A RESTful Approach: Clean UPnP without SOAP

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 uses technologies developed specifically for UPnP but also borrows technologies such as SOAP from Web Services for remote procedure calls. SOAP has been criticised as inappropriate for Web Systems and this paper shows that these criticisms are equally applicable for UPnP. It shows how mechanisms built on REST principles can produce smaller, faster and simpler UPnP systems.

Keywords: Middleware, UPnP, network protocols

1. Introduction

UPnP 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 SOAP [2] is included in this as a remote procedure call (RPC) mechanism.

SOAP has been borrowed as an RPC mechanism from Web Services [3]. However, there is a strong community who have criticised SOAP on technical grounds in the arena of Web Services [4, 5]. The general opinion within this critical group is that SOAP is a poor implementation of twenty-year old technology, and is neither done well nor in a manner fitting the intended application area, and offers no novel features to justify its adoption.

This paper considers these criticisms as applied to using SOAP within UPnP. It comes to a similar conclusion: SOAP is not appropriate to UPnP and represents a poor technical solution to RPC for UPnP control points and devices. It presents a solution based on REST principles [6] that is substantially more lightweight for devices and is generally a simpler and "cleaner" model.

The next section reviews SOAP as an RPC technology. We follow that with a critique of the technology from the viewpoint of REST. Next we examine how UPnP uses SOAP, and in section 5 suggest a simpler alternative based on REST principles. Section 6 gives a comparison of the two methods.

2. SOAP

A remote procedure call (RPC) is a mechanism whereby an application can make calls on a remote, networked service by making what appear to be local procedure calls. The RPC layer translates this local call on a client-side *proxy* into TCP or UDP packets that are forwarded to a server-side *stub* where it executes a procedure on the server. The result is returned back to the proxy where it is presented as the result of the local call.

There are many RPC systems, from Sun's RPC (renamed as IETF ONC) [7], DCE [8] and COM+ [9] through to O/O versions such as CORBA [10] and mobile object versions such as Java RMI [11]. SOAP (Simple Object Access Protocol) is neither simple nor object-based and corresponds to Sun's RPC in nature.

The WWW Consortium has standardised SOAP and has also defined a protocol transport, namely SOAP over HTTP [2]. This uses HTTP POST messages to convey a SOAP procedure call (which is an XML document) and uses the HTTP response to return another XML document containing the procedure call results.

3. REST critique of SOAP

REST was invented by Fielding as a term to describe the "ideal" architecture of the Web as a means of delivering documents over HTTP [6]. The criticisms of SOAP include [4, 5]

- SOAP documents have their own protocol of procedure calls, which is layered over HTTP. This increases the complexity of SOAP procedure calls as against simple document fetches using GET and POST. It is also more highly layered than other RPC protocols, which typically sit directly above TCP or UDP, which may increase the possibility of system errors

- The semantics of GET versus POST are that a document request using GET is not intended to make state changes on the server, whereas a request using POST is expected to do so. SOAP requests must use POST since they are uploading an XML document, but there is no requirement that a SOAP request be a state-change request. This blurs the meaning of POST. While apparently trivial, this removes a weapon from the armory of firewall controllers who have been able to distinguish the intent of a request by its HTTP type
- A SOAP response may be a complex data type representing several “objects”. Alternatively, it could be a simple data type or a URL representing a SOAP server (which is *not* a SOAP object). That is, a SOAP reply could be of three different types, but these types do *not* include other addressable objects!. This differs from other RPC systems which (rightly) claim an object base: a CORBA call can return a CORBA reference to other objects; a Java RMI call can return a Java object or a proxy object for a remote object. There is no “type-safety” concept for the URL’s from SOAP replies
- Still on this issue, the Web has thrived on the ability to make links between documents. In order to this, every entity must have an *address* as a URI. This requirement is lost when SOAP can return nested data structures representing entities without addresses and has no mechanism to return addresses of SOAP addressable objects (since there is no such concept in SOAP). Other middleware systems allow calls to return objects with the same middleware addressing scheme so that calls can be made to these in turn

By way of example, consider a database query on persons. A SOAP request might look like (simplified)

```
POST url HTTP/1.1
HOST: ...
CONTENT-LENGTH: ...
CONTENT-TYPE: ...
SOAP-ACTION: ...

<Envelope>
  <Body>
    <CUSTOMERS>
    </CUSTOMERS>
  </Body>
</Envelope>
```

with response

```
HTTP/1.0 200
CONTENT-TYPE: text/xml
CONTENT-LENGTH: ..

<?xml version=”1.0”?>
<Envelope>
  <Body>
    <CUSTOMERS/>
      <CUSTOMER>
            <ID>11</ID>
            <FIRSTNAME>Julia</FIRSTNAME>
            ...
      </CUSTOMER>
      ...
    </CUSTOMERS>
  </Body>
</Envelope>
```

In this, no customers have addresses. They cannot be accessed individually; they cannot be passed to other clients as either objects or addresses since they have no defined existence outside of the return document.

The REST version is to issue a request
GET http://host/CUSTOMERS
with response consisting of addresses

```
<resources xmlns:xlink=”http://www.w3.org/1999/xlink”>
  <resource xlink:href=”http://host/CUSTOMER/0/”>0</resource>
  <resource xlink:href=”http://host/CUSTOMER/1/”>1</resource>
  <resource xlink:href=”http://host/CUSTOMER/2/”>2</resource>
  <resource xlink:href=”http://host/CUSTOMER/3/”>3</resource>
</resources>
```

Further requests are then made of addresses such as
GET http://host/CUSTOMER/1/
with response

```
<resources xmlns:xlink=”http://www.w3.org/1999/xlink”>
  <ID>11</ID>
  <FIRSTNAME>Julia</FIRSTNAME>
  ...
</resources>
```

which consists of primitive values.

4. UPnP and SOAP

UPnP consists of a number of components [12], just like other discovery systems such as Jini [13] and Salutation. (The IETF Service Location Protocol does not have an equivalent of SOAP, since it expects each service to have its own communication protocol.)

- A format for service/device description. UPnP uses XML

- A mechanism for advertising services/devices. UPnP uses a new multicast protocol called HTTPMU (HTTP multicast over UDP) [14], based on HTTP syntax but (of course) not HTTP semantics. This protocol introduces new verbs such as NOTIFY and M-SEARCH to replace GET, POST, etc

- A mechanism for searching for devices/services. This also uses HTTPMU

- Devices/services in the first place are accessed by their *address* as a URL. At that address is a detailed description of the device or service. This is an XML document

- A device description includes vendor-related information, a list of services and a URL address for each service

- A service description contains names of methods and the data types to be passed as "in" parameters and returned as "out" parameters

- Finally, UPnP specifies that SOAP is to be used for method calls and return of results

5. UPnP with REST

Most of UPnP is in accordance with REST principles. A fairly small set of verbs is used for service advertisement and discovery (NOTIFY and M-SEARCH). An appropriate protocol is used: an adaptation of HTTP for a multicast environment. All devices and even their nested services are accessible through URLs. Access to device/service information is through a standard HTTP GET, and this returns an XML document. There are no issues with this.

The initial specification of UPnP data-types adopts the XML data-type specification of primitive types, but with no aggregation types such as arrays or records. The types include integers, booleans, floats, dates and strings. There is no possibility of unaddressed internal data structures in this. But the audio-visual specifications break this data-type restriction and use the complex MPEG-21 hierarchy of data-types [15]. This issue is considered in more detail later.

However, if the UPnP organisation is prepared to allow new UPnP services to break with established specifications then it will need to place some limits on this. It will also need to place limits on vendor extensions to data-types to ensure that each data element can be addressed.

The first two criticisms of SOAP (extra protocol layer and blurring of semantics) are applicable to UPnP. UPnP has "actions" which are generally expected to change state and "queries" which ask for the values of state variables and are not expected to change state. Using SOAP, both actions and queries must be called using POST, whereas REST would say that actions should use POST and queries should use GET. In general, SOAP just supplies a "noise" layer that increases traffic and obscures semantics without adding anything to functionality.

For example, the Device Architecture 101 contains an example query for the value of a state variable in a service (we simplify it a bit)

```
POST controlUrl HTTP/1.1
HOST: ...
CONTENT-LENGTH: ...
CONTENT-TYPE: ...
SOAP-ACTION: ...

<Envelope>
  <Body>
    <QueryStateVariable>
      <varName> vblName </varName>
    </QueryStateVariable>
  </Body>
</Envelope>
```

which under REST could simply be

```
GET controlUrl/vblName HTTP/1.1
HOST: ...
```

In a similar manner, actions can be encoded as POST requests with the parameters as an `application/x-www-form-urlencoded` string. For example, a SOAP encoding of an action as

```
POST controlUrl HTTP/1.1
HOST: ...
CONTENT-LENGTH: ...
CONTENT-TYPE: ...
SOAP-ACTION: ...

<Envelope>
<Body>
<actionName>
<argumentName> value </argumentName>
</actionName>
</Body>
</Envelope>
```

would under REST be encoded as

```
POST controlUrl/actionName HTTP/1.1
HOST: ...
CONTENT-LENGTH: ...
CONTENT-TYPE: application/x-www-form-urlencoded

argumentName=value
```

The proposed encodings have the service control point as part of the address (as does the SOAP encoding). But the proposed encodings also have the query variable or control action as part of the address, unlike the SOAP encodings where this information is buried within the XML SOAP request document. This proposed encoding fits better with the REST philosphy. It should be noted that this not only allows quicker processing of valid queries, but also makes it easier for the server to detect errors such as invalid action or invalid state variable without needing to parse an XML document.

6. Implementation

CyberGarage has an implementation in Java for UPnP devices and control points [16]. This allows devices and control points to be written in Java and run as applications. This is available in source code form. CyberGarage is able to use two XML parsers, the Xerces parser from the Apache project and KXML, a lightweight parser better suited to enbedded systems. We modified CyberGarage to use the simpler GET and POST commands of form encoded data as described earlier. We then had three versions to test: using SOAP with both lightweight and heavyweight XML parsers and a REST-based version using no SOAP in method calls.

We ran a test of querying a service's state variable one thousand times in a loop. In the first test the device and control point were run as separate processes on a Linux computer using Sun's JDK 1.4. In the second test the device was run on one computer and the control point on another, connected directly by crossover cable. The first case is slower because the tasks were time-sharing, each using about 40% of the CPU, whereas in the second case each task was using over 90% of the CPU on each machine.

The results are presented in Table one. It can be seen that the REST-based solution is about twice as fast as the fastest of the SOAP versions, and this includes overheads (presumably similar for both) of networking, task switching, etc. There is also a difference in packet sizes. For example, a query on a variable `Text` with value `text00` has a payload as XML document or as form encoded values, both wrapped in an HTTP packet. The sizes for this typical query are given in Table 2. The sizes are significantly smaller for the form encoded encoded packets.

	SOAP/Xerces	Soap/KXML	REST
standalone	196	137	67
network	165	114	64

Table 1. Time to make 1000 queries (secs)

In addition to these factors, memory is also consumed by running an XML parser or an x-www-form-urlencoded parser, and the corresponding data structures built. We tested this by just parsing the SOAP payload of an action against the payload of the same action form encoded.

		SOAP	REST
Request	payload	350	4
	packet	535	84
Response	payload	365	11
	packet	526	171

Table 2. Query request size (bytes)

	SOAP/Xerces	Soap/KXML	REST
classes+data	416,336	114,808	6,072

Table 3. Size of parsers (bytes)

7. Audio-visual proposal

UPnP includes an extensive package for discovering media services and information [17]. This differs in nature from existing UPnP standards in that it uses a far more extensive range of data-types. For example, to describe audio-visual data a ContentDirectory uses a subset of the MPEG-21 Digital Item Declaration Language called DIDL-Lite.

DIDL-Lite allows a complex hierarchy of resources to be stored in a directory. A directory may contain other directories such as "My Music" and "My photos", and in turn "My Music" might list music albums, while inside one of these might be individual music tracks. The representation of any part of the hierarchy is given as an XML document conforming to the DIDL-Lite DTD. However, this structured document is effectively invisible in the device and service descriptions since it is simply represented as a string.

For example, the `Browse` action to search through a directory has an "out" parameter of `A_ARG_TYPE_Result` as a string, but this string is to be interpreted as a DIDL-Lite document. In its format as a string, it offers nothing new to the model proposed above. The string can simply be returned as a parameter in a form encoded string.

However, regarded as an XML document it is necessary to look more closely at this structure. Each object in a content directory is required to have a unique identifier "item". Calls to actions such as `Browse` must give this "item" identifier of the object it is browsing (so that it can browse subdirectories and individual items). The returned XML document contains a representation of objects in a directory, and this representation contains the identifier of each object. For example, a browse of item "0" (the root directory) might return a list of objects containing identifiers "1", "2" and "30". Requests to these objects must include this identifier.

Effectively this identifier is the address of the object within the directory service. However, the current proposal hides this address deep within the SOAP call

```
POST controlUrl
...
   Browse(id, ...)
...
```

The identifier is a “hidden” address in that knowledge of the adress is not enough to get access to it: it is neccessary to use a SOAP method call. The identifier should be made part of an address for the object, so that it can be accessed directly. A possibility might be `controlUrl/id`.

Requests to the object could then be made directly to its address, qualified by parameters giving the request details. For example, a browse request does not change state so could be sent as a GET rather than POST:

```
GET controlUrl/id?requestType=browse...
```

The current type of the identifier is string. This would allow a URL to be given in place of just the identifier without changing the specification. But it would be better to change the specification to be a url instead of any string.

In addition to making the address visible, there is also the side-benefit that addresses are no longer restricted to just the device - an object could be anywhere on the network. Currently, the specification only allows a final resource such as a CD track to be a url. By allowing urls to be the value of an “item” would allow more flexible use of devices. For example, if a mobile media service joins the network then it could (a) remain as a separate service; (b) have all its media information copied to an existing “central” server; or (c) under this proposal could simply copy a top-level container address to the central server. Requests in this last case would be referred to the device actually carrying the service. This would reduce traffic in amalgamating playlists, etc, from multiple sources.

8. A double standard

The UPnP organisation has mandated SOAP for UPnP v1.1. Discussions for UPnP v2.0 are now underway. It is possible for the REST proposals to coexist with the existing SOAP. If a device receives a `GET` request then it must be a REST request since SOAP does not allow `GET`. If it is a `POST` request then the `Content-Type` can be used to distinguish between SOAP and REST. In this paper we have used the content type of `x-www-form-urlencoded`, but to avoid any potential ambiguity another name could be invented. Thus there is no difficulty in transitioning from one mechanism to the other.

9. Conclusion

It has already been noted that the allowable data-types are not bound by the original specification documents. REST can offer some pragmatic guidelines particularly in regard to addressability of objects for future work in this area.

There is a significant advantage to using REST instead of SOAP. Devices using SOAP need to be able to parse XML documents; devices using REST do not have this requirement. Although devices may be required to *produce* XML documents (as in media content) this can be done simply by string operations that do not need an XML parser.

Thus there is a significant memory and processing requirement caused to devices by using SOAP that is not present using REST. UPnP systems using REST techniques are demonstrably faster and lighter than those using SOAP.

10. References

[1] UPnP Forum, http://www/upnp.org
[2] WWW Consortium, SOAP 1.2 Protocol, http://www.w3.org/TR/soap12
[3] WWW Consortium, Web Services home page, http://www.w3.org/2002/ws/
[4] P. Prescod, “A new direction for Web Services” XML Journal, http://www.sys-con.com/xml/article.cfm?id=454
[5] J. Newmarch, “A critique of web services” [6] R. Fielding, “Architectural Styles and the Design of Network-based Software Architectures”, http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm
[7] IETF, “ONC Remote Procedure Call” http://www.ietf.org/html.charters/oncrpc-charter.html
[8] DCE
[9] COM+
[10] Object Management Group, http://www.omg.org/
[11] Sun Microsystems, “Java Remote Method Invocation¡” http://java.sun.com/j2se/1.4.1/docs/guide/rmi/spec/rmiTOC.html
[12] UPnP, “UPnP Device Architecture”
[13] K. Arnold et al, “The Jini Specification”, Addison-Wesley, 1999
[14] Y.Y. Goland, et al, “Multicast and Unicast UDP HTTP Messages”, expired IETF draft
[15] MPEG, “ MPEG-21 Overview v.5” http://www.chiariglione.org/mpeg/standards/mpeg-21/mpeg-21.htm
[16] S. Konno, “Cyberlink for java” http://www.cybergarage.org/net/upnp/java/index.html
[17] UPnP, “MediaServer V 1.0 and MediaRenderer V 1.0” http://www.upnp.org/standardizeddcps/mediaserver.asp