BRIDGING BETWEEN MIDDLEWARE SYSTEMS: OPTIMISATIONS USING DOWNLOADABLE CODE

Jan Newmarch
Faculty of Information Technology, Monash University
Melbourne, Australia

Keywords: Middleware, UPnP, Jini, Service oriented architecture, downloadable code, proxies.

Abstract: There are multiple middleware systems and no single system is likely to become predominant. There is therefore an interoperability requirement between clients and services belonging to different middleware systems. Typically this is done by a bridge between invocation and discovery protocols. In this paper we introduce three design patterns based on a bridging service cache manager and dynamic proxies. This is illustrated by examples including a new custom lookup service which allows Jini clients to discover and invoke UPnP services. There is a detailed discussion of the pros and cons of each pattern.

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 (UPnP Consortium, 2006), Jini is designed for adhoc environments with the capability of handling short as well as long-lived services (Waldo, 2005) while Web Services are designed for long running services across the Web (WWW Consortium, 2002). There are many other middleware systems such as CORBA, Salutation, HAVi etc each with their own preferred application space, and these different application spaces will generally overlap to some extent[1].

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 middleware on one side and another middleware on the other. Examples include Jini to CORBA (Newmarch, 2001), Jini to UPnP (Allard et al., 2003), 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 (the bridge) performs translation from one protocol to the other.

[1]The middleware systems we are interested in involve discovery of services, rather than just transport-level middleware such as HTTP connecting web browsers and HTTP servers

Newmarch (Newmarch, 2005) has investigated how a Jini lookup service can be embedded into a UPnP device to provide an alternative to the bridging architecture. However, in practical terms this is an invasive mechanism which requires changes to the UPnP device and cannot be easily retro-fitted into devices.

Jini (Arnold, 2001) is apparently unique in production-quality middleware sytems with service discovery in that rather than giving some sort of remote reference to clients it downloads a proxy object into the client (the proxy is a Java object). Many of the obvious security issues in this have already been addressed by Jini. It has also been claimed that this will lead "to the end of protocols" (Waldo, 2000). In this paper we investigate the implications of downloadable code for bridging systems, and show that it can lead to many optimisations.

Some of our work can be applied to middleware systems which support downloadable code but not discovery, such as JavaScript in HTML pages.

We illustrate some of these optimisations with a Jini-to-Web Services bridge and others with Jini- to-UPnP bridge.

The principal contribution of this paper is that it proposes and demonstrates a number of optimisations that could be considered to be additional architectural patterns that can sometimes be applied to bridge between different middleware systems. The validity of these patterns are demonstrated by discussion of several example systems and through an implementation for bridging between UPnP services and Jini clients. However, the patterns do have strong requirements on the client-side middleware: it must be possible to dynamically download code to clients and to dynamically determine the content of this downloaded code.

The structure of this paper is as follows: the next section discusses some general properties of bridging systems and the following section discusses downloadable code in this context. Section 4 introduces the first of three optimisations, one for transport-level bridging. Section 5 considers service cache management and the following section applies this to the second optimisation, for service-level transport. This is followed by a section on device-level optimisation. Successive sections deal with event handling and the implementation of a Jini-UPnP bridge based on these principles. We then assess the proposals and consider the value and generality of our work, before a concluding section.

Background knowledge of Jini may be found in Newmarch (Newmarch, 2001) and on the UPnP home site (UPnP Consortium, 2006).

2 BRIDGING

Nakazawa et al (Nakazawa et al., 2006) discuss general properties of middleware bridges. They distinguish three features

- Transport-level bridging concerns translation between two invocation protocols where a client makes a request of a service. Examples of invocation protocols include SOAP and CORBA's IIOP. Transport-level bridging is concerned with translating from the invocation of a request to its delivery, and also between any replies.
- Service-level bridging involves the advertisement and discovery of services. Examples of discovery protocol include CORBA's use of a Naming service and UPnP's Simple Service Discovery Protocol.
- Device-level bridging concerns the semantics of services.

Transport level bridging includes translating between the data-types carried by each protocol. For example for Web Services using SOAP, these are XML data-types while for Java RMI using JRMP these are serialisable Java objects. There are usually problems involved in such conversions. Vinoski (Vinoski, 2005) points to the mismatch between Java data-types and XML data-types. While he goes on to examine the consequences for JAX-RPC, the same issues cause problems converting from SOAP data-types to Java objects on JRMP. Newmarch (Newmarch, 2005) discusses the mismatch between UPnP data-types and Java objects and concludes that the UPnP to Java mapping is generally okay but the opposite direction is not. There is no general solution to the data-mapping problem, and indeed the use of the so-called ”language independent” XML in some middleware systems appears to have exacerbated this. Services where the data-types are not convertable cannot be bridged. This paper does not address this issue.

While the transport protocol is usually end-to-end, the discovery protocol may be either end-to-end as in UPnP or involve a third party. Dabrowski and Mills (Dabrowski and Mills, 2001) term this third-party a service cache manager (SCM). Examples of such a manager are the Jini lookup service, the CORBA and RMI Naming service and UDDI (although this does not seem to be heavily used). The implications for service-level bridging involve the discovery protocol: in an end-to-end discovery system the service-level bridge will need to understand how to talk directly to services and/or clients, while with a service cache manager the bridge will need to understand how to talk to the service cache manager.

Device-level bridging concerns the meaning of “service” in different middleware systems, and how services (and devices) are represented.

In general a bridge system will look like Figure 1.

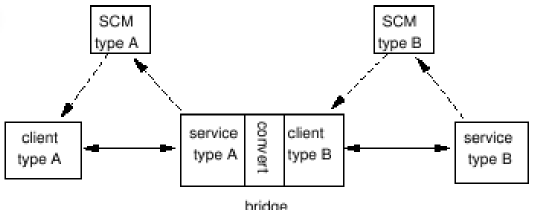

Figure 1: Typical bridge system.

3 DOWNLOADABLE CODE

There are many examples where code is downloaded from one computer to execute in another. These include JavaScript in HTML pages, Safe-Tcl (Levy et al., 1997) and Erlang (Brown and Sablin, 1999). Jini as a service-oriented architecture makes use of RMI to download a proxy object representing a service into a client. This changes the nature of the client/service transport protocol since that is now managed by the proxy object, not by the client-the client just makes local calls on the proxy. The Java Extensible Remote Invocation framework (Jeri) in Jini 2.0 allows the proxy and service to use any protocol that they choose.

Proxy/service communication in Jini can be represented in Figure 2.

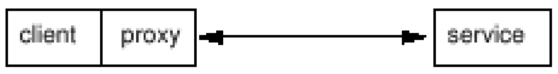

Figure 2: Proxy communication.

The pattern of communication of Figure 2 can also be employed by JavaScript using the Ajax extensions (Garrett, 2005), and is used by Google Maps and Google Mail for example, although the communication is restricted to HTTP calls.

4 OPTIMISING TRANSPORT-LEVEL BRIDGING

Transport-level bridging involves the bridge receiving messages from a client using the client's transport protocol, translating them into messages for the service and sending them using the service's transport protocol. Responses are handled in a similar way.

Many internet protocols specify all components of the interaction between clients, services and service cache managers. 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 was shown by Java RMI over CORBA's IIOP instead of JRMP, and also by CORBA's use of Naming and Trader services, there is no necessary link between discovery and invocation. As long as a client and service are using the same invocation protocol they can interact directly.

For UPnP and many systems there is little choice since the invocation protocol is fixed by the middleware specification. 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 a lookup service (service cache manager) to a client, and this can carry code to implement any desired communicaration protocol.

In a similar but less flexible way, the Ajax `XMLHttpRequest` object can exchange any type of data with its originating service. Usually this is XML data, but could be other types such as JSON (JSON, 2006)

In the most common situations, the service proxy communicates with its bridge service. However, a transport-level bridge is just there to translate and communicate between the client and the service. If the code to do this translation is moved into the proxy, then the transport-level component of the bridge service becomes redundant. That is, the client makes local calls on the proxy, which makes calls directly to the service using the service's transport protocol. One leg of the middleware has been removed. This is illustrated in Figure 3.

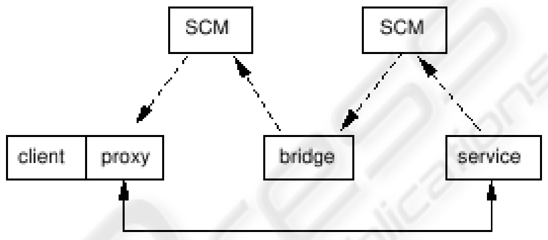

Figure 3: Removing one transport step.

This optimisation improves performance by

- removing one serialisation step
- removing one deserialisation step
- removing one network transport leg

In addition, the conversion to the destination protocol is performed once at the client-side. There are some systems such as that of Nakazawa et al (Nakazawa et al., 2006) in which the bridge performs conversion from source data-types to an intermediate "standard" type and from there to the destination type. This (or even just conversion from source transport data-types to destination transport data-types) introduce possibilities for semantic problems which are mitigated by a single conversion step at the client-side.

This pattern has been used by Newmarch (Newmarch, 2006) to show how a Jini client can communicate with a Web Service. The proxy uses SOAP, the transport protocol for the Web Service. The conversion from Java data-types to XML data-types is performed by the JAX-RPC package (which cannot do a perfect conversion job, as mentioned earlier). The role of the bridge is just there to advertise the Web Service to the Jini federation and to upload a proxy to the Jini lookup service.

This pattern can also be used by Jini clients to talk to CORBA services, since Jini can directly generate proxies that use IIOP.

Casati (Casati, 2006) shows how JavaScript downloaded into a browser can talk directly to Web Services instead of the more usual HTML-Servlet-Web Service (or similar) bridge (as typified by the web site `www.xmethods.com`). Casati employs the `XMLHttpRequest` object which allows a browser to communicate with an HTTP server asynchronously. This is usually used to exchange data between the browser and original page server. But as Web Services typically use SOAP over HTTP, Casati gives JavaScript for the object to be used as a proxy to talk directly to the Web Service.

In a later section we discuss how we use this pattern for a Jini client to talk to a UPnP service.

5 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 for clients (`lookup()`). UDDI similarly has two sets of messages, but there are more of them since UDDI has a more complex structure (Bellwood, 2002). Conceptually, there should be one protocol for services discovering caches and another for clients discovering them, with different interfaces exposed to each.

6 OPTIMISING SERVICE-LEVEL BRIDGING

The standard bridge acts as a client to one discovery protocol and as service to the other. For example, in a Jini/UPnP bridge (Allard et al., 2003) UPnP device advertisements are heard by a bridge acting as a UPnP control point, which re-advertises the service as a Jini service. In addition, it also acts as a transport-level bridge.

As a second optimisation we propose folding the service cache managers into the bridge, to just leave service-level bridging as in Figure 4.

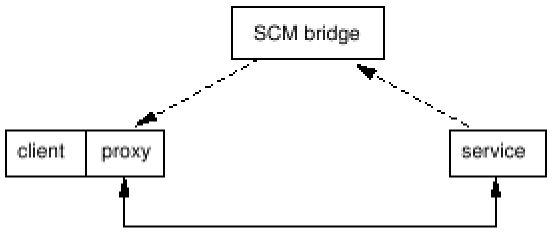

Figure 4: Optimised service-level bridging.

As an illustration of this, we have built a lookup service as a service-level bridge which 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 service-level bridge handles requests from Jini clients, primarily a discovery request for the lookup service.

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 be a normal Jini client and uses the lookup service to search for a service using the standard Jini API. 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.

This optimisation is only useful in conjunction with the first one. Transport-level bridging or its replacement will still need to be in place. If there is no replacement then little is gained by separating transport-level and service-level bridging. However, when the transport-level bridge is replaced by a smart proxy then it is possible to just keep the service-level bridge.

This is at present a practical restriction on the applicability of this pattern, since there do not appear to be many middleware systems in practical use apart from Jini that support both downloadable proxies and discovery services. However, this could be expected to change with future development of more advanced service oriented frameworks (for example, see Edwards (Edwards et al., 2005)).

7 DEVICE-LEVEL BRIDGING

Different middleware systems have different basic ideas of services. Many systems such as CORBA, Jini and WebServices only have the notion of services. Others like UPnP and Bluetooth have devices. UPnP devices contain a number of services (and possibly other devices, recursively).

The different systems give different meanings to discovery. For example, the UPnP on/off light is a `BinaryLight` device containing a `SwitchPower` service. Jini has no concept of `BinaryLight`'s and can only look for a `SwitchPower` service. So a Jini client cannot search for a binary light device but only some subset (as a collection) of the service interfaces offered. On the other hand, UPnP advertises the binary light device and the services, but with separate messages for each service, rather than as a group. UPnP devices usually only have one service although some may have more. For example, an internet gateway device may have several services and embedded devices. This device has a total service list of `Layer3Forwarding`, `WANCommonInterfaceConfig`, `WANDSLLinkConfig` and `WANPPPConnection`. In general, a Jini service may implement a number of service interfaces, and a Jini client may request a service that simultaneously implements a number of interfaces.

In the case of UPnP, services are described by XML documents, while Jini services are described by Java interfaces. We have defined a standard mapping from UPnP services to Jini services. For example, the UPnP service description for a `SwitchPower` service is

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:...">
   ...
   <actionList>
    <action>
    <name>SetTarget</name>
      <argumentList>
        <argument>
          <name>newTargetValue</name>
          <relatedStateVariable>Target
          </relatedStateVariable>
          <direction>in</direction>
        </argument>
      </argumentList>
    </action>
    ...
  </actionList>
  ...
</scpd>
```

Our mapping translates this into the Java interface (along with a suitable definition of `Target`)

```
public interface SwitchPower
       extends Remote {
   void SetTarget(Target newTargetValue)
           throws RemoteException;
   ...
}
```

The service-level bridge will need to be able to translate from one representation to the other. A direct approach is to store a table mapping each service. In the case of UPnP and Jini, the table would just hold UPnP service names matched to Java class files. At this stage, the service-level bridge will be responsible for creating the proxy, and to do this it needs the class files for the service interfaces. While it would be fine for the bridge to have class files for the set of "standard" devices and services maintained by the UPnP Consortium, it would not allow for new, unknown services to be managed.

New UPnP services would require the service-transport bridge to examine in detail the UPnP service description and generate source code for the Java interface. Then compile this on the fly using a local Java compiler (such as `javac` or Kirby's dynamic compiler (Kirby, 2005)). This is similar to dynamic compilation of JSP and servlets by servlet engines such as Tomcat. The resultant class files can be cached against repeated use.

Similar mechanisms may be needed for bridging between any middleware systems where new and unknown services may be presented to the service-level bridge. This will depend on what information is required by the bridge in order to create a proxy.

8 OPTIMISING DEVICE-LEVEL BRIDGING

In the architecture proposed so far, the service-level bridge needs to be able to generate a proxy to represent the original service. For a Jini client, this requires class files on the lookup service for the Java interfaces, and for unknown service types these will need to be generated by the bridge. This will involve detailed introspection of the service descriptions and use of a Java compiler. While dynamic compilation of JSP pages demonstrates that this is feasible, it nevertheless has overheads.

The Jini client on the other hand has to know the service interface, otherwise it cannot ask for a service proxy. So if knowledge of the Java interfaces can be deferred to the client side, then it just becomes a lookup of already instantiated classes. The name of the interface is all that is required for the client to find the interface class[2] .

The Jini lookup service already downloads a proxy to the client to represent it. This has not been shown in the figures so far as it is a Jini-specific (but standard) detail. Usually this proxy just makes remote calls back to the lookup service. However, just like any downloaded code, the proxy can be designed to perform any functions on the client side (subject to security constraints). In particular, on a lookup operation the proxy could just pass back to the lookup ser-

[2]The client has to know the interfaces it is interested in. It should not know the implementation classes. This is addressed in the implementation section.

vice enough to allow a match to be made, and on success the lookup service could pass back just enough for the lookup service's proxy to create a proxy for the original service. In the case of a UPnP/Jini bridge, the minimal information is the names of the interfaces required, and the returned information just needs to be the URL of the UPnP device description. These are enough for a proxy to be created on the client-side that can talk to the UPnP service. See Figure 5 for the final system.

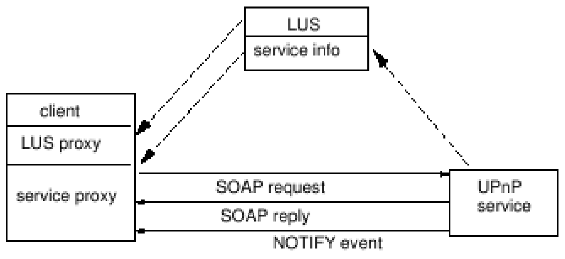

Figure 5: Optimised service-level bridging.

9 EVENT HANDLING

The discussion so far has used the remote procedure call paradigm. However, there are other possibilities such as an asynchronous callback mechanism where the service makes calls back to the client. This is easily handled by the proposed systems, as the proxy just registers itself as the callback address.

10 IMPLEMENTATION OF OPTIMISED JINI-UPNP BRIDGING

There is an open source implementation of UPnP devices and control points by CyberGarage (Konno, 2006). This is very closely modelled on the UPnP Device Architecture specification (UPnP Consortium, 2006a). It exposes an API to allow a client to create a `ControlPoint` which can listen for device announcements, to determine the services within the device and it 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 treats UPnP devices and services using a DOM-oriented model, unlike the SOA-oriented manner of Jini. We use the UPnP to Java mapping discussed earlier to translate between the two representations.

In our implemention, we use the Java `Proxy class` to give a dynamic proxy. This proxy implements all of the services on a UPnP device that are requested by the client. The proxy is supplied with the device URL so that it can access the device description. This description contains the URLs for action calls, for registering listeners and for the presentation. The Jini proxy requires an invocation handler. We use the CyberGarage classes to build a generic handler to deal with SOAP calls to the device. The CyberGarage classes and this handler are downloaded from the bridge to the client. This avoids the need for the bridge to know the service interfaces at all and allows the client to only know the service interfaces.

The proxy implementation 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.

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 has to import almost all of the library, as does the service proxy. It should be possible to produce a lighter-weight version for each with only the required partial functionality.

11 ASSESSMENT

Any "optimisation" often has both positive and negative sides. We try to offer a balanced viewpoint on the advantages and disadvantages of our pattern

11.1 Transport-level Optimisation

In transport-level optimisation, we place the code to perform service invocation directly in the proxy downloaded to the client. The principal advantages of this are

- perfomance improvement by removing one serialisation step

- perfomance improvement by removing one deserialisation step
- perfomance improvement by removing one network transport leg
- reducing the risk of semantic mismatches between client and service data-types by reducing the number of data conversion steps.

The ma jor disadvantage is that code has to be downloaded to the client that is capable of talking directly to the service. This is generally downloaded from an HTTP server. Some examples follow

- Casati(Casati, 2006) gives JavaScript that can be downloaded to a web browser such as Firefox or IE that can make function calls on Web Services. This requires just 10kbytes of JavaScript source code. This relies on the extensive libraries and support within the browsers for many of the library calls made.
- Newmarch (Newmarch, 2006) discusses a Jini proxy that can make function calls on Web Services. The particular implementation used there makes use of the Apache Axis objects `Call`, `QName` and `Service`. These classes and all the classes they depend on are substantial in size–over 900kbytes. There are clear redundancies in this: for example, there are many classes which deal with WSDL document processing, and this is not needed by the proxy.
- For the Jini UPnP proxy discussed here, the CyberGarage classes are used. 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 much more reasonable 20kbytes and there is even a light version of this. This gives a total of 180kbytes which is acceptable for any Jini client–the reference implementation of Sun's lookup service takes 50kbytes just by itself.

The actual amount of code downloaded depends on the complexity of the proxy and the degree of support that already exists in the client. These three examples show variations from 10kbytes to nearly 1Mbyte.

11.2 Service-level Optimisation

The standard bridge requires upto two service cache managers, one for each discovery protocol. In addition, the bridge has to act as a client to discover the original service and as a service to advertise to the original client. Service-level optimisation reduces this to two halves of two SCMs: one half to listen to service adverts, the other half for the original client to discover the service. UPnP does not have an SCM and control points listen directly to service adverts, which reduces the savings somewhat.

On the downside, it is necessary to write parts of service cache managers. Although this is not inherently difficult, knowledge of how to do this and API support by middleware systems is not so widespread as for writing simple clients and services. Jini has the necessary classes, but there are no tutorials on how to write a lookup service. CyberGarage has support for control points, but this is tightly woven with the device code and so contains redundant code.

In addition, the need to possibly perform introspection on service descriptions, to generate appropriate client-side definitions and to compile them are disadvantages.

11.3 Device-level Optimisation

This optimisation gains by removal of some code (introspection, generation of interfaces and compilation) completely. On the other hand, code to generate the proxy is just moved into the client. In the case of Jini, most of this code is already present in the client from the Jini libraries and does not represent much of an overhead. For other systems it may be more costly.

11.4 Generality

The design patterns discussed in this paper rely on a number of properties of the two middleware systems in order to be applicable

- it must be possible for a service cache manager to be used in each middleware system. In practise this is not an onerous provision and it can be applied even to systems such as UPnP which do not require an SCM.
- There must be a (sufficiently good) mapping of the datatypes from service system to client system. This allows UPnP services to be called from Jini clients, but would limit the scope of Jini services that could be invoked by UPnP clients. As another example, the flexibility of XML data-types means that it should be possible to mix Jini clients with Web Services, and Jini services with Web Service clients.
- It must be possible to download code from the SCM to run in either the client or service. In our case study, we have downloaded code to the client that understands the service invocation protocol, but it would work equally well if code could

be downloaded to the service that understands the client invocation protocol. Without this, the recipient would already need to know how to deal with a foreign invocation protocol, which would largely defeat the value of the pattern.

The third point is the most difficult to realise in practise. Many languages support dynamic code execution: most interpreted languages have an equivalent of the `eval()` mechanism, through to dynamic linking mechanisms such as dynamic link libraries of compiled, relocatable code.However, the only major language supporting dynamic downloads of code across a network appears to be Java, and the principal middleware system using this is Jini. Given some level of dynamic support, adding network capabilities to this is not hard: the author wrote a few pages of code as proof of concept to wrap around the Unix C `dlopen()` call to download compiled code across the network into a C program.

12 VALUE OF WORK

The value of mixing different middleware systems can be seen by a simple example. Through UPnP, various devices such as hardware-based clocks and alarms can be managed. A stock exchange service may be available as a Web Service. A calendar and diary service may be implemented purely in software as a Jini service. Using the techniques described in this paper, a Jini client could access all of these. Acting on events from UPnP clocks to trigger actions from the Jini diary the client could query the Web Service stock exchange service and ring UPnP alarms if the value of the owner's shares has collapsed.

In addition to extending the use of clients and services, there are also some side benefits:

- 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. Work is ongoing within the UPnP Consortium to bring WSDL descriptions into the UPnP world. Jini clients on the other hand are agnostic to any hardware or software base, and can mix services of any type.

Both middleware systems have limitations–in the case of Jini, in the types of services that can be accessed, and in the case of UPnP, in the range of services that can be offered. Other middleware systems will have similar limitations. For example, Web Services tend to deal with long-lived services at well-known addresses whereas Jini can handle transient services

13 CONCLUSION

We have proposed a set of alternative architectures to bridge between different middleware systems which uses a service cache bridge and a downloadable proxy understanding the service or client invocation protocol. In addition, we have used this between Jini and UPnP and 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 techniques are applicable to any client protocol which supports downloadable code and any service protocol.

REFERENCES

Allard, J., Chinta, V., Gundala, S., and Richard III, G. G. (2003). Jini meets upnp: In *Proceedings of the Applications and the Internet (SAINT)*.

Arnold, K. (2001). *The Jini Specification*. Addison-Wesley.

Bellwood, T. (2002). Uddi version 2.04 api specification. Retrieved July 7, 2006 from `http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm`.

Brown, L. and Sablin, D. (1999). Extending erlang for safe mobile code execution. In *Lecture Notes in Computer Science, vol 1726*.

Casati, M. (2006). Javascript soap client. Retrieved July 7, 2006 from `http://www.codeproject.com/Ajax/JavaScriptSOAPClient.asp`.

Dabrowski, C. and Mills, K. (2001). Analyzing properties and behavior of service discovery protocols using an In *Proc. Working Conference on Complex and Dynamic Systems Architecture*.

Edwards, W. K., Newman, M. W., Smith, T. F., Sedivy, J., and Izadi, S. (2005). An extensible set-top box platform for home media applications. *IEEE Transactions on Consumer Electronics*, 4(51).

Garrett, J. J. (2005). Ajax: a new approach to web applications. Retrieved July 7, 2006 from `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

JSON (2006). Json in javascript. Retrieved July 7, 2006 from `http://www.json.org/js.html`.

Kirby, G. (2005). Dynamic compilation in java. Retrieved July 7, 2006 from `http://www-ppg.dcs.st-and.ac.uk/Java/DynamicCompilation`.

Levy, J. Y., Ousterhout, J. K., and Welch, B. B. (1997). The safe-tcl security model. Technical report, Sun Microsystems. Retrieved July 7, 2006 from `http://research.sun.com/technical-reports/1997/abstract-60.html`.

Nakazawa, J., Edwards, W., Tokuda, H., and Ramachandran, U. (2006). A bridging framework for universal interoperability in pervasive systems. In *ICDCS*. Retrieved July 7, 2006 from `www-static.cc.gatech.edu/~keith/pubs/icdcs06-bridging.pdf`.

Newmarch, J. (2001). *A Programmers Guide to Jini*. APress.

Newmarch, J. (2005). Upnp services and jini clients. In *ISNG, Las Vegas*.

Newmarch, J. (2006). *Foundations of Jini 2 Programming*. APress.

UPnP Consortium (2006). Upnp home page. Retrieved July 7, 2006 from `http://www.upnp.org`.

Vinoski, S. (2005). Rpc under fire. *IEEE Internet Computing*.

Waldo, J. (2000). The end of protocols. Retrieved July 7, 2006 from `http://java.sun.com/developer/technicalArticles/jini/protocols.html`.

Waldo, J. (2005). An architecture for service oriented architectures. Retrieved July 7, 2006 from `http://www.jini.org/events/0505NYSIG/WaldoNYCJUG.pdf`.

WWW Consortium (2002). Web services home page. Retrieved July 7, 2006 from `http://www.w3.org/2002/ws`.