Jini Meets UPnP: An Architecture for Jini/UPnP Interoperability

J. Allard, V. Chinta, S. Gundala, G. G. Richard III
Department of Computer Science
University of New Orleans
New Orleans, LA 70148
Contact: golden@cs.uno.edu

Abstract

A service discovery framework provides a collection of protocols for developing dynamic client/server applications, allowing clients to find and use services without any previous knowledge of the locations or characteristics of the services. There are currently many service discovery technologies available or in development, including Jini, UPnP, SLP, Salutation, Bluetooth SDP, and Ninja. These have similar high-level goals, but quite different architectures. Each software or hardware product utilizing service discovery will typically use only one of these protocols, meaning that clients and services using different technologies will not be able to cooperate. Since it is likely that several protocols will be widely used, there is a need for interoperability frameworks that allow clients and services written using different service discovery technologies to cooperate.

This paper presents a Jini/UPnP interoperability framework that allows Jini clients to use UPnP services and UPnP clients to use Jini services, without modification to service or client implementations. As service specifications are typically developed independently for each protocol, a fully automatic interoperability solution is not currently practical, so we introduce service-specific proxies to bridge Jini and UPnP. Our goal is to reduce the amount of effort required to support new service types and our framework includes a substantial amount of support for rapid proxy development. A modest development effort is required to support each new service type, and our initial (and highly unscientific) measurements reveal that the level of effort is typically on the order of one day by a member of our team.

Keywords: service discovery protocols, Jini, Universal Plug and Play, interoperability.

1. Introduction

Broadly, a service discovery framework is a collection of protocols for developing dynamic client/server applications [8, 9]. A number of component protocols are typically included, which allow services to advertise their availability and for clients to search for needed services. In a service discovery-enabled network, devices that are plugged in become part of the community and may be discovered and used by clients with a minimum of manual configuration. Services that crash, are replaced, or move, gracefully disappear. Catalogs track available services. Garbage collection facilities rid the system of outdated information. Device drivers—at least in the sense that they are visible to users—are eliminated and highly dynamic interactions between clients and services become the norm. Administrative hassles are minimized, since devices can be replaced and moved about easily. The functionality of mobile devices such as palm-sized computers, which necessarily trade functionality for small form factors, is greatly enhanced, since needed services can be discovered on-demand, whether the user happens to be at home or in a coffee shop.

In a service discovery world, a client in need of a printer invokes service discovery to find available devices, evaluates their characteristics (e.g., resolution, color capability, cost per page), chooses a device, and initiates a print job. A PDA in desperate need of additional storage searches for a remote file storage service, offloads some of its content to free available memory, roams away, then returns to claim the files. Virtually anything can be a client or a service—clients need things, and services provide them. A digital camera might participate in a network as both service and client, providing pictures to clients, but also engaging an enabled light if the picture is too dark. Connecting the needy—clients—and the providers—available services—is the point of service discovery. These technologies directly attack the "I don't know where you are" and "I don't know how to talk to you" issues of client/server. But it can't be this easy, right? In general, it's not, because there are many competing, incompatible service discovery technologies. More on that in a bit. First, the common ground.

1.1. Common Ground

The major commercial players in service discovery are currently Jini [1], Universal Plug and Play [11], Salutation [10], and SLP [5]. Bluetooth Service Discovery Protocol (SDP) [2,3], which is substantially lighter weight and has more limited functionality, as well as a number of research oriented systems round out the list. One of these,

Ninja [4], addresses a number of security concerns lacking in other protocols. There is a lot of common ground among these technologies, and we'll concentrate on the commonality before addressing the major differences. Not all service discovery technologies embody all the concepts that follow, but most include a majority of them. All support the concepts of *client* and *service*, which are simply entities that need and offer some functionality (e.g., printing), respectively. Clients perform *discovery* in order to find needed services. In some cases, they may directly seek the needed services themselves; in others, they may contact one or more *service catalogs*, which maintain directories of available services. A discovery attempt generally classifies the service by type, and may optionally include requirements such as a manufacturer, serial number, or other service attributes.

Whether services are sought directly or a catalog is consulted, a client needs very little information about its environment—it can locate services (or service catalogs) dynamically (via multicast), with little or no static configuration. When a service enters the network, it will perform *service advertisement*, either directly to clients or to one or more service catalogs. These advertisements include necessary contact information and also descriptive attributes (or information allowing the attributes to be discovered). An eventing mechanism is typically available, which allows clients or services to be informed of interesting events in the network (e.g., a printer running out of paper). A garbage collection facility rids the system of outdated information, such as descriptions of services that have disappeared from the network, or requests from clients that no longer need services. Typically, an implementation of a particular service discovery framework provides a robust environment for developing applications, but developed clients and services are quite non-portable between *different* service discovery technologies.

1.2. Smart People Don't All Think Alike: The Need for Interoperability

None of the current technologies for service discovery is a superset of the others, and none is mature enough to dominate the market. The fact that none dominates isn't a sign of poor engineering, or an excuse to sit down and wait for the "next big thing"—it's simply a sign that smart people don't necessarily tackle problems in the same way. In the long term, market pressures may result in the field being trimmed substantially. These pressures haven't yet materialized, because the ink is still drying on many of the specifications and few commercial products have emerged. Of course the service discovery approach that succeeds in the marketplace may do so for reasons quite unrelated to "best technical approach". In the short term, there are options, developers have to choose among them, and as a result of various factors, different choices will be made. This puts users in a quandary: developers using Jini may be interested in contacting UPnP services. It may be necessary to install a Jini-enabled peripheral in a network in which UPnP is the primary service discovery technology. Or a user with an SLP-enabled laptop might need to use a printer in a Salutation coffeeshop. Interoperability frameworks allow service discovery technologies to be bridged, so that clients and services of different types can interact. Without interoperability, the presence of half a dozen competing technologies will limit the usefulness of service discovery "in the large".

In this paper we concentrate on interoperability between UPnP and Jini. Despite the high-level similarities between these protocols, interoperability turns out to be quite difficult. The language-centric nature of Jini and distinct differences in *what* has to be standardized to define a service are substantial obstacles. Since Jini relies heavily on mobile code, the thing to be standardized is an interface, which specifies the methods that a Java client can expect a service implementation to provide. Jini services can make use of a wide spectrum of Java technologies, including support for audio, video, and the transfer of complex Java objects through object serialization. Further, complicated types can bleed over into the interfaces that define services, making interaction with non-Java applications quite daunting. In UPnP, on the other hand, the things to standardize are XML device and service descriptions. Unlike Jini, the protocols spoken between UPnP clients and services tend to be based on simple, primitive types such as strings, booleans, and integers. The details of device discovery, advertisement, and eventing are also quite different, meaning that bridging Jini and UPnP clients and services from scratch on a case-by-case basis places a heavy burden on programmers. Our architecture provides tools to make bridging the protocols much easier, with the key idea being to introduce service type-specific proxies that perform discovery of UPnP services on behalf of Jini clients and Jini services on behalf of UPnP clients. These proxies are essentially "virtual services" and are instantiated automatically as supported service types are detected. Enabling bridging for new service types cannot be fully automated, but in our experience the effort required per service type is quite an improvement over coding interoperability from scratch. Our architecture provides proxy skeletons that provide much of the functionality needed to support a new service type. Details of the architecture are presented in Section 4, after a look at related work and present an introduction to Jini and UPnP.

1.3. Related Work

Some initial work on interoperability for service discovery has been done, though the number of efforts to date is relatively small. Despite the importance of interoperability, much work remains, and the scarcity of

related work is due primarily to the youth of many of the technologies. A whitepaper available from the Salutation Consortium [10] describes mapping the Salutation architecture for service discovery to Bluetooth SDP. Bluetooth is an attractive target for interoperability efforts because it brings low-cost wireless to mobile devices, eliminating cables. In fact, to our knowledge none of the other groups developing service discovery technologies rule out Bluetooth interoperability. Mapping Jini, UPnP, and SLP to Bluetooth is relatively painless because PPP (and thus IP) can be run over Bluetooth and current implementations of Jini, UPnP, and SLP all target IP-based networks. Salutation is also interoperable with SLP in one direction, for discovery beyond the local network.

A Jini/SLP bridge is proposed in [6] that allows Jini clients to make use of SLP services. Properly equipped service agents advertise the availability of Java driver factories that may be used to instantiate Java objects for interacting with an SLP service. A special SLP user agent discovers these service agents and registers the driver factories with available Jini lookup services. An advantage of this architecture is that the service agents do not need to support Jini—in fact, they do not even need to run a Java virtual machine. Bridges of this sort seem to be the most appropriate way to foster operability between the various service discovery technologies. As with our architecture, some programming is required for each service type. This is generally necessary because the client/service interfaces are so different between the various service discovery approaches. Note that all of the interoperability efforts to date are unidirectional, providing interoperability in only one direction. Our architecture provides bi-directional interoperability, allowing UPnP and Jini clients and services to mingle freely.

2. Jini

Jini is a service discovery technology based on Java, developed by Sun Microsystems. Because of the platform-independent nature of Java, Jini can rely on mobile code for interaction between clients and services. *Lookup services* provide catalogs of available services to clients in a Jini network. On initialization, Jini services register their availability by uploading proxy objects to one or more of these lookup services. The proxy objects are essentially "device drivers" written in Java—they allow the client to control the service. Protocols are included for connecting to lookup services with known locations and for dynamically discovering lookup services within multicast range. Once a client has contacted a lookup service, it can search for interesting services and then download the corresponding service proxy objects. In Jini, searching is based on the type of the proxy object—generally specified using a Java interface—and on sets of descriptive attributes. For example, a client interested in discovering a remote file storage facility might search for services whose proxy objects implement the following interface:

```
public interface StorageService extends Remote {
  public boolean open(String username,
    String password, boolean newAccount)
        throws RemoteException;
  public boolean close(String username,
    String  password) throws RemoteException;
  public boolean shutdown(String username,
    String password) throws RemoteException;
  public boolean store(String username,
    String password, byte[] contents,
    String pathname) throws RemoteException;
  public byte[] retrieve(String username,
    String password, String pathname)
        throws RemoteException;
  public boolean delete(String username,
    String password, String pathname)
        throws RemoteException;
  public String[] listFiles(String username,
    String password) throws RemoteException;
  public String name() throws RemoteException;
}
```

After downloading an object implementing the above interface, the client can invoke the methods to create accounts, store and retrieve files, etc.

Garbage collection in Jini relies on leases, which allow lessors to grant access to a resource (e.g., proxy object storage on a lookup service) on a timed basis. Eventing is provided for asynchronous notification of interesting things occurring in the network—examples include the discovery of a new lookup service and the availability of a needed service. Development of clients and services in Jini is at a high-level, with extensive class support hiding the details of the multicast-based discovery and advertisement protocols, leasing, etc. from programmers. Further, RMI-based service proxies tend to reduce the need to create custom client/service protocols (e.g., based on sockets), except when performance is a concern.

Note that Jini specifically requires at least one accessible lookup service—there is no "directory-less" mode where clients and services discover each other and then interact directly. This is in sharp contrast to UPnP, where no directories are used. Jini includes other tools, such as a transaction service and a JavaSpaces implementation, which provides a Linda-like "bag-of-objects" abstraction. These also have no peer in UPnP. Code mobility allows the creation of very interesting services in Jini—the proxy object downloaded by a client may perform computation on the client side, on the service side, or both. We currently use the standard Jini distribution v1.1 from Sun Microsystems for development.

3. Universal Plug and Play (UPnP)

Universal Plug and Play is a set of protocols for service discovery under development by the Universal Plug and Play Forum, an industry consortium led by Microsoft. UPnP standardizes the protocols spoken between clients (called control points in the UPnP lingo) and services rather than relying on mobile code. Device

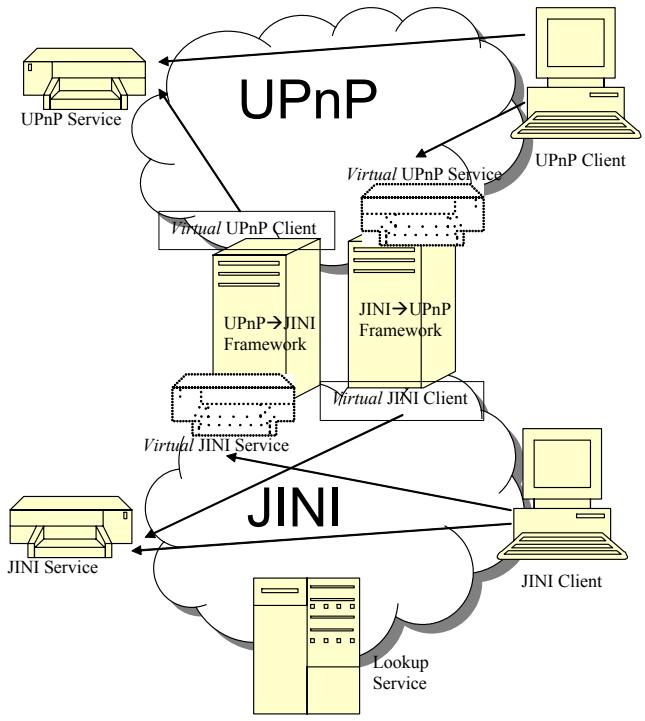

Figure 1: UPnP/Jini framework. Arrows represent client→service usage, e.g. a client can use a service when an arrow connects them. This diagram only shows one entity of each type but the framework supports multiple clients and services simultaneously.

and service descriptions are coded in XML, and a number of protocols for local autoconfiguration, discovery, advertisement, client/service interaction, and eventing—Auto-IP, SSDP (Simple Service Discovery Protocol), SOAP (Simple Object Access Protocol), GENA (General Event Notification Architecture)—are included in the specification. These protocols tend to be based loosely on existing standards (e.g., HTTP).

The UPnP specification addresses six areas. The first (numbered 0) is Addressing, and is related to device initialization—in order to interact with other UPnP entities, a device or control point must have an IP address. IP addresses will generally be provided by a DHCP server or configured statically, but for environments with little infrastructure (e.g., a home LAN), UPnP uses a protocol named Auto-IP to automatically generate non-routable IP addresses. The second area is Discovery, which allows control points to discover UPnP devices offering services of interest. Discovery also covers device advertisement, used to announce the availability of devices as they enter the network. The discovery phase provides control points with existential information, but little additional information about the discovered devices. Discovery is the concern of SSDP. Description, or the ability to inquire about device specifics (e.g., manufacturer, serial numbers, specific services offered, etc.) is the third phase. Using a URL obtained in the Discovery phase, a control point can download a document called the device description document, which fully describes a device of interest. This document is platform-neutral, expressed entirely in XML. For example, the description document for a remote file storage device might look like the one specified in Appendix A (placed there in single column format for readability).

If a control point determines, after examining a device's description document, that it wishes to interact with services provided by a device then Control is used to send the device commands and receive results. All Control interactions are via a protocol based on HTTP, called SOAP (Simple Object Access Protocol), which runs over TCP. Commands sent via SOAP are addressed to control URLs for the services provided by a device—these control URLs are specified in the device's description document, which is obtained during the Description phase. The API for a service is also expressed in XML. A partial specification of the remote file service "rfsService" is given in Appendix A (A.2).

To be informed of important state changes, UPnP control points subscribe to the event services offered by a device. This mechanism makes up the Eventing portion of the UPnP specification and is handled by the GENA (General Event Notification Architecture) protocol. Eventing consists of notifications of state variable changes. Finally, the Presentation aspect of UPnP allows devices to define a *presentation URL*, which is the location of an HTML document which provides a graphical interface to the device (a "virtual" remote control).

Unlike Jini and SLP, UPnP does not support service directories—communication between devices and clients is always direct. Thus UPnP is aimed at smaller environments where the benefits of directories are reduced due to the small number of devices typically found in the network.

We currently do UPnP development using Intel's UPnP SDK, an open source implementation of the UPnP protocols for Unix, and Siemen's Java UPnP stack**.**

4. A Jini/UPnP Interoperability Framework

In this section we discuss the design of our UPnP/Jini interoperability framework. The architecture introduces service-specific proxies that allow Jini and UPnP clients and services to mingle—UPnP clients can use Jini services and vice versa. Each new service type requires a modest amount of code to be written, but the Jini/UPnP clients and services themselves do not require modification. Our goal, as might be expected, is to reduce the per-service implementation effort as much as possible.

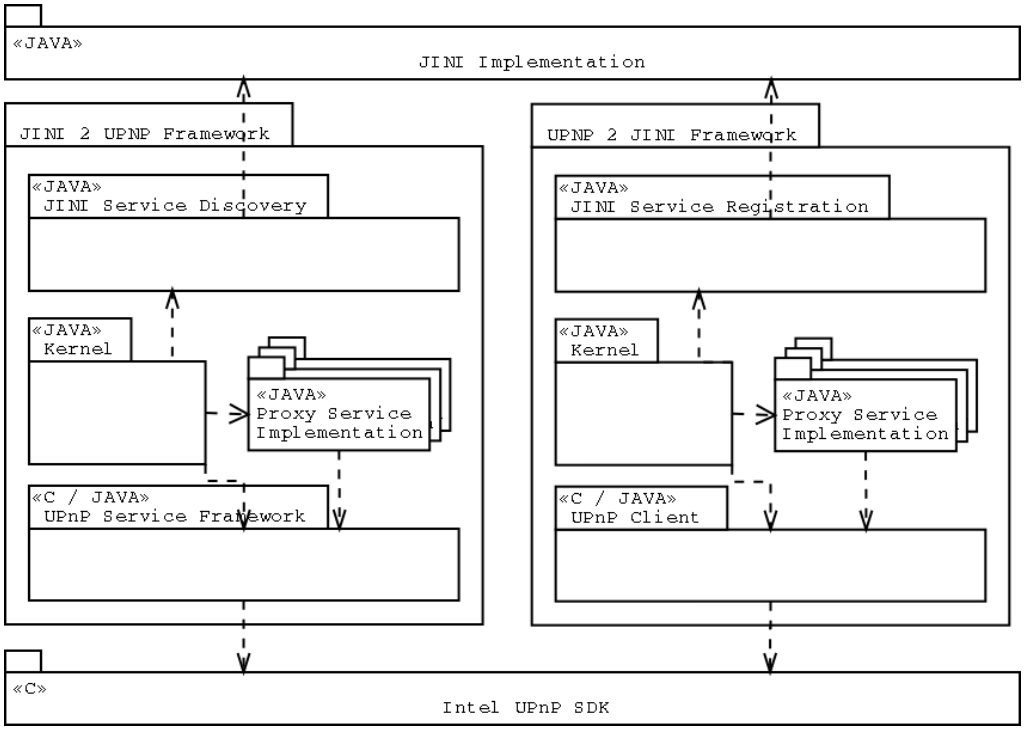

Figure 2: Jini/UPnP framework design. Arrows indicate dependencies.

To achieve this goal, we need to fool Jini clients into believing they are interacting with Jini services even when the services are in fact UPnP, and similarly for UPnP clients. The framework generates virtual Jini services, one for each instance of a recognized UPnP service. This concept is similarly applied in the other direction (UPnP client using a Jini service). This process is depicted in Figure 1. For each supported UPnP service instance, the architecture automatically generates a virtual Jini service instance and registers the instance with available Jini lookup services. For each supported Jini service instance, the architecture generates a virtual UPnP instance which performs UPnP advertisement for the service. The virtual services perform bridging, translating between Jini method calls and UPnP actions. When a UPnP or Jini service is removed from the network, the architecture automatically destroys the corresponding virtual services.

4.1. Design

The architecture is illustrated in Figure 2. The major components are:

- UPnP Service Framework, which provides basic UPnP service functionality to Java through a JNI interface to the Intel UPnP SDK. This component supports the development of proxies for new service types.
- UPnP Client, which provides basic UPnP client functions, such as sending an action, retrieving state variables, and service discovery. These functions are needed to transmit commands to a UPnP service on behalf of a Jini client.
- Jini Service Discovery, which is responsible for the discovery of known Jini service types.
- Jini Service Registration, which registers virtual Jini services with available lookup services.
- Proxy Service Implementation, which is the only part specifically dependent on the specific type of a supported service. This component is used to implement a "virtual" service, which interacts with a real service to perform service functions. A pair of proxies are required for each supported service type, one to support Jini → UPnP interoperability, and one to support UPnP → Jini.
- Kernel, which manages the other modules, coordinates discovery of services, instantiation of appropriate proxies to bridge UPnP and Jini service implementations, registration and advertisement of virtual services and garbage collection. The Kernel uses Java reflection to enumerate supported service types, eliminating the need for modifying core components when new services types are added. The Kernel also ensures that virtual services are not bridged again—e.g., a UPnP service that is made available to Jini clients should not then be bridged to make it available to UPnP clients!

The main idea behind this design is that the framework should provide as much support as possible for rapid proxy development, reducing the amount of code that must be written to support new service types. To verify this claim we developed proxies for two independently developed remote file services, one in Jini and the other in UPnP, paying attention to the effort required to get proxies running. The design and development of the proxies took one afternoon for one person. Each

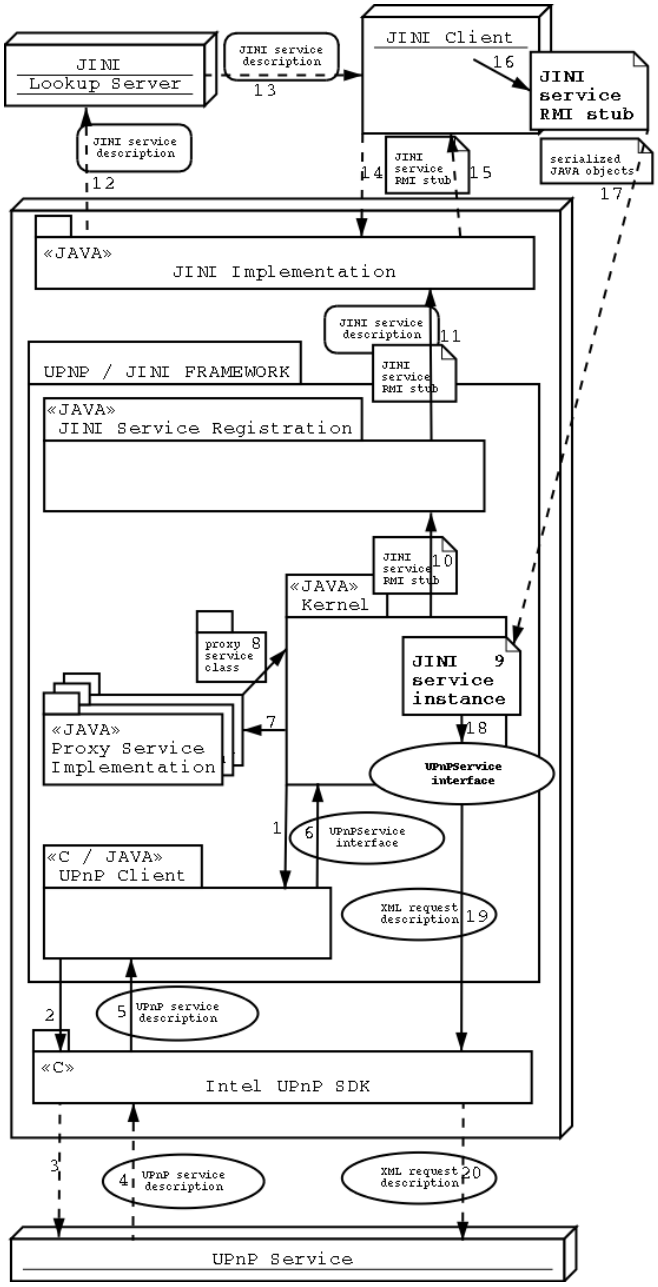

Figure 3: A Jini client interacts with a UPnP service. Plain arrows represent local method calls. Dashed arrows represent network messages.

represents about a hundred lines of Java code. This is very light development effort, considering that the Jini and UPnP services have reasonably different interfaces. As an example of these differences, the Jini interface for remote file storage contained methods to create a new account and to login into an existing account, whereas the UPnP specification uses a single action for logging into an account, which implicitly creates the account if it does not already exist. Further, the representation of a transmitted file is different. In addition to remote file storage, we have also applied our architecture to interoperability for GPS and LCD projection services.

4.2. The Details

In this section we present a case study to better explain how the framework functions. Figure 3 details how the framework works to allow a Jini client to use a UPnP service. We follow this process from initialization, to the discovery of an UPnP service, the creation and registration of a corresponding virtual Jini service, through the use of the service by a Jini client.

We elaborate on the interesting steps here. The steps correspond to the numbers in Figure 3.

1. The Kernel uses the UPnP Client component to discover available services. Only supported service types are discovered (that is, service types for which a proxy has been developed).
2. The request is forwarded to UPnP SDK.
3. Service request messages are transmitted.
4. Service description documents are retrieved.
5. The UPnP Client component parses the description documents.
6. The UPnP Client component creates a UPnPService interface to interact with the UPnP service and sends it to the kernel
7. The kernel locates an appropriate proxy service class for the discovered service.
8. The proxy service class is found.
9. The kernel creates a Jini service proxy instance.
11. A Jini service registration instance is created.
12. The service description is registered with available lookup services.
13. A client performs a service request and downloads the service's stub.

17-20. Methods invoked in the service's stub cause the client to interact with the concrete UPnP service through the virtual service.

4.3. Issues

In this section we detail some of the issues encountered when bridging UPnP and Jini. Bridging from scratch for each service type is not trivial, since the two technologies differ in several critical ways. The most important difference is service standardization. Each service type (printer, remote file service, LCD projection service, etc.) will have a standardized Java interface and a standardized UPnP XML specification, which may or may not offer equivalent functionality. Further, standard types in Java, such as vectors, hashtables, etc. tend to find their way into Java interfaces, while UPnP service specifications tend to be based on simpler types.

The other issues are:

- Service discovery: Jini uses a set of attributes to search for specific services, UPnP does not. The possible Jini attributes (manufacturer, model,

capabilities) must be derived from the UPnP XML description document, which must be downloaded before it can be examined.

- Programming languages: Only Java is available for Jini development. A number of languages are available for UPnP, notably C, C++, and Java.
- Eventing: Jini eventing is quite different from the eventing mechanism in UPnP, where state variable values are transmitted directly to clients. Equivalent functionality can be provided using a remote callback mechanism or the proxy can use a thread to poll the Jini service periodically and detect state changes.

5. Conclusions and Future Work

We have presented an architecture for Jini/UPnP interoperability. Our architecture introduces virtual services that allows Jini and UPnP clients to mingle freely. Efforts of this sort are important because in the near term, a number of service discovery technologies will compete for dominance. We are currently working on interoperability with other service discovery protocols, particularly SLP and Salutation. An open question is how to further automate the support of new service types, thus reducing the per-service programming effort even further. The code for the architecture and some sample proxy implementations is freely available. Further, a longer version of this paper that contains sample proxy code is available; the code was omitted from the current version of the paper to save space. Contact the authors for more information on these items.

References

[1] K. Arnold, R. W. Scheifler, J. Waldo, A. Wollrath, B.O 'Sullivan, The Jini Specification, 1999.

[2] Bluetooth specification, available from http://www.bluetooth.org/.

[3] "Bluetooth Extended Service Discovery Profile," http://www.bluetooth.org/specifications.htm.

[4] S. E. Czerwinski, B. Y. Zhao, T. Hodes, A. D. Joseph, R. Katz "An Architecture for a Secure Service Discovery Service,"Fifth Annual International Conference on Mobile Computing and Networks (MobiCOM '99).

[5] E. Guttman, C. Perkins, J. Veizades, M. Day, "Service Location Protocol, Version 2," RFC 2608, http://www.ietf.org/rfc/rfc2608.txt.

[6] Automatic Discovery of Thin Servers: SLP, Jini and the SLP-Jini Bridge, Erik Guttman, James Kempf, IECON, San Jose, 1999.

[7] B. Miller, R. Pascoe, "Mapping Salutation Architecture APIs to Bluetooth Service Discovery Layer," www.salutation.org/whitepaper/BtoothMapping.PDF.

[8] G. G. Richard III, Service and Device Discovery: Protocols and Programming, McGraw-Hill, 2002.

[9] G. G. Richard III, "Service Advertisement and Discovery: Enabling Universal Device Cooperation," IEEE Internet Computing, vol. 4, no. 5, September/October 2000.

[10] Salutation Service Discovery Architecture. www.salutation.org.

[11] Universal Plug and Play Specification, v1.0, at www.upnp.org.

Appendix A: Sample UPnP Remote File Service Specification

A.1 Device description

```
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <URLBase>http://sarah.remotefs.com:6599</URLBase>
  <device>
    <deviceType>urn:schemas-upnp-org:device:rfs:1</deviceType>
    <friendlyName>UPnP Remote File Server</friendlyName>
    <manufacturer>
       Remote File Services, Inc. </manufacturer>
    <manufacturerURL>http://www.remotefs.com/</manufacturerURL>
    <modelDescription>UPnP RFS</modelDescription>
    <modelName>RFS Test</modelName>
    <modelNumber>1.0</modelNumber>
    <modelURL>http://www.remotefs.com/</modelURL>
    <serialNumber>999954321</serialNumber>
    <UDN>uuid:Upnp-RFS-1_0-1212121212120</UDN>
    <UPC>123456789</UPC>
    <serviceList>
      <service>
        <serviceType>
              urn:schemas-upnp-org:service:rfsService:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:rfsService1</serviceId>
        <controlURL>/upnp/control/rfs1</controlURL>
```

```
        <eventSubURL>/upnp/event/rfs1</eventSubURL>
        <SCPDURL>/rfsSCPD.xml</SCPDURL>
      </service>
    </serviceList>
   <presentationURL>/rfspres.html</presentationURL>
</device>
</root>
```

A.2 Partial SCPD for remote file storage service

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <actionList>
    <action>
      <name>LogIn</name>
      <argumentList>
        <argument>
        <name>Login</name>
          <relatedStateVariable>A_ARG_string</relatedStateVariable>
          <direction>in</direction>
        </argument>
        <argument>
        <name>Password</name>
          <relatedStateVariable>A_ARG_string</relatedStateVariable>
          <direction>in</direction>
        </argument>
        <argument>
        <name>Connection</name>
          <relatedStateVariable>A_ARG_i4</relatedStateVariable>
          <direction>out</direction>
        </argument>
      </argumentList>
    </action>
      <name>PutFile</name>
      <argumentList>
        <argument>
       <name>Connection</name>
          <relatedStateVariable>A_ARG_i4</relatedStateVariable>
          <direction>in</direction>
        </argument>
        <argument>
       <name>Name</name>
          <relatedStateVariable>A_ARG_string</relatedStateVariable>
          <direction>in</direction>
        </argument>
        <argument>
       <name>Data</name>
          <relatedStateVariable>A_ARG_bin.hex</relatedStateVariable>
          <direction>in</direction>
        </argument>
      </argumentList>
    </action>
        …
  </actionList>
  <serviceStateTable>
    <stateVariable>
      <name>A_ARG_string</name>
      <dataType>string</dataType>
    </stateVariable>
        …
    <stateVariable>
      <name>A_ARG_boolean</name>
      <dataType>boolean</dataType>
    </stateVariable>
  </serviceStateTable>
</scpd>
```