UPnP code for Windows 8

ljw1004

14 Sep 2012 Public Domain

How to use UPnP on Windows 8

Download source code - 40.3 KB

Introduction

This code plays audio on UPnP speakers. If you have DLNA speakers ("Digital Lifestyle Networking Alliance") they are based on UPnP and it will also work on them.

Instead of following this article, there's a key alternative technique that's better - using the new Windows 8 support for "Play To" (this link goes to Windows SDK sample code for the PlayTo contract+API). PlayTo is far simpler, better-tested, and preferable to implementing all this UPnP stuff yourself. It takes only a few lines of code. Users of your app-store app will then be able to use the Devices charm to make your app stream its video or audio to their TV or speakers or picture-frames, so long as the device's manufacturer has provided Windows-8-certified drivers.

(Note that certified drivers are only needed for app-store apps; they're not needed for desktop apps. There's also one older technique, using a COM library called *UPNP.dll* dating back to XP, but it isn't allowed for app-store apps).

I wrote this code because my device doesn't yet have windows-8-certified drivers, and because I wanted to learn about UPnP. More generally, the code might help you understand the UPnP underpinnings of how things work under the hood when you use PlayTo.

Contents

2. Next it shows code that implements those raw UDP and TCP communications explicitly, both for device-discovery (SSDP) and for device control (UPnP).

   There are two forms of the code: .NET45 code (runs on any .NET45 machine but not allowed on app store) and WinRT code (runs on Windows tablets, allowed on app-store, requires Windows8).

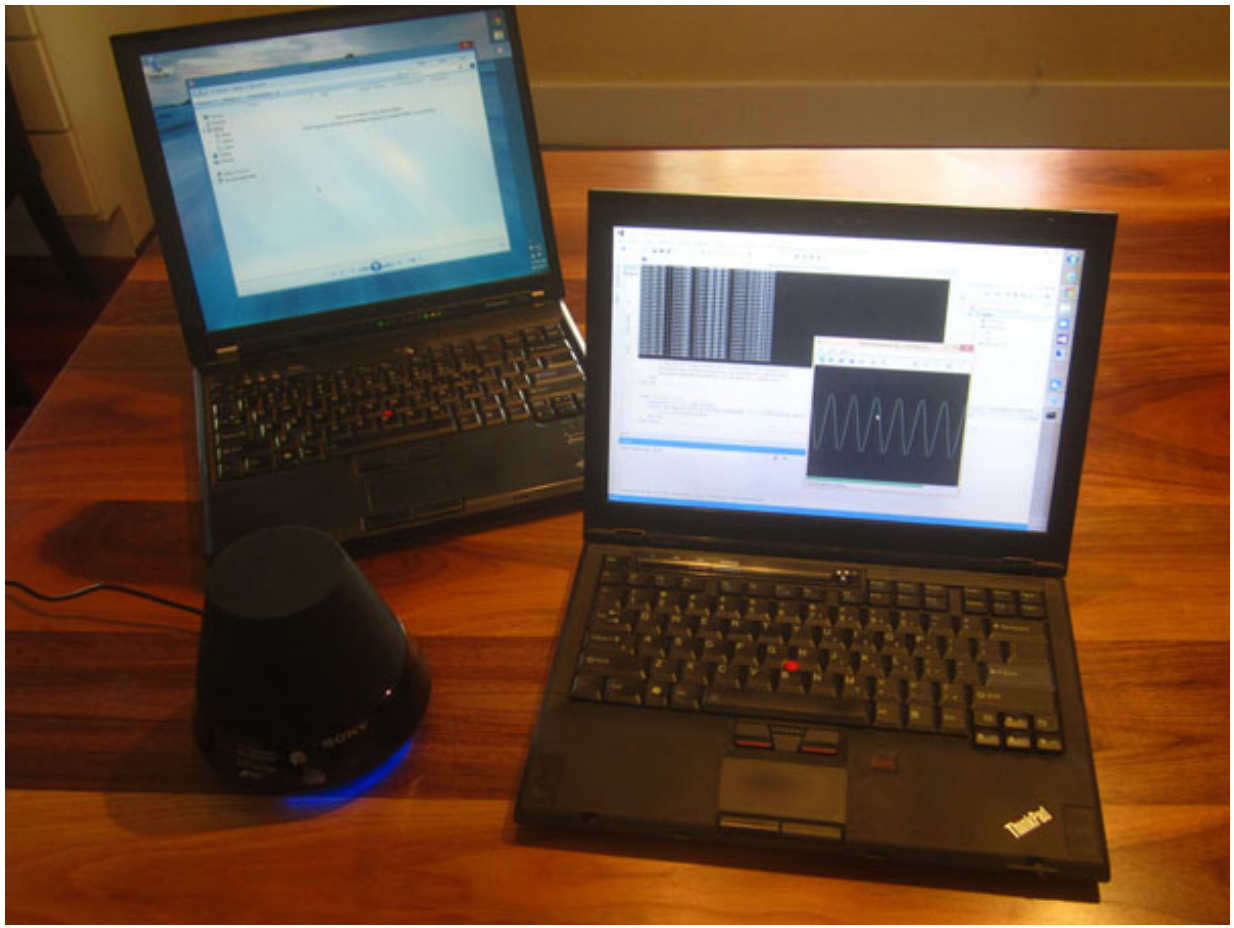
3. Next it shows where you can use HttpClient and VB's XML-literals, for a higher-level abstraction rather than doing raw TCP programming.
4. Next it shows how to use WinRT device-enumeration APIs, rather than implementing SSDP manually. This is a cleaner way of doing things, and will run on Windows tablets and is allowed on the app-store, but requires Windows 8.

Testing with UPnP devices

Here are three UPnP devices you can test this code on:

- I went out and bought a Sony SA-NS300 wifi speaker. This is a UPnP device (but doesn't yet as of September 2012 have windows-8-certified drivers).
- I downloaded Developer Tools for UPnP Technologies. This has a program called *MediaRenderer.exe* in it, which you can test against (but it also isn't yet windows-8-certified).
- I set up a different machine on the same network, ran Windows Media Player on it, and did Stream > AllowRemoteControl. (This has to be on a different machine: it doesn't work on the same machine). This technique is also described in the Windows SDK "PlayTo sample". It is windows-8-certified.

Here's a picture of my laptop simultaneously sending audio wirelessly to all three devices!



Pairing in WinRT enumeration: The WinRT device-enumeration APIs will only report "paired" devices. On home/private networks, devices will be automatically paired, but on public networks you have to pair them manually (Start Screen > Control Panel > Devices > AddNewDevice). Perhaps this "pairing" feature was to stop you accidentally broadcasting where you shouldn't on a public network? My SSDP code lacks this feature.

Limitation with manual SSDP search: My manual SSDP-discovery code doesn't seem to find as many devices as WinRT is able to enumerate, or show in the Network Neighborhood. Windows is likely enumerating devices discovered with multiple discovery protocols beyond SSDP.

UPnP protocol

UPnP has superb documentation at www.upnp.org.

- UPnP Device Architecture - explains how the shared UPnP protocols work, including service discovery. This is common information for all UPnP protocols. The rest of the documents are specific to media-playing.
- UPnP AV Architecture - explains the particular case of media rendering, from a DMS (Digital Media Source) to a DMR (Digital Media Renderer)
- ConnectionManager:1 - documents the API of the `ConnectionManager` interface, used to determine which protocols are supported
- AVTransport:1 - documents the API of the `AVTransport` interface, used to tell the DMR which media-source to play, and to start/stop it.

When media is played over UPnP, there are three parties involved:

Control Point - this party tells the others when and what to play

Digital Media Renderer (DMR) - the speakers/tv. They can be told what URL to play, and can be told to start/stop.

Digital Media Source (DMS) - where the media comes from, i.e. it serves up the URL that was requested by the DMR.

You can imagine lots of ways this might work. Maybe your computer is the control point, and the DMS is some archive on a server in your attic, and the DMR is your television set. Or maybe your computer is the DMS, and the control point is a tablet or cell phone or remote control, and the DMR is your set of Wi-Fi speakers.

In this article, we will write a console app which acts as both control-point and DMS. Actually it's a lightweight DMS that merely serves up a single piece of media over HTTP: it won't support the full range of searching functionality &c. that a DMS is normally supposed to do.

The UPnP Protocol

UPnP devices find each other by a protocol called **SSDP**, documented in the UPnP Device Architecture document. It specifies how new devices can notify existing listeners when they join a network. It also specifies a simpler form, which we describe here, where you can find out all devices currently present.

>>> outgoing multicast UDP message on 239.255.255.250:1900] as follows, with line terminator CRLF and encoding UTF8. Note there's a blank line at the end of the message. The MX parameter tells devices to respond at some random time interval up to 1 second, so we should keep listening for at least that long. The format of this packet, and the response, is all in the UPnP Device Architecture document.

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
ST:urn:schemas-upnp-org:device:MediaRenderer:1
MAN: "ssdp:discover"
MX: 1
```

<<< receive UDP response from each device, sent by them from their ip address and from whichever port they happened to use, received by whatever IP+port we happened to bind our UDP socket to. The response also uses CRLF and UTF8 and includes a blank line at the end (but since it fits into one packet, this doesn't matter).

```
HTTP/1.1 200 OK
ST: urn:schemas-upnp-org:device:MediaRenderer:1
CACHE-CONTROL: max-age=900
EXT:
USN: uuid:93997fb5-25f6-4385-8356-c705f0e7e0d9::urn:schemas-upnp-org:device:MediaRenderer:1
SERVER: Windows NT/5.0, UPnP/1.0
LOCATION: http://192.168.0.128:35142/
Content-Length: 0
```

>>> outgoing TCP message, for each device, to do a HTTP-GET to retrieve the URI specified in the device's LOCATION: response. In this case we create a TCP socket, connect it to 192.168.0.128:35142, and send the following message (again with CRLF and UTF8 and a blank line at the end).

```
GET / HTTP/1.1
```

<<< response over that same TCP socket. As always for HTTP responses, the header is read in UTF8, and is terminated by a blank line, and then there are "Content-Length" bytes which have to be interpreted in the specified encoding. (Incidentally, there's a bug in `HttpClient.GetStringAsync` and `HttpContent.ReadAsStringAsync` which makes them throw an exception when the charset is enclosed in quotes, like here.) In this message, look out for UDN (the unique device name), and for the ControlURLs of the three services it supports. The format of this discovery-response message is in the UPnP Architecture Document.

```
HTTP/1.1 200 OK
CONTENT-TYPE: text/xml;charset="utf-8"
X-AV-Server-Info: av="5.0"; cn="Sony Corporation"; mn="SA-NS300"; mv="1.00"
X-AV-Physical-Unit-Info: pa="SA-NS300"
CONTENT-LENGTH: 3169

<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion><major>1</major><minor>0</minor></specVersion>
  <device>
    <deviceType>urn:schemas-upnp-org:device:MediaRenderer:1</deviceType>
    <UDN>uuid:5f9ec1b3-ed59-1900-4530-0007f521ebd6</UDN>
    <friendlyName>SA-NS300</friendlyName>
    <presentationURL>http://192.168.0.122/</presentationURL>
    <manufacturer>Sony Corporation</manufacturer>
    <manufacturerURL>http://www.sony.net/</manufacturerURL>
    <modelDescription>Network Speaker</modelDescription>
    <modelName>SA-NS300</modelName>
    <iconList>
      <icon><mimetype>image/jpeg</mimetype><width>48</width><height>48</height>
            <depth>24</depth><url>/speaker 48.jpg</url></icon>
      <icon><mimetype>image/jpeg</mimetype><width>120</width><height>120</height>
            <depth>24</depth><url>/speaker 120.jpg</url></icon>
      <icon><mimetype>image/png</mimetype><width>48</width><height>48</height>
            <depth>24</depth><url>/speaker 48.png</url></icon>
      <icon><mimetype>image/png</mimetype><width>120</width><height>120</height>
            <depth>24</depth><url>/speaker 120.png</url></icon>
```

```
    </iconList>
    <serviceList>
      <service>
        <serviceType>urn:schemas-upnp-org:service:RenderingControl:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:RenderingControl</serviceId>
        <SCPDURL>/RenderingControl/desc.xml</SCPDURL>
        <controlURL>/RenderingControl/ctrl</controlURL>
        <eventSubURL>/RenderingControl/evt</eventSubURL>
      </service>
      <service>
        <serviceType>urn:schemas-upnp-org:service:ConnectionManager:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:ConnectionManager</serviceId>
        <SCPDURL>/ConnectionManager/desc.xml</SCPDURL>
        <controlURL>/ConnectionManager/ctrl</controlURL>
        <eventSubURL>/ConnectionManager/evt</eventSubURL>
      </service>
      <service>
        <serviceType>urn:schemas-upnp-org:service:AVTransport:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:AVTransport</serviceId>
        <SCPDURL>/AVTransport/desc.xml</SCPDURL>
        <controlURL>/AVTransport/ctrl</controlURL>
        <eventSubURL>/AVTransport/evt</eventSubURL>
      </service>
    </serviceList>
  </device>
</root>
```

The next phase of the protocol is to interact with the device over its control interfaces. For this particular device, a media-renderer, the overall protocol is documented in the "UPnP AV Architecture" document. Let's chose first to query the device about which media formats it supports:

>>> outgoing TCP message, to do a "GetProtocolInfo SOAP action" on the device's `ConnectionManagerUrl` which we retrieved above. You can do this over the same TCP socket as used previously, if it's still open and is on the same host. HTTP/1.1 sockets are kept alive by default for a limited period; HTTP/1.0 sockets are closed by default unless they have the Keep-Alive header, in which case they'll be kept alive for a limited period. In this case we chose to close the above TCP socket, and open another one to the 192.168.0.122:8080 (with any local ip/port), and send the following message. As always, CRLF and UTF8 and a blank line at the end, and Content-Length is the size in non-encoded bytes of the body; "UPnP Architecture" document goes into exactly what's required of SOAP messages in UPnP. Note that the SOAP action is specified twice, once in the SOAPAction header (with its schema), and once as `<u:GetprotocolInfo>` (again with its schema).

```
POST /ConnectionManager/ctrl HTTP/1.1
Host: 192.168.0.122:8080
Content-Length: 299
Content-Type: text/xml; charset="utf-8"
SOAPAction: "urn:schemas-upnp-org:service:ConnectionManager:1#GetProtocolInfo"

<?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:GetProtocolInfo xmlns:u="urn:schemas-upnp-org:service:ConnectionManager:1">
    </u:GetProtocolInfo>
  </s:Body>
</s:Envelope>
```

<<< response over that same TCP socket. As described in the UPnP Architecture document, each possible action has some IN parameters (which would have been provided on the outgoing message, except that `GetProtocolInfo` doesn't have any). Also OUT parameters, which are provided in the response. In this case, there are two out parameters Source and Sink. Their meaning and format is documented in the "ConnectionManager:1" document, $2.5.2 "Protocol Concept". In this case, the device is telling us what kind of media sources it is able to play. It can issue http-get requests, over any network, and can accept the mime-type "audio/L16;rate=44100;channels=1" or "...channels=2". There are also about twenty other mime-types which I truncated, including mp3 audio.

```
HTTP/1.1 200 OK
CONTENT-LENGTH: 4114
CONTENT-TYPE: text/xml; charset="utf-8"
X-AV-Server-Info: av="5.0"; cn="Sony Corporation"; mn="SA-NS300"; mv="1.00"
X-AV-Physical-Unit-Info: pa="SA-NS300"
EXT:
SERVER: KnOS/3.2 UPnP/1.0 DMP/3.5

<?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:GetProtocolInfoResponse xmlns:u="urn:schemas-upnp-org:service:ConnectionManager:1">
     <Source></Source>
     <Sink>http-get:*:audio/L16;rate=44100;channels=1:
        DLNA.ORG_PN=LPCM;DLNA.ORG_FLAGS=9D3000000000000000000000000000000,
        http-get:*:audio/L16;rate=44100;channels=2:DLNA.ORG_PN=LPCM;
        DLNA.ORG_FLAGS=9D3000000000000000000000000000000...[truncated]...</Sink>
   </u:GetProtocolInfoResponse>
 </s:Body>
</s:Envelope>
```

>>> outgoing TCP message to do a "`SetAVTransportURI`" on the device's `AVTransport` URL which we fetched previously. How do I know which actions to send to which device, and when, and in what order? It's all described in the "UPnP AV Architecture" document. In this case, I'm using `SetAVTransportURI` to tell the device which URI it's going to request audio from. (Hopefully, the audio will come in one of the mime-types that the device can accept!) I could have picked any URI here. I could have picked the URI of an mp3 off the internet, for instance, and the speakers would get ready to play that. But instead I picked "http://192.168.0.128:62890/dummy.l16", which is a URI I invented on an IP+port that my console application itself has set up a mini-http-server on. This way, the speakers will get their audio straight from my own app. The meaning of the `SetAVTransportURI` action and response, and all its IN and OUT parameters, is explained in the "AVTransport:1" document.

```
POST /AVTransport/ctrl HTTP/1.1
Host: 192.168.0.122:8080
Content-Length: 449
Content-Type: text/xml; charset="utf-8"
SOAPAction: "urn:schemas-upnp-org:service:AVTransport:1#SetAVTransportURI"

<?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:SetAVTransportURI xmlns:u="urn:schemas-upnp-org:service:AVTransport:1">
      <InstanceID>0</InstanceID>
      <CurrentURI>http://192.168.0.128:62890/dummy.l16</CurrentURI>
      <CurrentURIMetaData></CurrentURIMetaData>
```

```
    </u:SetAVTransportURI>
  </s:Body>
</s:Envelope>
```

<<< response on the same TCP socket. There were no OUT parameters for this action, so the response doesn't have any.

```
HTTP/1.1 200 OK
CONTENT-LENGTH: 332
CONTENT-TYPE: text/xml; charset="utf-8"
X-AV-Server-Info: av="5.0"; cn="Sony Corporation"; mn="SA-NS300"; mv="1.00"
X-AV-Physical-Unit-Info: pa="SA-NS300"
EXT:
SERVER: KnOS/3.2 UPnP/1.0 DMP/3.5

<?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:SetAVTransportURIResponse xmlns:u="urn:schemas-upnp-org:service:AVTransport:1">
    </u:SetAVTransportURIResponse>
  </s:Body>
</s:Envelope>
```

>>> outgoing TCP message to do a "Play" soap action on the device's `AVTransport` URL, the same as we retrieved and used earlier. Once again it's possible to re-use the same TCP socket as for previous requests, but I chose to create a fresh socket each time (or use a more powerful library to abstract all this away). Once again, the meaning and parameters of this `Play` action are documented in the "AVTransport:1" document.

```
POST /AVTransport/ctrl HTTP/1.1
Host: 192.168.0.122:8080
Content-Length: 329
Content-Type: text/xml; charset="utf-8"
SOAPAction: "urn:schemas-upnp-org:service:AVTransport:1#Play"

<?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 xmlns:u="urn:schemas-upnp-org:service:AVTransport:1">
      <InstanceID>0</InstanceID>
      <Speed>1</Speed>
    </u:Play>
  </s:Body>
</s:Envelope>
```

<<< response. In response to the `Play` message, the device starts requesting data from the URI we told it to use, and starts playing it! We'll get to that soon. In the meantime, it gives acknowledgement.

```
HTTP/1.1 200 OK
CONTENT-LENGTH: 306
CONTENT-TYPE: text/xml; charset="utf-8"
X-AV-Server-Info: av="5.0"; cn="Sony Corporation"; mn="SA-NS300"; mv="1.00"
X-AV-Physical-Unit-Info: pa="SA-NS300"
EXT:
SERVER: KnOS/3.2 UPnP/1.0 DMP/3.5

<?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:PlayResponse xmlns:u="urn:schemas-upnp-org:service:AVTransport:1">
    </u:PlayResponse>
  </s:Body>
</s:Envelope>
```

<<< Request using http-get from the speakers. We told the speakers to use "*http://192.168.0.128:62890/dummy.l16*" as their media source. So, they follow this protocol and open a TCP connection to this ip and port and make a GET request for the specified path. In my console app, I implemented a TCP server which listens for requests on this IP+port, so I can see exactly what request my speakers made. This is a standard http request, so it's in UTF8 and uses CRLF and we have to keep reading the request until we get a blank line at the end. It's interesting to see these headers! We see that Sony used WinAmp code to implement their speakers.

```
GET /dummy.l16 HTTP/1.1
Host: 192.168.0.128:36743
Icy-MetaData: 1
Connection: close
transferMode.dlna.org: Streaming
User-Agent: WinampMPEG/2.8
Accept: */*
X-AV-Client-Info: av="5.0"; cn="Sony Corporation"; mn="SA-NS300"; mv="1.00"
X-AV-Physical-Unit-Info: pa="SA-NS300"
```

>>> response from our TCP-server to the speakers. This is a normal response to an http-get message: it has the headers (in UTF8 with CRLF and finished with a blank line). After that, we send the body of our response -- that's to say, we send raw PCM audio over the TCP socket. The mime-type we specified in the Content-Header tells the speakers what they're getting; specification is at http://tools.ietf.org/html/rfc2586. It's PCM data, two channels: first 16bit sample from left channel, then first 16bit sample from right channel, then second 16bit sample from left channel, and so on. Each sample is a 16bit twos-complement number. I think it's in "network byte order", most significant byte first, but I get confused. We can just continue sending data over this TCP stream for as long as we like. We can send at some rate. Some clients will try to slurp up as much in advance as they possibly can. Other clients will only receive data for up to a few seconds ahead (so as to cope with random network delays).

```
HTTP/1.1 200 OK
Content-Type: audio/L16;rate=44100;channels=2

....raw PCM data, in audio/L16 format.....
....Trallalla..BOOM..deeay..............
.........
...
```

```
.
.
```

There are other useful actions in the attached code sample -- `GetTransportInfo`, `GetPositionInfo`, `Stop`. They all follow the same pattern as here.

Implementing the server half of the protocol

Now we have to implement the protocol. I'm going to explain the implementation "backwards" - first I'll show the implementation of the mini-HTTP server which serves up media, and next I'll show the rest of the protocol.

The server will listen for incoming TCP requests. We have to decide which network adapter to listen on, i.e. what local IP address to use. I've hard-coded it here, but in the attached code we actually figure out the best local IP address during the SSDP discovery phase.

```
Dim LocalIp = "192.168.0.128"
```

.NET code: Next, infrastructure for setting up the server TCP-listener that accepts connections. Here's one version that uses .NET 4.5 and runs on the desktop (e.g., console apps) on any machine that has .NET45 installed. It uses `data.Start(1)` to only queue up to 1 pending request. The final line gives us the "`dataLocation`" URI that we're going to tell the `AVTransport` to play from.

```
data.Start(1)
Dim dataTask = Task.Run(
    Async Function()
        While True
            Using dataConnection = Await data.AcceptTcpClientAsync(),
                  dataStream = dataConnection.GetStream()
                Try
                    Await DoUpnpDataDialogAsync(dataStream, dataStream, progress)
                    Exit While
                Catch ex As IO.IOException When ex.HResult = -2146232800
                    ' Why would a DigitalMediaRenderer open a data connection only to
                    ' close it again immediately and reopen another one? I don't know,
                    ' but some do so we offer them a second opportunity.
                End Try
            End Using
        End While
    End Function)

Dim dataLocation = New Uri("http://" & data.Server.LocalEndPoint.ToString() & "/dummy.l16")
```

WinRT code: The above code isn't allowed in app-store apps. Here's what you'd use instead:

```
Using data As New Windows.Networking.Sockets.StreamSocketListener()
    AddHandler data.ConnectionReceived,
        Async Sub(sender As Windows.Networking.Sockets.StreamSocketListener, _
              args As Windows.Networking.Sockets.StreamSocketListenerConnectionReceivedEventArgs)
            Using args.Socket
                Using data_reader = IO.WindowsRuntimeStreamExtensions.AsStreamForRead(
```

```
                                    args.Socket.InputStream),
                data_writer = IO.WindowsRuntimeStreamExtensions.AsStreamForWrite(
                                    args.Socket.OutputStream)
                Try
                    Await DoUpnpDataDialogAsync(data_writer, data_reader, progress)
                Catch ex As Exception When ex.HResult = -2147014842
                End Try
            End Using
        End Using
    End Sub

    ' CAPABILITY: privateNetworkClientServer
    Await data.BindServiceNameAsync("")

    Dim dataLocation = New Uri("http://" & device.LocalUri.Host & _
          ":" & data.Information.LocalPort & "/stream.l16")

    ...
End Using
```

DoUpnpDataDialogAsync. The above code merely sets up sockets. The meat of the server is in this function:

```
Async Function DoUpnpDataDialogAsync(writer As IO.Stream, reader As IO.Stream,
                                     progress As IProgress(Of String)) As Task
    Dim request = ""
    Using sreader As New IO.StreamReader(reader, Text.Encoding.UTF8, False, 1024, True)
        While True
            Dim header = Await sreader.ReadLineAsync()
            If String.IsNullOrWhiteSpace(header) Then Exit While
            request &= header & vbCrLf
        End While
    End Using

    Dim response = "HTTP/1.1 200 OK" & vbCrLf &
                   "Content-Type: audio/L16;rate=44100;channels=2" & vbCrLf & vbCrLf
    Dim responsebuf = Text.Encoding.UTF8.GetBytes(response)
    Await writer.WriteAsync(responsebuf, 0, responsebuf.Length)

    Dim MaryHadALittleLamb = {247, 220, 196, 220, 247, 247, 247, 220, 220, 220, 247, 294, 294}
    Dim phase = 0.0
    Dim buf = New Byte(4096 * 4 - 1) {}
    For imusic = 0 To MaryHadALittleLamb.Length * 8 - 1
        Dim freq = If(imusic Mod 8 = 0, 0, MaryHadALittleLamb(imusic \ 8))
        For i = 0 To buf.Length \ 4 - 1
            phase += freq / 44100 * 2 * Math.PI
            Dim amplitude = If(freq = 0, 0, Math.Sin(phase) * (Short.MaxValue - 1))
            Dim bb = BitConverter.GetBytes(CShort(amplitude))
            buf(i * 4 + 0) = bb(1) ' left channel, MSB
            buf(i * 4 + 1) = bb(0) ' left channel, LSB
            buf(i * 4 + 2) = bb(1) ' right channel, MSB
            buf(i * 4 + 3) = bb(0) ' right channel, LSB
```

```
        Next
        Await writer.WriteAsync(buf, 0, buf.Length)
    Next
    Await writer.FlushAsync()
End Function
```

Implementing the client side of the protocol

In this article, I'll start by showing low-level code to implement the protocol. After that, I'll show some higher-level wrappers that achieve the same end.

I'm writing the article this way because I believe it's always important for you, the developer, to understand from the ground up how a protocol behaves. No network abstraction/wrapper is ever clean enough that you can ignore what's underneath.

WinRT code for SSDP: I'm giving the WinRT version of this code first, because the WinRT networking APIs are cleaner than .NET. The code is very straightforward. It sends out the muticast message, and waits up to 1200ms for a response. It gathers up all the responses it receives. (There's a race-condition here that should be fixed: the MessageReceived handler might be called simultaneously on different threads if replies come together).

```
Dim remoteIp As New Windows.Networking.HostName("239.255.255.250"), remotePort = "1900"
Dim reqbuf = AsBuffer(CreateSsdpRequest(remoteIp.RawName() & ":" & remotePort)).AsBuffer()
Dim locs As New HashSet(Of Tuple(Of Uri, Uri))

Using socket As New Windows.Networking.Sockets.DatagramSocket()
    AddHandler socket.MessageReceived,
        Sub(sender, e)
            If e.LocalAddress.IPInformation.NetworkAdapter.IanaInterfaceType = 24 Then Return
            ' don't show anything from the loopback adapter
            Using reader = e.GetDataReader()
                Dim responsebuf = New Byte(CInt(reader.UnconsumedBufferLength - 1)) {}
                reader.ReadBytes(responsebuf)
                Dim location = ParseSsdpResponse(responsebuf, responsebuf.Length)
                locs.Add(Tuple.Create(location, New Uri("http://" & e.LocalAddress.RawName)))
            End Using
        End Sub

    ' CAPABILITY: PrivateNetworks
    Await socket.BindEndpointAsync(Nothing, "")
    socket.Control.OutboundUnicastHopLimit = 1
    socket.JoinMulticastGroup(remoteIp)
    ' There's no WinRT equivalent of ReuseAddress, but it seems not to be needed

    Using stream = Await socket.GetOutputStreamAsync(remoteIp, remotePort)
        Await stream.WriteAsync(reqbuf)
    End Using
    Await Task.Delay(1200)
End Using
```

.NET code for SSDP: This code is a little uglier than the WinRT equivalent. As far as I can tell, it has to explicitly call out which network interface it's listening on...

```
Dim remoteEp = New Net.IPEndPoint(Net.IPAddress.Parse("239.255.255.250"), 1900)
Dim locs As New HashSet(Of Tuple(Of Uri, Uri))

Dim localIps =
     From network In Net.NetworkInformation.NetworkInterface.GetAllNetworkInterfaces()
     Where network.OperationalStatus = Net.NetworkInformation.OperationalStatus.Up
     Where network.NetworkInterfaceType <> Net.NetworkInformation.NetworkInterfaceType.Loopback
     Let localAddr = (From uaddr In network.GetIPProperties.UnicastAddresses _
         Where uaddr.Address.AddressFamily = Net.Sockets.AddressFamily.InterNetwork).FirstOrDefault
     Where Not localAddr Is Nothing
     Select localAddr.Address

For Each localIp In localIps
    Using socket As New Net.Sockets.Socket(Net.Sockets.AddressFamily.InterNetwork,
                                           Net.Sockets.SocketType.Dgram,
                                           Net.Sockets.ProtocolType.Udp)
        socket.SetSocketOption(Net.Sockets.SocketOptionLevel.Socket,
                               Net.Sockets.SocketOptionName.ReuseAddress, True)
        socket.SetSocketOption(Net.Sockets.SocketOptionLevel.IP,
                               Net.Sockets.SocketOptionName.AddMembership,
                               New Net.Sockets.MulticastOption(remoteEp.Address))
        socket.SetSocketOption(Net.Sockets.SocketOptionLevel.IP,
                               Net.Sockets.SocketOptionName.MulticastTimeToLive, 1)

        socket.Bind(New Net.IPEndPoint(localIp, 0))
        Dim receiverTask = Task.Run(
            Sub()
                Dim responsebuf = New Byte(51200) {}
                Do
                    Try
                        Dim ep As Net.EndPoint = New Net.IPEndPoint(Net.IPAddress.Any, 0)
                        Dim len = socket.ReceiveFrom(responsebuf, ep)
                        Dim location = ParseSsdpResponse(responsebuf, len)
                        locs.Add(Tuple.Create(location, New Uri("http://" & localIp.ToString)))
                    Catch ex As Net.Sockets.SocketException When ex.ErrorCode = 10004
                        Return ' WSACancelBlockingCall, when the socket is closed
                    End Try
                Loop
            End Sub)

        Dim request = CreateSsdpRequest(remoteEp.ToString())
        socket.SendTo(request, remoteEp)
        Await Task.Delay(1200)

        socket.Close()
        Await receiverTask
    End Using
Next
```

Helper functions for SSDP packets: one function to construct the SSDP request packet, and one function to parse the responses and extract out that crucial LOCATION: header.

```
Function CreateSsdpRequest(authority As String) As Byte()
    Dim request = "M-SEARCH * HTTP/1.1" & vbCrLf &
                  "HOST: " & authority & vbCrLf &
                  "ST:urn:schemas-upnp-org:device:MediaRenderer:1" & vbCrLf &
                  "MAN: ""ssdp:discover""" & vbCrLf &
                  "MX: 1" & vbCrLf &
                  "" & vbCrLf
    Return Text.Encoding.UTF8.GetBytes(request)
End Function

Function ParseSsdpResponse(responsebuf As Byte(), len As Integer) As Uri
    Dim response = Text.Encoding.UTF8.GetString(responsebuf, 0, len)
    Return (From line In response.Split({vbCr(0), vbLf(0)})
            Where line.ToLowerInvariant().StartsWith("location:")
            Select New Uri(line.Substring(9).Trim())).FirstOrDefault
End Function
```

Retrieve service description: What we have so far is a `HashSet` with the Location: URIs where we can learn about devices, and also the LocalIp where we found each one. Now we need to retrieve those URIs, and parse them to discover the `ConnectionManager` and `AVTransport` control URIs:

```
Imports <xmlns:ud="urn:schemas-upnp-org:device-1-0">

Dim desc_request = MakeRawGetRequest(deviceLocation)
Dim desc_response = Await GetXmlAsync(desc_request, progress)
Dim desc_friendlyName = desc_response.<ud:root>.<ud:device>.<ud:friendlyName>.Value
Dim desc_services = desc_response.<ud:root>.<ud:device>.<ud:serviceList>.<ud:service>

Dim connectionManagerUri =
     (From service In desc_services
      Where service.<ud:serviceType>.Value = "urn:schemas-upnp-org:service:ConnectionManager:1"
      Select New Uri(deviceLocation, service.<ud:controlURL>.Value)).FirstOrDefault

Dim avTransportUri =
     (From service In desc_services
      Where service.<ud:serviceType>.Value = "urn:schemas-upnp-org:service:AVTransport:1"
      Select New Uri(deviceLocation, service.<ud:controlURL>.Value)).FirstOrDefault
```

Do Soap Actions on the ConnectionManager and AVTransport. The rest of the protocol is plain sailing now.

```
Imports <xmlns:uc="urn:schemas-upnp-org:service:ConnectionManager:1">
Imports <xmlns:ut="urn:schemas-upnp-org:service:AVTransport:1">


Dim getprotocol_request = MakeSoapRequest(connectionManagerUri, <uc:GetProtocolInfo/>, {})
Dim getprotocol_response = Await GetSoapAsync(getprotocol_request, progress)

Dim mimeTypes = getprotocol_response.<Sink>.Value.Split({","c})

Dim seturi_request = MakeSoapRequest(avTransportUri, <ut:SetAVTransportURI/>,
```

```
        {"InstanceID", "0", "CurrentURI", _
         dataLocation.ToString(), "CurrentURIMetaData", ""})
Dim seturi_response = Await GetSoapAsync(seturi_request, progress)

Dim play_request = MakeRawSoapRequest(avTransportUri, <ut:Play/>,
       {"InstanceID", "0", "Speed", "1"})
Dim play_response = Await GetSoapAsync(play_request, progress)
```

Helper functions for making the various TCP requests:

```
Function MakeRawGetRequest(requestUri As Uri) As Tuple(Of Uri, Byte())
    Dim s = "GET " & requestUri.PathAndQuery & " HTTP/1.1" & vbCrLf &
            "Host: " & requestUri.Host & ":" & requestUri.Port & vbCrLf & vbCrLf
    Return Tuple.Create(requestUri, Text.Encoding.UTF8.GetBytes(s))
End Function

Function MakeRawSoapRequest(requestUri As Uri, soapAction As XElement,
                            args As String()) As Tuple(Of Uri, Byte())
    Dim soapSchema = soapAction.Name.NamespaceName
    Dim soapVerb = soapAction.Name.LocalName
    Dim argpairs As New List(Of Tuple(Of String, String))
    For i = 0 To args.Length - 1 Step 2
        argpairs.Add(Tuple.Create(args(i), args(i + 1)))
    Next

    Dim s = "POST " & requestUri.PathAndQuery & " HTTP/1.1" & vbCrLf &
            "Host: " & requestUri.Authority & vbCrLf &
            "Content-Length: ?" & vbCrLf &
            "Content-Type: text/xml; charset=""utf-8""" & vbCrLf &
            "SOAPAction: """ & soapSchema & "#" & soapVerb & """" & vbCrLf &
            "" & vbCrLf &
            "<?xml version=""1.0""?>" & vbCrLf &
            "<s:Envelope xmlns:s=""http://schemas.xmlsoap.org/soap/" & _
            "envelope/"" s:encodingStyle=""http://schemas.xmlsoap.org/soap/encoding/"">" & vbCrLf &
            "  <s:Body>" & vbCrLf &
            "    <u:" & soapVerb & " xmlns:u=""" & soapSchema & """>" & vbCrLf &
            String.Join(vbCrLf, (From arg In argpairs Select "      <" & _
            arg.Item1 & ">" & arg.Item2 & "</" & arg.Item1 & ">").Concat({""})) &
            "    </u:" & soapVerb & ">" & vbCrLf &
            "  </s:Body>" & vbCrLf &
            "</s:Envelope>" & vbCrLf
    '
    Dim len = Text.Encoding.UTF8.GetByteCount(s.Substring(s.IndexOf("<?xml")))
    s = s.Replace("Content-Length: ?", "Content-Length: " & len)
    Return tuple.Create(requestUri, Text.Encoding.UTF8.GetBytes(s))
End Function

Async Function GetXmlAsync(request As Tuple(Of Uri, Byte())) As Task(Of XDocument)
    Dim requestUri = request.Item1, requestBody = request.Item2
    Using socket As New Net.Sockets.TcpClient(requestUri.DnsSafeHost, requestUri.Port), stream = socket.GetStream()
        If Not progress Is Nothing Then progress.Report("--------->" & _
```

```
            vbCrLf & Text.Encoding.UTF8.GetString(requestBody))
        Await stream.WriteAsync(requestBody, 0, requestBody.Length)
        Await stream.FlushAsync()
        '
        Dim headers = "", body = ""
        Using sreader As New IO.StreamReader(stream, Text.Encoding.UTF8, False, 1024, True)
            Dim len = 0
            While True
                Dim header = Await sreader.ReadLineAsync()
                If String.IsNullOrWhiteSpace(header) Then Exit While
                If header.ToLower().StartsWith("content-length:") Then len = CInt(header.Substring(15).Trim())
                headers &= header & vbCrLf
            End While
            Dim buf = New Char(1024) {}
            While Text.Encoding.UTF8.GetByteCount(body) < len
                Dim red = Await sreader.ReadAsync(buf, 0, 1024)
                body &= New String(buf, 0, red)
            End While
        End Using
        '
        If Not headers.StartsWith("HTTP/1.1 200 OK") Then
            Throw New Net.Http.HttpRequestException(headers & vbCrLf & body)
        End If
        Return XDocument.Parse(body)
    End Using
End Function

Async Function GetSoapAsync(request As Tuple(Of Uri, Byte())) As Task(Of XElement)
    Dim requestLines = Text.Encoding.UTF8.GetString(request.Item2).Split({vbCrLf})
    Dim soapAction = (From s In requestLines Where _
      s.ToLower().StartsWith("soapaction:")).FirstOrDefault.Substring(11).Trim(" "c, """"c)
    Dim soapResponse As Xml.Linq.XName = "{" & soapAction.Replace("#", "}") & _
        "Response" ' e.g. {schema}actionResponse
    '
    Dim xml = Await GetXmlAsync(http, request)
    Dim body = xml.<soap:Envelope>.<soap:Body>.Elements(soapResponse).FirstOrDefault
    If body Is Nothing Then Throw New Net.Http.HttpRequestException("no soap body")
    Return body
End Function
```

Higher level of abstraction: Use HttpClient and XML-literals

The code helpers above constructed HTTP and SOAP requests out of raw strings and bytes. That's obviously a bad solution -- clumsy and error-prone. It's better to use `HttpClient` instead, and VB's support for XML literals:

```
Function MakeGetRequest(requestUri As Uri) As Net.Http.HttpRequestMessage
    Return New Net.Http.HttpRequestMessage(Net.Http.HttpMethod.Get, requestUri)
End Function
```

```
Function MakeSoapRequest(requestUri As Uri, soapAction As XElement,
                         args As String()) As Net.Http.HttpRequestMessage
    Dim m As New Net.Http.HttpRequestMessage(Net.Http.HttpMethod.Post, requestUri)
    m.Headers.Add("SOAPAction", """" & soapAction.Name.NamespaceName & _
                  "#" & soapAction.Name.LocalName & """")

    Dim c = <?xml version="1.0"?>
            <soap:Envelope soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
                <soap:Body>
                      <<%= soapAction.Name %>>
                          <%= Iterator Function()
                                  For i = 0 To args.Length - 1 Step 2
                                       Yield <<%= args(i) %>><%= args(i + 1) %></>
                                  Next
                              End Function() %>
                      </>
                </soap:Body>
            </soap:Envelope>

    m.Content = New Net.Http.StringContent(c.Declaration.ToString() & vbCrLf & _
                    c.ToString() & vbCrLf, Text.Encoding.UTF8, "text/xml")
    Return m
End Function

<Runtime.CompilerServices.Extension>
Async Function GetXmlAsync(http As Net.Http.HttpClient,
                           request As Net.Http.HttpRequestMessage) As Task(Of XDocument)
    Using response = Await http.SendAsync(request)
        response.EnsureSuccessStatusCode()

        ' workaround for bug in HttpClient:
        If Not response.Content.Headers.ContentType.CharSet Is Nothing Then
            response.Content.Headers.ContentType.CharSet = _
               response.Content.Headers.ContentType.CharSet.Trim("""c)

        Dim body = Await response.Content.ReadAsStringAsync()
        Return XDocument.Parse(body)
    End Using
End Function

<Runtime.CompilerServices.Extension>
Async Function GetSoapAsync(http As Net.Http.HttpClient,
                            request As Net.Http.HttpRequestMessage) As Task(Of XElement)
    Dim soapAction = request.Headers.GetValues("SOAPAction").FirstOrDefault.Trim("""c) ' e.g. schema#action
    Dim soapResponse As Xml.Linq.XName = _
        "{" & soapAction.Replace("#", "}") & _
        "Response" ' e.g. {schema}actionResponse
    '
    Dim xml = Await GetXmlAsync(http, request)
    Dim body = xml.<soap:Envelope>.<soap:Body>.Elements(soapResponse).FirstOrDefault
```

```
    If body Is Nothing Then Throw New Net.Http.HttpRequestException("no soap body")
    Return body
End Function
```

Higher level of abstraction: use WinRT Device Enumeration

Actually, Windows 8 has SSDP discovery built in. We can remove our existing SSDP code and replace it with "WinRT Device Enumeration". I described win8 device-enumeration more fully in a separate article: http://www.codeproject.com/Articles/458550/Device-enumeration-in-Windows-8.

No WinRT probes by device-type. Recall that we send an SSDP probe for all devices that have the device-type "urn:schemas-upnp-org:device:MediaRenderer:1". Well, Windows sends out SSDP probes for absolutely all devices -- *but it not longer lets us query its results by device-type*.

What we'll do instead is rely on the fact that all MediaRenderer:1 devices have a `RenderingControl` service and a `ConnectionManager` interface, and (optionally) an `AVTransport` interface. So we'll look for any device at all that has these interfaces.

It raises an interesting philosophical question -- what if there exist some devices whose type is not MediaRenderer:1, but they expose ConnectionManager and AVTransport interfaces? Should our code work with them? What will they do? I don't know!

No WinRT probes by service-type string. Also, our own SSDP implementation looked for services with type "urn:schemas-upnp-org:service:ConnectionManager:1". That's how the UPnP protocol says to find the `ConnectionManager` service. Well, WinRT Device Enumeration calls them *interfaces* rather than *services*. And it doesn't let you look them up by string. Instead you have to look them up by GUID. I've written the GUIDS out below. How did I discover what the GUIDs are? There isn't a good way. You just have to plug in some actual physical devices which you know have the correct service type, and test out the WinRT device enumeration APIs, and discover which GUIDs they report for their `DeviceInterface::System.Devices.InterfaceClassGuid` property.

```
Dim RenderingControlInterfaceClass = New Guid("8660e926-ff3d-580c-959e-8b8af44d7cde")
Dim ConnectionManagerInterfaceClass = New Guid("ae9eb9c4-8819-51d8-879d-9a42ffb89d4e")
Dim AVTransportInterfaceClass = New Guid("4c38e836-6a2f-5949-9406-1788ea20d1d5")
```

WinRT way to find control URL. In our own SSDP implementation, we looked for the `service.<ud:controlURL>` element of the XML description. This gave us the control-URL for the `ConnectionManager` and `AVTransport` services. In WinRT, the `controlURL` property is hidden away. I found it by looking in the Windows SDK header files for PKEYs, and figuring out by experiment that this particular PKEY refers to the `controlURL` property of a device:

```
Dim PKEY_PNPX_ServiceControlUrl = "{656A3BB3-ECC0-43FD-8477-4AE0404A96CD},16388"
```

WinRT device-enumeration code for UPnP. Here then is the code you'd use in WinRT to discover the control-URLs of UPnP devices:

```
im RenderingControls =
     Await Windows.Devices.Enumeration.Pnp.PnpObject.FindAllAsync(
               Windows.Devices.Enumeration.Pnp.PnpObjectType.DeviceInterface,
               {"System.Devices.DeviceInstanceId", "System.Devices.InterfaceClassGuid", "System.Devices.ContainerId"},
               "System.Devices.InterfaceClassGuid:=""{" & RenderingControlInterfaceClass.ToString() & "}""")

For Each device In RenderingControls
    If Not device.Properties.ContainsKey("System.Devices.DeviceInstanceId") Then Continue For
    If Not device.Properties.ContainsKey("System.Devices.ContainerId") Then Continue For
    Dim id = CStr(device.Properties("System.Devices.DeviceInstanceId"))
    Dim containerId = CType(device.Properties("System.Devices.ContainerId"), Guid)
```

```
    Dim ConnectionManagerInterface =
         (Await Windows.Devices.Enumeration.Pnp.PnpObject.FindAllAsync(
                   Windows.Devices.Enumeration.Pnp.PnpObjectType.DeviceInterface,
                   {"System.Devices.DeviceInstanceId", _
                   "System.Devices.InterfaceClassGuid", PKEY_PNPX_ServiceControlUrl},
                   "System.Devices.DeviceInstanceId:=""" & id & _
                   """ AND System.Devices.InterfaceClassGuid:=""{" & _
                   ConnectionManagerInterfaceClass.ToString() & "}""")).FirstOrDefault

    If ConnectionManagerInterface Is Nothing Then Continue For
    If Not ConnectionManagerInterface.Properties.ContainsKey(PKEY_PNPX_ServiceControlUrl) Then Continue For
    Dim connectionManagerUrl = New Uri(CStr(ConnectionManagerInterface.Properties(PKEY_PNPX_ServiceControlUrl)))


    Dim AVTransportInterface =
         (Await Windows.Devices.Enumeration.Pnp.PnpObject.FindAllAsync(
                    Windows.Devices.Enumeration.Pnp.PnpObjectType.DeviceInterface,
                    {"System.Devices.DeviceInstanceId", _
                    "System.Devices.InterfaceClassGuid", PKEY_PNPX_ServiceControlUrl},
                    "System.Devices.DeviceInstanceId:=""" & id & _
                    """ AND System.Devices.InterfaceClassGuid:=""{" & _
                    AVTransportInterfaceClass.ToString() & "}""")).FirstOrDefault

    If Not AVTransportInterface Is Nothing AndAlso Not _
           AVTransportInterface.Properties.ContainsKey(PKEY_PNPX_ServiceControlUrl) Then
               AVTransportInterface = Nothing
    Dim avTransportUrl = If(AVTransportInterface Is Nothing, Nothing, _
        New Uri(CStr(AVTransportInterface.Properties(PKEY_PNPX_ServiceControlUrl))))


    Dim Container = Await Windows.Devices.Enumeration.Pnp.PnpObject.CreateFromIdAsync(
                            Windows.Devices.Enumeration.Pnp.PnpObjectType.DeviceContainer,
                            containerId.ToString(),
                            {"System.Devices.Connected", "System.Devices.FriendlyName"})

    If Container Is Nothing Then Continue For
    If Not Container.Properties.ContainsKey("System.Devices.Connected") Then Continue For
    If Not Container.Properties.ContainsKey("System.Devices.FriendlyName") Then Continue For
    Dim connected = CBool(Container.Properties("System.Devices.Connected"))
    Dim friendlyName = CStr(Container.Properties("System.Devices.FriendlyName"))
    If Not connected Then Continue For

    ...
Next
```

WinRT code to discover local IP for a discovered device. There's one final task. We needed to know our local IP, the IP that the device would connect to us over, so that we could construct the right URL for the device to fetch from. In our manual SSDP code, we got this because we knew which IP address we'd received the UDP packet from the device. But in the WinRT device enumeration code, we need an alternative:

```
Try
    Dim localUri As Uri
    Using c As New Net.Sockets.TcpClient(connectionManagerUrl.DnsSafeHost,
                                         connectionManagerUrl.Port)
        Dim addr = CType(c.Client.LocalEndPoint, Net.IPEndPoint).Address
        Dim localHost = addr.ToString()
        If addr.AddressFamily = Net.Sockets.AddressFamily.InterNetworkV6 Then
            localHost = "[" & localHost & "]"
        End If
        localUri = New Uri("http://" & localHost)
    End Using
Catch ex  As Net.Sockets.SocketException
    ' oh well, I suppose we can't connect - sometimes the "connected" flag is incorrect
End Try
```

Note on code

None of this is robust code. I tried to write the absolute minimum of code that would explain the APIs involved. The idea is that you should read this article to learn the APIs, and then use them within your own application's infrastructure.

Disclaimer: Although I work at Microsoft on the VB/C# language team, this article is strictly a personal amateur effort based on public information and experimentation - it's not in my professional area of expertise, has been written in my own free time, and neither Microsoft nor I make any claims about its correctness...

Apart that is from the extensive use of "`Await`" and "`Async`" in this code. I was on the Microsoft VB/C# language team that designed these keywords, and I did a large chunk of their implementation into the VB and C# compilers. I'm jolly proud of them, I'm glad to see that they make these protocols so much easier to implement, and this code uses them in the Microsoft-recommended best-practice way!

License

About the Author

ljw1004

Technical Lead

United States

Lucian studied theoretical computer science in Cambridge and Bologna, and then moved into the computer industry. Since 2004 he's been paid to do what he loves -- designing and implementing programming languages! The articles he writes on CodeProject are entirely his own personal hobby work, and do not represent the position or guidance of the company he works for. (He's on the VB/C# language team at Microsoft).

Comments and Discussions

11 messages have been posted for this article Visit **https://www.codeproject.com/Articles/458807/UPnP-code-for-Windows-8** to post and view comments on this article, or click **here** to get a print view with messages.