Roku DIAL Programming

Version 1.1

February 2014

Roku DIAL Programming

DISCOVERY AND LAUNCH

DIAL (Discovery and Launch) is a simple network protocol for discovering first screen devices and applications from a second screen (such as a mobile iOS or Android application,) and for launching first screen applications on the first screen device from the second screen app. In the context of the Roku platform, the first screen device is the Roku device itself. A first screen application is a DIAL-aware channel installed on the Roku device. Complete details of the DIAL specification can be found here: http://www.dial-multiscreen.org/dial-protocol-specification.

Many current Roku developers are familiar with the Roku external control protocol (ECP) which includes functionality similar to DIAL. An experienced Roku developer may thus fairly ask the question "why do I need DIAL?" One reason is that you may already have a DIAL based second screen implementation for use with other platforms. DIAL support on Roku means that you don't need to add a second protocol to your current application for discovery and launch.

USING DIAL WITH ROKU

The Roku firmware implements the UPnP server that handles DIAL requests from second screen applications. Thus all of the server side DIAL protocol implementation is handled for you. As a developer, supporting DIAL in a Roku BrightScript channel requires two things. The first is a second screen application such as a mobile application that acts as the DIAL client. This application makes DIAL device discovery requests to find the set of Roku devices. It also makes launch requests to tell a specific Roku device to start a DIAL aware Roku channel. The DIAL specification also provides for a command to stop a running application. Roku also supports this command. The second thing that is required is to make the Roku channels that you want to control from the second screen app DIAL aware. As this second requirement is simpler, let's start with that.

Making Your BrightScript Channel DIAL Aware

To register a BrightScript channel with the Roku DIAL protocol, a new entry needs to be added to the channel manifest:

```
title=2DVideo
major_version=1
minor_version=0
build_version=0
…
dial_title=2DVideo
```

The title used in the dial_title property is used by the Roku DIAL server to identify the channel. This title is used in the application resource URL defined by the DIAL specification as the identifier of the application. For example, the DIAL server might identify the channel with the dial_title 2DVideo as: http://10.0.0.14:8060/dial/2DVideo

The exact format of this URL is [DIAL Application URL]/[appname], where [appname] is the dial_title specified in the manifest, and [DIAL Application URL] is the specified by the Roku device in response to the DIAL discovery query. We will cover the DIAL discovery process in detail later. With a dial_title registering your channel with the Roku DIAL server, a second screen application can make channel launch (and stop) requests. However, the DIAL protocol also specifies a way for the second screen application to pass parameters to the first screen application to customize application startup. For example, you may want your second screen application to launch a Roku channel, and specify a particular video to play, or a specific screen to navigate to in the Roku channel. Launch parameters are passed to your BrightScript channel's Main function as a dynamic array of name – value pairs:

```
Function Main(params as Dynamic) as void
    if (params <> invalid)
        LaunchVideo(params.videoURL)
    endif
    …
End Function
```

Implementing DIAL In Your Second Screen App

The bigger task associated with bringing DIAL support to your Roku experience is implementing DIAL in your second screen applications. This consists of two steps: device discovery and app launch.

Discovery

DIAL discovery is performed by a DIAL client by sending an M-SEARCH request via UDP to the IPv4 multicast address 239.255.255.250 on port 1900. An M-SEARCH request looks like this:

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: seconds to delay response
ST: urn:dial-multiscreen-org:service:dial:1
USER-AGENT: OS/version product/version
```

In response, the Roku DIAL server returns a response such as this:

```
HTTP/1.1 200 OK
LOCATION: http://10.0.0.14:52235/dd.xml
CACHE-CONTROL: max-age=1800
EXT:
BOOTID.UPNP.ORG: 1
SERVER: OS/version UPnP/1.1 product/version
ST: urn:dial-multiscreen-org:service:dial:1
```

There may be multiple Roku devices on your network. Therefore, after sending the M-SEARCH request, you should continue to read UDP response packets until all devices have been reported. The most information in this response is the LOCATION header. This URL is the location of the DIAL server on the host device. This location is queried in the second part of the discovery process to obtain the URL of the DIAL REST service for the device. This query is performed by issuing an HTTP GET request to the LOCATION URL:

```
GET /dd.xml HTTP/1.1
```

The device sends back a response similar to the following:

```
HTTP/1.1 200 OK
Application-URL: http://10.0.0.14:12345/apps
...
<UPnP device description in message body>
```

The Application-URL header contains the DIAL REST service endpoint. Application information requests and launch and stop commands are sent to this endpoint. As an example, here is how you would query the device discovered above for information about the 2DVideo channel:

```
GET http://10.0.0.14:12345/apps/2DVideo HTTP/1.1
```

In response, the Roku device sends the following information back to the DIAL client:

```
HTTP/1.1 200 OK
...
<?xml version="1.0"  encoding="UTF-8"?>
<service xmlns="urn:dial-multiscreen-org:schemas:dial">
      <name>YouTube</name>
      <options allowStop="true"/>
      <state>running</state>
      <link rel="run" href="run"/>
</service>
```

Below is an example of how to implement DIAL discovery from a second screen Android application:

```
String upnpLocation;
String M_SEARCH = "M-SEARCH * HTTP/1.1\r\nHOST: 239.255.255.250:1900\r\nMAN:
    \"ssdp:discover\"\r\nMX: seconds to delay response\r\nST: urn:dial
    multiscreen-org:service:dial:1\r\nUSER-AGENT: RokuCastClient";
DatagramSocket clientSocket = new DatagramSocket();
clientSocket.setSoTimeout(1000);
```

```
InetAddress IPAddress = InetAddress.getByName("239.255.255.250");
byte[] sendData = new byte[1024];
byte[] receiveData = new byte[1024];
sendData = M_SEARCH.getBytes();
DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
    IPAddress, 1900);
clientSocket.send(sendPacket);
String response = "";
DatagramPacket receivePacket = new DatagramPacket(receiveData,
    receiveData.length);
clientSocket.receive(receivePacket);
response = new String(receivePacket.getData());
clientSocket.close();
//…
//Parse out the LOCATION header from the response string
//and store it in the String upnpLocation
//…
//Get the Application-Url for the discovered device
String appUrl;
URL obj = new URL(upnpLocation);
HttpURLConnection con = (HttpURLConnection) obj.openConnection();
con.setRequestMethod("GET");
BufferedReader in = new BufferedReader(
    new InputStreamReader(con.getInputStream()));
String inputLine;
StringBuffer response = new StringBuffer();

while ((inputLine = in.readLine()) != null) {
    response.append(inputLine);
}
appUrl = getHeader(con, “Application-URL”);
in.close();

//getHeader method
protected String getHeader(HttpURLConnection con, String header) {
    java.util.List<String> values = new java.util.ArrayList<String>();
    int idx = (con.getHeaderFieldKey(0) == null) ? 1 : 0;
    while (true) {
        String key = con.getHeaderFieldKey(idx);
        if (key == null)
            break;
```

```
        if (header.equalsIgnoreCase(key))
            return con.getHeaderField(idx);
        ++idx;
    }
    return "";
}
```

Launch

Now that you have the DIAL REST endpoint for a Roku device, you can launch DIAL aware channels on that device. Launching a DIAL aware channel is done from your second screen application by sending an HTTP POST to the DIAL REST endpoint and specifying the dial title. Launch parameters can be sent in the body of the POST request. For example:

```
POST /apps/2DVideo
Content-Type: text/plain; charset="utf-8"
param1=value1&param2=value2…
```

The Roku device will reply to a launch request with an HTTP response similar to this:

```
HTTP/1.1 201 CREATED
LOCATION: http://10.0.0.14:12345/apps/2DVideo/run
```

It is important to save the URL in the LOCATION header in this response if you need to stop the channel from your second screen application. This is because to stop (exit) a channel, you send an HTTP DELETE to that URL. For example, to exit the 2DVideo channel launched with previously described HTTP POST, you would send this request:

DELETE http://10.0.0.14:12345/apps/2DVideo/run

Here is some sample Android Java code for launching the 2DVideo channel. In this example, once the launch command is successful, the Location header in the response will contain the URL http://10.0.0.14:8060/dial/2DVideo/run. This gets assigned to the variable `appRunLocation`. In addition to the videoUrl parameter to specify which video to play, we pass streamFormat, which the BrightScript code will use to set the stream format content meta data. A third parameter, play_start, is used to tell the channel how many seconds into the video to start playback. If a video partner keeps track of video playback position, this technique could be used to implement a resume feature.

```
String appRunLocation = "";
URL obj = new URL("http://10.0.0.14:8060/dial/2DVideo");
HttpURLConnection con = (HttpURLConnection) obj.openConnection();

//add request header
con.setRequestMethod("POST");
con.setRequestProperty("Content-Type", "text/plain; charset=\"utf-8\"");

String urlParameters = "videoUrl=" + URLEncoder.encode("video-url", "UTF-8");
urlParameters += "&streamFormat=ism";
urlParameters += "&play_start=" + play_start;
```

```
// Send post request
con.setDoOutput(true);
DataOutputStream wr = new DataOutputStream(con.getOutputStream());
wr.writeBytes(urlParameters);
wr.flush();
wr.close();

BufferedReader in = new BufferedReader(
    new InputStreamReader(con.getInputStream()));
String inputLine;
StringBuffer response = new StringBuffer();

appRunLocation = getHeader(con, RUN_LOCATION);
in.close();
```

Exiting the channel from Android would be done as follows, using the `appRunLocation` value retrieved when launching the channel:

```
URL obj = new URL(appRunUrl);
HttpURLConnection con = (HttpURLConnection) obj.openConnection();
//add request header
con.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
con.setRequestMethod("DELETE");
con.connect();

BufferedReader in = new BufferedReader(
new InputStreamReader(con.getInputStream()));
String inputLine;
StringBuffer response = new StringBuffer();

while ((inputLine = in.readLine()) != null) {
    response.append(inputLine);
}
in.close();
```