

A Custom Lookup Service for 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. There have been several approaches to linking Jini clients to UPnP services. This paper explores another approach using a custom lookup service that listens for UPnP device announcements and makes them available to Jini clients through a suitable proxy. The paper explores aspects of lookup services (service cache managers) and service proxies that have received only limited attention so far. This approach avoids some of the complexity of earlier approaches and demonstrates a technique that can be employed to bridge between services of any type and clients supporting dynamic proxies, such as Jini.

Keywords: Middleware, UPnP, Jini, Service oriented architecture

1. Introduction

There are many middleware systems which often overlap in application domains. For example, UPnP is designed for devices in zero-configuration environments such as homes [3], Jini is designed for adhoc environments with the capability of handling short as well as long-lived services [2] while Web Services are designed for long running services across the Web [4]. There are many other middleware systems such as CORBA, Salutation, HAVi etc each with their own preferred application space.

It is unlikely that any single middleware will become predominant, so that the situation will arise where multiple services and clients exist but belonging to different middleware systems. To avoid middleware “silos”, it is important to examine ways in which clients using one middleware framework can communicate with services using another.

This issue is not new: the standard approach is to build a “bridge” which is a two-sided component that uses one mid-

dleware on one side and another middleware on the other. Examples include Jini to CORBA [6], Jini to UPnP [14], SLP to UPnP, etc. These essentially replace an end-to-end communication between client and service by an end-to-middle-to-end communication, where the middle performs translation from one protocol to the other.

While the invocation protocol is usually end-to-end, the discovery protocol may be either end-to-end as in UPnP or involve a third party. Darowski *et al* term this a *service cache manager* and a major example of this is the Jini lookup service. In an end-to-end discovery system the bridge will also need to understand both discovery protocols, while with a service cache manager the bridge will need to understand how to talk to the service cache manager.

Newmarch [7] has investigated how a Jini lookup service can be embedded into a UPnP device to provide an alternative to the bridging architecture. However, this is an invasive mechanism which requires changes to the UPnP device and cannot be easily retro-fitted into devices.

In this paper we explore the role of a service cache manager in more detail and show that under certain conditions the bridging role can be placed in the service cache manager. The resulting system does not need a separate bridge component and so is often simpler than a standard bridge system. We consider in detail how a custom service cache manager can be built that can handle both the Jini and UPnP discovery protocols and from there how a custom proxy can be used to handle the invocation protocol. The implementation of this builds on open source software and is leading to a standardisation of UPnP devices as Java interfaces within a service-oriented framework.

The principal contribution of this paper is that it proposes and demonstrates another architectural pattern that can be applied to bridge between different middleware systems.

The structure of this paper is as follows: the next section consider background from UPnP and Jini. We then follow that with a discussion of architectural considerations. Section 4 discusses our proposed system and section 5 the implementation of this system. Then we assess the system,

and in section 7 consider the value of this work and the context in which it could operate.

2. UPnP and Jini

2.1. UPnP

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 sophisticated 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 [8]. Multicast 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.

UPnP advertisements and messages are illustrated in Figure 1.

2.2. Jini

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 has been used to build systems on both an enterprise and local scale, sometimes involving hardware, sometimes just software services. Systems built on Jini include a real-time telemetry system for F1 racing cars[8th comm,

<http://www.jini.org/nonav/meetings/eighth/J8abstracts.html#Daniel>], Nedap AEOS [8 comm] and FETISH to provide travel agency services across the European Union [<http://developers.sun.com/foryourbusiness/featurearchive/fueled.html>].

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. A typical system showing mobility of objects using JRMP or Jeri is shown in Figure 2. The resultant communication protocol between objects is not prescribed by Jini, although most commonly this is either JRMP or Jeri.

Newmarch [7] discusses in detail similarities and differences between UPnP and Jini. Briefly, Jini can handle all of the UPnP data-typs, but not vice-versa; Jini relies on mobile code whereas UPnP relies on URL’s for XML documents and service end-points as shown above.

3. Architectural considerations

3.1. Service cache manager

Service cache managers are expected to store “services” in some format and deliver them to clients. The stored service can be a simple name/address pair as in naming systems such as Java RMI or CORBA, complex XML structures linked to WSDL URLs for Web Services in UDDI directories, or other possibilities. The Jini lookup service stores service proxy objects, along with type information to locate them.

When clients and services are trying to locate a service cache manager, there is often an assumed symmetry, that the client and service are searching for the same thing. In our examples above, this occurs in all of naming services, UDDI registries and Jini lookup services.

Once found though, clients and service do different things: services register whereas clients look for services. The Jini `ServiceRegistrar` for example contains two sets of methods, one for services (`register()`) and one

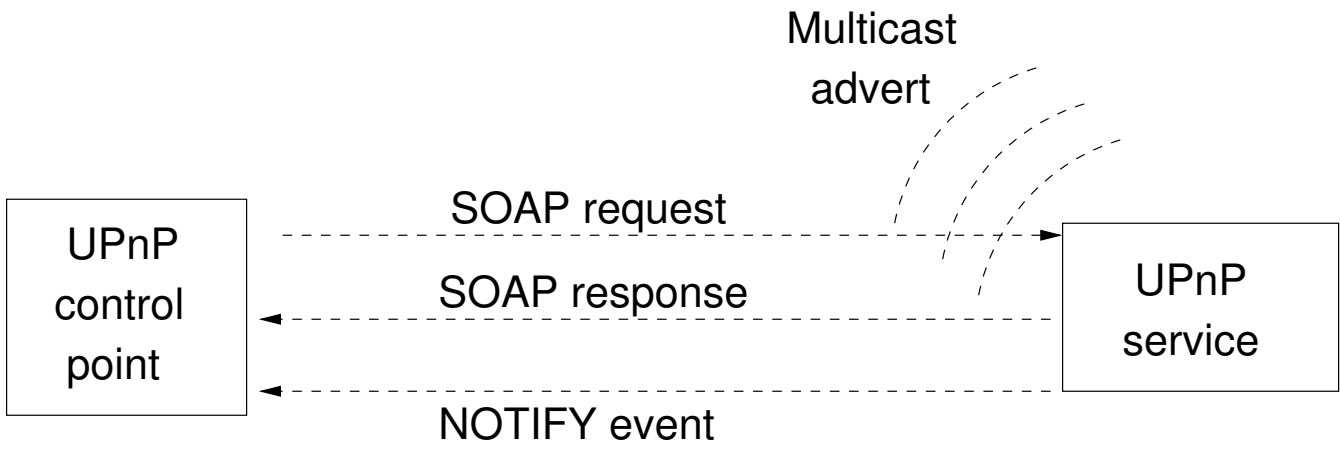


Figure 1. Messages in UPnP

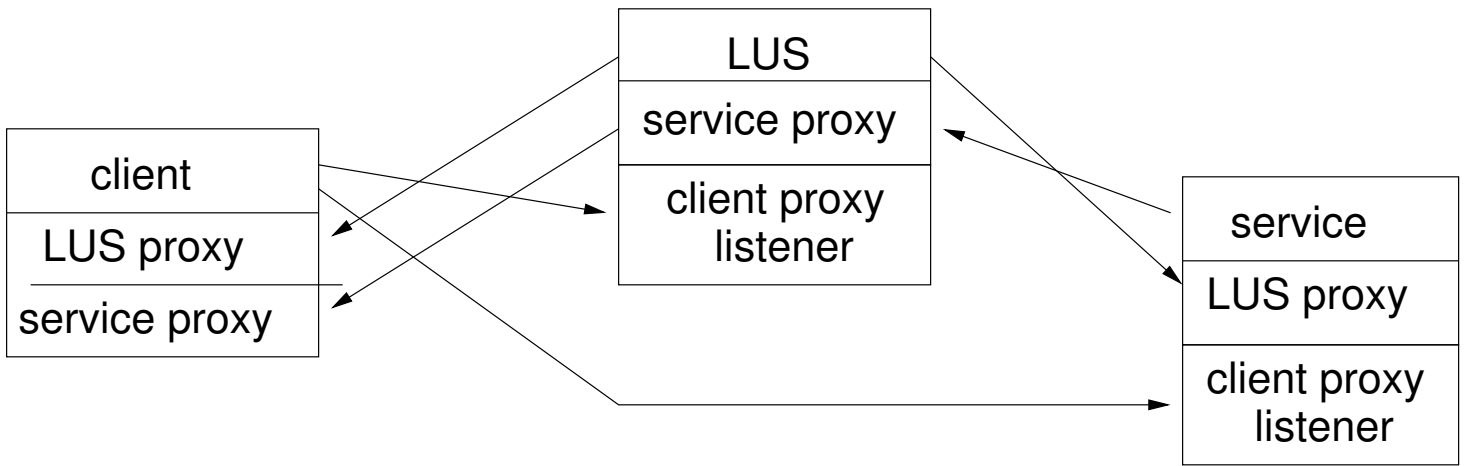


Figure 2. Mobile objects in Jini

for clients (`lookup()`). A service is searching for one in order to register itself while a client is searching for a service cache manager in order to look up services. Conceptually, there should be one protocol for services discovering caches and another for clients discovering them, with different interfaces exposed to each.

3.2. Announcement *versus* discovery

UPnP services *announce* their presence by multicast advertisements. UPnP clients *discover* services by multicast searches. Jini clients and services *discover* lookup services by multicast discovery, or a lookup service can *announce* itself by a multicast advertisement. While all of these use multicast, there is a directionality component of information flow that is usually not shown. In announcement, the information is outwards from the multicast source, while for discovery it is inwards towards the source. The flows for Jini and UPnP are shown diagrammatically in Figure 3. There is nothing inherently good or bad about either advertisement or search or about the direction of flow between any two components. It is a choice parameter in middleware that is not often discussed.

3.3. Service invocation

Many internet protocols specify all components of the protocol. For example, UPnP specifies the search and discovery protocols and also the protocol for procedure call interaction between client and service as SOAP. However, as Java RMI over Corba's IIOP instead of JRMP showed, there is no necessary link between discovery and invocation. As long as a client and service are using the same RPC protocol they can interact.

For UPnP and many systems, there is little choice. Newmarch [?] has shown that UPnP can use a protocol grounded in REST and this offers substantial memory and network savings, but this is not in the UPnP standard. However, Jini 2.0 allows a "pluggable" communications protocol. While most systems would require the client to have the communications protocol "hard coded" (or loadable from local files), Jini allows a service proxy to be downloaded from lookup service to client, and this can carry code to implement any desired communication protocol.

3.4. Bridging and alternatives

The majority of systems that allow one protocol to talk to another do so via a *bridge* at the invocation level. This is a two-sided component that understands one protocol on one side and the other protocol on the other side. Typically this runs as an independent module and acts as a client to one protocol and a service to the other. This is shown in Figure

4, with a service cache manager between client and service for one protocol. The bridge not only talks both protocols but also translates in each direction. Examples were cited earlier.

Bringing hard-coded modules into the client is an alternative that has been employed by many protocol implementations. For instance, a "CORBA to Java" converter will allow a Java client to link in IIOP communication modules, while a "WSDL to Java" converter will allow a similar client to link in SOAP communication modules. This requires explicit configuration steps on the client side.

However, as discussed in the previous section, downloadable code allows a client to use a downloaded proxy which talks a "foreign" protocol directly without requiring a translation unit.

As shown in Figure 4, to register a bridge between protocols X and Y, a bridge registers as a service of type X so that the client of type X can locate it. But as discussed earlier, a service cache manager need not be limited to a symmetric protocol, and could easily use multiple protocols. In particular, these protocols need not be of the same direction: one side of a service cache manager could search, the other side announce.

4. Proposed System

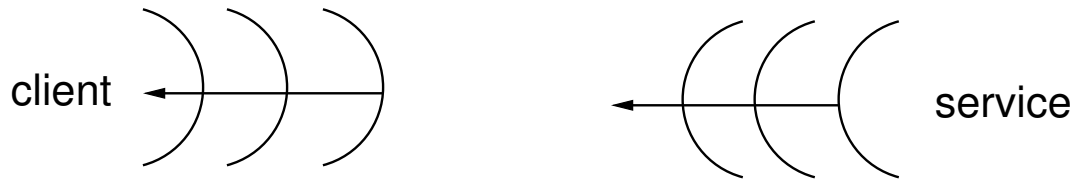
For UPnP, the format of messages and communication protocols are fixed and there is no service cache manager. For Jini, the protocol to locate a lookup service and download a proxy for it is fixed. The protocol is symmetric with respect to clients and services: they both use the same protocol to get the same type of proxy.

We propose a *two-sided lookup service* that acts between two protocols. However, unlike the bridges discussed earlier which act between the two invocation protocols, our two-sided lookup service acts between the discovery and announcement mechanisms. That is, one side finds *services* by announcements or listening for adverts using the *service* discovery protocol. The other side can be found by *clients* using adverts or listening using the *client* discovery protocol. In other words, our proposal is for a *bridging service cache manager*.

In our particular study, the bridging lookup service listens for UPnP device advertisements on one side. It can handle device registration and device farewells and will deal with device renewals, timing out if they are not received. In this respect it acts like a UPnP control point, but unlike a control point it does not send any action calls to the UPnP device or register itself for events.

The other side of the bridging lookup service handles requests from Jini clients, primarily a discovery request for the lookup service. This is shown in Figure 5.

UPnP



Jini

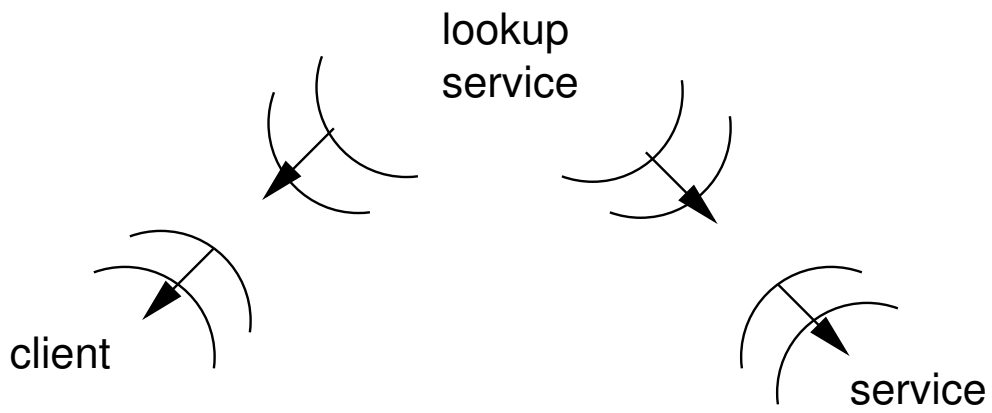


Figure 3. Directionality of information flow

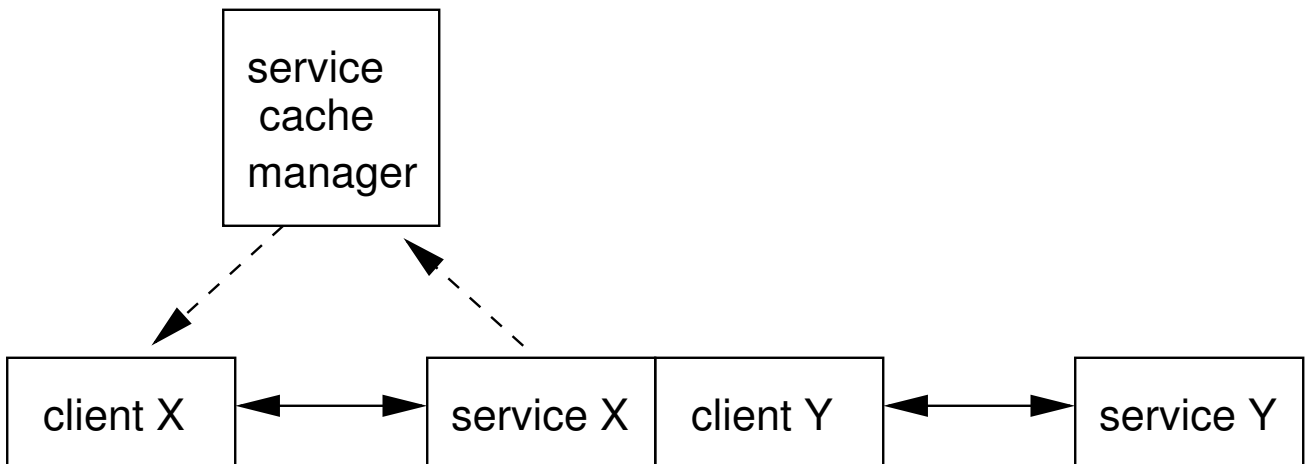


Figure 4. Bridging service protocols

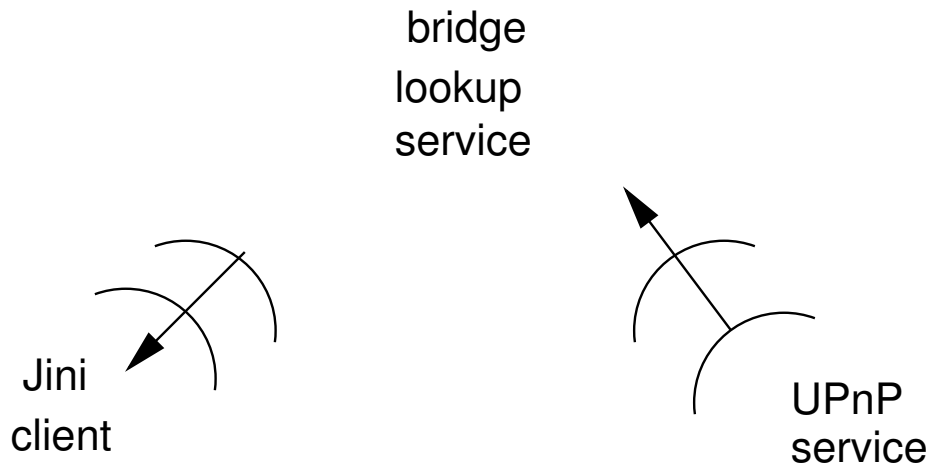


Figure 5. Bridging discovery protocols

The lookup service will act like a normal Jini lookup service as far as the Jini client is concerned and return a lookup service proxy. The Jini client will act like a normal Jini client and use this to search for a service. If the lookup service knows of UPnP devices that deliver the service, it will prepare a proxy for the UPnP device and send it back to the Jini client.

The pluggable nature of Jini proxies can be used to advantage here: since Jini does not specify invocation protocols, a proxy can be returned that uses any desired protocol. The device proxy does not use the common JRMP or Jeri protocols for Jini, but instead uses the SOAP protocol to talk directly to the UPnP device. These requests will be action requests and registrations for events. The resulting system is shown in Figure 6.

5. Implementation

There is an open source implementation of UPnP devices and control points by CyberGarage [12]. This is very closely modelled on the UPnP Device Architecture specification [8]. It exposes an API to allow a client to create a `ControlPoint` which can listen for device announcements, determine the services within the device and has methods to prepare parameters and make action calls on UPnP services. It also supports getting device information such as friendly name and registering as listener for state variable change events.

We use this in our lookup service to monitor UPnP devices and keep track of the services that are available, as well as device information.

The CyberGarage API does not treat UPnP devices and services in the service-oriented manner of Jini, RMI, CORBA or even Web Services. These specify single services using an IDL (interface definition language) of some

form. We have defined a mapping from UPnP device and service specifications into Java interfaces. For example, the Service Power service within a BinaryLight device is transformed into the Java interface

```
public service_power extends Remote {
    void SetPower(Boolean Power,
                 BooleanHolder Result)
                 throws RemoteException;
    void GetPower(BooleanHolder Power)
                 throws RemoteException;
}
```

The lookup service examines the UPnP device and service descriptions and extracts the UPnP service names and other information (such as friendly name). It stores this information along with the device URL.

In general a request by a Jini client for a service returns a proxy implementing one or more of these interfaces requested (a service might implement many interfaces). In our implementation, the proxy implements all of the services on a single device. The proxy is supplied with the device URL so that it can access the device information.

The proxy implementation also uses the CyberGarage library, but only for the control components of the CyberGarage `ControlPoint`. That is, it is used to prepare and make SOAP action calls and to register and listen for UPnP events. However, it does not listen for devices, since that is done by the bridging lookup service. When a method call is made on the service proxy it uses the control point to make a SOAP remote procedure call.

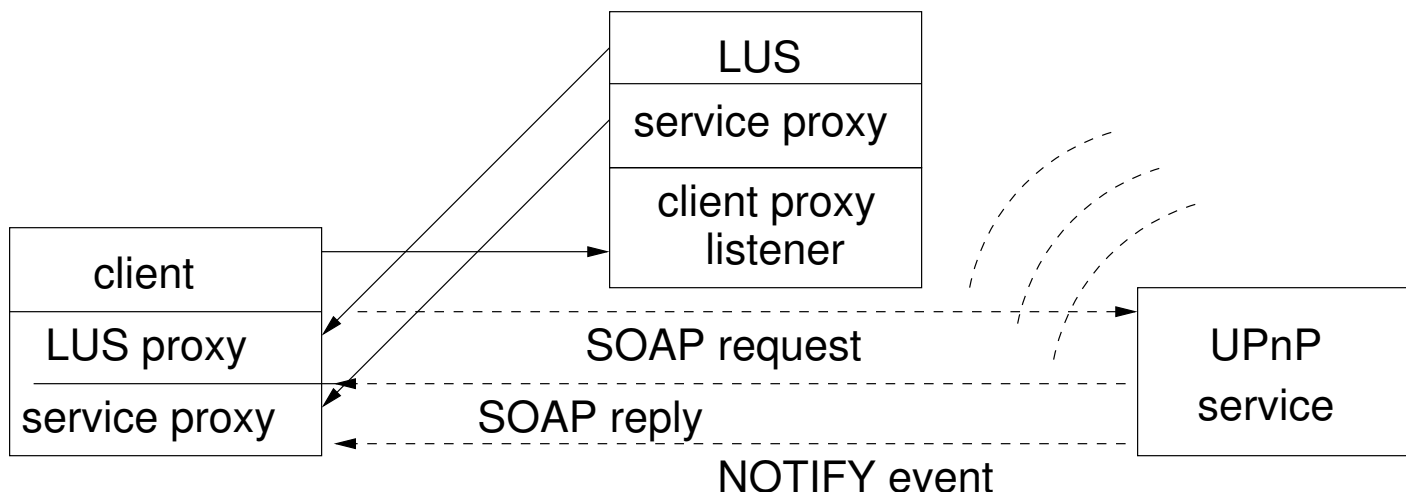


Figure 6. Jini/UPnP Lookup Service

6. Assessment

6.1. Implementation

Our current implementation relies heavily on the CyberGarage library, but only on the control point code. The device advertisement code is not used. Only a part of the control point code is used by the bridging lookup service to monitor devices while another part is used by the service proxy to make action calls and listen for events. However, the CyberGarage code is tightly interwoven, and it was not possible to use only the relevant parts. The lookup service had to import almost all of the library, as did the service proxy. It should be possible to produce a lighter-weight version for each with only the required partial functionality.

The lookup service can run with full knowledge of the classes it needs. In particular, the CyberGarage classes can be in its classpath. A Jini client cannot be expected to have such classes available, so the service proxy will have to download them using the standard Jini class dynamic class loading. This means accessing an HTTP server with the CyberGarage files and loading the classes from there. These classes are 270kbytes in size. However, the jar file also contains the source code for the package. Removing these reduces the size to 160kbytes and a specialised version could be even smaller.

CyberGarage also requires an XML parser to interpret SOAP responses. The default parser (Xerces) and associated XML API package are over 1Mbyte in size which is substantial for an HTTP download. The kXML package can be used instead, and this is a more reasonable 20kbytes and there is even a light version of this.

Nevertheless, a total of 180kbytes for downloadable code is acceptable: the reference implementation of Sun's

lookup service takes 50kbytes, for example.

6.2. Standardisation

A UPnP device and its services is defined by an XML document, similar in intent to WSDL for Web Services [11] but much more straightforward and better designed. The UPnP Consortium is in charge of defining standard devices and services.

There are now enough service-oriented bindings to Java to consider standardising a binding of Java to UPnP descriptions. The UPnP documents defining meta-device and service architecture are in [?, ?], while individual specifications can be found on the UPnP Web site. Although both UPnP and Web Services use SOAP, the UPnP specification and WSDL differ, so it is not just a simple matter of using WSDL to Java conversions.

We have defined a mapping of data types from UPnP to Java. Many cases are straightforward such as strings and signed numbers. We map these into Java `String` and object types such as `Byte`. Strings may sometimes be limited to a set of values, and we define a type which contains the string value and the set of possible values. Similarly for bounded numeric types. UPnP parameters may be `in` or `out` and we have defined holder types for the `out` parameters. Mappings of action names, service names and device names have been defined. Device information (such as friendly name) are given as a Jini `Entry`, and event information is given in a subclass of `Event`.

The proposal is under discussion in the Jini mailing list. Once agreed, this should be of value not only to this project but to other UPnP/Jini systems.

7. 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. It allows clients to use services on devices, but has no mechanism for UPnP clients to deal with software-only services since they cannot be readily expressed in UPnP. Jini clients on the other hand are agnostic to any hardware or software base, and can mix services of any type.

For example, a calendar service may be purely implemented in software, but can be used to set the alarm on hardware-based clocks. A Jini client can handle both of these, whereas UPnP by itself could only deal with the clock.

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.

8. Conclusion

We have proposed an alternative architecture to an invocation bridge between different middleware systems which uses a service cache bridge and a proxy understanding the service invocation protocol. In addition, we have automated the generation and runtime behaviour of this proxy from a UPnP specification. This has been demonstrated to give a simple solution for UPnP services and Jini clients. The technique is applicable to any client protocol which supports downloadable code and any service protocol. For example, it could be applied to a Jini/Web Service system.

References

- [1] UPnP Forum, “UPnP Home Page”, <http://www.upnp.org>.
- [2] J Waldo “An Architecture for Service Oriented Architectures” <http://www.jini.org/events/0505NYSIG/WaldoNYCJUG.pdf>
- [3] K. Arnold, et al, The Jini Specification, 2 nd ed., Reading, Mass.: Addison-Wesley, 2001.
- [4] WWW Consortium, Web Services Home Page, <http://www.w3.org/2002/ws/>
- [5] UPnP, “UPnP Device Architecture”, <http://www.upnp.org/resources/documents/>.
- [6] J. Newmarch, “A Programmers Guide to Jini”, APress, 2000.
- [7] J Newmarch “ UPnP Services and Jini Clients “ ISNG 2005, Las Vegas
- [8] UPnP Consortium “UPnP Device Architecture” <http://www.upnp.org>’
- [9] J Newmarch “A RESTful Approach: Clean UPnP without SOAP” IEEE Consumer Communications and Networking Conference, Las Vegas, 2005
- [10] WWW Consortium, “SOAP 1.2 Protocol,” <http://www.w3.org/TR/soap12>.
- [11] WSDL 1.0 Specification, <http://http://www.ibm.com/developerworks/web/library/w-wsdl.html>
- [12] S. Konno, “Cyberlink for java” <http://www.cybergarage.org/net/upnp/java/index.html>.
- [13] Intel, “Intel Software for UPnP Technology”, <http://www.intel.com/technology/upnp/>
- [14] 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)
- [15] 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.
- [16] Psinaptic “JMatos, Jini Network Technology for Embedded Processors” http://www.psinaptic.com/j_matos.jsp
- [17] Sun Microsystems, “J2ME Connected Device Configuration”, <http://java.sun.com/products/cldc/index.jsp>