper, we will not compare our implementation with those of PsiNaptic. At most we could compare external factors such as performance and memory consumption, but we have chosen not to do so due to time constraints.

3 Background

This section provides some background information regarding technologies that play an important role in the rest of this paper.

3.1 Java Object Serialization

Some Java platforms provide built-in object serialization support, as specified in [10]. The serialization mechanism facilitates the translation of object state data to bytes and vice versa. Serialized objects do not contain the definitions of the associated classes and interfaces, and thus, when deserializing an object, it is necessary to either have the relevant type definitions preloaded, or to know the location from which to load the classes and have the capability to do dynamic class loading.

By default, any classes to load at runtime are looked for by the class loaders created automatically by the JVM, from known locations on the local filesystem. Typically, the location of the classes of a locally installed application is specified by setting the `CLASSPATH` environment variable, which tells the so-called *system class loader* the location from which to automatically load the application classes as required. Generally, we shall refer to the parameter that tells a Java class loader from where to find the definition of the class to load as the *codebase* of that class, regardless of which class loader is being utilized.

3.2 Remote Method Invocation

Remote Method Invocation (RMI) [2] is a mechanism in which objects that reside in different JVMs may invoke methods on each other [7]. RMI uses Java object serialization for passing objects by value. When passing objects by "reference", serialization is still used for transferring so-called *stub* objects that have the same interface as the "referred" object.

Partly due to the heavy use of the Java serialization mechanism, RMI is rather tightly coupled with Java, and it would probably be infeasible to implement it for other platforms. On the other hand, when Java and RMI is available throughout a distributed system, it makes interaction between remote objects rather convenient and easy to implement. For instance, RMI transparently equips clients with the ability to download implementation code. It is therefore not a surprise that Sun's Jini implementation is built on top of RMI.

3.3 Jini

Jini [3] is a distributed computing framework that consists of libraries and server programs that provide a variety of functionality. Among other things, Jini offers support for leasing of shared resources, transactions, and service administration, but it is perfectly possible to construct Jini services that do not make direct use of any such functionality. That functionality is not within the scope of this paper, as we are rather focusing on what is required to offer services via Jini when the server platform is lacking certain features.

Jini provides support for registering and looking up services. We refer to entities offering services as servers, and those using them as clients; it is possible for the same entity to be both a server and a client. A set of all the Jini clients and servers in a network is commonly referred to as a *federation*. To enable clients to locate services, there must be one or more lookup servers in the network, and also within the multicast radius when multicast discovery is being relied on to locate one. If one does not want to risk a LUS not being available, it is safest to run a LUS along with any services that one is providing.

When a service is registered with a LUS, a Java object that implements the service API must be supplied to the LUS. A client that wants to utilize the service must first acquire the object from the LUS, as the service is accessed via that object. For the rest of this paper, we shall refer to such an object as a *proxy*, regardless of whether the object talks to a server or implements the whole service by itself.

3.4 XML-RPC

XML-RPC is a platform-independent remote procedure call (RPC) mechanism, whose specification is in [12]. XML-RPC defines a protocol where each message is an HTTP-POST request, and the body of the request is in XML format. Procedure call parameters and return values must be mapped to the limited number of data types and structures that XML-RPC supports. Any transparent translation between the data types of XML-RPC and those of a programming language are implementation specific. XML-RPC is rather language-agnostic, and there are implementations for a variety of languages and platforms.

4 Requirements for Jini Federation Members

This section discusses the functionality that the entities participating in a Jini federation are required to have or implement. A Jini federation typically consists of a number of clients and servers, and some of the servers must act as lookup servers to enable clients to locate services. We will cover the different types of entities separately.

4.1 Jini Clients

The proxy of a lookup service is sent to the client during the lookup server discovery process, which is based on a custom protocol. For all other services, the proxy is acquired by the client by a method call to the LUS proxy. Every LUS proxy implements the `net.jini.core.lookup.ServiceRegistrar` API. Regardless of which of these two methods is used to acquire a proxy, the proxy object needs to be converted from the transport format into a Java object instance within the client runtime. The class definitions of the deserialized classes are loaded from the LUS or some other server. Therefore Jini clients require deserialization and dynamic class loading capabilities.

Not only does the client need to have the capabilities required by the LUS proxy, but it should also be able to execute any code contained within any proxy if it wants to access the service associated with that proxy. A client that does not run on a J2SE environment is likely to be excluded from a large number of services, as are those clients with restrictive security policies that prevent certain operations even if the functionality would otherwise be available. Generally, the more APIs and the fewer restrictions a client platform has, the more likely is it to be able to access a service that it comes across.

However, it could be argued that to be Jini-enabled, a client merely needs to be able to discover lookup servers, and to acquire and deserialize LUS proxies. Whether it can actually use a proxy depends on the proxy not requiring capabilities that the client does not have.

4.2 Lookup Servers

A Jini lookup service must provide the following services:

1. Performing its part of the LUS discovery process. Both the unicast and multicast discovery protocols must be supported.
2. Maintaining information about registered services; the information must persist across server restarts.
3. Accepting service registrations, renewals, and deregistrations. Registrations must be deleted if not renewed in a timely manner.
4. Responding to queries regarding registered services.
5. Allowing registered clients to update the attributes associated with services that they have registered.
6. Notifying clients regarding changes in the service database if the clients have subscribed to such notifications.

Let us now consider how the implementation of the above services can be split between the server and its proxy. Item 1 is something that must be taken care of at the server side, as the proxy is not used during discovery. Item 2 must likewise be handled by some server, as the database contains information about multiple clients, and cannot be local to each proxy object; the database server would typically be the lookup server itself. Items 3-6 are such that they can be requested by the client via the proxy API, and therefore some of the implementation must be in the proxy. The database must also be accessed by the implementation, except perhaps with item 4 that only concerns queries, and not modifications to the database. If all the data was cached in the client side, and there was an update notification mechanism in place to ensure that the cache stays current, then item 4 could be implemented fully within the proxy. Apart from this possible exception, items 3-6 all require functionality both in the proxy and in the server.

If a lookup server was weak in terms of performance, the implementor would probably want to offload as much functionality as possible to the proxy. In such a case, the server would probably only handle discovery, data storage, deletion of expired service entries, and the originating of events regarding changes in the database. Everything else can be done in the proxy. According to [8], this is the approach taken by JMatos. If, on the other hand, the server had ample processing power and capabilities, then the proxy could be a mere stub that would just translate API calls into requests sent to the server. Sun's Jini implementation takes this approach; the proxy is just an RMI stub. These approaches are the two extremes, but naturally the functionality can be distributed in a more even manner as well.

When setting up a lookup service to run on a resource-constrained device, perhaps to ensure that the device and its services can be discovered, one might hope that some of the services listed above would be optional; however, that is not really the case. In particular, one might want to refuse registrations from external services, as the extra work involved might be prohibitive on low-powered devices. However, the Jini specification does not appear to provide any way to refuse a registration. The `ServiceRegistrar` method that allows registration is

```
public net.jini.core.lookup.ServiceRegistration register(
            net.jini.core.lookup.ServiceItem item,
            long leaseDuration)
    throws java.rmi.RemoteException
```

Returning `null` to indicate that no registration was made is not an option given by the Jini specification. One could always throw a `RemoteException` to indicate that registration may have failed, but the client might just keep retrying, which would result in a lot of unnecessary work by the client. This is obviously not desirable, and there thus does not appear to be a decent way to refuse registrations.

4.3 Other Jini Servers

In the case of application-specific services, those creating a service get to decide what the server does, and how it is accessed. Therefore the designer of the application is the one who decides what is required of the server. What is typically required is some method for the proxy to communicate with the server, but we will not speculate on what else might be required. Depending on the application, it may even be that a service can be implemented fully within its "proxy", in which case a corresponding server is not required at all.

5 Avoiding Server-Side Object Serialization

In Sections 4.2 and 4.3, we explained what functionality is required from Jini servers. In this section, we discuss how such functionality can be provided if the underlying platform does not provide object serialization support. As previously mentioned, this also means that we cannot use RMI. We shall mainly concentrate on lookup servers, as we know what functionality is required of them. The same or similar solutions may or may not apply to an application-specific server, depending on what the application is.

5.1 Lookup Server Discovery

In Jini, there are three different discovery protocols: multicast request protocol, multicast announcement protocol, and unicast discovery protocol. From our point of view, the good point is that none of the protocols utilizes RMI. Instead, in all of them, a simple, custom protocol is used, with TCP and/or UDP as the network protocol. The drawback is that messages of the protocols are composed of serialized Java objects and primitive data types. To construct and interpret such messages, even if the lookup server is not capable of full object serialization, it still needs to be able to understand and produce the serialized forms of the following Java objects:

- `int`
- `java.lang.String`
- `net.jini.core.lookup.ServiceID`

We believe it to be feasible to implement the required routines, as the above objects are relatively simple. However, we do not feel qualified to give a proper estimate as to the amount of work involved, as we have not implemented such routines.

During discovery, a serialized proxy object for the LUS must be sent to the client. If the lookup server has no serialization capability, it must have been serialized in advance, presumably by the LUS build process, and stored somewhere as a Java byte array. See Section 5.2 for advice on how to perform the serialization.

5.2 Proxy Serialization

The LUS proxy that is sent from a lookup server to a client must be wrapped in a `java.rmi.MarshalledObject`, as that is what the client expects to receive. The reason for the use of such a wrapper is presumably that in addition to a serialized object, a `MarshalledObject` also contains the codebase information for that object. As the codebase must be known during deserialization, it makes sense to use a `MarshalledObject` also when sending objects between a lookup server and its proxy, although the codebase information could also be passed differently, if required.

As far as we know, there is no method in any common JRE that would make it convenient to redefine the codebase of a class that has been already loaded. There is also no method in `MarshalledObject` that could be used to set the codebase associated with the wrapped object. However, there is a JVM property called `java.rmi.server.codebase` that will be used as the codebase for all classes loaded from the `CLASSPATH` and subsequently marshalled. We can either set the property to reflect the codebase, or, when creating the object to be wrapped, actually load the instantiated class from a location that will also be accessible by clients who download the serialized object. That location will then be stored within the `MarshalledObject` as the codebase.

The following code snippet demonstrates one way to load class `className` from codebase `codeBase`, to create an instance of it using the default constructor (which we are assuming it has), and then to write the object to the output stream `stream` in a serialized form. `OurClassLoader` could for instance be a subclass of `java.net.URLClassLoader`, with no changes apart from making sure that the `findClass(String)` method has such permissions that we can call it.

```
URL url = new URL(codeBase);
ClassLoader loader = new OurClassLoader(url);
Class clazz = loader.findClass(className);
Constructor con = clazz.getConstructor(new Class[0]);
Object object = con.newInstance(new Object[0]);
MarshalledObject mo = new MarshalledObject(object);
ObjectOutputStream out = new ObjectOutputStream(stream);
out.writeObject(mo);
out.flush();
```

For all the services registered with a LUS, the lookup server must store the proxy objects in persistent storage. Any services that are registered at the server side can be serialized in the manner described above, at build time, on a platform capable of Java object serialization. Those services that are registered by clients can be serialized at run time within the LUS proxy that runs on the client side, simply by wrapping the service proxy within a `MarshalledObject` and then serializing.

5.3 Service Lookup

During a service lookup, a client asks the LUS proxy for one or more services that match a template supplied by the client. The proxies of the matching services are then returned to the client. In Section 5.2 we discussed how the proxies can be prepared so that they are ready for sending to the client. Let us now consider how to select the services whose proxies are to be sent. The `ServiceRegistrar` methods that allow service lookup are shown below.

```
public java.lang.Object
       lookup(net.jini.core.lookup.ServiceTemplate tmpl)
    throws java.rmi.RemoteException

public net.jini.core.lookup.ServiceMatches
       lookup(net.jini.core.lookup.ServiceTemplate tmpl,
              int maxMatches)
    throws java.rmi.RemoteException
```

Let us take a closer look at `ServiceTemplate`, as that is what we need to compare to each of the records of a service database. In the discussion that follows, we shall frequently refer to the templates and the records being compared as *service templates* and *service items*, respectively. A `ServiceTemplate` has the following properties:

- `net.jini.core.entry.Entry[]` attributeSetTemplates
- `net.jini.core.lookup.ServiceID` serviceID
- `java.lang.Class[]` serviceTypes

In each of the above, a `null` value serves as a wildcard that matches every service item. Wildcards can be significant in terms of performance. For instance, if we found that the vast majority of templates never contained any attributes, then we would perhaps want to avoid caching of attributes within the proxy, as that would increase memory consumption on the client side without giving us much benefit. If, on the other hand, lookups with attributes were very frequent, then there would be a lot of traffic generated due to attribute lookups if no caching was done. Expected usage patterns should be considered while implementing caching, or one might even want to implement adaptive caching. However, discussing complex caching schemes is beyond the scope of this paper, and we will not go into any more detail regarding caching here.

When matching a service template to a service item, it is possible to do everything on the client side. It is also possible to do everything on the server side, with the possible exception of attribute set comparisons. Performing some of the comparisons at the client and some at the server is also an option, but one that we shall not discuss here, for brevity.

Perhaps the simplest alternative is to do all the matching on the client, within the proxy, as we are assuming here that the client is powerful and capable of serializing objects. If we simply ensure that we have an up-to-date copy of the service database within the proxy, perhaps by asking the server to notify us regarding any database changes (see Section 5.5), we can deserialize objects and utilize their methods to access their properties or to perform equality checks, for instance.

A more complicated approach is to attempt to resolve lookups entirely on the server side. In order to do that, both the template and the database must be in such a format that the server is capable of performing the comparisons. Let us consider each of the `ServiceTemplate` properties separately:

Service identifiers To compare service identifiers, one can simply do a byte comparison between the 16-byte service identifiers. The identifier data is normally wrapped in a `ServiceID` object, but the 16-bit value can be extracted and converted to a byte array before passing it to the server.

Service types One can simply use reflection to extract all the types of service proxy object, including interfaces and superclasses. If the fully qualified names are then given to the server as ASCII strings, the server merely needs to perform bytewise comparisons between the strings to determine whether a service is of the requested type.

Attribute sets An `Entry` can be thought of as an attribute set, with each of the fields of the entry being an attribute that belongs to the set. For an entry to match a template, the class of the template must be the same as, or a superclass of, the class of the entry [11]. We can handle the type comparisons similarly as in the case of service types. For every `Entry` in a service template, we must have the fully qualified name of its class. For every Entry in a service item, we must know the name of its class, as well as all superclasses.

During matching, we would also have to establish whether an entry has a matching field for each of the fields in a template entry. Field equality is defined by `MarshalledObject.equals(Object)`, which states that the serialized representations of the objects within must otherwise match, but that any differences in codebase annotations must be ignored [11]. If it was feasible to take a serialized object and to remove the codebase annotations, then we could simply store the result at the lookup server, and do a byte comparison to determine equality. However, we have not determined what is involved in the removal of the annotations, and that is left for future work.

5.4 Service Database Queries and Updates

Apart from the lookup methods, a `ServiceRegistrar` also has a number of other methods that can be used to query a service database. These can be implemented in a manner similar to what was described in Section 5.3. There also is a facility for setting and modifying service attribute sets; these can likewise be handled similarly, but if any database changes are actually caused, then those changes must be propagated from the proxy to the database server. For queries, it may not be necessary to contact the server if sufficient caching has been done to answer the queries.

5.5 Service Database Update Notification

The `ServiceRegistrar` API provides a facility that clients may use to ask for notification regarding service database changes. Thus far in our discussion, it has typically been the proxy that must contact the server; however, to implement the notification facility, it must also be possible for the server to contact any proxies that have registered for notifications. Unless one wants to keep a connection open for as long as such registration subscriptions are in effect, there must be a mechanism for the server to connect to the client. The proxy could, for instance, start listening to a certain port for connections, and supply the contact information to the server when subscribing.

Once implemented, the notification functionality can be useful not only for clients, but also within a lookup server. If service information is being cached, the notification mechanism can be used to tell LUS proxies to update their cache when the information gets updated. Of course, if no caching is used, and the client does not subscribe to notifications, the server does not need to be able to initiate contact with the proxy that resides on the client.

5.6 Communication Between a Jini Server and Its Proxy

Since we are looking at a situation where RMI is not available, there must be an alternative method of communication between a proxy and a server, unless the entire service is implemented within the proxy. One possibility is to always tailor application-specific, custom protocols, and especially in a resource-constrained environment such an approach may be the best as it allows for implementations optimized for a particular purpose.

However, since RMI is an RPC mechanism, another RPC mechanism is perhaps the most natural replacement for RMI. One of the most lightweight of RPC solutions is XML-RPC, which we briefly introduced in Section 3.4. The benefit of XML-RPC in this context is that it should be suitable for a variety of Jini servers, as implementations are available for a number of platforms, and at least the simpler implementations tend to be small, which makes XML-RPC fairly well suited even for hosts with little storage space. If one implemented the same Jini server for a number of different platforms, then the ability to use the same RPC interface in each of them would make it an option to use the same proxy to access each server.

5.7 Service Registration

Since a Jini client must have object serialization capabilities, it is normally not a problem that when registering a service with a LUS, its proxy must be serialized within the LUS proxy. Now, we have been discussing Jini servers that do not require serialization capabilities, yet the services they provide must also be registered with a LUS to be found. Since those servers cannot use the standard Jini mechanism for registering themselves with a LUS, there must either be some other entity that takes care of the registration on their behalf, or there must be a different method for registering. Some of the possible, non-standard registration solutions include:

- Have the LUS itself register the services, either using a fixed set of services, or have the LUS acquire a list from a known place.
- Run the services in the same process as the lookup server, and include an API in the server that allows registration.
- Provide a custom registration facility that can be accessed from a separate process or even from a remote host. Such a facility could also be utilized by the LUS proxy to perform registrations requested by clients. For instance, an API accessible via XML-RPC could provide a suitable registration interface.

6 Jini Lookup Service Implementation

As a part of our work on this topic, we implemented the beginnings of a Jini LUS. Basically, we wanted to implement just enough of the functionality to do a lookup and to acquire the service object of the Hello World service that is presented in Section 7. To be more specific, we implemented one of the lookup methods, with the omission that it does not support attribute sets. We had the LUS itself register the Hello World service, which is one of the registration options mentioned in Section 5.7.

As we implemented our services, we applied some of the solutions presented in Section 5. We utilized the Apache XML-RPC implementation [1] for communications between the proxy and the server. We had the lookup server listen to a known port for XML-RPC connections. The port number was hardcoded within the LUS proxy, but the same was not done with the address of the LUS host. We decided that the LUS proxy codebase would reside on the same host as the server, and therefore the following code within the LUS can be used to determine the lookup server host:

```
Class clazz = getClass();
ProtectionDomain domain = clazz.getProtectionDomain();
CodeSource cs = domain.getCodeSource();
URL url = cs.getLocation();
String host = url.getHost();
```

As described as one possible approach in Section 5.3, we resolved lookups fully on the server side. The server has the following lookup API, where the parameter and return value types were chosen so that they are directly supported by Apache XML-RPC.

```
public byte[] lookup(byte[] serviceId, Vector types)
```

In addition to making XML-RPC API calls, all that was left for the proxy was to translate the parameters and return values so that they correspond to the `ServiceRegistrar` lookup API. To support this scheme we wrote a build tool that collects information regarding proxy objects; for instance, to collect the fully qualified type names we used the following code:

```
static void getAllSuperClasses(List list, Class clazz) {
    for (;;) {
        clazz = clazz.getSuperclass();
        if (clazz == null)
            break;
        list.add(clazz);
    }
}

static void getAllInterfaces(List list, Class clazz) {
    Class[] ifaces = clazz.getInterfaces();
    for (int i=0; i<ifaces.length; i++) {
        Class iface = ifaces[i];
        if (list.contains(iface))
            continue;
        list.add(iface);
        getAllInterfaces(list, iface);
    }
}

static List getAllTypes(Class clazz) {
    List list = new LinkedList();
    list.add(clazz);
    getAllSuperClasses(list, clazz);
    getAllInterfaces(list, clazz);
    return list;
}
```

We targeted all of our prototype implementations for the J2SE platform. The client JVM is not required to have the XML-RPC classes installed locally; the system, application, and service interface classes are enough. The proxy class definitions as well as the definitions of any classes used within the proxy (such as the XML-RPC implementation classes) are retrieved from the lookup server.

7 Hello World Service Implementation

As already mentioned, we implemented a simple Jini service that we registered with our LUS implementation. The service is called Hello World, and its proxy object provides a method that returns the string "Hello World!" to the caller. This functionality would have been easier to implement fully within the proxy, but instead we also implemented a server that sends the string to the proxy using XML-RPC. The XML-RPC communications were set up in the same manner as with the LUS described above.

The following code snippet should give an idea of how the Hello World proxy can be acquired and used to access the service, when `registrar` is the LUS proxy:

```
ServiceTemplate tmpl = new ServiceTemplate(
    null, // any service ID
    new Class[] { HelloWorldService.class },
    null); // any attribute sets
Object proxy = registrar.lookup(tmpl);
HelloWorldService service = (HelloWorldService)proxy;
System.out.println(service.getString());
```

As our lookup server does not support discovery, we "faked" the discovery by reading the serialized LUS proxy from a file, instead of receiving it during the discovery process, as follows:

```
File file = new File(LUS_PROXY_FILE_NAME);
ServiceRegistrar registrar;
InputStream input = new FileInputStream(file);
try {
    ObjectInputStream os = new ObjectInputStream(input);
    MarshalledObject mo = (MarshalledObject)os.readObject();
    registrar = (ServiceRegistrar)mo.get();
} finally {
    input.close();
}
```

8 Evaluation of the Implementations

As our LUS only implements a small subset of the functionality required by the Jini specification, and only makes use of some of the guidelines presented in this paper, it is some-what lacking as a proof of concept. Nonetheless, it would have been infeasible for us to implement a complete LUS given that we did not have much time to allocate for the implementation work.

Our choice of a platform, i.e. J2SE, is also less than perfect in this context, but we chose it as we deemed that the capabilities of the platform would decrease the amount of labor required. Of course, it would have made for a more effective proof-of-concept to target a server platform that does not support object serialization. However, we had to minimize the amount of work involved.

Hello World is a valid Jini service, albeit a simple one, and demonstrates that at least certain kinds of Jini servers can easily be implemented without RMI or object serialization support. Very little work on the part of the programmer was required to implement this simple service, in part because the Apache XML-RPC library uses Java's reflection facilities to automatically determine which methods to make available for calling remotely.

It should be reasonably straightforward to port our implementations for another platform, assuming that there is an XML-RPC implementation available for that platform. The Apache XML-RPC software that we used would not be suitable for all Java-based implementations, as it e.g. makes use of reflection, which is not available on all JREs. However, there are a number of other implementations, such as kXML-RPC [4], which is designed to be run on J2ME environment.

9 Conclusion

In this paper we have discussed how servers can participate in a Jini federation even when the utilization of a powerful Java platform with Java object serialization capabilities and RMI is either infeasible or undesirable. We have also described and evaluated our prototype implementations of a partial Jini LUS and a simple Jini service, neither of which require serialization support, and both of which use XML-RPC instead of RMI for client-server communication.

Acknowledgements

The author would like to thank Janne Jalkanen for advice, and in particular for the idea of using XML-RPC as an RMI replacement. Thanks are also due to Razvan Matei for providing feedback regarding this work.

References

[1] Apache XML-RPC.
URL `http://xml.apache.arg/xmlrpc/`

[2] Java Remote Method Invocation (RMI).
URL `http://java.sun.com/products/jdk/rmi/`

[3] Jini network technology.
URL `http://java.sun.com/jini/`

[4] kXML-RPC.
URL `http://kxmlrpc.enhydra.org/`

[5] PsiNaptic Inc.
URL `http://www.psinaptic.com/`

[6] STEVEN KNUDSEN, SERGE BRACHE; CMatos – Jini services for non-Java devices; white paper; PsiNaptic; 2002.

[7] SCOTT OAKS, HENRY WONG; Jini in a Nutshell; O'Reilly, USA; ISBN 1-56592-759-1; 2000.

[8] LAWRENCE (TIM) SMITH, CAMERON ROE, KNUD STEVEN KNUDSEN; A Jini lookup service for resource-constrained devices; in 4th IEEE International Workshop on Networked Appliances; Gaithersburg, MD, USA; 2002.

[9] Sun Microsystems, Inc.; Connected, Limited Device Configuration Specification Version 1.0a; 2000.

[10] Sun Microsystems, Inc.; Java Object Serialization Specification Revision 1.4.4; 2001.

[11] Sun Microsystems, Inc.; Jini Technology Core Platform Specification Version 1.2; 2001.

[12] DAVE WINER; XML-RPC specification; 1999.
URL `http://www.xmlrpc.com/spec`