electric monk

Ferry Boender

Programmer, DevOpper, Open Source enthusiast.

Blog

Exploring UPnP with Python

Tuesday, July 5th, 2016

UPnP stands for Universal Plug and Play. It's a standard for discovering and interacting with services offered by various devices on a network. Common examples include:

- Discovering, listing and streaming media from media servers
- Controlling home network routers: e.g. automatic configuration of port forwarding to an internal device such as your Playstation or XBox.

In this article we'll explore the client side (usually referred to as the *Control Point* side) of UPnP using Python. I'll explain the different protocols used in UPnP and show how to write some basic Python code to discover and interact with devices. There's lots of information on UPnP on the Internet, but a lot of it is fragmented, discusses only certain aspects of UPnP or is vague on whether we're dealing with the client or a server. The UPnP standard itself is quite an easy read though.

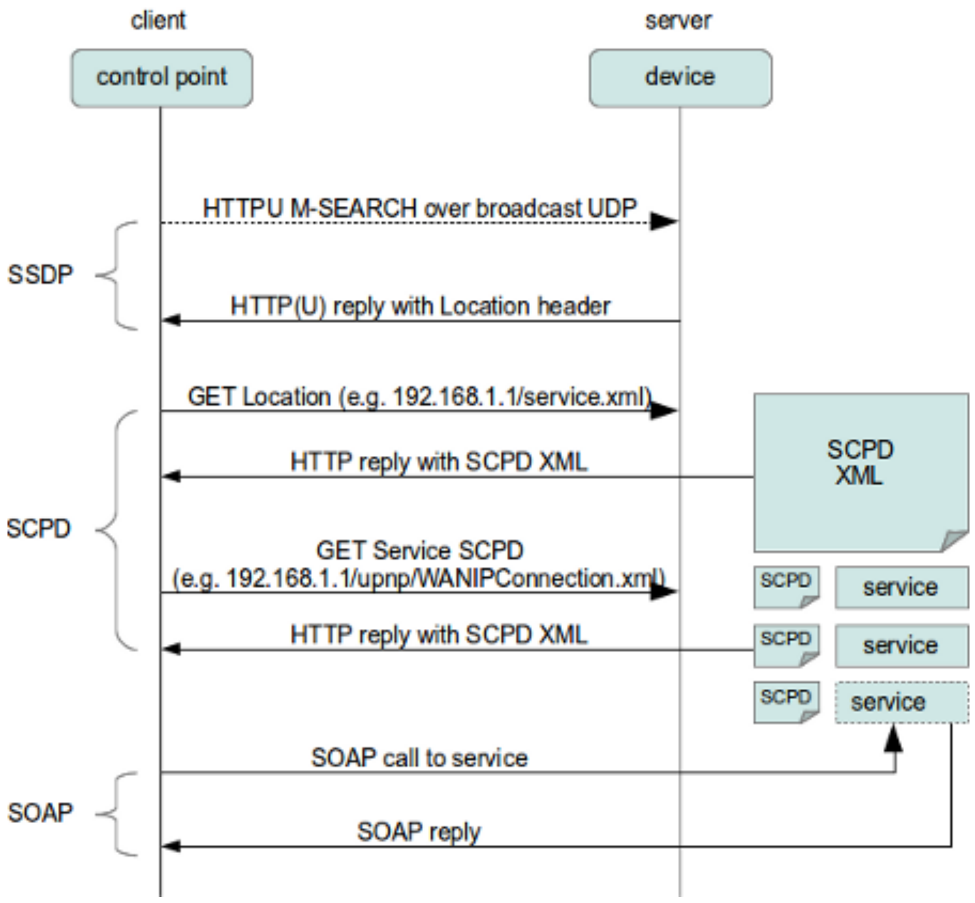
Disclaimer: The code in this article is rather hacky and not particularly robust. Do not use it as a basis for any real projects.

Protocols

UPnP uses a variety of different protocols to accomplish its goals:

- **SSDP**: *Simple Service Discovery Protocol*, for discovering UPnP devices on the local network.
- **SCPD**: *Service Control Point Definition*, for defining the actions offered by the various services.
- **SOAP**: *Simple Object Access Protocol*, for actually calling actions.

Here's a schematic overview of the flow of a UPnP session and where the different protocols come into play.


client
server
control point
device
HTTPU M-SEARCH over broadcast UDP
SSDP
HTTP(U) reply with Location header
GET Location (e.g. 192.168.1.1/service.xml)
HTTP reply with SCPD XML
SCPD XML
SCPD
GET Service SCPD (e.g. 192.168.1.1/upnp/WANIPConnection.xml)
SCPD
service
HTTP reply with SCPD XML
SCPD
service
SCPD
service
SOAP call to service
SOAP
SOAP reply

The standard flow of operations in UPnP is to first use SSDP to discover which UPnP devices are available on the network. Those devices return the location of an XML file which defines the various services offered by each device. Next we use SCPD on each service to discover the various actions offered by each service. Essentially, SCPD is an XML-based protocol which describes SOAP APIs, much like WSDL. Finally we use SOAP calls to interact with the services.

SSDP: Service Discovery

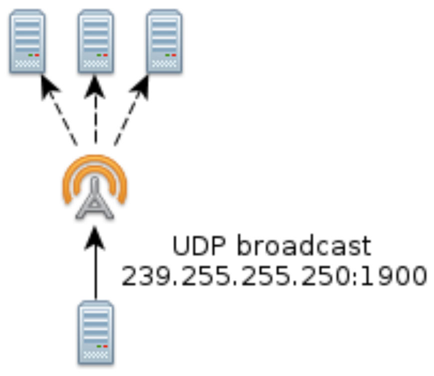
Lets take a closer look at SSDP, the Simple Service Discovery Protocol. SSDP operates over UDP rather than TCP. While TCP is a statefull protocol, meaning both end-points of the connection are aware of whom they're talking too, UDP is stateless. This means we can just throw UDP packets over the line, and we don't care much whether they are received properly or even received at all. UDP is often used in situations where missing a few packets is not a problem, such as streaming media.

SSDP uses HTTP over UDP (called HTTPU) in broadcasting mode. This allows all UPnP devices on the network to receive the requests regardless of whether we know where they are located. Here's a very simple example of how to perform an HTTPU query using Python:

```
import socket

msg = \
    'M-SEARCH * HTTP/1.1\r\n' \
    'HOST:239.255.255.250:1900\r\n' \
    'ST:upnp:rootdevice\r\n' \
    'MX:2\r\n' \
    'MAN:"ssdp:discover"\r\n' \
    '\r\n'

# Set up UDP socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
s.settimeout(2)
s.sendto(msg, ('239.255.255.250', 1900) )

try:
    while True:
        data, addr = s.recvfrom(65507)
        print addr, data
except socket.timeout:
    pass
```

This little snippet of code creates a HTTP message using the `M-SEARCH` HTTP method, which is specific to UPnP. It then sets up a UDP socket, and sends out the HTTPU message to IP address 239.255.255.250, port 1900. That IP is a special broadcast IP address. It is not actually tied to any specific server, like normal IPs. Port 1900 is the one which UPnP servers will listen on for broadcasts.

Next, we listen on the socket for any replies. The socket has a timeout of 2 seconds. This means that after not receiving any data on the socket after two seconds, the `s.recvfrom()` call times out, which raises an exception. The exception is caught, and the program continues.



You will recall that we don't know how many devices might be on the network. We also don't know where they are nor do we have any idea how fast they will respond. This means we can't be certain about the number of seconds we must wait for replies. This is the reason why so many UPnP control points (clients) are so slow when they scan for devices on the network.

In general all devices should be able to respond in less than 2 seconds. It seems that manufacturers would rather be on the safe side and sometimes wait up to 10 seconds for replies. A better approach would be to cache previously found devices and immediately check their availability upon startup. A full device search could then be done asynchronous in the background. Then again, many uPNP devices set the cache validaty timeout extremely low, so clients (if they properly implement the standard) are forced to rediscover them every time.

Anyway, here's the output of the M-SEARCH on my home network. I've stripped some of the headers for brevity:

```
('192.168.0.1', 1900) HTTP/1.1 200 OK
USN: uuid:2b2561a3-a6c3-4506-a4ae-247efe0defec::upnp:rootdevice
SERVER: Linux/2.6.18_pro500 UPnP/1.0 MiniUPnPd/1.5
LOCATION: http://192.168.0.1:40833/rootDesc.xml

('192.168.0.2', 53375) HTTP/1.1 200 OK
LOCATION: http://192.168.0.2:1025/description.xml
SERVER: Linux/2.6.35-31-generic, UPnP/1.0, Free UPnP Entertainment Service/0.6
USN: uuid:60c251f1-51c6-46ae-93dd-0a3fb55a316d::upnp:rootdevice
```

Two devices responded to our `M-SEARCH` query within the specified number of seconds. One is a cable internet router, the other is Fuppes, a UPnP media server. The most interesting things in these replies are the `LOCATION` headers, which point us to an SCPD XML file: `http://192.168.0.1:40833/rootDesc.xml`.

SCPD, Phase I: Fetching and parsing the root SCPD file

The SCPD XML file (`http://192.168.0.1:40833/rootDesc.xml`) contains information on the UPnP server such as the manufacturer, the services offered by the device, etc. The XML file is rather big and complicated. You can see the full version, but here's a grealy reduced one from my router:

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <device>
    <deviceType>urn:schemas-upnp-org:device:InternetGatewayDevice:1</deviceTyp
    <friendlyName>Ubee EVW3226</friendlyName>
    <serviceList>
      <service>
        <serviceType>urn:schemas-upnp-org:service:Layer3Forwarding:1</serviceT
        <controlURL>/ctl/L3F</controlURL>
        <eventSubURL>/evt/L3F</eventSubURL>
        <SCPDURL>/L3F.xml</SCPDURL>
      </service>
    </serviceList>
    <deviceList>
      <device>
        <deviceType>urn:schemas-upnp-org:device:WANDevice:1</deviceType>
        <friendlyName>WANDevice</friendlyName>
        <serviceList>
          <service>
            <serviceType>urn:schemas-upnp-org:service:WANCommonInterfaceConfig
            <serviceId>urn:upnp-org:serviceId:WANCommonIFC1</serviceId>
            <controlURL>/ctl/CmnIfCfg</controlURL>
            <eventSubURL>/evt/CmnIfCfg</eventSubURL>
            <SCPDURL>/WANCfg.xml</SCPDURL>
          </service>
        </serviceList>
        <deviceList>
          <device>
            <deviceType>urn:schemas-upnp-org:device:WANConnectionDevice:1</dev
            <friendlyName>WANConnectionDevice</friendlyName>
            <serviceList>
              <service>
                <serviceType>urn:schemas-upnp-org:service:WANIPConnection:1</s
                <controlURL>/ctl/IPConn</controlURL>
                <eventSubURL>/evt/IPConn</eventSubURL>
                <SCPDURL>/WANIPCn.xml</SCPDURL>
              </service>
            </serviceList>
          </device>
        </deviceList>
      </device>
    </deviceList>
  </device>
</root>
```

It consists of basically three important things:

- The URLBase
- Virtual Devices

- Services

URLBase

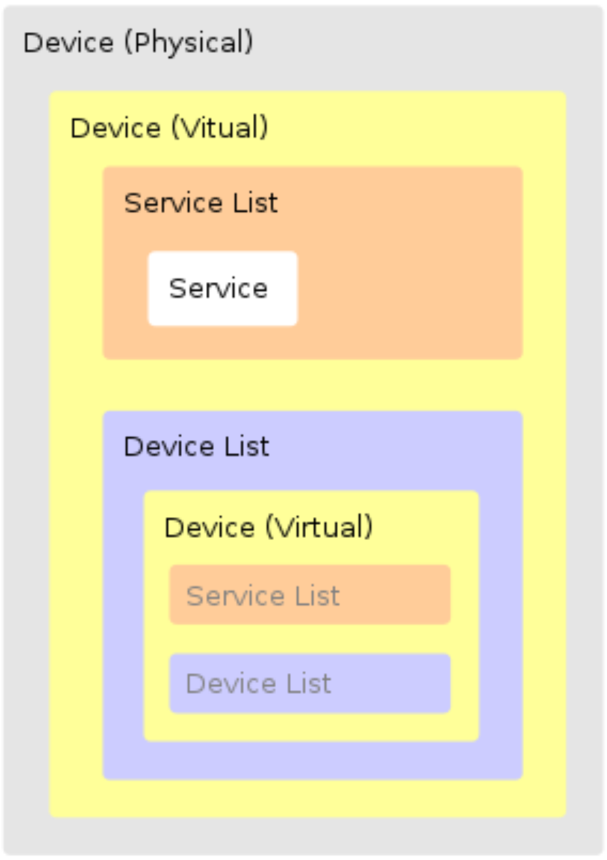
Not all SCPD XML files contain an URLBase (the one above from my router doesn't), but if they do, it looks like this:

```
<URLBase>http://192.168.1.254:80</URLBase>
```

This is the base URL for the SOAP requests. If the SCPD XML does not contain an URLBase element, the LOCATION header from the server's discovery response may be used as the base URL. Any paths should be stripped off, leaving only the protocol, IP and port. In the case of my internet router that would be: http://192.168.0.1:40833/

Devices

The XML file then specifies *devices*, which are virtual devices that the physical device contains. These *devices* can contain a list of services in the `<ServiceList>` tag. A list of sub-devices can be found in the `<DeviceList>` tag. The Devices in the deviceList can themselves contain a list of services and devices. Thus, devices can recursively contain sub-devices, as shown in the following diagram:



As you can see, a virtual Device can contain a Device List, which can contain a virtual Device, etc. We are most interested in the `<Service>` elements from the `<ServiceList>`. They look like this:

```
<service>
  <serviceType>urn:schemas-upnp-org:service:WANCommonInterfaceConfig:1</servic
  <serviceId>urn:upnp-org:serviceId:WANCommonIFC1</serviceId>
  <controlURL>/ctl/CmnIfCfg</controlURL>
  <eventSubURL>/evt/CmnIfCfg</eventSubURL>
  <SCPDURL>/WANCfg.xml</SCPDURL>
</service>
...
<service>
  <serviceType>urn:schemas-upnp-org:service:WANIPConnection:1</serviceType>
  <controlURL>/ctl/IPConn</controlURL>
  <eventSubURL>/evt/IPConn</eventSubURL>
  <SCPDURL>/WANIPCn.xml</SCPDURL>
</service>
```

The `<URLBase>` in combination with the `<controlURL>` gives us the URL to the SOAP server where we can send our requests. The `URLBase` in combination with the `<SCPDURL>` points us to a SCPD (Service Control Point Definition) XML file which contains a description of the SOAP calls.

The following Python code extracts the `URLBase`, `ControlURL` and `SCPDURL` information:

```
import urllib2
import urlparse
from xml.dom import minidom

def XMLGetNodeText(node):
    """
    Return text contents of an XML node.
    """
    text = []
    for childNode in node.childNodes:
        if childNode.nodeType == node.TEXT_NODE:
            text.append(childNode.data)
    return(''.join(text))

location = 'http://192.168.0.1:40833/rootDesc.xml'

# Fetch SCPD
response = urllib2.urlopen(location)
root_xml = minidom.parseString(response.read())
response.close()

# Construct BaseURL
base_url_elem = root_xml.getElementsByTagName('URLBase')
if base_url_elem:
    base_url = XMLGetNodeText(base_url_elem[0]).rstrip('/')
else:
    url = urlparse.urlparse(location)
    base_url = '%s://%s' % (url.scheme, url.netloc)

# Output Service info
for node in root_xml.getElementsByTagName('service'):
    service_type = XMLGetNodeText(node.getElementsByTagName('serviceType')[0])
    control_url = '%s%s' % (
        base_url,
        XMLGetNodeText(node.getElementsByTagName('controlURL')[0])
    )
    scpd_url = '%s%s' % (
        base_url,
        XMLGetNodeText(node.getElementsByTagName('SCPDURL')[0])
    )
    print '%s:\n  SCPD_URL: %s\n  CTRL_URL: %s\n' % (service_type,
                                                      scpd_url,
                                                      control_url)
```

Output:

```
urn:schemas-upnp-org:service:Layer3Forwarding:1:
  SCPD_URL: http://192.168.0.1:40833/L3F.xml
  CTRL_URL: http://192.168.0.1:40833/ctl/L3F

urn:schemas-upnp-org:service:WANCommonInterfaceConfig:1:
  SCPD_URL: http://192.168.0.1:40833/WANCfg.xml
  CTRL_URL: http://192.168.0.1:40833/ctl/CmnIfCfg

urn:schemas-upnp-org:service:WANIPConnection:1:
  SCPD_URL: http://192.168.0.1:40833/WANIPCn.xml
  CTRL_URL: http://192.168.0.1:40833/ctl/IPConn
```

SCPD, Phase II: Service SCPD files

Let's look at the `WANIPConnection` service. We have an SCPD XML file for it at `http://192.168.0.1:40833/WANIPCn.xml` and a SOAP URL at `http://192.168.0.1:40833/ctl/IPConn`. We must find out which SOAP calls we can make, and which parameters they take. Normally SOAP would use a WSDL file to define its API. With UPnp however this information is contained in the SCPD XML file for the service. Here's an example of the full version of the WANIPCn.xml file. There are two interesting things in the XML file:

- The `<ActionList>` element contains a list of actions understood by the SOAP server.
- The `<serviceStateTable>` element contains metadata about the arguments we can send to SOAP actions, such as the type and allowed values.

ActionList

The `<ActionList>` tag contains a list of actions understood by the SOAP server. It looks like this:

```
<actionList>
```

```
    <action>
      <name>SetConnectionType</name>
      <argumentList>
        <argument>
          <name>NewConnectionType</name>
          <direction>in</direction>
          <relatedStateVariable>ConnectionType</relatedStateVariable>
        </argument>
      </argumentList>
    </action>
    <action>
      [... etc ...]
    </action>
  </actionList>
```

In this example, we discover an action called `SetConnectionType`. It takes one incoming argument: `NewConnectionType`. The `relatedStateVariable` specifies which StateVariable this argument should adhere to.

serviceStateTable

Looking at the `<serviceStateTable>` section later on in the XML file, we see:

```
  <serviceStateTable>
    <stateVariable sendEvents="no">
      <name>ConnectionType</name>
      <dataType>string</dataType>
    </stateVariable>
    <stateVariable>
    [... etc ...]
    </stateVariable>
  </serviceStateTable>
```

From this we conclude that we need to send an argument with name "`ConnectionType`" and type "`string`" to the `SetConnectionType` SOAP call.

Another example is the GetExternalIPAddress action. It takes no incoming arguments, but does return a value with the name "NewExternalIPAddress". The action will return the external IP address of your router. That is, the IP address you use to connect to the internet.

```
  <action>
    <name>GetExternalIPAddress</name>
    <argumentList>
      <argument>
        <name>NewExternalIPAddress</name>
        <direction>out</direction>
        <relatedStateVariable>ExternalIPAddress</relatedStateVariable>
      </argument>
    </argumentList>
  </action>
```

Let's make a SOAP call to that action and find out what our external IP is.

SOAP: Calling an action

Normally we would use a SOAP library to create a call to a SOAP service. In this article I'm going to cheat a little and build a SOAP request from scratch.

```
import urllib2

soap_encoding = "http://schemas.xmlsoap.org/soap/encoding/"
soap_env = "http://schemas.xmlsoap.org/soap/envelope"
service_ns = "urn:schemas-upnp-org:service:WANIPConnection:1"
soap_body = """





""" % (soap_encoding, service_ns, soap_env)

soap_action = "urn:schemas-upnp-org:service:WANIPConnection:1#GetExternalIPAdd
headers = {
    'SOAPAction': u'"%s"' % (soap_action),
    'Host': u'192.168.0.1:40833',
    'Content-Type': 'text/xml',
    'Content-Length': len(soap_body),
}

ctrl_url = "http://192.168.0.1:40833/ctl/IPConn"

request = urllib2.Request(ctrl_url, soap_body, headers)
response = urllib2.urlopen(request)

print response.read()
```

The SOAP server returns a response with our external IP in it. I've pretty-printed it for your convenience and removed some XML namespaces for brevity:

```
<?xml version="1.0"?>
<s:Envelope xmlns:s=".." s:encodingStyle="..">
  <s:Body>
    <u:GetExternalIPAddressResponse xmlns:u="urn:schemas-upnp-org:service:WANI
      <NewExternalIPAddress>212.100.28.66</NewExternalIPAddress>
    </u:GetExternalIPAddressResponse>
  </s:Body>
</s:Envelope>
```

We can now put the response through an XML parser and combine it with the SCPD XML's `<argumentList>` and `<serviceStateTable>` to figure out which output parameters we can expect and what type they are. Doing this is beyond the scope of this article, since it's rather straight-forward yet takes a reasonable amount of code. Suffice to say that our extenal IP is 212.100.28.66.

Summary

To summarise, these are the steps we take to actually do something useful with a UPnP service:

1. Broadcast a HTTP-over-UDP (HTTPU) message to the network asking for UPnP devices to respond.
2. Listen for incoming UDP replies and extract the LOCATION header.
3. Send a WGET to fetch a SCPD XML file from the LOCATION.
4. Extract services and/or devices from the SCPD XML file.
    1. For each service, extract the Control and SCDP urls.
    2. Combine the BaseURL (or if it was not present in the SCPD XML, use the LOCATION header) with the Control and SCDP url's.
5. Send a WGET to fetch the service's SCPD XML file that describes the actions it supports.
6. Send a SOAP POST request to the service's Control URL to call one of the actions that it supports.
7. Receive and parse reply.

An example with Requests on the left and Responses on the right. Like all other examples in this article, the XML has been heavily stripped of redundant or unimportant information:

```
HTTPU broadcast
```

```
LOCATION:http://192.168.1.254:80/upnp/IGD.xml
SERVER:SpeedTouch 546 5.4.0.14 UPnP/1.0 (0612BH95K)
USN:uuid:UPnP_SpeedTouch546-1_00-14-7F-08-7E-0D::upnp:rootdevice
```

```
GET http://192.168.1.254:80/upnp/IGD.xml
```

```
<device>
  <serviceList>
    <service>
      <controlURL>/ctl/IPConn</controlURL>
      <SCPDURL>/WANIPCn.xml</SCPDURL>
    </service>
  </serviceList>
  <deviceList>
    <device>
      ...RECURSION...
    </device>
  </deviceList>
</device>
```

```
GET http://192.168.0.1:40833/WANIPCn.xml
```

```
<action>
  <name>GetExternalIPAddress</name>
  <argumentList>
    <argument>
      <name>NewExternalIPAddress</name>
      <direction>out</direction>
      <relatedStateVariable>ExternalIPAddress</relatedStateVariable>
    </argument>
  </argumentList>
</action>
<serviceStateTable>
  <stateVariable sendEvents="yes">
    <name>ExternalIPAddress</name>
    <dataType>string</dataType>
  </stateVariable>
</serviceStateTable>
```

```
POST http://192.168.0.1:40833/ctl/IPConn

<SOAP-ENV:Envelope xmlns:SOAP-ENV="..">
  <SOAP-ENV:Body>
    <m:GetExternalIPAddress xmlns:m="...:WANIPConnection:1">
    </m:GetExternalIPAddress>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
<Envelope>
  <Body>
    <GetExternalIPAddressResponse>
      <NewExternalIPAddress>212.187.48.56</NewExternalIPAddress>
    </GetExternalIPAddressResponse>
  </Body>
</Envelope>
```

Conclusion

I underwent this whole journey of UPnP because I wanted a way transparently support connections from externals networks to my locally-running application. While UPnP allows me to do that, I feel that UPnP is needlessly complicated. The standard, while readable, feels like it's designed by committee. The indirectness of having to fetch multiple SCPD files, the use of non-standard protocols, the nestable virtual sub-devices… it all feels slightly unnecesarry. Then again, it could be a lot worse. One only needs to take a quick look at SAML v2 to see that UPnP isn't all that bad.

All in all, it let me do what I needed, and it didn't take too long to figure out how it worked. As a kind of exercise I partially implemented a high-level simple to use UPnP client for python, which is available on Github. Take a look at the source for more insights on how to deal with UPnP.

Search this blog: Search