UPnP Services and Jini Clients

Jan Newmarch
School of Network Computing
Monash University
jan.newmarch@infotech.monash.edu.au

Abstract

UPnP is middleware designed for network "plug and play". It is designed to manage devices*, unlike more general middleware such as Jini which is designed to manage* services*. Since UPnP uses more simplistic technologies than Jini, it has appeared to be easier to build and deploy UPnP devices than Jini devices. This paper shows how UPnP devces can also expose themselves as Jini devices in a fairly easy manner, and discusses the advantages of having UPnP devices available within a more general purpose service middleware.*

Keywords: Middleware, UPnP, Jini, Service oriented architecture, Embedded systems

1. Introduction

UPnP (Universal Plug and Play) is an industry standard to allow devices to be added seamlessly into a small network such as a home network [1]. The technologies underlying UPnP have been developed steadily since 1998, and UPnP is now based on the following: TCP, UDP and multicast network protocols in addition to search protocols such as HTTPU and HTTPMU and method invocation by SOAP. The UPnP consortium has standardised a number of devices such as light switches through to A/V players. However, the UPnP middleware is "device-centric" and does not contain mechanisms to share in general service-based middleware systems.

There are an increasing number of service-based middleware systems, including Jini [2] and Web Services [3]. While Web Services are currently targetted at business systems with corporate-scale discovery and registration mechanisms, Jini was designed to support both the kind of adhoc discovery mechanisms that are needed for zero configuration home systems as well as the larger scale business service systems.

Nevertheless, Jini has not made inroads into the smaller device markets, in contrast to UPnP. This paper discusses the technical reasons for this apparent failure, and shows how a UPnP device can both advertise itself not only as a UPnP service but also as a Jini service and hence can be invoked by a Jini client, with minimal overheads on the device. The advantages of this "dual personality" are discussed.

The structure of this paper is as follows: in the next section we review UPnP and in the following section do the same for Jini. The next section summarises the differences. This is followed by a more detailed discussion of mobile objects in Jini. The main results are given in Section 6, where we restrict Jini to UPnP services. We show that it is possible to embed a Jini lookup service within a UPnP service with the resultant Jini service proxy able to talk directly to the UPnP service. The following section discusses our implementation and other possibilities. We then conclude with a discussion of the value to both Jini and UPnP of this work, and a look at future work. Our contribution is that we show that it is easy for a UPnP service to be a Jini service as well, and this increases the scope of use of UPnP devices within a more general service framework.

In related work, Allard et al [9] discuss a Jini/UPnP bridge. In this system, specialised software acts as bridge between the two middleware systems. For each UPnP service instance the bridge generates a Jini service which it registers with a Jini lookup service. Calls on this Jini service are transformed into calls on the UPnP device, and returns from this device are similarly translated back into returns from the Jini service. The advantage of this method is that it does not require any invasive code in the UPnP device. The disadvantage is that it requires specific code to be written for each UPnP device type, and a bridge object to translate between method calls of the two middleware systems. We do not require additional code to be written, have no bridge object and the Jini client makes calls directly on the UPnP service without any intermediate third party.

Pota et al [10] discuss the relationship between Jini and Grid Systems. They also talk about a bridge object, but consider the possibility of the Jini service proxy talking directly to the grid service using the grid service's protocol. However, they do not discuss any implementation features of their work.

JMatos [11] is an existing Jini system that builds a special purpose Jini lookup service. The proxy for this lookup service can be completely downloaded to the client, so that as in our system there is no need to have Java to run the lookup service. Their system is designed for general purpose low-resource systems, while ours is able to take specific advantage of the tighter environment of UPnP services.

2. UPnP and Jini

UPnP is designed as language agnostic middleware particularly suited for small network-aware devices in a zero configuration environment. A typical example of such an environment is the future home, where many mundane devices such as light switches, air conditioners, etc, through to more sophicated devices such as A/V storage devices will exist, all with IP networking capability. Other possibilities include automobile systems or sensor networks which require minimal configuration.

UPnP devices advertise themselves by multicast [3]. Muticast scope typically limits advertisements to the local network. UPnP devices have no protocol for unicast advertisements beyond this scope. Clients searching for UPnP devices also make requests by multicast, and again there is no unicast mechanism.

Advertisements and searches are performed by using a protocol derived from the principles of HTTP requests, but adapted to multicast: text based messages using a small number of verbs. Advertisements provide information about services by giving basic device and service information while providing URLs for further information such as how to control the device. These protocols, known as HTTPMU and HTTPU have been prescribed by the UPnP consortium.

Services on a device have functions that can be called directly and events which can be delivered to listeners. Service functions are invoked by SOAP calls, a protocol borrowed from Web Services. In addition, UPnP devices maintain state, and changes of state are signalled by changes in state variables. These changes are notified by unicast to a list of listeners who have explicitly subscribed to state changes.

Jini is Java-specific middleware. It relies on clients able to interpret Java bytecodes. It is designed for a general purpose environment, able to take advantage of multicast and zero configuration environments and yet also has unicast mechanisms for general internet services.

Jini makes use of a service registry called a "lookup service". Services and clients find a lookup service by local multicast or by unicast to known locations. Services register themselves with a lookup service and clients ask it for suitable services. Services are stored and moved around the network as Java marshalled objects, and are downloaded to clients where they run in the client's Java virtual machine.

Typically, a client will download a proxy for the service which will communicate using a protocol such as RMI back to the service. However, Jini does not mandate any particular proxy/service structure or communication protocol between them. While it happens to be most common and convenient to use RMI proxies communicating using an RMI protocol, other possibilities can exist.

Some services can also generate events, and clients add themselves as event listeners by sending a client proxy to the service. Again, while there is much flexibility possible, RMI proxies are usually sent.

3. Comparing UPnP and Jini

While there are some common features between UPnP and Jini, there are also many differences. These are summarised in Table 1. UPnP uses fixed protocols for all operations: advertisement, discovery and invocation. On the other hand, Jini only has a fixed protocol for searching for lookup services and this protocol does *not* require Java.

UPnP has a very limited set of data types that can be used in method invocation [3]. They are the primitive types such as integers, boolean values and strings. Jini can use any serialisable Java object. This includes structured types such as arrays in addition to the full set of Java objects as well as the primitive types. In adddition, proxies can also define their own protocols and move objects appropriate to that protocol (e.g. CORBA IIOP references).

There is a cost to this flexibility, in that it requires the movement of Java objects across the network, usually from one JVM to another.

Since UPnP is language agnostic and only supports simple data types it does not need to support any form of object mobility. Jini does, because it needs to get lookup service proxies to clients and services, and service proxies to clients. Once there, a Java runtime must be present to handle the proxies. In addition. method calls that have objects as arguments or return values also require mobility - either to transport serialisable objects or to transport proxies for remote objects.

The mobile objects in a typical Jini djinn are shown in Figure 1.

4. Restricting Jini to UPnP-like Services

4.1. Restricted datatypes

If a Jini service only uses the datatypes supported by UPnP, then there is no object mobility required for method

Property	*Jini*	*UPnP*
Service adverts	via a Lookup Service (LUS)	Direct multicast
Service discovery	via an LUS	Direct multicast
Discovery protocol	fixed	fixed
Service invocation protocol	Unspecified (JRMP, Jeri, IIOP, etc)	SOAP
Object references	Java proxy objects	URLs or XML documents
Mobility	LUS proxy (to client and service)	None
	Service proxy (to client)	
	Method call arguments (to service)	
	Method call result (to client)	
	Listener registration (to service and LUS)	
	Unknown class definitions downloaded from an HTTP server	
Language	Java only	Agnostic
Data types	Any, including Java objects	Small set, mainly primitive

Table 1. Differences between UPnP and Jini

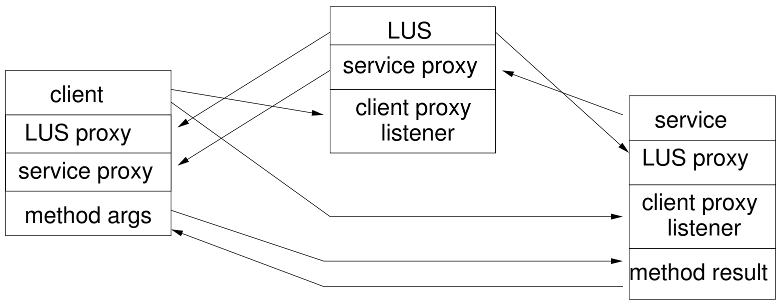

Figure 1. Mobile objects in Jini

arguments or return values. The primitive types such as integers and the content of strings can be sent across the network without requiring any Java object mobility. The parameters and return values do not require Java in either client or server for UPnP data types.

4.2. Invocation protocol

Jini does not specify the method invocation protocol. In fact, Jini 2.0 [4] has made this completely configurable so that a service proxy can use any desired invocation mechanism. In particular, a proxy can be set up to use SOAP, the UPnP invocation protocol [5]. In Jini 2.0 this configurability is used on both sides of the client/service connection: the proxy in the client may use Jeri to talk to a service also talking Jeri, or the proxy may talk JRMP to the service, and so on.

However, there is no need for the service to be implemented as a Jini service. The service endpoint just needs to understand the protocol used by the proxy. Thus a Jini service proxy using SOAP can directly invoke a UPnP service without needing a Jini version of the service at all. While this still requires Java on the client side to run the proxy, it does not require Java on the service side.

4.3. Embedded lookup service

The standard configuration of a Jini "djinn" is have separate clients, services and lookup services. In this mode, any service must be able to handle a proxy from a lookup service, requiring a Java runtime. However, it is possible for a service to run its *own* lookup service: the service could have a component that talks the Jini discovery protocol and on demand (from a client) downloads a lookup service proxy to the client. The service in this case would only need to listen and respond to the Jini multicast discovery protocols and these do not involve Java at all.

On request from a client, the service would need to download a lookup service proxy. From the service viewpoint, this is just a series of bytes obtained from serialising the lookup service proxy. While this is generally produced by a Java runtime serialising a Java object on the fly, it could also be done by a Java process serialising the object at some prior time and saving it in a file, while the service just delivers the byte stream from the file. So although a Java runtime would be required at some preparation staqe, the service would not need to be running Java in order to respond to client requests and download a lookup service proxy.

4.4. Service proxy

Such a specialised lookup service could be primed to "only" know about its own service. That is, in response to a request from a client for a service, it could answer affirmatively only if there is a match to the single service it represents. It could then return a proxy for the service. How it gets this proxy is not fixed: it could get it from the Jini service, or generate it directly itself. In the second case, this could avoid unneccessary conversation with the service.

In Java 1.4 a new class was introduced, the `Proxy` class. This class can be instantiated with a list of interfaces and will return an object implementing all of those interfaces. When a method is called on such an object it passes the call to an invocation object and, as we have already mentioned, this invocation object could make SOAP calls directly to the UPnP service.

The result of these changes is to reduce object mobility to that of Figure 2.

4.5. UPnP interfaces

A UPnP device and its services is defined by an XML document, similar in intent to WSDL for Web Services [6] but much more straightforward and better designed. When a UPnP device advertises itself it includes the URL of its device. UPnP devices use this to work out how to communicate with services. The URL for the device contains URLs for the descriptions of the services it offers and also URL endpoints for SOAP invocation of each service.

The service URLs can be used to generate suitable interfaces for Jini services, and since the service descriptions are often standardised by the UPnP Consortium they could become "well known" Jini service descriptions.

The device URL could be included as part of the specialised Jini lookup service proxy so that when it is asked for a service by a client it can examine the UPnP XML device and service description. From the service descriptions it can tell what Jini service interfaces it offers, and from the SOAP endpoints it can return a service proxy that can communicate with the UPnP services.

4.6. Class files

The final component that distinguishes the Jini and UPnP protocols is the Jini requirement to download class files that a client does not already have. With Jini 2.0 this requirement has been greatly reduced from earlier versions since the `Proxy` class can be used to generate a proxy on the fly, and as this is part of Java 1.4 the client will already have its classes and will not require a download. However, the client will need to find the classes for the SOAP invocation layer. If this becomes a common invocation class then clients may have this class in their class files in future versions of Jini. If not, then since the device has to run an HTTP server anyway, the classes can simply be placed on its HTTP server for downloading to clients.

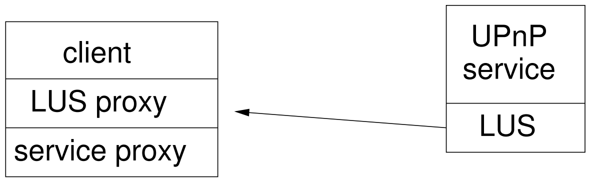

Figure 2. Reduced mobility for UPnP Services

5. Implementation

Based on the above considerations we have implemented a set of Java classes that will perform the lookup service role of Jini discovery and will return a proxy that can run entirely within a Jini client. This lookup service proxy is seeded with the URL for the device XML description. Upon request from the Jini client, it can return a Jini service proxy that implements all of the Java interfaces derived from the UPnP services. This service proxy uses SOAP to communicate directly with the UPnP services.

We have designed the service-side classes to run in the Cybergarage Java UPnP implementation [7]. Cybergaraqe allows services and devices to be written in the Java SDK and deployed within an IP network. it implements all of the UPnP protocols, so that its services can be found and invoked by UPnP clients. With the addition of our classes, the services can also be found and invoked by Jini clients.

Note that this is an invasive mechanism: the UPnP service needs to have code added to also make it into a Jini service. However, the overheads of this are small, less than 15k of class files. This is because UPnP already has to support TCP, UDP and multicast protocols and we are just adding support for another protocol and creating a lookup service proxy just in order to serialise it.

Although our implementation runs within the Java Cybergarage, this is done for convenience in serialising a lookup service proxy on the fly. It would be a straightforward matter to add this to, say, the C++ version of Cybergarage or to use a different UPnP service platform such as that provided by Intel [8]. This could reduce overheads to a few thousand bytes. However, configuration and deployment would be a more complex matter.

6. Value of work

Jini has suffered by a lack of standards work for Jini devices and device services, with a corresponding lack of actual devices. This work allows Jini to “piggyback” on the work done now and in the future by the UPnP Consortium and to bring a range of standardised devices into the Jini environment. Jini clients will be able to invoke UPnP services in addition to services specifically designed for Jini.

UPnP is a device-centric service architecture. As mentioned earlier, there are now many service architectures, including Jini, Web Services and Grid Services. These all offer environments which include both software and hardware services. However, for a client to use both a UPnP and a Jini service has previously required the client to talk both sets of protocols. Our work allows both types of service to be handled within a single framework, that of Jini.

For example, a calendar service may be purely implemented in software, but can be used to set the alarm on hardware-based clocks. If the two types of service use different middleware systems then it increases the difficulty of writing clients. Bringing UPnP into the Jini environment makes it easier to write clients which can access all the services from within Jini.

Note that it is not possible to bring general Jini services into the UPnP environment due to the probable Java dependencies of most Jini services.

7. Future Work

Our implementation uses Cybergarage for Java for the services and Jini for the clients. These both require the standard Java JDK. Neither of them will currently run on a “lightweight” Java J2ME microedition machine such as the KVM. Work is currently underway to adapt Cybergarage to this VM. This means defining a new profile at least, since there is no J2ME profile that supports multicast and the web services profile is not rich enough for UPnP.

Getting Jini to work as a client on the KVM is more difficult since the KVM does not support dynamically changing the class loader and the default loader does not support dynamic remote class loading [12]. However, the KVM specification states that an alternative default class loader may be used, presumably to allow for small machines with different file systems. It may be possible to use a Jini-aware class loader instead, but this will only work for specialised versions of the KVM, not for those currently existing. Security issues for the KVM will also become more important with such a classloader.

We have adopted an approach in which the lookup service is embedded within the UPnP device. Other authors have looked at bridges between middleware systems. A third alternative would be a specialised Jini lookup service that listens for UPnP device announcements and makes Jini service proxies of the nature that we have discussed available to Jini clients. This would eliminate the invasive nature of our current system, would not require custom coding, and would allow Jini clients to talk directly to UPnP devices without a third party in the middle. However, it would still need a separate lookup service to be running.

8. Conclusion

We have discussed ways in which UPnP services can also be made available as Jini services, with low overheads. This can be done both for UPnP devices that understand Java and for those that don't. We have an implementation for Java-aware devices. This has been shown to be of advantage both to Jini and to UPnP.

References

[1] UPnP Forum, UPnP Home Page, http://www.upnp.org.

[2] K. Arnold, et al, The Jini Specification, 2 nd ed., Reading, Mass.: Addison-Wesley, 2001.

[3] WW Consortium, Web Services Home Page, http://www.w3.org/2002/ws/

[4] UPnP, "UPnP Device Architecture", http://www.upnp.org/resources/documents/.

[5] J. Newmarch, "A Jini Tutorial", http://jan.netcomp.monash.edu.au/java/jini/tutorial.

[6] WWW Consortium, "SOAP 1.2 Protocol," http://www.w3.org/TR/soap12.

[7] WSDL 1.0 Specification, http://http://www.ibm.com/developerworks/web/library/w-wsdl.html

[8] S. Konno, "Cyberlink for java" http://www.cybergarage.org/net/upnp/java/index.html.

[9] Intel, "Intel Software for UPnP Technology", http://www.intel.com/technology/upnp/

[10] J. Allard, V. Chinta, S. Gundala, G. G. Richard III, "Jini Meets UPnP: An Architecture for Jini/UPnP Interoperability," Proceedings of the 2003 International Symposium on Applications and the Internet (SAINT 2003)

[11] Sz Pota, K. Kuntner and Z. Juhasz, "Jini Network Technology and Grid Systems" Proc. MIPRO 2003, Hypermedia and Grid Systems, Opatija, Croatia, 19-23. May 2003.

[12] Psinaptic "JMatos, Jini Network Technology for Embedded Processors" http://www.psinaptic.com/j_matos.jsp

[13] Sun MicroSystems, "J2ME Connected Limited Device Configuration", http://java.sun.com/products/cldc/index.jsp