CableLabs Revised Home Networking API

Home Network Service Messaging
July 26, 2011

Overview

This document provides an updated Home Networking API from the previous CableLabs submission. The interface has been simplified and the parameter specification has been changed to be flexible and independent of the underlying protocol. Please provide any comments or suggestions as appropriate and point out any use cases that would not be satisfied using this API.

General approach

There are two ideas common to most user-centric media home networking protocols and a third idea that is at least common to many.

- Discovery – When the device is plugged in or switched on, services on the device advertise themselves so services on other devices on the network can communicate with them.
- Request/response – Once communication channels are established, clients may want to issue requests to services. They can expect responses in return, but normally the arrival of these responses is asynchronous.
- Conditional event/response – The third idea is a conditional request. In this case, a condition of interest is defined and a callback function is specified. If the condition occurs, the callback is executed. This is useful in the notification and handling of asynchronous user actions or conditions. Conditional events are not available in some protocols.

The CableLabs proposed JavaScript API, which is protocol independent, is implemented as follows:

```
function discoveryControl(JSONString protocols);


function sendRequest(JSONString request, responseCallback);


function responseCallback(JSONObject response);
```

JSON (parameterization technique)

In the proposed set of APIs, JSON is used to pass parameters. JSON provides a compact method for passing an arbitrary list of name/value pairs within a string. An example of a JSON string is shown below.

```
JSONString =

‘{

        “protocol”:”upnp”,

        “serviceType”:”urn:schemas-upnp-org:service:ContentDirectory:1”,

        “uuid”:”00000000-0000-1010-8000-5442499C2FE3”,

        “deviceName”:”BRAVIA XBR-52LX900”,

        “response”:”found”,

        “requestID”:”-1”

}’;
```

Using JSON, a single protocol independent string can be used for exchanging parameters in the APIs, yet the content of the string can contain name/value pairs specific to a particular protocol.

Home Networking JavaScript API

The proposed API consists of two function calls and one or more callback functions. The discoveryControl() function call selects which protocols are to be discovered and identifies the discovery callbacks. The sendRequest() function sends commands to a service and specifies the callback to receive the response. The callback function has the same format regardless of the target protocol or service. It receives a single parameter that is a JSON object. The following is a list summarizing the API.

- discoveryControl(JSONString protocols);
- sendRequest(JSONString request, responseCallback);
- responsecallback(JSONObject response);

The free format of JSON allows for complete flexibility in specifying parameters. The protocol (e.g. “upnp,” “zeroconf,” etc.) is a required parameter for both requests and responses. The HTTP response code is required for the response callbacks. All other parameters are specific to the protocol and command.

DiscoveryControl

The process of discovery requires that services announce their arrival and departure on the home network. A listener callback receives these messages and maintains a list of available services. The specifics of discoveryControl() are outlined below.

```
protocols =

‘{

        “upnp”:”upnpDiscoveryCallback”,

        “zeroconf”:”zeroconfDiscoveryCallback”,

        “someProtocol”:””  //Denotes stop discovery for someProtocol

}’


discoveryControl(JSONString protocols);
```

Each parameter of the “protocols” JSON string specifies a protocol of interest and the callback to receive the discovery messages for that particular protocol. The discoveryControl() function matches the requested protocols with those actually implemented in the user agent. If the requested protocol is available, the discovery process will be started and the callback specified will be set to receive the discovery message. If the callback string is left empty and the protocol is already running, it will be shut down.

Discovery Examples

The examples below provide a more detailed view of how discoveryControl() is implemented for two commonly used home network protocols.

UPnP Discovery Example

The following example shows how UPnP can be used in the discoveryControl() function. The discovery process is implemented in the user agent. When discoveryControl() is called with a protocol that is implemented, the discovery process begins. When a device using the specified protocol is discovered, the callback function is called. The callback function is called with a single argument that is a JSON object. It has two required parameters. The first is the “protocol” string. The second is the “status” string. All other parameters within the JSON object are particular to the specific protocol. For UPnP, the JSON object for a particular television is shown as an example below.

```
discoveryControl({“UPnP” : “upnpDiscoveryCallback”);


upnpDiscoveryCallback(jsonObject)

{

        //do something with the discovered service

};
```

Where:

```
jsonObject =

{

        “protocol”:”upnp”,

        “serviceType”:”urn:schemas-upnp-org:service:ContentDirectory:1”,

        “uuid”:”00000000-0000-1010-8000-5442499C2FE3”,

        “deviceName”:”BRAVIA XBR-52LX900”,

        “response”:”information about the UPnP interface, actions, events, etc.”,

        “status”:”found”,

};
```

The information passed in the JSON object is used to keep track of the details of the discovered device. The status parameter indicates that the device is either found or lost.

Zeroconf Discovery Example

The implementation of discovery for the Zeroconf protocol is very similar to discovery of UPnP or any other protocol.

```
discoveryControl({“Zeroconf” : “zeroconfDiscoveryCallback”);


zeroconfDiscoveryCallback(jsonObject)
```

```
{
        //do something with the discovered service
};
```

Where:

```
jsonObject =
{
        “protocol”:”zeroconf”,
        “serviceType”:”DAAP”,
        “deviceName”:” Firefly Media Server on Windows_daap”,
        “status”:”found”,
};
```

The discoveryProtocol() function and associated callbacks provide a service and protocol independent means to discover services that has sufficient flexibility to implement any protocols and services of interest. It allows for discovery to be event driven by the arrival and disappearance of devices.

SendRequest

After discovery, the sendRequest() function is used to issue commands to a particular service. It works in much the same way as the discoveryControl() function in that it passes parameters in a JSON string and specifies a callback to receive the relevant responses.

The sendRequest() function passes the “protocol” string as a required parameter. The other parameters in the JSON string are specific to the service and action requested.

The other parameter of sendRequest() is the callback to receive the response to the request. The callback approach allows for implementation flexibility in processing request responses. For example, a specific callback could be defined for each and every sendRequest to each service. Alternatively, a single callback could be used across all sendRequests and services. The application gets to decide how to organize request responses. The basic format of the JSON string passed to sendRequest() is illustrated below.

```
sendRequest(JSONString request, responseCallback);


request=
{
        “protocol”:”upnp”,          // the name of the protocol



        //other parameters as required
}
```

The response callback has the following format:

```
responseCallback(JSONObject response)



response =
{
        “protocol”:”upnp”, // the name of the protocol (e.g. “upnp”)
        “responseCode”:”” // http return code

        //other parameters as appropriate
};
```

The parameter “responseCode” is required in the JSON object response passed to the responseCallback() routine. It is the HTTP response code.

UPnP sendResponse() Example

Here is an example of a UPnP implementation of the sendRequest() function and the corresponding callback for the response.

The following code snippet sends a UPnP “play” action to an AVTransport renderer. The command is formatted into a JSON string, then sent to the specific device using the sendRequest() API.

```
jsonString =

‘{
        “protocol”:”upnp”,

        “serviceType”:”urn:schemas-upnp-org:service:AVTransport:1”,

        “uuid”:”00000000-0000-1010-8000-5442499C2FE3”,

        “action”:”#PLAY”,

        “body”:”<?xml version=\"1.0\" encoding=\"utf-8\"?><s:Envelope
        xmlns:s=\"http://schemas.xmlsoap.org/soap/envelope/\"
        s:encodingStyle=\"http://schemas.xmlsoap.org/soap/encoding/\"><s:Body><
        u:Play xmlns:u=\"urn:schemas-upnp-
        org:service:AVTransport:1\"><InstanceID>0</InstanceID><Speed>1</Speed
        ></u:Play></s:Body></s:Envelope>”

}’;


sendRequest(jsonString, upnpCallback);
```

The parameters of the JSON string are defined as follows:

- “protocol,” “serviceType” and “uuid” are returned when a UPnP device is discovered. They are used here to uniquely identify the device and service.
- “action” is the command parameter being requested. It determines how to format the body parameter.
- “body” contains the SOAP command specified by the Action parameter.

The following code snippet receives a UPnP response from a “play” request to an AVTransport renderer. The response is formatted as a JSON Object parameter in the receiveResponse() API.

```
upnpCallback(jsonObject);


jsonObject =

{

        “protocol”:”upnp”,

        “serviceType”:”urn:schemas-upnp-org:service:AVTransport:1”,

        “uuid”:”00000000-0000-1010-8000-5442499C2FE3”,
```

“friendlyName”:”BRAVIA XBR-52LX900”,

“response”:” HTTP/1.1 200 OKContent-Length: 261Content-Type: text/xml; charset=\"utf-8\"EXT: Connection: Keep-AliveDate: Wed, 29 Jun 2011 22:06:32 GMTServer: Linux/2.6 UPnP/1.0 XBR-52LX900/1.7X-AV-Server-Info: av=5.0; cn=\"Sony Corporation\"; mn=\"BRAVIA XBR-52LX900\"; mv=\"1.7\";X-AV-Physical-Unit-Info: pa=\"BRAVIA XBR-52LX900\";<?xml version=\"1.0\"?><s:Envelope xmlns:s=\"http://schemas.xmlsoap.org/soap/envelope/\" s:encodingStyle=\"http://schemas.xmlsoap.org/soap/encoding/\"><s:Body><u:Play Response xmlns:u=\"urn:schemas-upnp-org:service:AVTransport:1\"></u:PlayResponse></s:Body></s:Envelope>”,

“responseCode”:”200”

};

The parameters in the JSON object are as follows:

- “protocol,” “serviceType,” and “uuid” match the values supplied in the request.
- “friendlyName provides a user-readable name for the responding device
- “response” is the SOAP formatted response to the UPnP request
- “responseCode” is the html return code

Zeroconf sendRequest() Example

The following code snippet sends a Zeroconf directory browse request. The command is formatted into a JSON string, then sent to the specific device using the sendRequest() API.

As can be seen, the parameters passed in a Zeroconf request bear little resemblance to those passed for a UPnP request. This illustrates the protocol-independent nature of this approach.

jsonString =

‘{

"protocol":"zeroconf",

“deviceName":"Firefly Media Server on clarke-thinkpad-t43_daap",

"requestPath": "/databases/1",

"requestQuery":"/containers?meta=dmap.itemid,dmap.itemname,dmap.persistentid,com.apple.itunes.smart-playlist,dmap.haschildcontainers,dmap.parentcontainerid,com.apple.itunes.is-podcast-playlist,com.apple.itunes.special-playlist,com.apple.itunes.saved-genius"

}';

sendRequest(jsonString, zeroconfCallback);

The paramters of the JSON string are defined as follows:

- "protocol" and "deviceName" are the parameters returned when a Zeroconf device is discovered.
- For Zeroconf, we use the device name as a uuid since it is guaranteed to be unique on the local network segment.
- "requestPath" gets appended to the URL in building the request
- "requestQuery" gets appended to requestPath to pass the parameters

The following code snippet receives a Zeroconf directory browse response. The response is formatted as a JSON object parameter in the receiveResponse() API.

zeroconfCallback(jsonObject);

jsonObject =

'{

"protocol":"zeroconf",

"responseCode":"200",

"domainName":"Firefly Media Server on clarke-thinkpad-t43_daap",

"ip":"10.4.18.35",

"port":"3689",

"response":{"apso":{"mstt":200,"muty":false,"mtco":12,"mrco":12,"mlcl":[{"mlit":{"miid":6,"asfm":"mp3","minm":"Aint No Rest For The Wicked","assz":4888740,"astm":175647}},{"mlit":{"miid":7,"asfm":"mp3","minm":"Lotus","assz":5261624,"astm":196649}},{"mlit":{"miid":8,"asfm":"mp3","minm":"Free Love","assz":6085584,"astm":208561}},{"mlit":{"miid":9,"asfm":"mp3","minm":"Back Against The Wall","assz":5352834,"astm":228362}},{"mlit":{"miid":10,"asfm":"mp3","minm":"Back Stabbin' Betty","assz":4773362,"astm":219193}},{"mlit":{"miid":11,"asfm":"mp3","minm":"Tiny Little Robots","assz":6793873,"astm":250462}},{"mlit":{"miid":12,"asfm":"mp3","minm":"Soil To The Sun","assz":5423455,"astm":197433}},{"mlit":{"miid":13,"asfm":"mp3","minm":"In

```
One Ear","assz":5808321,"astm":241397}},{"mlit":
{"miid":14,"asfm":"mp3","minm":"Drones In The
Valley","assz":3377955,"astm":147853}},{"mlit":
{"miid":15,"asfm":"mp3","minm":"Judas","assz":5238404,"astm":206706}},{"mlit":
{"miid":16,"asfm":"mp3","minm":"James Brown","assz":4406341,"astm":200724}},
{"mlit":
{"miid":17,"asfm":"m4v","minm":"Angels_And_Demons_S0507_Promo.m4v","assz":6
6938904}}]}},

        "reqPath":"/databases/1/containers/1“}

}’;
```

The protocols for the JSON object are described below:

- “protocol” and “requestPath” match the values supplied in the request.
- “responseCode” is the html return code
- “domainName provides a user-readable name as well as a uuid for the responding device
- “ip” provides the IP address of the responding device
- “port” provides the port number of the response
- “response” is the JSON formatted response to the Zeroconf request

The discoveryControl() and sendRequest() functions along with the callbacks completely define the API. The differentiation is determined by the format of the JSON strings and objects passed. One particular special case that is of interest is the handing of events (such as asynchronous variable changes).

Events

Events are handled using the same sendRequest() function and callback approach, since a sendRequest’s associated response is asynchronous.

SendRequest() is used to register interest in a specific event. The JSON string parameter defines the event. Occurrence of the event results in the sendRequest callback being invoked. Every time the event occurs the callback will be called. The syntax used in the sendRequest to register interest in an event is service specific. Support for events is specific to individual protocols and services.

UPnP sendRequest for Events Example

The following code snippet registers a UPnP evented variable. The request is formatted into a JSON string, then sent to the specific device using the sendRequest() API.

```
jsonString =

‘{

        “protocol”:”upnp”,

        “serviceType”:”urn:schemas-upnp-org:service:AVTransport:1”,

        “uuid”:”00000000-0000-1010-8000-5442499C2FE3”,

        “action”:””,

        “body”:””

}’;


sendRequest(jsonString, upnpCallback);
```

The parameters in the JSON string are defined as follows:

- “protocol,” “serviceType” and “uuid” are the parameters returned when a UPnP device is discovered.
- “??” is the command parameter that determines how to format the body parameter
- “body” contains the SOAP command to register the event
- receiveResponse for Events (UPnP example)

The following code snippet receives a UPnP response from a registered event. The response is formatted as a JSON object parameter in the upnpCallback() API.

```
upnpCallback(jsonObject);


jsonObject =

{

        “protocol”:”upnp”,

        “serviceType”:”urn:schemas-upnp-org:service:AVTransport:1”,

        “uuid”:”00000000-0000-1010-8000-5442499C2FE3”,

        “friendlyName”:”BRAVIA XBR-52LX900”,
        “response”:””,

        “responseCode”:”200”

};
```

The parameters for the receiveResponse callback for UPnP events are defined as follows:

- “protocol,” “serviceType,” and “uuid” match the values supplied in the request.
- “friendlyName provides a user-readable name for the responding device
- “response” is the SOAP formatted response to the UPnP event
- “responseCode” is the html return code

Security

One of the key issues in allowing access to the home network is security. A user may not want a web interface to have unrestricted access to all available home devices or services. CableLabs has implemented a two-tier approach.

First, the user agent is implemented as a signed Java Applet. If the applet has an authentication chain back to a source already authorized by the user, it will load without interruption. If not, the browser will ask the user for permission to load the applet. If permission is granted, execution will continue. Otherwise, the applet will not be run.

The second tier of security is at the service level. As services are discovered by the Applet, but before the Applet invokes the callback function, the user is asked to authorize that the Web page can have access to that service (for example, through a pop-up dialog box). If authorization is granted, execution continues. If not, the callback will not be called and it will be as if the service was never discovered. Additional levels of security can be invoked by the particular implementation.

Implementation Notes

The API proposed in this document can be implemented in a number of different ways. CableLabs has implemented it as a signed Java Applet that is downloaded with the HTML/CSS web page that provides the user interface. However, it could also be implemented as a plug-in or completely integrated into the browser source code.