Universal Control v.0.6.0 Tutorial

Tutorial for how to work with a Universal Control Server and write a client

J P Barrett

Universal Control v.0.6.0 Tutorial

James P. Barrett

Abstract

The Universal Control API Specification[1] described the behaviour of a Universal Control server accurately, but is very dry and hard to work with. This document is intended as a simple tutorial in how to make use of a Universal Control server and how to write a simple client.

Contents

Universal Control v.0.6.0 Tutorial

James P. Barrett

1 Introduction (or "What is Universal Control, anyway?")

The Universal Control API is a mechanism developed by BBC Research and Development to allow software running on other devices (such as applications running on mobile phones, widgits running in web-browsers, or even embedded systems running on physical remote controls) to act as a sort of "interface" for a set-top box, digital radio, or similar device.

It allows all of the standard "remote-control" operations one would expect (like changing channel, altering the volume, turning the box on and off, etc . . .) but it also provides all of the information that's needed to indicate to users what channel the box is currently tuned to, what other content is available, when particular on-demand programmes will be available to download, and a wide variety of other things which would otherwise require navigating the box's built-in menus to find out.

As a result many Universal Control clients are quite different from a traditional "remote control". For example one very unusual client developed during the R&D phase is a device which plugs into the local network, descovers a nearby set-top box and uses text-to-speach to "read-out" the name of the currently displayed channel.

2 Getting Started

In order to make use of this document you will need:

- A set-top box (or similar device) running a Universal Control server compliant with version 0.6.0 of the specification.
- A development machine with a web-browser[1] and an install of python v.2.6 or greater (2.7 recommended).

Throughout this tutorial I will give examples using my own install of the Universal Control example server which is running on a Mythbuntu machine connected to a UK DVB-T receiver. Other servers will producve very similar results.

Throughout this tutorial I will assume that the reader is familar with XML and has a passing familiarity with Python. Python is a very easy language to learn, and a basic Python tutorial (such as the one found at python.org[2]) will cover far more python syntax than will be needed here.

3 Making a few basic requests

You should already have a Universal Control server running on your network. We'll cover how to discover a server later – for now make sure that the server is running with the Security Scheme turned off (for the example server the security scheme is switched off by default) and take a note of the IP address of the set-top-box on which it's running, and what port number it's on (by default this will be 48875).

Example: On my system the server is running on IP address `192.168.1.105` on the default port number: `48875`.

[1]We recommend using Mozilla Firefox v.3.6 or higher for the web-browser. Other browsers should work just as well, but some may behave slightly differently, for example in Google Chrome all URLs for server resources will need to be prepended with 'view-source:' in order to force the browser to display the XML code.

Armed with this information I'm able to make my first request to the UC server. A UC server is a kind of HTTP server (much like a web-server), and hence everything it does is in response to a "request" received over the network. Here we can use a web-browser to make a simple request to the server, and we do this by entering a URL of the following form into the browser:

`http://`{IP Address of server}`:`{port number}`/`{path of resource}

For our first request the path of the resource will be `uc`.

Example: I make my request to: `http://192.168.1.105:48875/uc`

The web-browser should immediately display an XML file looking something like the following[2]:

```
<response resource="uc">
    <ucserver name="MythTVBox-12"
          security-scheme="false"
          server-id="076c537e-180c-11e0-b105-90fba687f588"
          version="0.6.0">
        <resource rref="uc/power"/>
        <resource rref="uc/time"/>
        <resource rref="uc/events"/>
        <resource rref="uc/sources"/>
        <resource rref="uc/source-lists"/>
        <resource rref="uc/outputs"/>
        <resource rref="uc/feedback"/>
        <resource rref="uc/remote"/>
        <resource rref="uc/acquisitions"/>
        <resource rref="uc/search"/>
        <resource rref="uc/storage"/>
        <resource rref="uc/credentials"/>
        <resource rref="uc/categories"/>
        <resource rref="uc/apps"/>
    </ucserver>
</response>
```

What does this tell us? Well, for starters we spot that the root element of the document is a `response` element, which has a `resource` attribute with `"uc"` as the value. This is a common feature throughout the UC API that all representations of the state of system, whether sent back by the server or provided by the client (more on that later) take the form of an XML document with a `response` element as the root element, which has the path of the resource as the `resource` attribute. Other than that we get a few bits of information about the box:

- The box is running a Universal Control server (else we wouldn't be getting a valid response).
- The box's name is "MythTVBox-12" – this name should be unique on the local network, and can be used by a client to identify the box to a user.
- The box's id is "076c537e-180c-11e0-b105-90fba687f588" – this id is globally unique, and can be stored to allow a client application to check if it's really talking to the same server.
- The box doesn't currently have the security scheme turned on.

[2]On Google Chrome you'll need to add 'view-source' to the start of the URL in order to make the browser display the file

- The server supports version 0.6.0 of the Universal Control protocol. That's good, because that's the version this tutorial is written about.

It also tells us that the server implements a number of the optional resources described in the specification, in this case the "power", "time", "events", "sources", "source-lists", "outputs", "feedback", "remote", "acquisitions", "programmes", "storage", "credentials", "categories", and "apps" resources. Most of these resources will be described later in this document when they come up.

Now, all of that is useful information (and checking it is important before proceding), but it's not really what we want when we want to know the state of a set-top box. In fact what we really want is to know *what's currently on*, and a few details about how it's being shown. For that we need to access a different resource on the box: the `uc/outputs/main` resource. So plug the URL for that resource into your web-browser.

Example: I make a request to `http://192.168.1.105:48875/uc/outputs/main`.

The response should look something like this:

```
<response resource="uc/outputs/0">
    <output name="Main Screen">
        <settings volume="0.8000"/>
        <programme sid="5287" cid="2010-08-16T15%3a00%3a00Z"/>
        <playback speed="1.00"/>
    </output>
</response>
```

which is, perhaps, a little more informative, but by itself not very. The question is what is it trying to tell us? Well firstly we note that although we sent out request to `uc/outputs/main` the response was from `uc/outputs/0` – that's normal "main" is just an alias for another output – in this case the main screen attached to the MythTV box.

This output is currently showing content, the audio volume is `"0.8000"`, and the content being shown has source-id `"5287"` and content-id `"2010-08-16T15%3a00%3a00Z"`, it's playing at normal speed.

The problem then, clearly, is that we have no idea what source-id `"5287"` or content-id `"2010-08-16T15%3a00%3a00Z"` mean. So it's time for a brief discussion of the data-model which Universal Control uses.

3.1 Outputs, Sources, and Content (Oh my!)

The Universal Control data model was designed to map reasonably naturally to the kinds of data which a television set-top-box, internet radio, or similar device might want to expose. Such a device is basically a tool for presenting content coming from one of a number of sources to the use by means of some sort of output (a screen for a TV, speakers for a radio, etc . . .). As such the most important elements in the UC data model are thus **Outputs** (which represent means of presenting content to the user), **Sources** (which represent distinct "places" from which content might come), and **Pieces of Content** (which represent individual pieces of content which would sensibly be presented, including both television/radio programmes and also interactive applications which the user might want to access).

For the Myth TV server the mapping is fairly straight-forward:

- There is one **Output** corresponding to the default MythTV Frontend for the system. In future the capability to treat multiple Frontends as seperate outputs may be added.

- Each television channel or radio station is a **Source**.
- Each programme on each channel/station is a **Piece of Content** within the **Source**.
- The list of recordings on the MythTV backend is also a **Source**, and each recording within it is a **Piece of Content**.
- The list of videos on the MythTV backend is also a **Source**, and each video within it is a **Piece of Content**.
- The Myth TV menus are represented by a **Source**, and each individual menu page within then is a **Piece of Content**.
- The Myth Game subsystem is represented by a **Source** and each individual game is a **Piece of Content**.
- Each MythNetVision content source configured on the box (such as BBC iPlayer) is a **Source**, and each programme available from it is a **Piece of Content**.

Each source is identified with a source-id, and each piece of content is identified with a content-id. Colloquially we'll often refer to pieces of content which aren't "interactive" as "programmes". You can rely upon each source-id being unique within the server, and you can rely upon each content-id being unique within the source – but two programmes on different television channels *might* have the same content-id. For that reason to properly identify a programme uniquely you need to refer to it using a pair of a source-id and a content-id.

Example: The programme being displayed on my Myth TV box as I type this is identified by the source-id "5287" and the content-id "2010-08-16T15%3a00%3a00Z". Currently I don't know any more information about it than that, but read on, all will be revealed...

In order to find out a little more about a source we need to retrieve its details, and we do this by making a request to the resource `uc/sources/{source-id}`

Example: I make a request to `http://192.168.1.105:48875/uc/sources/5287`.

The result is a document of the form:

```
<response resource="uc/sources/5287">
    <source sid="5287"
          name="BBC TWO"
          sref="dvb://233a..10bf"
          default-content-id="2010-08-16T15%3a00%3a00Z"
          logo-href="http://192.168.1.105:48875/images/sources/5287.jpg"
          live="true"
          linear="true"
          follow-on="true"
          lcn="002"/>
</response>
```

This document, for the first time, tells us which source we're looking at in a way which makes a bit more sense to a human being. Specifically it tells us that we're watching "BBC TWO", and also that this is channel number "002" in the box's tv guide. That's useful information if we wanted to implement "channel up" and "channel down" in a way that the user might expect. The properties live, linear, and follow-on together tell us that this source will behave much the way we'd expect a TV channel to behave – ie. it's got a concept of "current" time (ie. what's being transmitted right now), it has a linear schedule with start and end times for its programmes (which don't overlap), and when one programme ends the output that's showing it will continue to show the next programme on the channel, rather than going back to a menu like a TV-on-demand source might. Finally the default-content-id attribute tells us which programme would be shown if we tuned an output to that source right now. It's the programme that's currently showing.

So, to get a bit more information about the programme we make a request to uc/search/sources/5287.

Example: I make a request to http://192.168.1.105:48875/uc/search/sources/5287.

The result is of the form:

```
<response resource="uc/search/sources/5287">
    <results more="true">
        <content sid="5287"
              cid="2010-08-16T15%3a00%3a00Z"
              global-content-id="crid://fp.bbc.co.uk/4j52me"
              global-series-id="crid://fp.bbc.co.uk/knpis0"
              series-id="fp.bbc.co.uk%2fknpis0"
              title="Diagnosis Murder"
              interactive="false"
              duration="2700.00000"
              start="2010-08-16T15:00:00Z"
              acquirable-until="2010-08-16T15:00:00Z"
              presentable-from="2010-08-16T15:00:00Z"
              presentable-until="2010-08-16T15:45:00Z">
            <synopsis>Detective series. A telethon is held to raise funds
                  for the hospital. However, Dr Sloan loses his cool when
                  the host of the show is murdered on air.</synopsis>
            <category id="chi"/>
            <media-component id="audio"
                  type="audio"
                  name="Primary Audio"
                  default="true"/>
            <media-component id="video"
                  type="video"
                  name="Primary Video"
                  aspect="16:9"
                  default="true"/>
            <media-component id="subtitles"
                  type="subtitles"
                  name="Primary Subtitles"
                  default="false"/>
        </content>
    </results>
</response>
```

Now we're talking! This Response gives us what's currently on the channel, and tells us a lot of useful information about the programme. With the output of this resource a client would be able to display a "What's on Now" display similar to the one normally presented on a set-top-box interface. To get information to produce a "Now and Next" display similar to many set top boxes we'd instead make a request to the resource `uc/search/sources/5287?results=2`. This makes use of a query parameter to control the results returned by a source. Query paramters are a part of the way HTTP URLs are formed, and the UC API uses them in a fairly standardised way. After the end of the "path" segment of the resource URL we put a `?` character followed by a series of parameters seperated by `&` characters. Each parameter is of the form {name}={value}.

The result from this request would be:

```
<response resource="uc/search/sources/5287?results=2">
    <results more="true">
        <content sid="5287"
              cid="2010-08-16T15%3a00%3a00Z"
              global-content-id="crid://fp.bbc.co.uk/4j52me"
              global-series-id="crid://fp.bbc.co.uk/knpis0"
              series-id="fp.bbc.co.uk%2fknpis0"
              title="Diagnosis Murder"
              interactive="false"
              duration="2700.00000"
              start="2010-08-16T15:00:00Z"
              acquirable-until="2010-08-16T15:00:00Z"
              presentable-from="2010-08-16T15:00:00Z"
              presentable-until="2010-08-16T15:45:00Z">
            <synopsis>Detective series. A telethon is held to raise funds
                  for the hospital. However, Dr Sloan loses his cool when
                  the host of the show is murdered on air.</synopsis>
            <category id="dra"/>
            <media-component id="audio"
                  type="audio"
                  name="Primary Audio"
                  default="true"/>
            <media-component id="video"
                  type="video"
                  name="Primary Video"
                  aspect="16:9"
                  default="true"/>
            <media-component id="subtitles"
                  type="subtitles"
                  name="Primary Subtitles"
                  default="false"/>
        </content>
        <content sid="5287"
              cid="2010-08-16T15%3a45%3a00Z"
              global-content-id="crid://fp.bbc.co.uk/4cdjvk"
              global-series-id="crid://fp.bbc.co.uk/kqn1ix"
              series-id="fp.bbc.co.uk%2fkqn1ix"
              title="In the Night Garden"
              interactive="false"
              duration="1800.00000"
              start="2010-08-16T15:45:00Z"
              acquirable-until="2010-08-16T15:45:00Z"
              presentable-from="2010-08-16T15:45:00Z"
```

```
                presentable-until="2010-08-16T16:15:00Z">
              <synopsis>Series for young children. Upsy Daisy teaches Igglepiggle
                    how to play hiding. Upsy Daisy is very good at hiding.
                    Igglepiggle is impressed.</synopsis>
              <category id="chi"/>
              <media-component id="audio"
                    type="audio"
                    name="Primary Audio"
                    default="true"/>
              <media-component id="video"
                    type="video"
                    name="Primary Video"
                    aspect="16:9"
                    default="true"/>
              <media-component id="subtitles"
                    type="subtitles"
                    name="Primary Subtitles"
                    default="false"/>
          </content>
      </results>
  </response>
```

By specifying different parameters to these sorts of requests a client can retrieve enough information from this resource to fill out a whole electronic programme guide. Provided that one went in already knowing the source-ids for all the sources one wanted to ask about.

So far we've only seen one way of finding out a source-id, and that's to check which one is being presented on an output. That's a bit of a problem, since it'll be hard to tell the box to change to a particular channel if you have no idea what source-id the other channel has! Naturally there's a solution to this problem.

3.2 Source-lists

Most servers are likely to have a lot of sources asosciated with them. The API provides a mechanism to divide those sources into lists for convenience of discovery. All boxes which implement the `uc/source-lists` resource are guaranteed to implement the list `uc_default`, which will contain the "default" list of sources the server designer thought most clients would be interested in. To access this on the Myth TV server send a request to `uc/source-lists/uc_default`.

Example: I send a request to `http://192.168.1.105:48875/uc/source-lists/uc_default`. I'm also going to stop telling you what URL I'm sending requests to from now on, since I assume you've got the idea.

The response should be of the form shown below (though most real servers will produce a list much longer than this one, I've trimmed it for easier demonstration):

```
<response resource="uc/source-lists/default">
    <sources>
        <source sid="mythtv"
              name="The MythTV Menus"
              default-content-id="mainmenu"
              live="false"
```

```
                linear="false"
                follow-on="false"/>
          <source sid="mythgame"
                name="Games Supported by MythGame"
                live="false"
                linear="false"
                follow-on="false"/>
          <source sid="SG_1"
                name="Recordings Stored by MythTV"
                sref="myth:Default@192.168.1.105/var/lib/mythtv/recordings"
                live="false"
                linear="false"
                follow-on="false"/>
          <source sid="SG_2"
                name="Videos Stored by MythTV"
                sref="myth:Videos@192.168.1.105/var/lib/mythtv/videos/"
                live="false"
                linear="false"
                follow-on="false"/>
          <source id="5168"
                name="BBC ONE"
                sref="dvb://233a..1048"
                default-content-id="2010-08-16T16%3a05%3a00Z"
                logo-href="http://192.168.1.105:48875/images/sources/5168.jpg"
                live="true"
                linear="true"
                follow-on="true"
                lcn="001"/>
          <source id="5287"
                name="BBC TWO"
                sref="dvb://233a..10bf"
                default-content-id="2010-08-16T16%3a45%3a00Z"
                logo-href="http://192.168.1.105:48875/images/sources/5168.jpg"
                live="true"
                linear="true"
                follow-on="true"
                lcn="002"/>
      </sources>
  </response>
```

Here we've got two television channels and several built-in facets of the MythTV box, each with an id, a name, a default cid, and indeed all of the other information we got when we make a request to `uc/sources/5287` before. In fact the `source` element in this document representing BBC Two is *identical* to the one in the response to the previous request – that's not a coincidence, the API requires it.

The box may have other sources, though, which aren't listed in the default list. To find those we should take a look at what over lists the box supports by making a request to `uc/source-lists`. When we do that we get a response of the form:

```
  <response resource="uc/source-lists">
      <source-lists>
```

```
        <list list-id="uc_default"
              name="Default"
              description="The default list of sources"/>
        <list list-id="uc_storage"
              name="Recordings and Videos"
              description="The recordings and videos on the MythTV box"/>
        <list list-id="mythtv_apps"
              name="Applications"
              description="Interactive Applications"/>
        <list list-id="mythtv_mythnetvision"
              name="MythNetVision"
              description="MythNetVision on-demand services"/>
        <list list-id="mythtv_radio"
              name="Radio"
              description="All Radio Stations"/>
        <list list-id="mythtv_tv"
              name="TV"
              description="All TV channels"/>
    </source-lists>
</response>
```

Which lists all of the source-lists available on this server. The "mythtv_tv", "mythtv_radio", "mythtv_apps", and "mythtv_mythnetvision" source-lists are custom defined by the Myth TV server (they aren't manadated in the spec), and just contain the TV channels, radio stations, interactive application sources, and internet on demand services known to the box. The "uc_default" list is always required (as mentioned above), and the "uc_storage" list is required for any box which supports "pvr-like" capabilities to record and playback content from a local storage device. Since Myth TV has that capability the server implements that list.

Making a request to the `uc/source-lists/uc_storage` resource gives us a response of the form:

```
<response resource="uc/source-lists/storage">
    <sources>
        <source sid="SG_1"
              name="Recordings Stored by MythTV"
              sref="myth:Default@192.168.1.105/var/lib/mythtv/recordings"
              live="false"
              linear="false"
              follow-on="false"/>
        <source sid="SG_2"
              name="Videos Stored by MythTV"
              sref="myth:Videos@192.168.1.105/var/lib/mythtv/videos/"
              live="false" linear="false"
              follow-on="false"/>
    </sources>
</response>
```

The two sources here serve to encapsulate the recording and storage capabilities of the MythTV backend. It's also worth noting that the source-ids here aren't just numbers like the ones for TV channels and radio stations are. That's an implementation decision made by the author of the server, and shouldn't make a difference to a client developer, since source-ids should only be used to make requests to the server, and should *never* be presented directly to a user.

Armed with all of this information it's time to actually try changing the channel on the box.

4 Changing Things

So far all the requests we've made to the server have been made using a web-browser by entering the URL of the resource we wanted to access directly into the browser's address bar. This is taking advantage of the fact that, like a web-server, a Universal Control server uses the HTTP protocol to handle requests made to it. However a Universal Control server is not really a web-server, and it does some things a web-server doesn't do. In particular it handles several different types of reqests with different HTTP "verbs".

The way HTTP is designed every request made to a server must have a few pieces of information attached to it, including the URL of the resource the request is aimed at, the query string from the end of the URL (which we've seen used when making a request to the programmes resource), a number of "headers" which contain other information about the request (like what application made it, what kinds of content are acceptable as responses, what username to use for a secure server, etc...), and also the type of request it is, what is called the "verb". Although there are lots and lots of different HTTP verbs out there the UC API only uses four of them for ordinary requests:

- `GET` requests are made by the client to "get" the current state of the resource. The server sends back a "document" containing that information. This is the kind of request which a web-browser makes when you type a URL into the address-bar and hit enter. All the requests we've made so far have been of this type, and most of the requests you'll ever make will be as well. It's garaunteed by the API specification that no UC server will ever cause changes to happen to the box its running on as a result of a `GET` request (unless there's an error when processing it, of course, then all bets are off). `GET` requests are used in the API to provide data about the state of the box such as telling teh client what's currently being presented on an output.

- `PUT` requests are made by the client to "put" data onto the server. Specifically a "put" request asks the server to replace the resource referred to by the URL with the data that the client provides in the "body" of the request. `PUT` requests are used in the API to make changes. It's guaranteed, however, that the result of two identical `PUT` requests to the same resource with the same body one after another will always be the same as just sending one. `PUT` requests are used by the API to make changed to the state of the box, such as changing channel.

- `DELETE` requests are made by the client to "delete" the contents of a resource. Like `PUT` requests it's garaunteed that making two `DELETE` requests in a row to the same resource will result in the same result as making only one (ie. deleting something which has already been deleted is always succesful, since the thing is not there). `DELETE` requests are used by the API to remove things, for example to cancel an upcoming recording.

- `POST` requests are made by the client to "post" data to the server. They often (though not always) require a "body" be provided like in a `PUT` request, but unlike a `PUT` request there are no garauntees regarding a `POST`. For this reason it's important to be careful when making a request with this verb. `POST` requests are used by the API for a variety of purposes which aren't covered by the other verbs, such as booking a new recording.

In order to change the channel being shown on the box's output we're going to need to make a `POST` request to `uc/outputs/main`. In principle we could do this using a web-browser and query parameters, but instead we're going to do it in the more fully featured way using a message body and to do so we're going to need to fire up Python.

4.1 Making a Request Using Python

Fire up the python interpreter (on a unix system this can be done just by typing `python` in a shell) and enter the following code, which will make a `GET` request to `uc/outputs/main` (you will want to replace the IP address below with the one for your box):

```
>>> import httplib
>>> connection = httplib.HTTPConnection('192.168.1.105',48875)
>>> connection.request('GET','uc/outputs/main')
>>> response = connection.getresponse()
>>> response.status
200
>>> response.read()
```

The final command will print out a response in the form of a string, it should look something like `'<response resource="uc/outputs/0"><output name="Main Screen"><settings volume="0.8000"><programme sid="5287" cid="2010-08-16T15%3a00%3a00Z"/><playback speed="1.00"/></output></response>\n'`

Which, when formatted nicely comes out as:

```
<response resource="uc/outputs/0">
    <output name="Main Screen">
        <settings volume="0.8000"/>
        <programme sid="5287"
              cid="2010-08-16T15%3a00%3a00Z"/>
        <playback speed="1.00"/>
    </output>
</response>
```

exactly the same as our earlier `GET` request to the same resource.

The python code is fairly simple. First we import the library used for making http requests, then we open a connection to the server, then we make a `GET` request to the desired URL. Finally we retrieve the "response" which the server sent back and check the "status code" and the "body" of the response. The body is the xml document we've seen before, the status code is a number which indicates if the request has been succesful or if an error has ocured. Here are a few status codes which a request to a UC server might return:

- **200** – "OK". The request was successful and returned a body.
- **204** – "OK". The request was successful, but no body was returned.
- **400** – "Client Error". There was something wrong with the request you made – perhaps the query parameters or the body you supplied were formatted wrongly.
- **402** – "Validate". This server is asking for some sort of security credentials to continue. This won't happen on the MythTV server unless you turn on the security scheme (which isn't covered in this tutorial).
- **404** – "Not Found". The resource requested could not be found.
- **405** – "Not implemented". The resource does not exist on this server.
- **500** – "Server Error". An Error occured inside the server, and hence the request has failed.

In this case we got back a "200", meaning that the request was successfull and a body was returned (which is normally the case when a GET request is successful).

So, now, let's make a PUT request to change the channel from BBC Two to BBC One.

From earlier we know that my server uses source-id "5168" to represent BBC One. Yours may well use a different source-id, so check what the correct one is before making this request! I fire up python and enter the following:

```
>>> import httplib
>>> connection = httplib.HTTPConnection('192.168.1.105',48875)
>>> connection.request('POST','uc/outputs/main',"""\
... <response resource="uc/outputs/0">
... <programme sid="5168" cid=""/>
... </response>\n""")
>>> response = connection.getresponse()
>>> response.status
204
```

And the result is that my Myth TV box switches to showing BBC One.

Now as you can see we made this change by providing a body for our POST request which contained a programme element with a sid attribute that was equal to "5168" and a cid attribute which was blank. In this way we specify that the box should change to the source represented by source-id "5168", but we don't specify what programme to show – leaving the box to decide for itself. In the case of this TV channel the box naturally picks the programme that's currently showing (that's really the only choice it can make). If we were trying to get the box to play back some previously recorded content the behaviour could be quite different.

In this case the same effect could be achieved by making a POST request with no body to the uri `uc/outputs/main?sid=5168` as the specification says that this is equivalent in most cases (there are some subtle differences in behaviour which have to do with what happens if the selected source has an interactive application as its default piece of content, that isn't the case for a TV channel).

4.2 Understanding Outputs

So, here's a nearly complete rundown of the XML for the `uc/outputs/main` resource. This is the format you'll get back from GET requests:

- A single output element with:
  - A single name attribute holding a human readable string identifying the output. This is read-only.
  - A single settings element with:
    - (OPTIONALLY) A single volume attribute holding a decimal number between "0.000" and "1.000" representing the audio volume.
    - (OPTIONALLY) A single mute attribute which is "true" if the audio is muted, and "false" otherwise. If this attribute is missing the server will assume the volume is not muted.
    - (OPTIONALLY) A single aspect attribute which can hold the values "4:3", "14:9", "16:10", "16:9", "21:9", or "source" (which indicates that the screen's aspect ratio should change to match that of the content being displayed).
  - (OPTIONALLY) A single programme element which contains the details of what audio-visual content is currently being presented on this output:
    - A single sid attribute which contains the source-id of the current source.

    * A single cid attribute which contains the content-id of the current prgramme.
    * There can be component-override elements, as well, but we'll cover those later.
  - (OPTIONALLY) A single app element which contains the details of what interactive content is currently being presented on this output:
    * A single sid attribute which contains the source-id of the current source.
    * A single cid attribute which contains the content-id of the current app.
    * There can be controls elements, as well, but we'll cover those later.

- (OPTIONALLY) A single playback element with:
  - A single speed attribute set to a positive or negative decimal number. The value "1.0" represents normal speed forward, "0.5" represents half-speed, "-3.0" is tripple-speed in reverse, etc...

With those values, and the knowledge of how to look up source-ids and content-ids using the `uc/source-lists` and `uc/search/sources/{id}` resources you should be able to do most of the simple television control operations which you'd expect to want to do: changing channel, etc.... To change volume you need to access the resources `uc/outputs/main/settings` and `uc/outputs/main/playhead`, both of which accept GET requests to retrieve their current state, and PUT requests with a body of the same form as returned by the GET requests to set their state. For example, the code:

```
>>> import httplib
>>> connection = httplib.HTTPConnection('192.168.1.105',48875)
>>> connection.request('PUT','uc/outputs/main/settings',"""\
... <response resource="uc/outputs/0/settings">
... <settings volume="1.0"/>
... </response>\n""")
>>> response = connection.getresponse()
>>> response.status
204
```

will change the volume to maximum.

More advanced capabilities (like seeking within streams and pausing live TV) will require an understanding of some of the other resources that the UC API makes available. Which is what the rest of this tutorial will cover.

5 Writing a simple client

In this section we're going to write a simple Universal Control client written in Python. In a real environment chances are that you'll be using a different language (for example javascript for web-based or widget-based clients, actionscript for flash-based ones, Objective-C for native iPhone applications or C/C++ for native applications on other platforms) but the nature of the Python language makes the code very simple and easily readable, hence my decision to use it here.

Eventually this client will:

- Discover servers on the network.
- Allow user selection of a server.
- Display the current state of the main output of the selected server.
- Update the display when the output's status changes.

In order to make this example as simple as possible I'm not going to pay much attention to the interface of the client here (this isn't a GUI writing tutorial) a full set of the code of the client can be found in the universal control code repository.

It would be good for the client to also display a simple electronic programme guide and allow a user to change the channel and book future recordings – but the fact is that whilst writing such a client is very easy it would require use of some sort of UI library in order to properly design such a client, and that's way beyond the scope of this tutorial. Instead I'll include some guidance on how to do these things, and some code which does specific examples of these things in a non-general kind of way in the next section.

5.1 Discovering Servers 1 – Manual

The UC API specifies two different ways of discovering UC servers on the local network: an automatic scheme and a manual one. Since the manual one is a little simpler (and mandatory) I'll explain that first.

The scheme works by asking the user to enter a "Pairing Code" which is generated by the box to which you are trying to connect. Currently if you're running the modified MythTV frontend included in this repository then the pairing code can be accessed from the built-in menus by going to `Utilities/Setup > Setup > UC Pairing Screen`. Otherwise you can generate the pairing code which your set-top-box *would* display by using the python script "PairingCode.py" which is contained in the `utilities` directory of the Universal Control public archive. I suppose you could also generate it by hand based on the code in the Spec, but I wouldn't recommend that.

Example: I execute this by typing:

```
python ./PairingCode.py -i 192.168.1.105:48875
```

and I get the response:

```
6JG  ===  192.168.1.105:48875 SSS=None
```

indicating that the code is `6JG`.

The pairing-code decoding algorithm for version 0.6.0 is specified in Appendix A of the Universal Control Spec[1, Appendix A] in a python-like pseudocode form. Below I've included some python code which will prompt the user for a pairing code, and then decode it, printing the result:

```python
import re

bit = 0

def decode_pairing_code(code):
    global bit
    bit = 0

    #Start off by defining a list of the characters used in the base32 encoding
    #used for pairing codes.
    __chars = '0123456789ABCDEFGHJKMNPQRSTVWXYZ'

    # A sanity check to ensure that the "pairing code"
    # contains only the valid characters
    if not re.match('[%s]+' % (__chars,),code.upper()):
        raise ValueError

    # Interpret the code as a base-32 integer
    l = len(code)
    code = reduce(lambda x, y : x+y,
                  ((__chars.index(code.upper()[n]) << (l-1-n)*5) for n in range(l)))

    # Helper function to read bits from the code
    def __input(n):
        global bit

        signal = (code >> bit) & ((1 << n) - 1)
        bit += n
        return signal


    # If the first bit of the code is a 1 raise an exception
    if __input(1) == 1:
        raise ValueError

    # If the code is for a server which uses the security scheme
    # raise an exception
    if __input(1) == 1:
        raise ValueError

    signal = __input(2)
    if signal == 0:
        # ip address is of the form 192.168.*.*

        A = 192
        B = 168

        signal = __input(2)

        if signal == 0:
            # ip address if of the form 192.168.0.*

            C = 0

        elif signal == 1:
            # ip address if of the form 192.168.1.*

            C = 1

        elif signal == 2:
            # ip address if of the form 192.168.2.*

            C = 2

        else:
            C = __input(8)

        D = __input(8)

    elif signal == 1:
        # ip address is of the form 172.16-31.*.*

        D = __input(8)
        C = __input(8)
        B = __input(4) + 16
        A = 172

    elif signal == 2:
        # ip address is of the form 10.*.*.*

        D = __input(8)
        C = __input(8)
        B = __input(8)
        A = 10
    else:

        D = __input(8)
        C = __input(8)
        B = __input(8)
        A = __input(8)

    if __input(1) == 1:
        # Non-standard port
        port = __input(16)

    else:
        # Standard port
        port = 48875

    return ("%d.%d.%d.%d" % (A,B,C,D),port)


if __name__ == "__main__":
    print "Please Enter Pairing Code : ",
    pairing_code = raw_input()
    print '%s:%d' % decode_pairing_code(pairing_code)
```

This code will decode any pairing code produced by any server which *isn't* using the Security Scheme defined in the UC API specification. The algorithm is fairly straightforward, and relatively simple to implement in most languags.

The output produced by this code is in the form of a pair consisting of the IP address as a string, and the port number as an integer. Hence armed with this information you should now be able to make requests to the server which was identified by the pairing code exactly as you did in the first sections of this tutorial.

5.2 Discovering Servers 2 – Automatic

The UC API also allows an automatic mechanism for discovering a server, which can be done with far less human interaction. This makes use of the technology called "Multicast DNS and DNS Service-Discovery (mDNS/DNS-SD)" an implementation of which is produced by Apple under the tradename "Bonjour"[3] and for which an open-source implementation called "Avahi" also exists[4][3]. The code below searches the network for existing Universal Control servers which are advertising themselves via this method, and presents a list of them to the user, when servers appear or disappear from the network the list is redisplayed. This is all handled via "Avahi" which the python code accesses using DBus and the dbus-python bindings. Because this code uses dbus it will only work on Linux, similar code can be written on other platforms using other technologies:

```
"""This file uses the DBus interface which avahi exposes to discover
Universal Control servers on the local network via DNS-SD."""

import dbus, gobject, avahi, threading
from dbus import DBusException
import dbus.glib
from dbus.mainloop.glib import DBusGMainLoop

__all__ = ['UCServerList',
           'start',]

dbus.set_default_main_loop(DBusGMainLoop())
gobject.threads_init()
dbus.glib.init_threads()

class UCServerList (list):
    """This object behaves like a lis of servers, but in fact is a proxy
    for the DNS-SD process which is searching for servers.

    Each server is stored in the list as a tuple of the form:

    (uuid,name,ip-address,port)"""

    def __init__(self,servers=[], zeroconf=True):
        """The object can be initialised with Zeroconf turned off, in which
        case it behaves just like an ordinary list, it can also be given a
        list of pre-configured servers, including ones which have been set
        up manually"""

        self.zeroconf = zeroconf

        for s in servers:
            self.append(s)

        if self.zeroconf:
            # This code sends a request over DBus to Avahi asking Avahi to
            # create a "service browser" for the local network looking for
```

[3]As far as the author is aware no mechanism currently exists on any platform for accessing mDNS/DNS-SD services from a script running inside a web-browser. Sorry javascript/flash/etc... developers, for the moment you seem to be stuck with the manual method

```
            # services which support the _universalctrl._tcp protocol. The
            # We connect things up so that the method "self.handler" will
            # be called anytime a new server is discovered.

            try:
                bus = dbus.SystemBus()
                self.server   = dbus.Interface(
                    bus.get_object(avahi.DBUS_NAME,
                                   '/'),
                    avahi.DBUS_INTERFACE_SERVER)
                self.sbrowser = dbus.Interface(
                    bus.get_object(avahi.DBUS_NAME,
                                   self.server.
                                   ServiceBrowserNew(avahi.IF_UNSPEC,
                                                     avahi.PROTO_UNSPEC,
                                                     '_universalctrl._tcp',
                                                     'local',
                                                     dbus.UInt32(0)
                                                     )
                                   ),
                    avahi.DBUS_INTERFACE_SERVICE_BROWSER)

                self.sbrowser.connect_to_signal("ItemNew",
                                                self.handler)
            except:
                raise

    def handler(self,interface, protocol, name, stype, domain, flags):
        """This method is called whenever a new server is discovered on the
        network. It makes a call over dbus to try and resolve more detailed
        information about the server, and asks for the method
        "self.service_resolved" to be called with that information."""

        self.server.ResolveService(interface, protocol, name, stype,
                                   domain, avahi.PROTO_UNSPEC, dbus.UInt32(0),
                                   reply_handler=self.service_resolved,
                                   error_handler=self.print_error)

    def service_resolved(self, *args):
        """This method is called for each new server when detailed information is
        returned about it. It parses that information and then stores them in the
        list this object represents."""

        text=str(avahi.txt_array_to_string_array(args[9]))
        if len(text) != 50:
            return
        if text[2:11] != 'server_id':
            return
        uuid = text[12:48]

        self.append((uuid,
                          str(args[2]),
                          str(args[7]),
                          int(args[8])
                          ))

    def print_error(self,*args):
        pass


class DBusThread(threading.Thread):
    """This class is just a thread which runs in the background
    and handles dbus calls"""

    daemon = True

    def run (self):
        gobject.MainLoop().run()

def start():
    DBusThread().start()




def display_servers(l):
    """This function is called to parse the list of servers into a form
    which can be displayed to the user."""

    output = """\
UC Servers on the network
-------------------------

"""
    i = 1
    for server in l:
        output += '%2d -- %10s (%s,%d)\n' % (i,server[1],server[2],server[3])
        output += '        [%s]\n' % server[0]
        output += '\n'

    return output



if __name__ == '__main__':

    l = UCServerList()

    output = display_servers(l)
    print output

    start()

    while True:
        new_output = display_servers(1)

        if output != new_output:
            output = new_output
            print output
```

The UCServerList class which is defined here can be imported by other python code and treated very much like python's built-in list class (from which it inherits). The difference is that the contents of this "list" will change over time as new servers appear and disappear from the menu. A normal user-interface mechanism for dealing with service discovery like this is to present the user with a menu of options from which to select, and to add new entries to the menu or remove them as the servers on the network change.

This example also makes it clear that the mDNS/DNS-SD discovery mechanism provides more information than just what's stored in the pairing-code – in particular it provides both the serve name and server id which can also be discovered by making a GET request to the uc resource. In this way a client can make sure that the server they've connected to is the same one they discovered using mDNS/DNS-SD.

For those planning to implement this discovery mechanism in another language which has its own library for Zeroconf networking (such as using the Objective-C and C++ bindings for Bonjour or accessing Avahi via DBus from a language other than python) the following is the technical details which you'll need to provide when searching for servers on the network:

- All UC servers will advertise themselves with the protocol `_universalctrl._tcp`.
- All UC servers will advertise themselves on the network `local`.
- All UC servers will advertise themselves with weight and priority `0`.

Armed with that information, and with the documentation for your chosen platform's mDNS/DNS-SD implementation, you should be able to seek out new servers on the network with no user interaction other than choosing from a list – which can make for a very convenient user experience.

5.3 Making the first request

Once a server has discovered a server on the local network it's time to make the first request to that server. The initial request made to a server should always be a GET request to the resource `uc`, so that the client can check that the server is compatible with its capabilities (no point trying to use a client which requires spec version 0.5.0 or above with a server using spec version 0.3.0, and no point in using a client which requires the `uc/sources` resource with a server which doesn't implement it, for example). This is pretty similar to what we did manually back in Section 3. In order to handle the response properly the client needs to be able to parse XML and extract data from it. There are libraries for doing this for most languages, and the python standard library comes with several of them. For the purpose of this tutorial we'll be using `xml.dom.minidom`, which is a light-weight "Domain Object Model" (DOM) style parser. This is the easiest type of parser to use for a tutorial because it leads to simpler easier to read code, for an actual client on a resource-constrained device many developers will likely prefer to use a "Simple API for XML" (SAX) style parser instead.

The python-code shown here will ask the user for a pairing code, decode it (using the function from the manual discovery example) and then make a GET request to `uc`. Upon receiving a response it'll parse the XML and check that the version number is 0.6.0. Finally it'll print the capabilities of the server in a pretty manner for the user.

```python
from xml.dom.minidom import parseString
import httplib
import re

# Import the function for decoding a pairing code
# from the previous example
from pairing_code_decode import decode_pairing_code

def request(server, url, verb='GET', port=48875, body=None):
    """This function simply makes a request to the specified
    server at the specified resource using the specified
    verb and returns a tuple of the status code and the body
    of the response."""

    connection = httplib.HTTPConnection(server,port)
    connection.request(verb,url,body)
    response = connection.getresponse()
    return (response.status,
            response.read())

def parse_to_bool(string):
    """This function takes a string, and returns a bool,
    anything starting with 1,t or y is True, anything
    starting 0,f, or n is False."""

    assert isinstance(string, basestring)

    if string[0] in ('1','t','y','T','Y'):
        return True
    elif string[0] in ('0','f','n','F','N'):
        return False
    else:
        raise ValueError

def parse_base_resource(xml):
    """This method parses the xml returned by the base resource
    and turns it into a standard python dictionary containing
    the server details."""

    details = dict()

    # Start by feeding the xml into the dom parser
    dom = parseString(xml)

    # Check that the root element is a response
    # and that it contains a single ucserver
    assert dom.documentElement.tagName == "response"
    assert dom.documentElement.firstChild.tagName == "ucserver"

    ucserver = dom.documentElement.firstChild

    # Parse the attributes in turn
    details['name']            = str(ucserver.getAttribute('name'))
    details['server-id']       = str(ucserver.getAttribute('server-id'))
    details['version']         = str(ucserver.getAttribute('version'))
    details['security-scheme'] = parse_to_bool(
        ucserver.getAttribute('security-scheme'))

    # The logo-href attribute is optional
    if ucserver.hasAttribute('logo-href'):
        details['logo-href'] = str(ucserver.getAttribute('logo-href'))
        assert re.match('^http(s){,1}://.+',
                        details['logo-href'])

    # Now loop through all the child nodes and interpret them as
    # implemented optional resources.
    details['resources'] = []
    for element in ucserver.childNodes:
        assert element.tagName == "resource"
        details['resources'].append(str(element.getAttribute('rref')))

    return details

def pretty_print_server_details(details):
    """This function just prints out the server details in a pretty
    human readable format."""

    print "==%s %s %s==" % ('='*int((34 - len(details['name'][:34]))/2),
                            details['name'][:34],
                            '='*(-int((len(details['name'][:34]) - 34)/2)))

    print """\
 (%(server-id)s)

UC version: %(version)s)

Optional Resources :""" % details

    if len(details['resources']) == 0:
        print "  None"
    else:
        for resource in details['resources']:
            print "  %s" % resource

def get_details_for_server(server):
    """This function makes a request to the base resource, parses the
    response checks that the version matches 0.6.0 and returns a dictionary
    containing the details of the server.

    If there's an error then it throws an exception."""

    #make the request
    try:
        (status,body) = request(server[0],'uc','GET',server[1])
    except:
        print "There was an error connecting to that server"
        raise

    if int(status/100) != 2:
        print "The request failed with code: %d" % status
        raise ValueError

    if status != 200:
        print "The request succeeded, but with code: %d" % status
        raise ValueError

    # So parse the XML
    server_details = parse_base_resource(body)

    # Check the version number (using standard regexp library)
    if not re.match('^0\.6\.0$',server_details['version']):
        print """\
The server implements non-supported version: %s\
""" % server_details['version']
        raise Exception, "Version Error"

    return server_details



if __name__ == "__main__":
    print "Please Enter Pairing Code : ",
    server = decode_pairing_code(raw_input())

    # Now get the details of the given server
    server_details = get_details_for_server(server)

    # Pretty print the output
    pretty_print_server_details(server_details)


```

The version check here makes use of the `re` python library, which implements perl-like regular expressions[4].

5.4 Displaying the State of the Main Output

So, building on the code of the previous example let's make this client actually do something more useful: to whit, let's make it display the status of the main output instead of the server's base resource. Now this requires a little more thought. Since the details of this output will be displayed to the user we won't be wanting to display source-ids or programme-ids. Instead we'll be wanting to display human readable names.

[4]You don't need to know regular expressions to parse the xml from the server, but they make it a lot easier, so I'd recommend using them. They also don't take very long to learn.

It would be possible to read the source-id from the output and then make a request to `uc/sources/`{source-id} to determine what it's human readable name is – however that's not the approach I'm going to take here. Looking ahead we would like to eventually enable this client to present the user with a list of sources to choose from when changing channel – and so it makes sense for the client to cache a list of sources acquired from the `uc/source-lists/uc_default` resource first, and then look up the human readable name in that list once the output has been queried. With that in mind our first code snippet in this section is a python script which will make a request to `uc/source-lists/uc_default`, parse the response, and then prettily print-out the list of sources for the user:

```python
from xml.dom.minidom import parseString
import re

# Import the function for decoding a pairing code
# making a request and parsing the uc resource
# from the previous examples
from pairing_code_decode import decode_pairing_code
from request import parse_to_bool, request, get_details_for_server

def parse_source_element(element):
    """This method parses an xml element representing a 'source'
    element in the return from a request to the 'sources' or
    'source-lists' resources and returns a standard python dictionary
    containing the source details."""

    details = dict()

    # Parse the element attributes to fill out the details
    details['sid']  = str(element.getAttribute('sid'))
    details['name'] = str(element.getAttribute('name'))

    # All other attributes are optional, first we parse the strings
    for attribute in ('sref','owner','default-content-id'):
        if element.hasAttribute(attribute):
            details[attribute] = str(element.getAttribute(attribute))

    # Then the booleans ...
    for attribute in ('linear','live','follow-on'):
        if element.hasAttribute(attribute):
            details[attribute] = parse_to_bool(element.getAttribute(attribute))

    # Then the URLs ...
    for attribute in ('logo-href','owner-logo-href'):
        if element.hasAttribute(attribute):
            details[attribute] = str(element.getAttribute(attribute))
            assert re.match('^http(s){,1}://.+',
                            details[attribute])

    # And finally the lcn, which is an integer
    if element.hasAttribute('lcn'):
        details['lcn'] = int(element.getAttribute('lcn'))

    return details


def parse_source_list_resource(xml):
    """This method parses the xml returned by a source-list
    resource and turns it into a standard python dictionary of
    dictionaries containing the source details."""

    sources = []

    # Start by feeding the xml into the dom parser
    dom = parseString(xml)

    # Check that the root element is a response
    assert dom.documentElement.tagName == "response"

    # Now find all elements of type "source"
    srcs = dom.getElementsByTagName('source')

    # Now loop over each one and parse it
    for element in srcs:
        details = parse_source_element(element)
        sources.append(details)

    return sources

def get_sources_for_list(server,list='default'):
    """This function makes a request to the specified
    uc/source-lists/{list-id} resource and returns a
    dictionary of dictionaries containing the details of
    the sources, indexed by the id."""

    # Make the request
    (status,body) = request(server[0],
                            'uc/source-lists/%s' % list,
                            'GET',
                            server[1])

    # Check that the status was a 200 OK (a 204 is useless to
    # us here)
    assert status == 200

    # So parse the XML
    sources = parse_source_list_resource(body)

    return sources

def pretty_print_sources(sources):
    """This function takes in a dictionary of sources and prints
    then in a pretty and human-readable fashion."""

    print """== SOURCES =="""

    sources = dict((source['sid'],source) for source in sources)

    # This just creates a list of source-ids sorted by lcn
    # followed by sources with no lcn
    sids_by_lcn = sorted([ key for key in sources if 'lcn' in sources[key] ],
                         key=lambda sid : sources[sid]['lcn'])

    for sid in sids_by_lcn:
        print '%03d -- %s%s [ %s ]' % (sources[sid]['lcn'],
                                        sources[sid]['name'][:40],
                                        ' '*(40 - len(sources[sid]['name'][:40])),
                                        sid)
    for sid in [ key for key in sources if 'lcn' not in sources[key] ]:
        print 'XXX -- %s%s [ %s ]' % (sources[sid]['name'][:40],
                                        ' '*(40 - len(sources[sid]['name'][:40])),
                                        sid)



if __name__ == "__main__":
    print "Please Enter Pairing Code : ",
    server = decode_pairing_code(raw_input())

    # Now get the details of the given server
    server_details = get_details_for_server(server)

    # Check server implements sources and source-lists
    assert 'uc/source-lists' in server_details['resources']

    # Now get the details of the sources in the
    # default source-list
    sources = get_sources_for_list(server,'uc_default')

    # Finally display the sources to the user
    pretty_print_sources(sources)

```

Armed with that code we're able to store the sources known to the server in a convenient look-up-table format, and hence we should be able to represent the "currently displayed source" for the output in a nice human readable fashion. Our next code snippet does precisely that, it requests a pairing code, connects to the server, caches the sources from the default list, makes a request to `uc/outputs/main`, and then presents the response to the user in a neatly formatted way.

```
from xml.dom.minidom import parseString
import re

# Import useful functions from the previous examples
from pairing_code_decode import decode_pairing_code
from request import parse_to_bool, request, get_details_for_server
```

```python
from check_sources import get_sources_for_list

def parse_output_resource(xml):
    """This method parses the xml returned by an output
    resource and turns it into a standard python dictionary
    containing the output details."""

    details = dict()

    # Start by feeding the xml into the dom parser
    dom = parseString(xml)

    # Check that the root element is a response
    # and that it contains a single output
    assert dom.documentElement.tagName == "response"
    assert dom.documentElement.firstChild.tagName == "output"

    details['rref'] = str(dom.documentElement.getAttribute('resource'))

    output = dom.documentElement.firstChild

    # Set some details from the attributes
    details['name'] = str(output.getAttribute('name'))

    # The remaining attributes are optional

    # Now loop over the elements inside the output element
    for element in output.childNodes:
        # If the element is a "settings" element
        if element.tagName == "settings":
            details['settings'] = dict()
            if element.hasAttribute('volume'):
                details['settings']['volume'] = float(
                    element.getAttribute('volume'))
                assert 0.0 <= details['settings']['volume'] <= 1.00
            if element.hasAttribute('aspect'):
                details['settings']['aspect'] = str(
                    element.getAttribute('aspect'))
                assert details['settings']['aspect'] in ('4:3',
                                                          '14:9',
                                                          '16:10',
                                                          '16:9',
                                                          '21:9',
                                                          'source')
            if element.hasAttribute('mute'):
                details['settings']['mute'] = parse_to_bool(
                    element.getAttribute('mute'))


        # If the element is a "programme" element
        if element.tagName == 'programme':
            sid = str(element.getAttribute('sid'))
            if element.hasAttribute('cid'):
                cid = str(element.getAttribute('cid'))
            else:
                cid = ''

            # I set the sid and pid here as a pair simply because
            # they work together as a pair to identify an individual
            # programme
            details['programme'] = (sid,cid)

        # If the element is an "app" element
        if element.tagName == 'app':
            sid = str(element.getAttribute('sid'))
            if element.hasAttribute('cid'):
                cid = str(element.getAttribute('cid'))
            else:
                cid = ''

            # I set the sid and pid here as a pair simply because
            # they work together as a pair to identify an individual
            # programme
            details['app'] = (sid,cid)


        # If the element is a playback element
        if element.tagName == 'playback':
            if element.hasAttribute('speed'):
                speed = float(element.getAttribute('speed'))
            else:
                speed = 1.0
            details['playback'] = speed

    return details

def get_output_details(server,output_id='main'):
    """This function makes a request to the specified
    uc/outputs/{id} resource and returns a dictionary
    containing the details of the output."""

    # Make the request
    (status,body) = request(server[0],
                            'uc/outputs/%s' % output_id,
                            'GET',
                            server[1])

    # Check that the status was a 200 OK (a 204 is useless to
    # us here)
    assert status == 200

    # So parse the XML
    return parse_output_resource(body)

def pretty_print_output(details,sources=None):
    """This method prints the details of an output in a pretty human
    readable fashion."""

    sources = dict((source['sid'],source) for source in sources)

    print ""
    print "==%s %s %s==" % ('='*int((34 - len(details['name'][:34]))/2),
                            details['name'][:34],
                            '='*(-int((len(details['name'][:34]) - 34)/2)))
    print ""
    if sources is not None:
        if ('programme' in details
            and details['programme'][0] in sources):
            print "Currently Showing: %21s" % (
                sources[details['programme'][0]]['name'][:21],)
            print ""
        elif ('app' in details
              and details['app'][0] in sources):
            print "Currently Using: %21s" % (
                sources[details['app'][0]]['name'][:21],)
            print ""
    if 'settings' in details:
        if 'volume' in details['settings']:
            print "Volume            : %21s" % (("%3d%%" % int(
                        details['settings']['volume']*100 + 0.5))[:21],)
    if 'playback' in details:
        print "Playback          : %21s" % (("%1.1fx speed" % (
                    details['playback'],))[:21],)
    print ""


if __name__ == "__main__":
    print "Please Enter Pairing Code : ",
    server = decode_pairing_code(raw_input())

    # Now get the details of the given server
    server_details = get_details_for_server(server)

    # Check server implements sources and source-lists
    assert 'uc/source-lists' in server_details['resources']

    # Now get the details of the sources in the
    # default source-list
    sources = get_sources_for_list(server,'uc_default')

    # Check server implements outputs
    assert 'uc/outputs' in server_details['resources']

    # Now get the details of the main output
    output = get_output_details(server,'main')

    # Finally display the output state for the user
    pretty_print_output(output,sources=sources)

```

As you can see, we're well on our way to producing a workable – if very simple – "display" client (ie. one which displays the state of the box but doesn't allow the user to modify it). The thing that's missing right now is having the display update when the box state does. That's the subject of the next section.

5.5 Event Notification

Building on the last example it might be very tempting to write some code like this to make a GET request every second and update the display with a new version of the output display:

```
from xml.dom.minidom import parseString
import re
import os
import time

# Import useful functions from the previous examples
from pairing_code_decode import decode_pairing_code
from request import parse_to_bool, request, get_details_for_server
from check_sources import get_sources_for_list
from show_output import get_output_details, pretty_print_output

if __name__ == "__main__":
    print "Please Enter Pairing Code : ",
    server = decode_pairing_code(raw_input())

    # Now get the details of the given server
    server_details = get_details_for_server(server)

    # Check server implements sources and source-lists
    assert 'uc/source-lists' in server_details['resources']

    # Now get the details of the sources in the
    # default source-list
    sources = get_sources_for_list(server,'uc_default')

    # Check server implements outputs
    assert 'uc/outputs' in server_details['resources']

    n = 0
    while True:
        # Clear the screen
        os.system('clear')

        # Now get the details of the main output
        output = get_output_details(server,'main')

        print "Update Count: %d" % n

        # Finally display the output state for the user
        pretty_print_output(output,sources=sources)

        time.sleep(1)
        n += 1




```

This will work, but I strongly recommend against doing it. It's inelligent and puts a lot of strain on the server (especially if multiple clients are behaving this way all at once). Luckily the API provides a better way of handling feedback from the server in the form of the `uc/events` resource.

So, let's take a look at this resource the old-fashioned way, using a web-browser. The response we get from a `GET` request to `uc/events` looks something like this:

```
<response resource="uc/events">
    <events notification-id="4c56ced500000588"/>
</response>
```

which isn't particularly informative. In fact it's down-right uninformative. The reason for this is that the `uc/events` resource requires a query parameter in order to return useful results. That query parameter is called `since` and it take the form of an "event-id". We've been given an event-id in the response above, so it would seem natural to use that one now. I'm not going to do that (you'll see why in a moment, trust me, there's a good reason). Instead I'm going to make the request with the event-id set to `"0000000000000000"`. When I make a request to `uc/events?since=0000000000000000` the response is then as follows:

```
<response resource="uc/events?since=0000000000000000">
    <events notification-id="4c56ced500000589">
        <resource rref="uc/power"/>
        <resource rref="uc/outputs/0/playhead"/>
        <resource rref="uc/storage"/>
        <resource rref="uc/outputs/0"/>
        <resource rref="uc/source-lists/uc_storage"/>
    </events>
</response>
```

That's a lot more helpful! What this tells us is that since the time identified by event-id `"0000000000000000"` the resources `uc/power`, `uc/outputs/0`, `uc/outputs/0/playhead`, `uc/storage`, and `uc/source-lists/uc_storage` have all undergone "notifiable changes" – the UC term for the kind of changes in state which a client might want to know about.

The response also tells us that the current time is identified by the event-id `"4c56ced500000589"`. So now let's make a requst to `uc/events?since=4c56ced500000589`. The browser makes the request, but no new XML page appears! In fact it reports that it's waiting for a response.

What's going on here?

The server won't actually send a response back to the browser until such a time as it has something to send (or a timeout occurs). To demonstrate this, whilst the browser is hanging around waiting for a response make a change to one of these resources (you can do that by firing up python and making a PUT request to `uc/outputs/main` or going over to the box and changing volume or channel manually or using an IR remote. On my example server I'm changing the channel using he Myth TV frontend). As soon as you make the change the browser will receive a response and display something like the following:

```
<response resource="uc/events?since=4c56ced500000589">
    <events notification-id="4c56ced50000058a">
        <resource rref="uc/outputs/0"/>
    </events>
</response>
```

Which tells us that the current event-id is now `"4c56ced50000058a"` and that since event-id `"4c56ced500000589"` there has been a notifiable change to the resource `uc/outputs/0` (which is the main, and indeed only, output on my example server).

The event-id has been incremented because a response was sent back to the user. It's not an actual time measurement and doesn't just change unless it has a reason to.

This whole mechanism provides a method for clients to "listen" to see if resources have changed, and only make `GET` requests to fetch their state when notified that they have. This is a much better way of dealing with changes than polling once per second would be.

With this in mind the next code snippet is a python script which presents the current state of the output, and updates it only when notified that it has changed. To do this I've employed python's `threading` module in a fairly simple manner. Since multithreading is often considered a more advanced coding topic I'll run through the details of how the multithreading here works (I've actually used threading already in a previous example, but only in a very minor way):

- `Thread` is a class which represents a thread of execution which will start when the `start` method is called and run the `run` method.

- `Condition` is a class which represents a type of "Lock" which can "acquired" by a thread and later released. Whilst one thread is executing code which is marked as being `with` the lock no other code which is marked as being `with` the lock can execute at the same time. A `Condition` has another capbility, one thread can `wait` for the Condition to have `notify` (or `notifyAll`) called on it by another thread, and delay any further execution until this happens.

With that, here's the code:

```python
from xml.dom.minidom import parseString
import re
import os
import time
from threading import Thread, Condition, RLock

# Import useful functions from the previous examples
from pairing_code_decode import decode_pairing_code
from request import parse_to_bool, request, get_details_for_server
from check_sources import get_sources_for_list
from show_output import get_output_details, pretty_print_output

```

```python
def parse_events_resource(xml):
    """Parse the xml from the response from a request to
    the uc/events resource.

    Returns a tuple of the current event-id and a list of
    resource URIs."""

    details = []
    id = None

    dom = parseString(xml)

    assert dom.documentElement.tagName == 'response'
    assert dom.documentElement.firstChild.tagName == 'events'

    events = dom.documentElement.firstChild

    id = str(events.getAttribute('notification-id'))

    for element in events.childNodes:
        assert element.tagName == 'resource'
        details.append(str(element.getAttribute('rref')))

    return (id,details)

def get_events_details(server,id=None):
    """This function simply makes a request to the uc/events resource
    using the supplied id (if one is supplied), parses the output
    and returns a tuple of the new event-id and a list of resources
    which have changed"""

    resource = 'uc/events' if id is None else 'uc/events?since=%s' % id

    # Make the request
    (status, body) = request(server[0],
                             resource,
                             'GET',
                             server[1])

    # Check that the status was a 200 OK ( a 204 is useless to
    # us here)
    assert status == 200

    # Parse the XML
    return parse_events_resource(body)

class EventListeningThread (Thread):
    """This class is initialised with the details of a server and
    also the relative URI of a resource to watch for changes to.

    It maintains a Condition which will be notified whenever the
    requested resource has undergone a notifiable change."""

    daemon = True

    def __init__(self, server, rrefs):
        self.server = server
        self.rrefs  = rrefs
        self.lock   = Condition()
        Thread.__init__(self)


    def run(self):
        (id, events) = get_events_details(self.server)

        while(True):
            (id, events) = get_events_details(self.server,id)

            with self.lock:
                if any([ rref in events for rref in self.rrefs ]):
                    self.lock.notifyAll()

if __name__ == "__main__":
    print "Please Enter Pairing Code : ",
    server = decode_pairing_code(raw_input())

    # Now get the details of the given server
    server_details = get_details_for_server(server)

    # Check server implements sources and source-lists
    assert 'uc/source-lists' in server_details['resources']

    # Now get the details of the sources in the
    # default source-list
    sources = get_sources_for_list(server,'uc_default')

    # Check server implements outputs
    assert 'uc/outputs' in server_details['resources']

    # Check server implements events
    assert 'uc/events' in server_details['resources']

    # Clear the screen
    os.system('clear')

    # Now get the details of the main output
    output = get_output_details(server,'main')

    n = 0
    print "Update Count: %d" % n

    # Display the initial output state for the user
    pretty_print_output(output,sources=sources)

    # Start the Event Listening Thread
    event_listening_thread = EventListeningThread(server, [ output['rref'],])
    event_listening_thread.start()

    while True:
        with event_listening_thread.lock:
            # Wait 20 seconds or until the output is updated, whichever
            # comes first
            event_listening_thread.lock.wait(20)

            n += 1

            # Clear the screen
            os.system('clear')

            # Now get the details of the main output
            output = get_output_details(server,'main')

            print "Update Count: %d" % n

            # Finally display the output state for the user
            pretty_print_output(output,sources=sources)
```

Much like the previous example it prints an "update count" which is incremented each time the display is updated, but now instead of being updated once per second it's updated once every 20 seconds, or when a notifiable change occurs in `uc/outputs/0` whichever comes first.

The mechanism for implementing the events notification used here was done using multithreading and synchronous HTTP requests, but the same events mechanism can be easily implemented using asynchronous HTTP requests in a single-threaded environment (such as javascript or actionscript).

6 Building an Electronic Programme Guide

In Section 3.1 I showed that a `GET` request made to the `uc/search/sources/5287` resource can be used to get information about an upcoming TV channel schedule. In fact these resources are very flexible, with a number of options which can be supplied as query paramters and a number of different forms which searches can take. So for example the `uc/search/outputs/0` will return the content currently presenting on output 0, whilst `uc/search/source-lists/uc_storage?text=Doctor%20Who` will return the first programme from each of the sources listed in the "uc_storage" source-list which includes the literal string "Doctor Who" somewhere in its title or synopsis.

Some of the options available to determine how to search when making a "uc/search/{foo}/{bar}" request are:

- `uc/search/sources/{sid1};{sid2};...`: is used to specify source-ids to search in.
- `uc/search/source-lists/{slid}`: is used to specify a source-list to search all of the sources in.

- `uc/search/outputs/{id}`: is used to specify that you want to see the content being presented on a particular output.
- `uc/search/text/{text string}`: is used to search for content containing a particular text string in its title or synopsis.
- `uc/search/categories/{id}`: is used to search for content in a particular "category" (more on what this means later).

there are several others, but they're less important at the current time in the tutorial and would require introducing a lot of new concepts.

There are also a number of options available for filtering returned results by their properties. Two of them make use of timestamps. As with everywhere else in the UC specification which deals with timestamps the format is as described in RFC 3339[5, Section 5.6], where it is refered to as "date-time". Simply put a UC compliant timestamp should be of the form such that the 35 minutes and 1.05 seconds past 12 noon GMT on the 29th of Februrary 2012 could be written as `"2012-02-29T12:35:01.05+00:00"`. The fractional part of the seconds (including the '.') can be omitted if it's zero, and if present can have any number of digits following the decimal point (although it's highly unlikely that higher precission than 1/60th of a second will have much use in near-future implementations of the protocol), the final portion of the code is an inidcator of the timezone, expressed as a number of hours and minutes difference from UTC (in this case `"+00:00"` since GMT and UTC are coincident), the whole timezone section could also be replaced with the character `"Z"` – a special character allowed by the format to indicate that the time is in UTC[5].

The options available for filtering the results include:

- `start`: is used to specify a start time. Only programmes which end or stop being available after this time (or programmes which have no end or last-availability times) will be included in the results. If `start` isn't specified then the server will treat the current time when the request was received as the start time of the request.
- `end`: is used to specify an end time. Only programmes which start or start being available before this time (or programmes which have no start or first-availability times) will be included in the results.
- `days`: is used instead of an end time to specify the duration for which results will be returned. It takes the form of a positive integer, and if set to `"1"` then the results will be the same as if `end` had been specified as the next midnight after the start time *in the timezone the box uses.* If the number is higher then it extends the end time by extra full days.
- `sid`: is used to specify a source-id which must be matched. It can be specified more than once in which case only content which matches at least one of the specified sources will be included.
- `cid`: is used to specify a content-id which must be matched. It can be specified more than once, in which case only content which match at least one of the specified content-ids will be included in the response.
- `series-id`: is used to specify a series-id which must be matched. Only programmes which match the specified series-id will be included in the response.

[5]This format lacks any mechanism for describing dates prior to `"0000-01-01T00:00:00+23:59"` (ie. one minute after midnight UTC on 31st December 2BC) or after `"9999-12-31T23:59:60-23:59"` (ie. one minute before midnight, the 1st of January 10000). This is not expected to cause a problem!

- `gcid`: is used to specify a global-content-id which must be matched. It can be used more than once on the same request. Only programmes which match one of the specified global-content-ids will be included in the response. Since this is a global id it needs to be percent-encoded before passing it to the server.

- `gsid`: is used to specify a global-series-id which must be matched. It can be used more than once on the same request. Only programmes which match one of the specified global-series-ids will be included in the response. Since this is a global id it needs to be percent-encoded before passing it to the server.

- `gaid`: is used to specify a global-app-id which must be matched. It can be used more than once on the same request. Only programmes which match one of the specified global-app-ids will be included in the response. Global app ids are only used for interactive content.

- `category`: is used to specify a category(see Section 9). What this is for will be explained later, but for now suffice it to say that it's a string, can be included more than once, and if it's used then only programmes who fall into one of the specified categories will be included in the output.

- `text`: is used to provide a text string to search the title and synposis of the programmes which are found for. This is a case-sensitive string search, and if multiple strings are provided only resulst which match all of them will be returned.

- `interactive`: is used to specify whether or not to include interactive content in the output. The default is "true", but you can set it to "false".

- `AV`: is used to specify whether or not to include non-interactive content in the output. The default is "true", but you can set it to "false".

In addition the parameter `field=title` can be specified to ensure that the `text` option (or the main search term of `uc/search/text`) only searches the titles of content, and the parameter `field=synopsis` can be specified to ensure that it only searches the synopses of the content.

If no filter options are specified the only filtering that will be done will be done based upon the "implicit" start-time (equal to the time when the server processed the request).

Finally there are two options which are used to control how many results are returned by a request:

- `results`: is an integer which limits the response to the specified number of results. On requests to `uc/search/sources/{sids}` this number of results will be returned *per source.*

- `offset`: is the number of programmes from the start of the results list (for each source if multiple sources were specified) to skip before starting to include programmes. Hence if a request to `uc/search/sources/5768;5769?results=2&offset=5` will return the sixth and seventh programmes from the (filtered) list of results for source 5768 followed by the sixth and seventh programmes from the filtered list of programmes from the source 5769.

If no numbers are specified then the server will try to return one piece of content (or one from each source). It's strongly recommended that most requests be made with `results` set to a higher value, as one is rarely the number of results wanted! It's always possible that the server might run out of programmes to return before returning the number you specified, in that case it'll return a smaller number.

A python script which will list the current and next programme on all live sources is shown below:

```python
from xml.dom.minidom import parseString
import re
import os
import time
from threading import Thread, Condition, RLock
import datetime

# Import useful functions from the previous examples
from pairing_code_decode import decode_pairing_code
from request import parse_to_bool, request, get_details_for_server
from check_sources import get_sources_for_list
from show_output import get_output_details, pretty_print_output

def pretty_print_content_list(l,sources=None):
    """This method prints a list of content in a pretty fashion."""

    if sources is not None:
        source_name = [ source['name'] for source in sources if source['sid'] == l[0]['
        if not source_name:
            return
        print "%s :" % ((source_name[0] + ' '*88)[:88],)
        print "-"*90

    for c in l:
        if 'start' in c:
            print "%s -- " % (c['start'].strftime('%H:%M'),),
        else:
            print "XX:XX -- ",

        print '%s' % (c['title'] + ' '*80)[:80]
        if 'synopsis' in c:
            for n in range((len(c['synopsis']) + 79)//80):
                print "          %s" % (c['synopsis'][n*80:(n+1)*80],)

    print ""

def parse_content_lists(xml):
    """This method parses an XML response from the uc/search subresources
    and returns it as a list of lists of dictionaries."""

    def __fromisodate(iso):
        match = re.match(
            r"(\d{4})-(\d{2})-(\d{2})T(\d{2}):(\d{2}):(\d{2})(?:\.(\d+))?(Z|(?:(\+|-)(\
            iso)
        micros = 0
        if match.group(7):
            micros = int(((match.group(7) + '0000000')[:7] + 5)//10)
        d = datetime.datetime(int(match.group(1)),
                              int(match.group(2)),
                              int(match.group(3)),
                              int(match.group(4)),
                              int(match.group(5)),
                              int(match.group(6)),
                              micros)
        if match.group(8) != 'Z':
            de = datetime.timedelta(hours=int(match.group(10)),
                                    minutes=int(match.group(11)))
            if match.group(9) == '+':
                d -= de
            if match.group(9) == '-':
                d += de
        return d

    lists = []

    # Start by feeding the XML into the dom parser
    dom = parseString(xml)

    # Check that the root element is a response
    assert dom.documentElement.tagName == "response"

    # Now loop over the results elements
    for results_element in dom.getElementsByTagName('results'):
        l = []

        # Now loop over the content elements in the list
        for content_element in results_element.getElementsByTagName('content'):
            content = dict()

            # Read out the attributes from the content
            for key in ('sid','cid','title','logo-href',):
                if content_element.hasAttribute(key):
                    content[key] = content_element.getAttribute(key)
            for key in ('interactive',):
                if content_element.hasAttribute(key):
                    content[key] = (
                        content_element.getAttribute(key).lower() in
                        ('true','yes','1'))
            for key in ('duration',):
                if content_element.hasAttribute(key):
                    content[key] = datetime.timedelta(seconds=float(content_element.g
            for key in ('start','acquirable-until','presentable-from',
                        'presentable-until'):
                if content_element.hasAttribute(key):
                    content[key] = __fromisodate(
                        content_element.getAttribute(key))

            # Now parse the synopsis
            synopses = content_element.getElementsByTagName('synopsis')
            if synopses:
                content['synopsis'] = synopses[0].firstChild.data

            # Parse the category information
            content['categories'] = []
            category_elements = content_element.getElementsByTagName('category')
            for category_element in category_elements:
                content['categories'].append(category_element.getAttribute('id'))

            # Parse Media-components
            if 'interactive' in content and not content['interactive']:
                content['media-components'] = []
                media_components = content_element.getElementsByTagName('media-componen
                for mce in media_components:
                    mc = {}
                    for key in ('id','type','name','aspect','vidformat','subformat','la
                        if mce.hasAttribute(key):
                            mc[key] = mce.getAttribute(key)

                    for key in ('default',):
                        if mce.hasAttribute(key):
                            mc[key] = ( mce.getAttribute(key) in
                                        ('true','yes','1'))


            # Append the content to the list
            l.append(content)

        # Append this list to the return set
        lists.append(l)

    return lists


def search_source_list(server,slist,params):
    """This method makes a request to the uc/search/source-lists/{id}
    resource with the provided details and returns a list of lists of
    content."""

    params = '&'.join('%s=%s' % (key,value) for (key,value) in params.items())
    if params != '':
        params = '?' + params

    # Make the request
    (status,body) = request(server[0],
                            'uc/search/source-lists/%s%s' % (slist,params),
                            'GET',
                            server[1])

    # Check that the status was 200
    assert status == 200

    # So parse the XML
    return parse_content_lists(body)
```

```python

if __name__ == "__main__":
    print "Please Enter Pairing Code : ",
    server = decode_pairing_code(raw_input())

    # Now get the details of the given server
    server_details = get_details_for_server(server)

    # Check server implements sources and source-lists
    assert 'uc/source-lists' in server_details['resources']

    # Now get the details of the sources in the
    # default source-list
    sources = get_sources_for_list(server,'uc_default')

    # Check server implements outputs
    assert 'uc/outputs' in server_details['resources']

    # Check server implements search
    assert 'uc/search' in server_details['resources']

    # Now get the details of the content
    content = search_source_list(server,
                                 'uc_default',
                                {'results' : '2'})

    for l in content:
        if len(l) > 0:
            pretty_print_content_list(l,sources=sources)
```

7 Channel Up and Down

Ok, so I've included this because everyone asks to be able to do it early on. I'm not recommending it as a good interface design (selecting from a list of human readable names is generally much nicer than simple up/down buttons), but it's fairly easy to achieve.

The following code will run, retrieve a list of sources, and change the channel of the output one "up" the list.

A much better interface would be to wait for a keypress to control the channels, but that would require using something like the `curses` library to provide the UI, and whilst that's not too tricky at all it's definitely beyond the scope of this tutorial.

```python
from xml.dom.minidom import parseString
import re

# Import useful functions from the previous examples
from pairing_code_decode import decode_pairing_code
from request import parse_to_bool, request, get_details_for_server
from check_sources import get_sources_for_list
from show_output import get_output_details

def set_output_source(server,output_id='main',sid=None):
    """This function makes a POST request to the specified
    uc/outputs/{id}?sid={sid}."""

    sid = (sid if sid is not None else 'mythtv')

    (status, body) = request(server[0],
                             'uc/outputs/%s?sid=%s' % (output_id,
                                                       sid),
                             verb='POST')
    return (status%100 == 2)

if __name__ == "__main__":
    print "Please Enter Pairing Code : ",
    server = decode_pairing_code(raw_input())

    # Now get the details of the given server
    server_details = get_details_for_server(server)

    # Check server implements sources and source-lists
    assert 'uc/source-lists' in server_details['resources']

    # Now get the details of the sources in the
    # default source-list
    sources = get_sources_for_list(server,'uc_default')

    # Check server implements outputs
    assert 'uc/outputs' in server_details['resources']

    # Now get the details of the main output
    output = get_output_details(server,'main')

    # Now sort the sources into order based on their logical channel number
    # and then alphabetically for any that lack one
    sources_in_order = (sorted([ sid for sid in sources.keys()
                                 if 'lcn' in sources[sid] ],
                               key=lambda sid : sources[sid]['lcn']) +
                        sorted([ sid for sid in sources.keys()
                                 if 'lcn' not in sources[sid] ]))

    # Now find the index of the current source in the list
    index = -1
    if 'programme' in output and output['programme'][0] in sources_in_order:
        index = sources_in_order.index(output['programme'][0])


    # Send the change channel instruction to the server
    set_output_source(server,
                      'main',
                      sources_in_order[(index +1)%len(sources_in_order)])


```

The code itself is, as you can see, not at all complex.

A few python functions have been used here which I haven't used before, but they're all fairly self-explanatory: `sorted` is used to sort a list into order. Passing in a `key` parameter is a way of controling what is used to sort the elements of the list; in this case we pass in a dynamically created function (which is what the `lambda` statement does) which takes a single parameter which is a key in the list and then returns the logical channel number of the asosciated source, thus ensuring that `sorted` will sort the keys based on an integer sort of the logical channel numbers. Similarly the method `index` just returns the firt index at which a particular entry appears in a list.

8 Accessible Remote Control

So far we've covered the good ways to do most of the simple control things people tend to want to do with their servers: changing channel, changing volume, etc.... The ways we've covered to do that are designed to be RESTful, and to allow the remote client to implement its own interface designs regardless of what choices the set-top-box manufacturer has made in designing their own interface.

There is another way to control the set-top-box, though.

A Bad way.

You can send key-codes to simulate remote-control buttons being pushed on an infrared remote.

Now, obviously you won't want to this, but unfortunately you may have to. The simple fact is that a lot of "red-button" services provided on digital TV can currently only be controlled by sending button pushes to them. This isn't accessible for people with visual impairments, and it's pretty terrible for a lot of devices, but it is an option.

So, with that caveat, here's how you do it: by making requests to `uc/remote`. The result of a `GET` request to this resource is shown below:

```
<response resource="uc/remote">
  <remote>
    <controls profile=":uk_keyboard"/>
  </remote>
</response>
```

showing that the MythTV server implements a single "control profile" (which is a profile listing button codes which the server will respond to), namely the profile `":uk_keyboard"`. This profile is defined in the spec(see [1]), but for the purposes of a simple button-mashing remote simulation we don't need it, instead we can make use of a few standard key-codes which are also defined in the spec which are considered to not be part of any profile:

- `"::POWER"` for a power button.
- `"::0"` to `"::9"` for number buttons.
- `"::RED"`, `"::GREEN"`, `"::YELLOW"`, and `"::BLUE"` for "colour" buttons like those found on UK digital remotes.
- `"::EPG"` for a button which brings up an onscreen electronic programme guide.
- `"::MENU"` for a button which brings up the main menu.
- `"::TEXT"` for a button which activates teletext if its available.
- `"::CURSOR_UP"`, `"::CURSOR_DOWN"`, `"::CURSOR_LEFT"`, and `"::CURSOR_RIGHT"` for arrow buttons for navigation.

- `"::OK"` for the button used to select items from menus.
- `"::BACK"` for a button which goes back one menu level.
- `"::VOLUME_UP"` and `"::VOLUME_DOWN"` for volume control.
- `"::CHANNEL_UP"` and `"::CHANNEL_DOWN"` for channel changing.
- `"::MUTE"` for a button which mutes the sound.
- `"::AUDIO"` for a button which turns audio description on and off.
- `"::SUBTITLE"` for a button which turns subtitles on and off.

there's no garauntee that a box will respond to all of these button presses, but there is a garauntee that *if* the box accepts input from a remote with a button which matches the specified description then it will respond to the apropriate key-code.

The MythTV box will respond to most of these, and we can send them to the box using a simple `POST` request such as one to the resource `uc/remote?button=::MUTE` to send the button-press intended to mute or unmute the audio.

The python to do this is left as an exercise to the reader, but it's fairly straightforward.

By itself this mechanism for controlling the box is completely unusable by users with accessibility requirements, but the spec has a second mechanism designed to make it slightly more widely accessible, in the form of the `uc/feedback` resource.

Unfortunately this resource isn't very useful on the MythTV implementation.

In theory it's intended that in future a client will be able to make a GET request to `uc/feedback` to get a response of the form:

```
<response resource="uc/feedback">
  <feedback>"Watch Live TV" selected.</feedback>
</response>
```

which would enable the client to convey the feedback text supplied directly to the user (possibly using text-to-speach software) and would thus enable a visually impaired user to navigate the built-in interface of the box more easily. As a mechanism this requires the built-in-interface of the box to be built with accessible text for conveying in such a way – and currently the MythTV Frontend isn't.

If you want to play around with this mechanism then the current MythTV server registers itself on the DBus Session Bus using the well-known name `uk.co.bbc.UniversalControl` and provides an object with path `/UniversalControl/Feedback` which implements an interface called `uk.co.bbc.UniversalControl.Feedback` which has a single method `setFeedbackText` which takes a string as a parameter. By calling this method you can set the text returned by `GET` requests to `uc/feedback` to be whatever you like, and each time such a change is made a notifiable change is triggered for that resource (which can be picked up by clients monitoring the `uc/events` resource).

9 Categories

As well as being able to group content by source the server also has the capability to group content using a more abstract concept of "categories". To find out what categories are supported on this server we make a `GET` request to `uc/categories` and get a response like:

```
<response resource="uc/categories">
  <categories>
    <category name="Genres">
      <category name="Sport" category-id="spo"/>
      <category name="Lifestyle" category-id="lif"/>
      <category name="Entertainment" category-id="ent"/>
      <category name="Educational" category-id="edu"/>
      <category name="Drama" category-id="dra"/>
      <category name="Children's" category-id="chi"/>
      <category name="Movie" category-id="mov"/>
      <category name="News and Factual" category-id="new">
        <category name="Children's News" category-id="chn"/>
      </category>
      <category name="Show/Game Show" category-id="sho"/>
    </category>
  </categories>
</response>
```

which indicates that this server supports 9 different categories with specific ids, all of which are considered subcategories on the “Genres” category. In general this xml response will contain one or more top level categories, each represented by a category element. Category elements can contain zero or more category elements within them, have a mandatory “name” attribute indicating a human readable name, and have an optional “category-id”. Categories without an id will never be used directly, they're just a tool for grouping other categories together. Categories with an id can certainly have subcategories, though.

The main use of category ids is to search for content using the resource `uc/search/category`, and to filter content searched for using other `uc/search` subresources.

10 Acquisitions (ie. scheduled recordings, downloads, all that jazz)

So, one thing which I haven't covered yet is how to book a recording on your PVR, something you probably will want to do.

In UC-land a booked a recording of a TV programme is called an “acquisition”. It's not the only thing that can be an acquisition, though. As far as UC is concerned booking a recording for later is much the same as asking a system to download a programme from an internet server (or even asking it to download an “app”) – all that matters is that there is some “source” for the content (ie. a TV channel or internet download service) and that the content will be copied to the storage on the box for later viewing/running/listening/whatever.

To look at the list of currently scheduled acquisitions we make a GET request to `uc/acquisitions` (sensibly enough), and get a response somewhat like this:

```
<response resource="uc/acquisitions">
  <acquisitions>
    <content-acquisition
        acquisition-id="61__5168__2011-03-23T15%3a05%3a00Z" sid="5168"
        cid="2011-03-23T15%3a05%3a00Z" interactive="false"
        series-id="fp.bbc.co.uk%2fkq9gkf" start="2011-03-23T15:05:00Z"
        end="2011-03-23T15:35:00Z" series-linked="true" active="false"/>
    <series-acquisition acquisition-id="061" series-id="fp.bbc.co.uk%2fkq9gkf"/>
  </acquisitions>
</response>
```

which tells us that currently there is a single “series-acquisition” booked – which will cause the box to acquire every piece of content which becomes available which has the series-id `"fp.bbc.co.uk%2fkq9gkf"` (you can find out the series-id of a piece of content by looking for it in the response to a `uc/search` request), furthermore there is also currently one “content-acquisition” booked – meaning that when it’s able to box will acquire the content identidied by the cid `"2011-03-23T15%3a05%3a00Z"` from the source `"5168"`; since this content id has its `series-linked` attribute set to `"true"` we know that it was booked automatically by the box as a result of the series-acquisition; the start and end attributes tell us when the box expects to start acquiring the content, and when it expects to finish doing so; the active attribute tells us that it’s not currently in the process of being acquired.

We can book a new acquisition simply by sending a `POST` request to the same resource with one of the following query parameter combinations specified:

- Specify **sid** and **content-id** to create a new content-acquisition to acquire the content identified by a particular cid from a particular source.
- Specify **global-content-id** to create a new content-acquisition to acquire the content identified by the gcid from whatever source is convenient at the next available oportunity. This is the recommended way to make such bookings.
- Specify **series-id** to create a new series-acquisition to acquire all content in the specified series.

Similarly there are sub-resources underneath this resource of the form `uc/acquisitions/{acquisition-id}` (eg. `uc/acquisitions/061`) which will give the user data about the specified acquisition if a `GET` request is made, and will cancel it if a `DELETE` request is made.

When `DELETE`ing a “series-linked” content-acquisition only that particular acquisition is canceled – not the series-acquisition which created it, so be careful to `DELETE` the right thing!

11 Stored Content

Once content has been acquired by an acquisition the acquisition disappears. The content itself then becomes available from the internal storage media of the box, and will likely appear added as a piece of content on one of the sources in the “uc_storage” source-list (take a look with a request to `uc/search/source-lists/uc_storage`). If all you want to do is view/use/listen to the content then that’s all you need, and you can treat it exactly like any other content you might want to present on an output – the fact that it’s on the local storage media doesn’t really matter.

There is one time when you’ll need to treat it differently, though – when you want to delete content from the local drive in order to save space.

The way to do that is using `uc/storage` and its subresources. A `GET` request to this resource returns something like this:

```
<response resource="uc/storage">
  <storage size="10147993032" free="10076378104">
    <stored-content cid="5168_20101111180000" sid="SG_1"
        lobal-content-id="crid://fp.bbc.co.uk/5a6sex"
        created-time="2010-11-11T18:30:00Z" size="512654776"/>
    <stored-content cid="20080531_180000_bbcone_doctor_who.ts" sid="SG_2"
        created-time="2010-10-14T16:45:43Z"
```

```
            size="604216020"/>
        <stored-content cid="20100811_200000_bbctwo_the_normans.ts" sid="SG_2"
            created-time="2010-09-21T14:32:08Z"
            size="603265891"/>
        <stored-content cid="20101118_203000_bbctwo_autumnwatch.ts" sid="SG_2"
            created-time="2011-01-10T13:58:23Z"
            size="808665453"/>
    </storage>
</response>
```

which tells us how much storage space the device has, how much of it is free, and how much storage space is used up by each of several stored pieces of content. If we want to narrow things down to only one particular source of stored content we can do so by specifying a subresource using a source id from the "uc_storage" list, so a GET request to uc/storage/SG_2 will give us:

```
<response resource="uc/storage/SG_2">
  <storage>
    <stored-content cid="20080531_180000_bbcone_doctor_who.ts" sid="SG_2"
        created-time="2010-10-14T16:45:43Z"
        size="604216020"/>
    <stored-content cid="20100811_200000_bbctwo_the_normans.ts" sid="SG_2"
        created-time="2010-09-21T14:32:08Z"
        size="603265891"/>
    <stored-content cid="20101118_203000_bbctwo_autumnwatch.ts" sid="SG_2"
        created-time="2011-01-10T13:58:23Z"
        size="808665453"/>
  </storage>
</response>
```

and then we can make a request directly for the details of a specific piece of content by, for example, making a GET request to uc/storage/SG_2/20080531_180000_bbcone_doctor_who.ts which gives:

```
<response resource="uc/storage/SG_2/20080531_180000_bbcone_doctor_who.ts">
  <stored-content cid="20080531_180000_bbcone_doctor_who.ts" sid="SG_2"
      created-time="2010-10-14T16:45:43Z"
      size="604216020"/>
</response>
```

and finally, to delete this piece of content we merely have to send a DELETE request to the same uri. Be warned that this will almost certainly immediately delete the content without prompting to make sure you're sure – don't send the request if you aren't sure! (and if you're implementing a client then make sure *you* ask the user if they're sure before sending the request)

If you are looking for assistance with some facet of the API not yet covered in this tutorial please email james.barrett@bbc.co.uk *with requests.*

References

[1] J. Barrett, M. Hammond, and S. Jolly. The Universal Control API v. 0.5.1. White Paper 194, BBC R&D, 2011.

[2] The python tutorial. http://docs.python.org/tutorial/.

[3] Bonjour. http://developer.apple.com/networking/bonjour/faq.html.

[4] Avahi. http://avahi.org/.

[5] Ed. Klyne, G. and C. Newman. Date and Time on the Internet: Timestamps. RFC 3339, IETF, 2002.