Research White Paper

WHP 193

June 2011

The Universal Control API version 0.6.0

An Overview

J.P. Barrett, M.E. Hammond, S.J.E. Jolly

BRITISH BROADCASTING CORPORATION

BBC Research & Development
White Paper WHP 193

The Universal Control API version 0.6.0

James P. Barrett Matthew Hammond Stephen Jolly

Abstract

The Universal Control API is a RESTful Web API designed to allow the core functionality of any network-connected set-top box, personal video recorder, Internet radio or similar device to be controlled from client software running on another device on the home network. This note provides a high-level introduction to the API and explains the decisions taken in its design, placing it in the context of the technologies it relies on and is similar to. This White Paper replaces a previous one describing an earlier version of the API.

Additional key words: remote, control, interface, HTTP, REST, accessible, UPnP, DLNA, IP

BBC Research & Development
White Paper WHP 193

The Universal Control API version 0.6.0

James P. Barrett Matthew Hammond Stephen Jolly

Contents

White Papers are distributed freely on request.

Authorisation of the Head of Research is required for publication.

BBC Research & Development
White Paper WHP 193

The Universal Control API version 0.6.0

James P. Barrett Matthew Hammond Stephen Jolly

1 Introduction

The Universal Control (UC) project is an attempt to define an API for televisions, set-top boxes (STBs) and similar home media devices (such as Internet radios), hereafter referred to as "boxes", whereby a client running on external hardware communicates with a server hosted on the box. For example, a remote control application could be written to run on a user's mobile phone or laptop, connect to one or more boxes over the home network, display status information and control them. The API does not restrict clients to a particular style of user interface; rather, it exposes the underlying *state* of the box on which it runs: what the box is currently displaying, the contents of its electronic programme guide (EPG), the list of recordings on its disk(s), etc. Clients can read the aspects of its state that the box chooses to let them access, request changes to the box's state, and receive near-realtime notifications when aspects of the box's state change.

In section 2 of this White Paper we will set out some of the purposes of the API: the things that it has been designed to permit the users of home media devices to do. In section 3 we then talk about some of the existing technologies (such as DLNA) by which some of the API's purposes can be fulfilled, and set out the case for a new home network API that complements them. In section 4 we then set out a brief high-level overview of that API's design and features. In section 5 this Paper will then describe the technologies selected as the basis for communication between client devices and the box, and explain some of the key decisions made in the API design process.

2 Purposes and Motivation

In this section we set out the key purposes of enabling communication between the set-top box and other electronic devices in the home: the "usage scenarios" for that capability that would improve the experience for the end user. One obvious and important purpose for such communication is the transfer of actual media assets between devices, but because ensuring that the transferred media can be transcoded to a format supported by the target device is complex, because real-world implementations would probably need a comprehensive Digital Rights Management system, and because this is an area that is already well addressed by existing technologies, in particular by DLNA (see section 3.1), we have not addressed this usage scenario in our work.

2.1 Improving the Accessibility of Television

The user interfaces (UIs) that are built-in to televisions and set-top boxes are, conventionally, based on infra-red remote controls and the overlaying of visual user interface elements (such as menus and Electronic Programme Guides (EPGs)) onto the television screen. This system is rarely optimal for the needs of people with physical and cognitive impairments, and presents particular issues for people who are blind or visually impaired. They are at a disadvantage when it comes to finding a dedicated remote control device in the first place, and the built-in menu systems of televisions and STBs typically only provides visual feedback in response to user actions (with some obvious exceptions, such as changing the volume). A far better solution would be a remote control software

application running on the mobile phone that many blind people already carry with them at all times. Running on an RNIB[1]-recommended smartphone, for example, it could present an optimised UI using speech synthesis to convey information to the user.

Other users may have more specialist requirements. For example, users with severe motor impairments may be limited to interaction via just a few buttons, or a single switch. The ability to customise the interface for efficient use by such a limited input mechanism is very important. Users with cognitive impairments may benefit from highly simplified UIs—again, the ability to customise the interface is key.

An important aspect to the practicality of developing such specialist UIs would be their reusability. Ideally, a single client application could be written and installed on a single device owned by the user (such as their mobile phone), and then used to control multiple home media devices using the same user interface.

Although the example used in this section is that of television, the UIs built-in to many other home media devices are often similar in that they present information to the user exclusively visually, and/or have dedicated "remote control" hardware that may be difficult to operate or locate.

2.2 Creating New Kinds of Remote Control

Users without specific accessibility needs can also benefit from being able to control boxes from their mobile phones, tablets and other devices. A particular benefit is that the television screen need not be taken over by user interface elements when a user is interacting with the box: for example, one person can be scheduling the recording of a programme while another is watching television, but users may also benefit from the ability to control multiple devices from a single UI and, depending on the nature of the client device, the increased expressiveness afforded by touchscreen or pointer inputs.

Putting the user interface for a home media device onto the mobile phones of each person in the home also means that those interfaces can adapt themselves to reflect the preferences of each user, delivering personalised programme recommendations, for example, or presenting that user's favourite broadcast services to them first. In some cases, remote user interfaces could incorporate additional information or media from an Internet source (if available), or be incorporated into appropriate applications whose primary purpose is not remote control. (For example, a smartphone client for a video on demand service could be given the ability to control televisions and STBs; if the same video on-demand (VOD) service is available on both the smartphone and the second device, then media playback can be synchronised on the two devices, or "transferred" from one to the other without requiring a direct stream of media between the two.)

As well as software remote control applications, new kinds of hardware remote control are also of interest. For example, an intentionally simplified hardware remote control could provide only the functions that the user is interested in, expressed in terms of the tasks the user wishes to achieve, not a set of individual actions that they must ask the box to perform in the correct sequence to achieve that task. For example, a child using a dedicated hardware device (perhaps incorporated into a toy) could press a button representing a particular children's television show to view the latest episode, irrespective of the channel, on-demand service, or recording it is being sourced from. (Similar devices could benefit those with cognitive impairments, including those associated with age.)

2.3 Enabling Multi-Device "Orchestrated Media" Experiences

If external devices can obtain information about the state of the box, remote control is just one of a number of new applications that could benefit users:

[1]The Royal National Institute for the Blind (RNIB) is a UK charity for the blind and visually impaired.

- Broadcaster and third-party websites could be enhanced with awareness of what the user is currently viewing and what services and recordings they have access to; providing content customised to their current experience, and supporting control of the programme currently being viewed and the scheduling of recordings on a Personal Video Recorder (PVR) device. Users could "tag" programmes they were watching, whether from a live broadcast or a recording or on-demand service, to engage in a social media experience relating to them, with specific programme segments identified for discussion and on-demand playback.

- It would be possible to synchronise to the playback of content on the box with the display of related information or applications on mobile phones, laptops and other devices.

- Audience measurement could be automated by an appliance simply connected anywhere on the home network that monitors the content displayed on the box.

- Multiple boxes in a house could be synchronised, letting the viewer wander from room to room without missing any of a programme—perhaps even routing the soundtrack of a TV programme (with audio description mixed in) to the kitchen radio. A programme could be "transferred" mid-playback to a mobile phone, and taken with the viewer when they leave the room. An application running on the box could turn the television into a second screen for the mobile phone.

- Alternative audio for a television programme could be sent to one viewers mobile phone, to be played back synchronised with the television, on headphones. The alternative audio could be a different mix of the main programme audio with quieter music and sound effects for improved intelligibility, or it could be a director's commentary track.

- Interactive applications and services, previously confined to a single device, could extend to other devices. This could reduce on-screen clutter on a television, for example, and allow several viewers watching the same programme in the same room to take part in an interactive experience together. A quiz show, for example, could allow all family members to play along using their personal devices; with their scores collated on the television screen and compared to that of the studio contestants.

Due to the many ways in which existing terms like "second screen" are applied to different usage scenarios, we adopt the term "orchestrated media" in this document to describe these multi-device experiences. A number of the scenarios listed in this section were discussed with representatives from the BBC's FM&T TV Platforms, FM&T Mobile and R&D Prototyping teams, and their contributions to the development of the API are gratefully acknowledged.

3 Alternatives

3.1 UPnP and DLNA

When considering the above usage scenarios, certain existing technologies are obvious candidates for consideration. Universal Plug and Play (UPnP) and its its profiles and supplementary standards defined by the Digital Living Network Alliance (DLNA) are standards for communication between network-connected home media devices that in some cases date back to the 1990s, and have been implemented (and to varying degrees, deployed) on a wide range of platforms since then. At the time of writing this paper, DLNA compliance was a common feature of televisions, home media servers and games consoles, and even some models of mobile phone.

UPnP defines much but not all of the functionality required for the "remote control" subset of the usage scenarios set out above. One notable omission includes support for selecting alternative media components, such as subtitles or alternative audio tracks. No explicit mechanism is defined for accessing device-specific features: although the UPnP Remote UI standard defines a way for

boxes to render their own user interfaces on remote clients[2], which could of course be used to control any device feature, there is no means to ensure that said box-generated UIs are simple, accessible, customisable or personalised.

DLNA profiles and subsets UPnP considerably, making interoperable implementations more straightforward by defining (for example) required and optional media codecs and profiles, and media transport mechanisms. In particular, DLNA only facilitates streaming *between* devices: it lacks the capability to route media internally. It also lacks the ability to schedule recordings.

Neither UPnP nor DLNA support the control or communication with interactive applications that would be required to fulfil the final usage scenario.

In addition, because UPnP and DLNA are based on underlying protocols other than standard HTTP, they are not implementable on platforms which only support network communication via that protocol, notably J2ME (in certain older but still common versions), web widgets, and web browsers. The latter in particular was considered vital as a technology with which to implement affordable client platforms in all the usage scenarios.

Primarily for the above reasons, we did not consider UPnP or DLNA to be appropriate technologies for addressing the usage scenarios set out above.

3.2 Communicating via the Internet

In many cases, both the box and the client device will be connected to the Internet. Particularly given the decision (see section 5.3) to use the home network as the intended physical layer for UC communications, it will often be the case that communication between clients and servers could just as easily take place via the mediation of an Internet service. We assume here that such a scheme would still be implemented in a standardised fashion by different box manufacturers, to gain the benefits of interoperable client devices. The advantages of such a scheme are as follows:

- Communication need not be limited to the home network; there are genuine (but in our opinion, secondary) usage scenarios in which being able to communicate with a home media device from any physical location with an Internet connection would be advantageous, such as setting a PVR to record programmes that will be broadcast before you are next at home[3]. The possibility of defining an Internet proxy service for conveying local network API interactions between the implementing box and client devices elsewhere on the Internet is considered by the authors to be a good way to address these additional scenarios, but is not currently specified.

- The few remaining situations where non-HTTP application-layer protocols are required by the API described in this document (discovering the IP addresses of servers on the network, and reacting to a change to a known server's IP address—see section 6.2) would be eliminated.

- It would be possible to use HTTPS as a way to protect sensitive information in transit.

On the other hand, the scheme has a number of disadvantages:

- It implies that if a user loses connection to the Internet, they will no longer be able to control their home media devices using remote UIs of the kind described above. This is a particular issue if the remote UIs have been developed for people with accessibility requirements, since such people may have no other control method available to them.

- Somebody has to fund the development and long-term maintenance of an Internet service, which may or may not have a revenue stream associated with it.

[2]For example, the box could serve a set of web pages that contain user interface elements that control the box. This is very different from the concept of serving a web *API* from the box, such as the Universal Control API described in this document.

[3]A usage scenario that we expect to diminish in importance as the availability of video on-demand (VOD) services via PVRs, STBs and other home media devices grows

- Users and interactive application providers have no guarantees that they will continue to be able to interact with a box in the ways described above if the provider of the Internet service goes out of business, or decides for other reasons (eg the box manufacturer going out of business, if the two entities are not the same) to stop providing the service. This is again a particular issue for people with accessibility requirements.

- The additional jitter added by congestion and similar non-deterministic network behaviour in the many additional "hops" that communications between devices must traverse degrades the ability of devices to synchronise media accurately.

- A local network API carries with it a reasonable assumption that the devices are in close proximity, which can be used to avoid presenting options to the user that may be inappropriate, such as an ability to transfer playback of a piece of video from a mobile media device to a television. A local network API also facilitates the triggering of other

- The inherent (but limited) security benefits associated with restricting communications to the home network are lost.

Overall we felt that a local network API was strongly implied for enabling the (accessible) remote user interface usage scenarios, and had strong advantages for many of the multi-device experience scenarios, particularly those which involve more than two devices. An exception is the kind of "multi-device" experiences of the kind described in the last usage scenario above, which may require Internet services for other reasons, such as the provision of application-specific media or interactive elements, or as a "back-channel" to return data such as high scores to the application provider. They may also be required to operate on a set of platforms that includes home media devices both with and without a local network API such as the one defined in this document. The decision as to whether or not to ignore any local network APIs that may be available and channel all inter-device communication via an Internet service must be made on a case-to-case basis.

4 An Overview of the API

What follows here is a very high-level overview of the API and its main features and functionality. The UC API specification [1] should be consulted to obtain a more detailed understanding. Section 5 below explains some of the decisions made during the design of the API.

A box implementing the API is expected to be connected to a home IP network and incorporate a UC server component that is a HTTP/1.1 compliant server on said network. Clients query and control the functions of the box by making HTTP requests to the UC server. As Fig. 1 indicates, the UC server can be considered an intermediary that maps queries and requests from the client to internal state and functions of the box.

Remote control functionality does not rely on internet connectivity because communication is direct between client and server. Of course, some clients or the functions of some boxes may require internet connectivity; but this will be a consequence of their own functional requirements, and not of the API.

Servers implement the W3C Cross-Origin Resource Sharing (CORS) standard [2], which allows Javascript running in compliant browsers to connect to "cross-origin" resources (ie resources from other domains) that explicitly permit such access. A server will also implement a "/crossdomain.xml" resource to enable cross-domain requests from Adobe Flash content.

4.1 Data Model

The API models the box in terms of "source lists", "sources", "content", "media-components" and "outputs". It also includes a concept of content "categories" and "acquisitions".

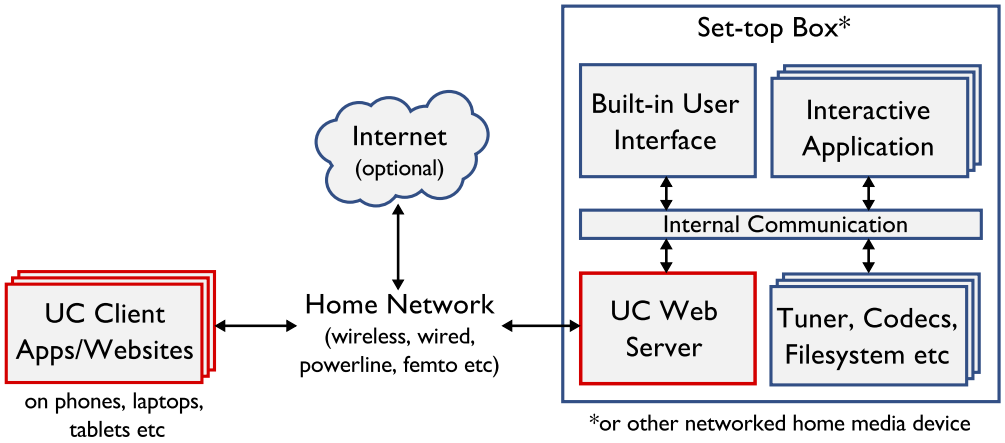

Figure 1: An illustration of the relationship between Universal Control API client, server and the box to be controlled.

Each output represents an individually-controllable means by which the box can render media: the main screen connected to it, for example, or a picture-in-picture capability. A box that is an STB with both SCART and HDMI sockets would probably model these as a single output, assuming that they always show the same thing.

Each source represents a source of media known to the box, such as a DVB or VOD service. In the context of the API, a source is a named entity that has one or more pieces of content associated with it. A source also has various properties. Some of these are descriptive properties, such as a human-readable name and a logo image. Others convey hints to the client about how to deal with the source. For example, a source can be defined to have a "live" property, which lets the client know that the source has a concept of "current position" that exists whether or not the source is being presented on an output. This information allows clients to manage user expectations regarding what will be presented on screen when such a source is selected: for example, a client can make it obvious that playback will not start at the start of a programme. A source may also have a "linear" property, which lets the client know that all the programmes in the source have both a start and an end time, and that no two programmes overlap in time. This property helps the client determine whether or not a "timeline" view of the source's programmes should be constructed, for example as part of a conventional "grid" electronic programme guide. Finally, a source may have a "follow-on" property, which tells clients that when one programme from a source ends, presentation of another programme from the same source will start immediately, without interrupting the flow of media to any output(s) presenting media from that source. This property gives clients information about the appropriateness of including programmes from that source in a "playlist", or similar entity, and lets them know that they must take explicit action at the end of the programme if they wish to prevent the "follow-on" behaviour.

Sources may be grouped into "source lists" that are meaningful to the user, such as "tv", "radio", or "on-demand services".

For a given "source" there will be many items of "content". For example, a television channel or VOD service will be host to many television shows or movies. Each item of content has associated metadata, including basic information such as title, synopsis and duration. Content may have several associated identifiers. Those that are local in scope are used within the context of the API implementation to refer to that item of content, or the series to which it belongs. Identifiers with global meaning can also be included. This enables clients to, for example, relate items of content to data available from other Internet based services.

It is notable that the term "content" here covers both "AV" content, such as traditional linear television, music, etc and "interactive" content such as web applications or MHEG content.

An item of content will have several "media-components" associated with it. Typical compo-

nents for “AV” content might include the main audio and video, as well as subtitles and alternative audio tracks.

A hierarchy of “categories” provides groupings of content with meanings relevant to the user of the client, such as programme formats or genres, or perhaps lists of favourites or newly available material.

“Acquisitions” model the ability of some boxes to take requests from users to obtain an item of content at a later point in time when it becomes available. For example, Personal Video Recorders (PVRs) include the ability to “book” a recording of a programme that is due to be broadcast in the near future. This can also represent other forms of acquisition behaviour, such as downloads or subscriptions to “feeds” from internet based services. An “acquisition” represent a given booking, download or subscription.

4.2 API Resources

The UC server implements the API as a RESTful web service. Specific HTTP resources map to logical aspects of the box’s state. Fig. 2 is an overview of the resources defined in the API.

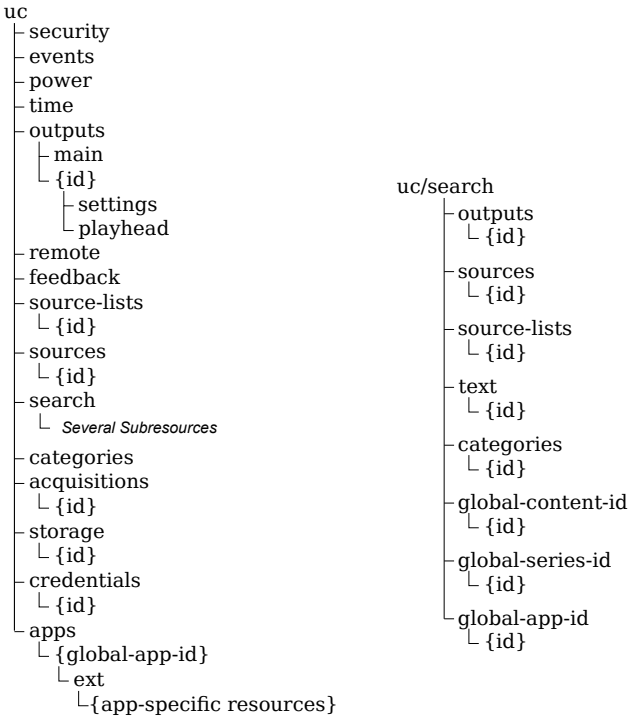

Figure 2: A tree diagram showing the resource hierarchy of the Universal Control API. Read down the tree from left to right to construct a valid URL path segment, eg “uc/outputs/main”.

Boxes may vary substantially in terms of the functionality they provide. For example, a box may, or may not have personal video recording capabilities. The API therefore adopts a modular design. At minimum, all boxes implement the ”uc” root resource and ”uc/security” resource. Other resources are implemented, as required, only if the box has corresponding functionality. The ”uc” resource enumerates the resources that a server implements, enabling clients to determine the available functionality.

4.3 Basic Control of Viewing

Using the “uc/power” resource, a client can check the power status of a box and command it to switch to an “on” or “standby” state.

The "uc/outputs" resource features "uc/outputs/{id}" sub-resources corresponding to each output. Selecting presentation of a particular piece of content is a matter of sending the box a new state for the appropriate resource, containing the IDs for that content and its source in the appropriate attributes. When receiving a "state-changing" request (indicated by an HTTP verb other than "GET"), the server is expected to make whatever changes are required to the underlying state of the box to make it match the state given in the client request. Clients can also make requests to this resource in order to enable or disable particular media-components, such as subtitles or alternative audio.

The "uc/outputs/{id}/settings" sub-resource provides the means by which clients can request changes to volume; and the "uc/outputs/{id}/playhead" sub-resource represents the playback position of the content currently being presented on the output. State changing requests can be used to seek to a different time index.

4.4 Time and Synchronisation

A basic mechanism has been defined to allow clients to synchronise a local clock to that of the box. A client can query the "uc/time" resource to achieve this. This, combined with the "playhead" resource already described, are sufficient in theory to permit clients to synchronise playback of media sourced from elsewhere with that of the box.

The accuracy with which clients and servers can synchronise playback using this mechanism has not yet been determined, and this will have an impact on the kinds of media that can be synchronised: director's commentary and web pages providing background information can tolerate perhaps a few seconds of misalignment, whereas the tolerance for audio that must be lip-synced to video is considerably tighter – around 10ms [3].

4.5 Source and Content Metadata

Clients must be able to determine what sources of content exist. Several resources play a roles in this. The "uc/source-lists" resource enumerates the names and ids of the available source-lists (groupings of sources). The sources in an individual list, identified by the list's id, can be retrieved from the "uc/souce-lists/id" sub-resources. Information on a specific individual source, identified by its "id", can be retrieved from the "uc/sources/id" resource.

Clients also need to discover what available content is available. The collection of sub-resources under "uc/search" resource make this possible. They facilitate search by output, source, source-list, category, identifier or text in the title and/or synopsis. Search requests can optionally include filtering arguments to filter result. This allows, for example, a client to specify that it only wishes to search of content items that are due to be available within the next 24 hours.

4.6 Acquiring and Managing Content

Many devices on which the UC API could be implemented now incorporate the ability to record, or otherwise acquire, content. The "uc/acquisitions" resource represents all requested acquisitions. A "state-changing" request can be used by the client to submit new acquisitions or update (through replacement) existing acquisitions.

The "uc/storage" resource is intended to provide "storage management" functions for content stored on any limited-capacity storage media that is accessible to the box. For example, if the box has an internal hard-drive which is used to store recorded programmes or downloads from the Internet then content stored on that hard-drive could be listed here. Each "stored-content" item held within this resource represents each locally stored piece of content. The primary storage management function is the ability to delete unwanted items, thereby freeing up storage space.

4.7 Notification

Developers are likely to want to be able to make UC client applications respond to changes in the server state in near-realtime. This requires the box to be able to notify the client of changes in its state on that basis.

The "uc/events" resource provides an "HTTP long-polling" based mechanism by which clients can be notified, in a timely fashion, when changes occur to the state of the box. A client can then re-query the affected resources to obtain new state information.

4.8 Discovery

A vital component of a remote control API that enables good user experiences is that of discovery: the ability of clients to identify servers that they can connect to with little or no user interaction. In order to make a successful connection to a UC server over an IP network, a client must know that server's domain name or IP address, and the TCP port on which the server is listening for connections. Sometimes it may be desirable to provide the client with additional information—authentication details, for example. We will refer to all this information collectively herein as "the discovery information".

A server implementing the API therefore provides two means of discovery. A UC server should implement the Bonjour protocol, which is built upon multicast DNS, to enable clients to discover servers in a fully automated fashion. In addition, the server should also implement manual discovery by means of a UC "pairing code", to be displayed on the box built-in user interface, and to be entered into the client by the user. The manual "pairing code" is designed to carry full discovery information, and is designed to be shorter for IP address ranges that are commonly used in domestic networks. The automatic discovery mechanism conveys only the IP address and TCP port.

4.9 Security

A UC server provides a means of remotely accessing and controlling a device in a person's home. In the future, STBs and similar devices are likely to become increasingly powerful and continue to gain additional functionality, and are likely to have more and more data stored upon them which might be considered private. Even for today's devices, it is likely that most users would like to be able to restrict the ability to access or control the box to a subset of the devices on the home network. Authentication in the context of Universal Control is intended to make sure that merely being connected to the same network as the box does not automatically give a client the ability to query and control it.

The API defines a security mechanism that servers can implement. It enables access to the API, or parts of the API, to be restricted to clients that have been explicitly authorised by the user. A "client" is defined to be an installation of a particular piece of client software on a particular hardware device. The discovery information conveyed in a "pairing code", described in section 4.8, can include an authentication credential that is to be used by the client to acquire credentials which can be used when formulating future requests.

The server may require that requests made to a particular API resource be authenticated. A server may respond to an unauthenticated request with an authentication challenge. The client must then resubmit the request, including the required authentication headers. This method of authentication incorporates a message digest hash and uses nonce values, providing a degree of assurance that the request originated from an authorised client and reducing the risk of replay attacks.

In addition individual requests can be made "restricted". This means that when the request is made a challenge will be issued, which will specify either that the request needs to be "confirmed" by resubmitting the request with a supplied nonce, or that the request needs to be "authorised" by repeating the request with a message digest calculated using a short PIN known to the user

and the server, but not communicated between client and server. Such a mechanism allows certain content to be restricted so that only users with access to a specific PIN may request it.

4.10 The API Extension Mechanism

To address the last scenarios listed in section 2.3, Universal Control servers can implement an internal mechanism by which interactive applications running on the box can expose their own web API as one or more sub-resources of the "uc/apps/{id}/ext' resource, using the UC server as an HTTP server. Note that while the advantages of RESTfulness set out in this document apply just as readily to these API extensions as they do to the UC API as a whole, these extensions do not have to be RESTful. The interactive application could just as easily implement an RPC-style interface by ignoring the HTTP verb and headers, and the path component and query string of the URL addressed by the client, and simply exchanging XML messages with the client.

The API specification does not specify how the box handles communication between the HTTP server and the interactive application, since this will be highly implementation-dependent. Both RPC and RESTful APIs are straightforward to implement on a box that handles communication between internal components using a message-passing paradigm, however. The HTTP verb, URI path, query string, headers and message body from the client can all be encapsulated in a standard format and sent as an internal message to the interactive application. If synchronous internal communication is supported, an HTTP response code can be returned immediately by the interactive app. Initiating communication from server to client must take place using the UC API's notification mechanism, however: a message must be sent internally from the interactive app to the HTTP server containing the URI path of a resource that the client can then request via an HTTP GET to receive the information from the server. This restriction is a consequence of the underlying RESTful nature of the UC API, and of the kinds of client device for which it has been designed.

4.11 Application Lifecycle Management

For the best possible user experience, it is desirable for the user to be able to specify their intention to start a multi-screen experience on just one of the devices that will implement it. If the server implements the notification mechanism described in section 4.7, devices running UC client software can get near-realtime notifications when the box starts presenting a linear programme to the user, or an interactive application becomes active on the box or starts extending the UC API. This allows client applications that are already running to take the actions necessary to join the experience immediately.

In addition to giving clients the information they need to know when API-extending apps are running, the "uc/apps" resource gives client applications substantial control over the lifecycle of interactive applications on the box, if the server permits it. Mechanisms for starting and stopping interactive applications are defined. Mechanisms for requesting the installation and uninstallation of particular applications are provided via the "uc/acquisitions" and "uc/storage" resources respectfully.

4.12 Accessibility for Conventional Interactive TV Applications

The "uc/remote" and "uc/feedback" resources are anomalous in the UC API. They do not facilitate clients in creating a remote user interface; rather, they give a client control over the box's built-in user interface. The `remote` resource is write-only, and allows clients to simulate button-presses on a box's infra-red remote control, or similar button-based input device. Unfortunately, this is a necessity if conventional interactive TV applications[4] are to be controlled via the UC API, since their user interface is inherently bound to the television or STB's IR remote and to the television

[4] "Red button" services, such as MHEG or MHP applications.

screen. The `feedback` resource is read-only, and provides a way for the box to send arbitrary text back to the client. The intention behind this resource is that, combined with the `remote` resource, interactive applications and the box's built-in interface can be made accessible to the visually impaired. The interactive application or built-in interface can describe its state in response to button presses in the form of text which is then read out to the user by the client using speech synthesis, or displayed to the user in large, high-contrast letters.

It should be noted that existing interactive television standards such as MHEG do not offer support for the generation of descriptive text of this kind.

5 The API Design Process

Our intention was to design an API that was easy to implement on a wide range of server and client platforms, simple to extend to support the functionality of new kinds of device, and capable enough to give users control over all the important functionality of the box. These considerations led to a decision to base the API on HTTP and XML: well-established, robust technologies, with good support on the key platforms that we chose and investigated. These standards are also well understood by developers, and supported with a range of good development tools. In this section, we present the investigations (carried out in 2009) and decisions that led to these conclusions, starting with our design principles (section 5.1) and the key platforms that were investigated for available technologies and tool support, both client (section 5.2.1) and server (5.2.2).

Various options for the API's underlying communication mechanism were considered before the IP-based home network was selected: the reasoning behind this choice is presented in section 5.3, along with some comments regarding the compatibility of the API with Bluetooth. Having selected Web technologies as the best fit to our key platforms (5.5), we had to consider whether clients should connect directly to the box or go via a web service on the Internet. Issues associated with access to web servers on the local network from web pages and widgets fetched from the Internet were also considered at this point.

Another design decision centred on the choice of an RPC-style or RESTful style of HTTP Web service requests—ultimately, we chose the latter. A brief introduction to these styles and their pros and cons in the context of the UC API is presented in section 5.6. A discussion of various options for the choice of payload format carried in the HTTP (XML, binary XML, JSON or a custom format) are given in section 5.7. We settled on XML, for the reasons given therein.

5.1 Design Principles

To help constrain the API design process, the following guiding principles were adopted:

1. Set-top box manufacturers won't implement anything that looks too complicated or too costly for a device's bill of materials. The API should not include functions that are not required by any compelling use case. It should minimise the number of libraries required to implement a compliant server, and should not rely on any technologies without mature support on most or all key server platforms.

2. Client developers should *want* to implement the API. The interface must be self-evidently simple and elegant. It should not rely on any technologies without mature support on most or all key client platforms.

3. Simple functionality should be simple to implement. If we want to define interfaces that do complicated things, they should be optional (for both clients and servers), and the complexity should be restricted to that part of the API.

4. We should make as few assumptions as possible about the nature of the server and client. We don't know what the set-top boxes of the future will look like. We don't know all the kinds of

device people will want to build client applications on today. A well-designed protocol could be used in all sorts of circumstances we can't imagine now. For example, what if the mobile phone you use to access audio description for TV programmes in your living room worked the same way in the cinema?

5. We should make as few assumptions as possible about the *developers* of servers and clients. We want any developer to be able to write implementations for any platforms under their control. Although the requirements of set-top box manufacturers are vital considerations, for example, we should also allow for server implementations in open source PVR software such as MythTV. Similarly, client developers could be mobile handset manufacturers, disability rights groups, small businesses, or enthusiasts.

5.2 Key Platforms

To further constrain the API design process, we decided to identify key client and server platforms. These were chosen because we and relevant experts within the BBC thought that they were the most likely platforms on which people would want to implement UC clients and servers. As implied in the guiding principles set out above, the implication of a platform being designated as "key" was that we made considerable efforts throughout the design process not to base the API on technologies that that platform does not support.

5.2.1 Client Platforms

The key client platforms we identified were as follows. In parentheses are details of the programming language, library and API options that we considered when evaluating the capabilities of each platform.

- Apple iPhone OS. (Applications developed in Objective-C using the iPhone SDK subset of the Cocoa libraries.)
- Nokia S60. (Applications developed in C++ using the Symbian libraries or HTML and Javascript using the Widget Runtime (WRT) APIs.)
- Google Android. (Applications developed in Java, using Android-specific APIs and libraries.)
- Palm (now HP) WebOS[5] (Applications developed in HTML and Javascript, using WebOS-specific APIs.)
- Java 2 Mobile Edition (J2ME). (Applications developed in Java, using the Connected Limited Device Configuration (CLDC) APIs and libraries.)
- Ubuntu Linux (or any popular modern desktop Linux operating system). (Applications developed in Java using the Java 2 Standard Edition (J2SE) class libraries and any other libraries available to the developer, C/C++ using the GNU libc, STL and any other libraries available to the developer, or Python 2.6 using any libraries available to the developer.)
- Windows XP. (Applications developed in Java using the J2SE class libraries and any other libraries available to the developer, C/C++ using the MFC libraries, the .NET libraries and any other libraries available to the developer, or Python 2.6 using any libraries available to the developer.)
- MacOS X 10.5. (Applications developed in Objective-C using the Cocoa libraries and any other libraries available to the developer, or Python using any libraries available to the developer.)

[5] At the time when the technologies to be used for the API were under consideration, very little information about the capabilities of WebOS was available, and hence it did not end up significantly constraining the design process.

- Desktop and mobile web browsers (Applications developed in HTML and Javascript, using W3C-standardised APIs.)
- Desktop and mobile web widgets (Applications developed in HTML and Javascript, using proprietary or W3C/OMTP-standardised APIs.)

The wide variation in the capabilities of these platforms ultimately led to the selection of HTTP and XML as the basis for the UC API, a decision detailed in sections 5.5 onwards.

A key omission from this initial list was Adobe Flash; at the time of writing it is believed that the API is fully compatible with that client platform, and it is planned to maintain that compatibility in future changes to the API.

5.2.2 Server Platforms

The primary intended server platform for Universal Control is an IP-enabled STB. Such a device is increasingly likely to be running a multi-tasking operating such as Linux, but to be significantly resource-constrained compared to a modern desktop PC, with perhaps 256MB of RAM rather than several gigabytes. Ideally, UC server implementations would be possible on even more limited devices, such as conventional STBs with real-time operating systems, 32MB of RAM (or less) and performance in the sub-500DMIPS range. Design decisions throughout the API development have been taken with such server platforms in mind, but at the time of writing, no implementations on such devices have been written.

In addition to set-top boxes, it was considered desirable to be able to implement UC servers on standard PC platforms (Windows, Mac and Linux)—both for the purposes of testing, and because a number of homes have "media centre PCs" running standard PC operating systems.

In terms of operating systems, programming languages, libraries and APIs, those assumed for the project's key server platforms are virtually a subset of those already defined for client platforms:

- Linux. (Applications developed in Java using the Java 2 Standard Edition (J2SE) class libraries and any other libraries available to the developer, C/C++ using GNU libc, STL and any other libraries available to the developer, or Python 2.6 using any libraries available to the developer.)
- Embedded Linux. (Applications developed in C using an embedded libc such as uClibc and any other libraries available to the developer.)
- Windows XP. (Applications developed in Java using the J2SE class libraries and any other libraries available to the developer, C/C++ using the MFC and .NET libraries and any other libraries available to the developer, or Python 2.6 using any libraries available to the developer.)
- MacOS X 10.5. (Applications developed in Objective-C using the Cocoa libraries and any other libraries available to the developer, or Python using any libraries availanle to the developer.)

5.3 Communications Technologies

It was assumed from the outset that use of an RF physical layer standard for communication between devices would be necessary, in order to give users of PC and mobile phone clients freedom from finding and plugging in cables, and from needing the direct line of sight required for reliable infra-red communications. Given the capabilities of the key client platforms listed in section 5.2.1, the obvious options for the physical layer were WiFi and Bluetooth. All the platforms support one of these two protocols, and many support both. Femtocells could in theory be used to give phones without WiFi access to the home network, but this possibility was not explored as part of the API design process.

Considering the pros and cons of the two options identified, firstly it is the case that the Apple iPhone only supports a very limited subset of the Bluetooth protocol, and Bluetooth support is far from universal on PCs. It may also be desirable for devices to be able to communicate with the set-top box when not within Bluetooth range (around 10m for most implementations, often less where the signal has to pass through walls, or the 2.4GHz spectrum is congested). On the other hand, many phones (mostly feature phones, at the lower end of the market) support Bluetooth but not WiFi. WiFi adoption in mobile devices is becoming more common however, and the case can be made that there is a correlation between ownership of a WiFi-enabled phone and likely interest in the use of a mobile phone as a remote interface device.

Adopting both protocols on an equal footing is undesirable: even small additional costs are a big deal in the low-margin world of consumer electronics, and it is anticipated that a very strong case would need to be made to persuade box manufacturers to absorb the costs of doing so. The additional development effort would be non-trivial, and the testing regime for implementations would be more complicated. It is becoming the norm for set-top boxes and televisions to be connected to home networks already, to give them access to IPTV and web services. That being the case, it is likely that for many boxes, only software changes would be required to implement an API that clients could access via a home WiFi access point (regardless of whether the box's connection to the home network is itself wired or wireless). Bluetooth, on the other hand, would require additional hardware in the box (adding to its bill of materials), or for users to buy a compatible plug-in Bluetooth device before they could control each device.

For these reasons, work on the API proceeded on the assumption that communication would take place via the home network, albeit without making assumptions , but the possibility of a version adapted to be carried over Bluetooth has not been ruled out.

One possible way to implement the API over a Bluetooth connection would be to use the "ObEx" protocol, which is similar to HTTP in several ways. A cursory investigation by the authors suggested that a direct mapping between an HTTP-based interface and an ObEx-based one should be quite simple (with HTTP GET requests mapped to ObEx GET instructions, and PUT and POST requests to PUT instructions in ObEx). ObEx also natively supports a "path" structure for referring to resources, which makes it very convenient for a RESTful style interface. Although the authors have not investigated the option in any detail, they consider it likely that a fairly straightforward mapping of most or all of the UC API could be made to the Bluetooth physical/datalink layer by taking advantage of these similarities.

5.4 Internet Layer

Having decided that the API would be built on Ethernet technologies, the use of IP networking was assumed. However there are currently two major versions of the Internet Protocol in use: version 4 [4], the first version to be widely adopted and the foundation of the current Internet, and version 6 [5], which is intended to replace it. IPv4 is likely to remain in widespread use for many years to come: realistically, even when IPv6 has become a mainstream protocol on the Internet (which was far from true at the time or writing), IPv4 is likely to remain in use on many local area networks, including most home networks. For this reason we concluded that designing the API for use on IPv4 networks needed to be the priority for an initial release of the API, reserving the option of extending the specification for use on IPv6 networks for a later release.

In practice most of the technologies used in the final design of the API work equally well on both IPv4 and IPv6. Indeed, from the perspective of clients and servers communicating using application layer protocols such as HTTP, the question of which version of the Internet Protocol is in use on the network never arises. The only place in the API specification in which the difference between the two protocols matters is in the discovery of servers by clients. This issue is discussed further in section 6.2).

5.5 Other Layers

A comprehensive review of various higher-level protocols and technologies on which to consider basing the API was then undertaken, the results of which are detailed in Appendix B. The purpose of this review was to determine the extent to which the various protocols, data formatting standards and pre-existing APIs were supported on the various predetermined key platforms. We were interested particularly in the availability of libraries for the technologies in question, rather than pre-existing client software, since good library support is a prerequisite for the kind of simple developer experience we were aiming for.

Where application-layer protocols were concerned, the deciding factor was the fact that not all the key platforms provide access to a full TCP/IP stack. J2ME (in its most limited, most widely-supported profile), web browsers and widget runtime engines are particularly restrictive, guaranteeing no network access other than a unicast HTTP client API. Browser- and widget-based implementations of a UC client application would face additional restrictions. There are three main ways in which an HTML/Javascript user agent (such as a web browser or widget runtime engine) could obtain an HTML/Javascript implementation of a Universal Control client: from an HTTP server running on the box, from an HTTP server on the Internet or as a pre-packaged "widget". Distinguishing between these three types of source is important because historically, user agents have applied the "same origin policy" when validating `XmlHttpRequest()` calls from client Javascript—the means by which browser- or widget-based UC clients would connect to UC servers. This prevents such applications from accessing web resources other than from the domain from which the Javascript itself was obtained. This is an issue for Universal Control clients downloaded from the Internet and run in a web browser, as it will prevent them from accessing the set-top box.

The W3C Cross Original Resource Sharing (CORS) standard [2] was selected as a solution to this. Although CORS is currently only at the public draft stage of standardisation, CORS is supported in recent versions of Firefox (v3.5+), Safari (v4+) and Chrome (v2+). Although Internet Explorer 8 implements a limited subset of the CORS standard, this subset would not be sufficient to implement a general purpose client for a local network web API. Implementation is less widespread on mobile browsers. CORS is supported in iPhone OS 3.0 and Android 2.1, but the Android 1.6, Symbian S60v5, Opera Mobile 9.80 and Opera Mini 4.2 browsers were tested and found not to support it.

Widgets present a different environment, with different challenges. The W3C (and other bodies) are in the process of standardising various aspects of widgets, and both OMTP's BONDI specification and W3C WARP define ways for widgets to declare static dependencies on network resources: lists of URIs that the widget needs access to. The advantage of this approach is that it allows operators and other handset providers to approve these lists before they allow a widget to be used on the phones they supply, and it allows users to know exactly what the widget will access before they even run it. The disadvantage is that there is no way of specifying the home network (or devices thereon) as a network resource on which the widget depends. It is however possible to write a widget that requests permission to access any network resource, including local network resources, although it is not clear that such widgets would be approved for use on all handsets, because of the security implications of this approach. Efforts are being made to change the standards in question to address this issue. At the present time, no mobile phones ship with support for any standards-compliant widgets, but a number of proprietary widget platforms are available. Nokia's Web Runtime (WRT) widgets, available on Symbian S60v3 FP1 devices and newer, place no restrictions on the hosts that a widget can connect to (but require widgets to declare a static dependency on network access). Windows Mobile 6.5 widgets behave similarly. Opera widgets have a more complicated security model, but not one that would preclude local network access.

The possibility of "workaround" solutions also exists: An HTML/Javascript implementation of the Universal Control API could be written within the constraints of the same-origin policy using a proxy running either on the set-top box or as a native application on the client device. The latter option would require the client device to be a PC or smartphone, since obviously the proxying

application must run at the same time as the widget. Neither option is elegant.

In addition to the same-origin security policy, RFC 2616 [6] restricts compliant HTTP clients to opening a maximum of two simultaneous connections to each origin. This restriction is enforced in most modern web browsers. The implication of this is that HTML/Javascript clients must take steps to ensure that the user experience is not harmed when resources take a prolonged period of time to return data.

To summarise a complicated set of restrictions, clients on the above platforms would be restricted as follows:

- J2ME clients would not face any explicit restrictions.
- Users of most older desktop web browsers and many current mobile browsers will only be able to access the server via web pages served from it, via it (acting as a proxy) or via proxy software written as a native application for their mobile phone.
- Users of contemporary widget platforms will not experience difficulties connecting to servers; users of future standardised widget platforms might effectively be prevented by their device providers from using software that can access local network resources. This is far from certain, however.
- An API would have to be based entirely on HTTP for the above platforms to be guaranteed access to it, and be written such that clients need not open more than two simultaneous connections to the server.

For these reasons, because of the importance of these client platforms, the API was built on HTTP.

5.6 REST vs RPC

There are at present two major competing philosophies for the design of HTTP-based APIs (better known as "web APIs", or "web services"), known as "REST" and "RPC". For readers without previous knowledge of web technologies, a brief introduction and comparison of the two can be found on Wikipedia [7]. The more established of the two is the "Remote Procedure Call" (RPC) approach, in which aspects of the state of the server (e.g. the channel to which a box is tuned, or the volume at which it is playing audio) are retrieved and altered by calling "functions", generally with atomic functionality. So for example, in an RPC-based UC API there might be a function to return the channel to which the box is tuned, and another to change it, along with many other functions to retrieve or modify other aspects the box's state. Often, the whole API can be accessed via a single resource on the web server (perhaps "`http://stb.example/api`", for example).By contrast, in a RESTful API, representations of the server state are exchanged between the client and server. The server state is divided up between a number of resources (for example, there could be one resource that is a list of the programmes recorded on a Personal Video Recorder (PVR), perhaps "`http://stb.example/storage`", another resource that represents the programmes in a box's Electronic Programme Guide (EPG), perhaps "`http://stb.example/programmes`", and so forth). Where appropriate, the client can send back a new state for a particular server resource, and the server then deduces what actions are necessary to change its state to that requested. Appendix A contains a sample sequence of interactions that illustrate how a RESTful API can be designed to enable remote control functionality.

We evaluated three standards for performing RPC over HTTP: XML-RPC, JSON-RPC and SOAP. Of these, JSON-RPC was poorly supported on a number of key platforms, but the other two standards had good (but not perfect) cross-platform support. We found that all three approaches had some disadvantages that were important from our perspective, however:

- In all cases, HTTP is being used as a little more than a "dumb bit-pipe"—the HTTP headers are generally unnecessary overheads. Particularly on mobile platforms, resource consumption

(bandwidth, memory and CPU) is a real concern, so the use of high-overhead technologies is troubling.

- As a consequence of this way of using HTTP, clients cannot make use of HTTP features to further reduce resource consumption. For example, HTTP client libraries cannot make effective use of caching to store the results of requests locally, and avoid redundant network requests.

- Adopting an RPC standard requires developers on most platforms to use third-party libraries that may be poorly documented, inconvenient to use or buggy. In an ideal world we would minimise library use to the greatest possible degree.

Adopting a RESTful approach (in particular, attempting to adopt the Resource Oriented Architecture (ROA) approach set out in O'Reilly's "RESTful Web Services" book [8]) eliminates most of these disadvantages, and introduces several other benefits (assuming that the API is well-designed):

- Simple interactions with the server can be *explored* very easily indeed—often by simply pointing a web browser at the server.

- We can take advantage of the inherent statelessness of HTTP to permit the client to drop all its network connections whenever it has reason to believe that the user is not paying attention, without requiring a resource-consuming reconnection process when the client reconnects. This is a particular benefit for conserving resources on mobile devices.

- Implementations can make use of HTTP's caching and "conditional get" mechanisms to further reduce resource utilisation where resources have not changed since a prior retrieval.

- A RESTful approach is inherently scalable. Multiple clients can be supported as easily as single clients (within the constraints of the number of simultaneous connections supported by the server), and changes made by one can be notified to all[6]

For these reasons, we adhered as closely as we felt able to the principles of REST and ROA.

5.7 Data Representation Formats

In common with the other design decisions made in the development of the API, the format in which to represent the data being carried by the API was subject to considerable investigation. We considered four main options: XML, binary XML, JSON or a custom format, eventually settling on XML. Where the availability of cross-platform support for an option was considered, this is summarised in Appendix B

Of the above options, a custom format (perhaps even a binary format) offers the ultimate in flexibility and efficiency. However, there are very significant disadvantages to such an approach. The format would have no developer familiarity, no software tools, no parsing libraries, little or no human-readability and would be prone to backwards-incompatibility. We were requested to consider the option, due to a concern that parsing a more verbose format would have a negative impact on performance and battery life on resource-constrained devices, but ultimately we decided that it was the wrong way to go.

XML was in many ways the "baseline" option. The standard has all the advantages that a custom format lacks: extensive developer familiarity, a wide range of parsing libraries available on all key platforms (including some which are very low-footprint), WADL and WSDL support

[6]Many of the scalability advantages of RESTful approaches that apply to Internet-scale web services do not apply to the UC problem space, of course. For example, the abilities to use caching proxies, or use multiple servers to handle requests to the same resource are unlikely to be relevant.

for XML as a representation format and good human-readability. In many cases, backwards-compatibility is possible in XML even without resorting to versioning resources: if new versions simply add elements or attributes, older clients can simply ignore the new information.

We also considered the possibilities of a binary XML format. A number of ways of representing XML in a compact binary form have been defined, including BiM, a proprietary format that has been adopted for the carriage of TV-Anytime metadata over DVB networks. Compared to XML, binary representations offer reduced bandwidth use and, potentially, faster parsing. Unfortunately, there is no generally-accepted standard with broad cross-platform support. wbXML is a binarisation format standardised by the WAP forum (now the Open Mobile Alliance). It was the format used in the original WAP standard. It has been adopted into other OMA formats such as SyncML and OMA DRM, and hence is supported (in some sense) on most mobile phones. Unfortunately that sense does not generally extend to the provision of native libraries on the project's key client platforms, and third-party library support appears patchy. In general, we were unable to find a binary XML format with good library support on all key platforms, and for that reason alone, rejected the option. (It should be noted that many HTTP clients and servers support lossless compression of HTTP payloads, which offers some of the same benefits for little or no developer effort.)

Finally, we considered JSON, a newer, lighter-weight format than XML. Parsers exist for almost all of the project's key client platforms, with the apparent exception of Symbian S60. However, a POSIX C library may be portable to this platform. A schema language also exists for JSON, although it does not appear to be as widely adopted as its equivalents for XML. JSON shares many of the developer-familiarity and human-readability benefits of XML. On the other hand, JSON itself does not appear to have been standardised by any international body, although as a subset of ECMAScript, it could be described as a subset of that ECMA standard. Overall, there appeared to be few benefits to using it in preference to XML, and a few minor disadvantages.

In coming to a decision regarding a data representation format, we bore in mind the following considerations:

- There's no point optimising an API for a particular class of client devices if doing so makes it significantly harder to implement for everyone, unless both that client device class and the optimisations in question are absolutely required for a particular use case. This is the biggest argument against going down the route of a proprietary format—there are many disadvantages to doing so, and no direct evidence that it's required.

- The Universal Control API may end up still being implemented in devices 10-20 years from now. It will inevitably need extending as the capabilities of devices change, and hence backwards-compatible extensibility is important.

- Even on mobile devices, binary protocols are not ubiquitous. In version 2.0, standardised as long ago as 2002, WAP moved from wbXML to XHTML MP (an XML format).

- We really do not want to be supporting more than one representation format unless it's absolutely necessary. If we do, then they should be two formats with a straightforward 1:1 bidirectional mapping between them.

Ultimately, of the options considered, only XML had the combination of mature, cross-platform support, a wide range of software tools and domain-specific languages built on top of it, developer familiarity and built-in extensibility that the project needed. The possibility exists of supporting multiple representation formats, identified by MIME type in the HTTP headers, but this is clearly a fall-back compromise. At the time of writing, following the implementation of a number of prototype clients and servers, no compelling reason for implementing a second representation format had been identified.

Having selected XML, we set about designing our resource representations with two goals in mind: keeping them concise, to minimise the resource utilisation associated with storing and

transmitting them, and making them easy to parse using the event-based "SAX" parsers that are common on mobile phones and other resource-restricted devices. To achieve the former, we recommend in the spec that clients and servers generate XML without namespace information, a `<?xml ... ?>` tag or unnecessary whitespace. To achieve the latter, we have designed the resource representations to minimise the use of nested elements, using attributes instead where possible, and to avoid the use of "mixed content", where an element can contain both text and other elements.

6 Implementation

In this section, we set out the design decisions behind various aspects of the API. This section is not intended to be a description of the API itself—a separate White Paper (R&D 194 [1]) exists for that purpose, and the reader is referred to that document.

6.1 Notification

The obvious mechanism for permitting this would be for the box to open a connection to an open port on the client and send information that way. Unfortunately, not all key client platforms would support such a feature: as described in Section 5.5, a number of these platforms guarantee no network access other than an HTTP client library, and hence cannot listen for incoming connections.

Two alternative mechanisms present themselves as solutions to this problem. The simplest is simply to have the client send requests for the necessary information repeatedly (once a second, perhaps) and take action when the response changes. There are many precedents for this approach — to give one, this is the way in which Google's browser-based GMail client works. A second option (used by Apple's Digital Audio Control Protocol (DACP) for remote control of iTunes instances over local networks, for example) is to have the client send an HTTP GET request asking to be informed of a particular event. This request then stalls, and no response is sent by the server until such a time as the event occurs. If the request times out before the event occurs then the client simply sends another request. This method, variously referred to elsewhere as "Bayeaux", "Comet" or "long polling", has the advantage of requiring less frequent requests and allowing more immediate response to notifications. A disadvantage is that the event may occur during the time between the connection timing out and the client re-requesting, but if the mechanism is well-implemented, this will simply result in a delay to the notification.

We chose to allow for both methods of notification, to give maximum flexibility to client developers. Clients can request resources repeatedly (possibly using HTTP's built-in cache and partial GET mechanisms to reduce resource utilisation), or take advantage of a single "notification" resource (`http://stb.example/notify`) that adopts the long-polling mechanism described above if they want state-change notifications in real-time. The notification mechanism is designed to be cheap to implement on the server, and for clients with limited numbers of simultaneous connections available to be able to connect to the `/notify` resource intermittently, without missing events. This latter feature also implies that clients can disconnect from the notification service (and drop all its network connections, as mentioned in section 5.6) when not in active use by the user, to conserve battery life and other resources.

6.2 Discovery

When considering the discovery problem, we made some assumptions about the nature of the network environment in which UC-capable devices would typically operate. We assumed that clients and servers would be in the same IP subnet (ie, traffic between the two is not routed). We assumed that IP addresses on that network would be self-assigned, or assigned unpredictably by a DHCP server. We assumed that the IP addresses of servers and clients might be subject to unpredictable changes. Finally, we assumed that there would be no managed DNS on the network.

These assumptions are believed to be valid for the vast majority of home networks in the UK at the time of writing.

We knew from the outset that fully-automatic general solutions (i.e. solutions where the client can discover the server with no assistance from either the user or other network devices) would not be possible on all key client platforms. As described in section 5.5, some platforms provide unicast HTTP client libraries and no other forms of network access. A robust and reliable automatic discovery protocol really requires some means of communicating with a local multicast group, which is not an option on these platforms. We therefore concentrated our efforts in two directions: identifying a fully-automatic discovery solution for platforms where doing so is possible, and designing a minimally-interactive alternative solution for other platforms.

6.2.1 Automatic Discovery

A number of solutions for automatic service discovery over IP networks have been developed over the years. At present, two main methods have been widely implemented: the multicast DNS (mDNS) and DNS Service Discovery (DNS-SD) standards developed by Apple and referred to by them as "Bonjour", and the Simple Service Discovery Protocol (SSDP) defined by Microsoft as part of UPnP. We investigated both technologies as candidates for incorporation into the Universal Control standard before eventually choosing Bonjour.

Both technologies involve the use of IP multicast and UDP to implement a scalable method of communicating with the other devices on a network. Neither set of technologies is fully-standardised. Details of Bonjour are freely available as an informational RFC from the IETF. Details of SSDP are freely available from the UPnP website, and have been standardised as part of ISO/IEC 29341 (which appears to be a snapshot of the UPnP specifications at a particular point in time). An expired IETF Internet Draft also describes SSDP.

Implementation-wise, both technologies build on existing standards. As might be expected from their names, the Bonjour technologies build on DNS and on the RFC 2782 standard for DNS SRV records. SSDP uses, unusually, the headers of multicast HTTP over UDP as its transport mechanism, and defines its own semantics. Although both support the discovery of specific services on devices running on the network, only Bonjour allows the advertisement of any kind of network service. SSDP is very much a component of UPnP as a whole, and compliant implementations only support the discovery of UPnP services. Since the UC API is not based on UPnP, SSDP could only be used to discover UC servers in a non-compliant fashion. This is certainly a possibility: the URL of the UC base resource could be advertised in the service URL field of an SSDP description or advertisement method, for example, or an additional header field could be used. The disadvantages of this approach include potential disruption to the functionality of third-party UPNP-compliant equipment and the probability that UPNP libraries would require modification to implement such extensions.

Looking at the library support available for the two technologies on key platforms (see Appendix B) we see that neither technology has universal support. In particular, platforms that lack multicast IP and/or UDP support (as detailed in section 5.5) cannot support implementations of either. Overall, Bonjour appears to be most widely supported.

Finally, SSDP appears to be a substantially more complex protocol than the mDNS/DNS-SD components of Bonjour. It supports service and discovery via both server advertisements and client searches, whereas mDNS only implements the latter. It also provides a method for notifying clients when devices are about to leave the network that relies on GENA, but implementing this is optional. Overall, both technologies are well-standardised, but Bonjour appears to be simpler and to have better cross-platform library support. It can also be used to advertise any web service. It is therefore our adopted solution for automatic discovery of Universal Control servers on local networks.

6.2.2 Manual Discovery

If clients cannot use automatic discovery mechanisms, then the user (or a third party) must provide information about the server to the client. A large number of possible methods were considered:

- The box could display the discovery information on an attached television screen, or on a screen built in to the box. The user would enter the information directly into the client when asking it to connect to the server. This has the advantage of not requiring any third parties or technologies that devices running on the key platforms are likely to possess, but requires the user to type in a string of alphanumeric characters and symbols with little meaning to them, and may require that this process be repeated each time the box is assigned a new IP-address.

- A central server might be maintained on the Internet to which boxes could submit part or all of the discovery information. Such a server would then pass on the information to a client upon being asked to do so. This mechanism has the advantage of not requiring special IP multicasting capabilities, but has the disadvantage of requiring an Internet connection to be present for both the box and the remote client device (and, of course, someone would have to run the central server). The client would also have to provide some kind of information to identify the box to the central server, so in many cases this solution may not be more convenient than obtaining the discovery information from the box via the television and then entering it into the client. The exception would be if the box was subject to frequent changes of IP address, in which case a central server could improve the user experience.

- The box could display a QR Code on an attached television which encodes the discovery information. This code could be read with a camera built in to the remote client device and a standard QR-code reading library. This is a well tested technology, but will not work on devices which lack a camera, or whose built-in camera is too low in quality. It may also present problems for visually impaired users. Also some (perhaps many) QR-code readers will respond to a QR-coded URI with an HTTP scheme by loading it in a web browser, which is unlikely to be the desired behaviour.

- The box could encode the discovery information as an audio-clip which could be played over built-in or attached speakers, and detected by a built-in microphone on the remote device. It is even possible that the discovery information could be embedded as an audio watermark in all the audio produced by the box. This option has similar advantages and disadvantages to the use of QR-codes, but there are fewer available libraries. It does have the advantage of working better than that option for visually impaired users.

- The box could send the discovery information to an email address or telephone number entered into its own built-in interface via the normal remote control. This method could well be extremely inconvenient and requires the client device to be able to receive text messages or emails.

- The box could use a unique IP-address assigned to it when manufactured regardless of what the local network settings were. This would violate best-practice when setting up local IP networks, and might lead to clashes with multiple boxes or other IP devices trying to use the same IP address.

- The client device could send a connection request to every valid IP-address on the local subnet and connect to the first one which responds correctly. This could take a very long time, and cause a great deal of network traffic.

Of the above options, the only one guaranteed to work for all client devices under all circumstances is the first in the list: to require the user to enter the discovery information manually. To improve the user experience (especially for mobile phone clients running on devices with

only a numeric keypad), we designed an entropy coding scheme that takes advantage of the fact that most home network IP addresses fall within the private ranges designated in RFC 1918 (eg 192.168.0.0/24). For simplicity, if the server requires clients to authenticate themselves using a client ID and shared secret (a possibility allowed for in the API, to permit device-based authentication for servers that do not want to allow unrestricted access —see section 6.4 for details) then information needed to acquire those credentials is incorporated into the code. The code is presented to the user as a "pairing code". This scheme is detailed in [1], but to give an example of its use, the information that the server has IP address "192.168.0.12", is listening on the default port (48875), and wishes the client to identify itself with the client password "1d" (in hexadecimal) can be conveyed to the client by the user entering the code "603P".

This scheme still has the disadvantage that if the box's IP address changes unexpectedly, the "pairing code" must be re-entered. On client platforms that do not support automatic discovery mechanisms, there is simply no good way round this requirement. Using the entropy coding scheme at least minimises the amount of information that must be re-entered. It is good practice for DHCP servers to minimise the frequency of IP address changes on networks, but there is always the risk that poor DHCP server design or configuration could lead to poor Universal Control user experiences on certain networks and certain platforms. If a box is being distributed by a service provider who also supplies the home DHCP server[7], it is possible that before it is supplied to the user, the latter could be configured to assign the same IP address to the box persistently.

6.2.3 Discovery Implementation

An obvious consequence of the fact that not all clients can support an automatic discovery mechanism is that servers must implement support for a manual one. However, the benefits of implementing an automatic mechanism alongside a manual one are considerable. In particular, doing so enables authorised clients to maintain a connection to a server indefinitely, with no further interaction from the user. There are two circumstances in which enabling this is particularly important. Firstly, we anticipate that many boxes that implement the UC API for accessibility purposes will be set up for users with impairments by volunteers. Re-entering a pairing code is likely to be particularly problematic for these users. Secondly, we want to enable "machine clients" such as audience monitoring devices, or standalone speech-synthesis units for the visually impaired that speak the actions taking place on the box to the user. These devices only make sense if servers support an automatic discovery mechanism that allows clients to keep track of any changes to the server's IP address over time. Therefore, we have declared support for pairing codes to be mandatory in both clients and servers, and support for Bonjour to be mandatory in servers.

6.2.4 IPv6 Considerations

As described in section 5.4, the only area of the API specification where the differences between IPv4 and IPv6 need to be considered is that of discovery. If discovery takes place via Bonjour then the version of IP in use is still immaterial, since the mDNS protocol [9] explicitly and transparently supports both versions . If the discovery takes place via a pairing code however, then the difference between IPv4 and IPv6 does matter, since the information encoded will need to be different in each case.

At present it seems too early to say with any certainty what kind of IPv6 address will typically be allocated to devices on home networks. One factor which may turn out to be relevant is that although device addresses on an IPv6 network may well all be globally unique (unlike on most IPv4 home networks) it is highly likely that the server and client will share the same IPv6 subnet-prefix (see [5]). It is current practice for IPv6 global unicast addresses to consist of a 64-bit subnet prefix and a 64-bit interface ID, requiring only the last 64 bits of the server's IPv6 address to be transmitted to the client. 64 bits is still an inconveniently large number to represent in duodecimal

[7] An increasingly common practice as ISPs attempt to build IPTV businesses

(requiring up to 18 characters), but it is anticipated that as with IPv4 addresses on local networks there will be a number of ranges of interface-IDs which are far more commonly used and hence can be assigned shorter codes.

It may be the case that at the point at which IPv6 becomes standard on home network the list of client platforms considered key for Universal Control usage scenarios (see section 5.2.1) no longer includes platforms without support for Bonjour, rendering the inclusion of discovery information in the pairing code unnecessary.

To facilitate the future expansion of the pairing code to support, amongst other possibilities, IPv6 addresses, a "version" bit has been included—currently required to be set to `0`.

The additional capabilities of IPv6 with regards to multicast and anycast addressing do not, at this time, appear to offer any enhanced capabilities which would be of use in the implementation of Universal Control over and above those provided by IPv4.

6.3 Security

As described in section 4.9, access to the API may be restricted to clients authorised by the user. The other aspect of API security—protecting communications from eavesdropping—has also been considered. Given the nature of the information carried by the UC API at present, we consider that the fact that communications are limited to the home network probably provides sufficient security. Given that encrypting communications between the client and server would involve a significant resource overhead, we do not recommend its use.

6.4 Authentication Options

A one-time authentication credential can be generated on the box, conveyed to the client device by a sufficiently secure non-network channel (eg by presenting it to the user via the "pairing code" mechanism that is also used by the discovery process) and used to obtain from the server a larger credential which can be used thereafter by the client to authenticate its requests to the server. To avoid user irritation the initial key can be kept reasonably short, and servers can protect themselves against "brute force" attacks by (for example) temporarily blocking access to devices that repeatedly fail to authenticate correctly within a short period of time.

The security of this approach is best suited to scenarios where any person with private physical access to the box is assumed to be authorised to control it via the local network—for situations where this is not the case, the ability to generate or manage client keys on the box could be protected via the common PIN mechanism that protects parental control settings, etc, referred to above.

Equally important as the question of authentication is the question of *deauthentication*. There are various circumstances in which a user might wish to prevent a previously authorised device from being able to connect to the server: if a device is lost or stolen, for example, or if domestic circumstances change. For this reason, it is important for the API to define a way for a server that requires authentication to be able to determine which authenticated device is the origin of a request. Boxes should probably allow devices to be deauthenticated using their built-in APIs, but it is also important for the API to support deauthentication of devices via the API itself, to extend that ability to people with accessibility requirements.

Associating authentication credentials with individual clients rather than individual users ensures that individual clients can be always be individually deauthenticated, but does not necessarily map on to a box's existing authentication model. In particular, some boxes may have a model whereby different privileges are given to different users. Servers that run on such boxes that wish to extend the concept of individual users into the way they handle API requests can do so by making a permanent association between a device and a user at the point when an authentication credential is generated for that device. This is a good way to treat personal devices like mobile phones, but not shared devices like family laptops. It may be appropriate for such servers to present users

at the point of credential generation with the choice not to associate a particular device with a particular user, to cater for this scenario.

The HTTP protocol specifies two forms of authentication which can be used to verify the identity of a client connecting to a host:

- HTTP Basic Authentication is very simple to implement, but involves sending a password unencrypted over the connection to the server. Unless that connection is encrypted (via a VPN or HTTPS, for example), this is insecure unless the underlying network offers protection against eavesdropping, which is not always the case in contemporary homes.

- HTTP Digest Authentication is slightly more secure, in that only hashes of passwords are actually sent over the network. Only passwords are concealed from eavesdroppers however: as with Basic authentication, the connection must be encrypted by other means if other components of the HTTP traffic are to be secured.

The advantage of using an authentication method built into HTTP would have been that support would likely be provided in many HTTP client libraries without modification. It would also have been an easy way of ensuring that the addition of authentication dis not affect the RESTfulness of the API.

There are major downsides to using these mechanisms, however. At time of writing many major browsers intercept authentication challenges issued as a response to requests made via AJAX, making it very hard for javascript clients to programmatically set username and password. In particular this behaviour was extremely troublesome when combined with CORS support.

Other authentication options are available, of course. Cookie-based authentication methods are commonly employed on Internet websites, although these are not RESTful, and not secure unless client-server communications are encrypted. HTTPS could be used for authentication (as well as security), although the resource overhead associated with HTTPS is considerably higher than for HTTP, and key management would be challenging. Alternatively, a custom solution could be implemented, perhaps a functional equivalent of HTTP Digest, but using custom HTTP headers, or even an implementation of a cryptographically secure key-exchange mechanism such as Diffie-Hellman [10].

Earlier versions of the API made use of HTTP digest authentication, but as of version 0.6.0 this has been replaced with a custom authentication mechanism based closely upon digest authentication (though using a different, more flexible hashing algorithm). This was decided upon because it was felt that with a sufficiently sized secure key this mechanism would be sufficiently secure for our purposes, though the authors cannot rule-out future changes to a more secure system. It was judged that the Diffie-Hellman algorithm, whilst mathematically simpler and significantly more secure, had insufficient library support to be a viable choice.

The final scheme involves the use of a short shared secret carried in the pairing code to encrypt the transfer from server to client of a much larger secret which is used as the actual authentication key. The size of this key makes off-line brute force attacks (to which digest-style schemes are vulnerable) unlikely unless the initial pairing exchange was intercepted by a malicious third-party.

6.5 Request Restriction

Even when a user has authorised a particular client to connect to a box, there are circumstances where it is appropriate to request further information from the user before fulfilling a user request. For example, the box may require explicit confirmation that the user wishes to perform a particular action (such as requesting the presentation of content that is unsuitable for younger viewers), or require proof that the user is a particular member of the household, by requiring the entry of a PIN before the action is carried out. This latter mechanism could be used to protect access to adult or

pay-per-view content, for example. The API implements mechanisms that fulfil these use cases.

The operation of the mechanism is as follows, for the flow of events in which no errors occur. Upon receiving a restricted request to a resource, the server responds with a restriction challenge instead of the usual response. Upon receiving the challenge, the client presents a message describing the restriction to the user, and asks them for either a confirmation, or a decimal PIN. Upon receiving the appropriate response from the user, the client repeats its request, adding a special header containing the user response. The server then instructs the box to carry out the requested action.

The mechanism is designed to be used rarely, and server implementors need to consider the impact on disabled users (particularly those with severe motor impairments) when restricting a kind of request with this mechanism. In general, it may be preferable to allow users to configure the extent to which a particular client device is subject to these restrictions: for example, an option could be provided in the built-in interface that would instruct the box that a particular UC client would only ever be used by a user over the age of 18, with a corresponding reduction in the number of restrictions that requests from that client would be subject to.

6.6 EPG Formats

There are a number of existing formats which have been developed for the transmission of Electronic Program Guide (EPG) information. We list them here with a brief summary of some of the advantages and disadvantages of each:

- TV Anytime is an ETSI-standardised format for EPG information and other media metadata [11]. The format is XML-based, but very complex. A schema exists for validating it. It is possible to specify a subset of TV Anytime for a particular application. TV Anytime requires the use of CRIDs as identifiers for individual programmes and groupings. The format can certainly be used to provide a list of services and also to provide schedule information (including titles and descriptions) for programmes being transmitted on a particular service. The same metadata format used for schedules can also be used for non-scheduled information, containing location information for programmes which can be stored with any kind of locator. On the down side, the table format is very normalised, rendering SAX parsing inefficient. It has no concept of channel numbers for TV services. This format could be used with only the addition of a mechanism for specifying logical channel numbers, but is not ideal.

- The BBC Programmes website [12] provides metadata and schedule information about BBC programmes. It includes an ontology of structured concepts which can be used to describe TV and radio metadata. The ontology is based on RDF, and can be used to construct an XML format for conveying the relevant information. This specification has not been standardised, but is published and widely used. It appears to have no concept of a channel number. Because it is RDF-based the level of normalisation used in the construction can be varied considerably for the needs of the project, however the level of verbosity in its description of even simple data is relatively high. Its vocabulary is based upon BBC usage, and does not always reflect the terms used elsewhere in the industry. This format could be used with the addition of a mechanism for specifying logical channel numbers, but is not ideal.

- The RadioDNS project [13] has defined a metadata format for radio which makes use of an XML metadata format standardised by ETSI in TS 102 818. This format uses a lot of the same ontological concepts as TV Anytime but is far less normalised, and hence is more suitable for SAX parsing. It appears to have no television equivalent, but carries most of the information which would be needed for such a purpose. It appears to have no concept of a channel number. Unlike TV Anytime it permits identifiers other than CRIDs. It would be

impossible for us to use this format unchanged, but it might serve as the basis for a more generalised format.

- The XMLTV project maintains an XML format [14] designed for carrying TV scheduling and metadata information. The format is simpler than TV Anytime, and ontologically incompatible with the previously listed formats. It also makes frequent use of context to determine the meaning of a tag, making SAX-style parsing more difficult. It has no concept of channel numbers, and includes date and duration in proprietary formats rather than following ISO standards. It is also heavily oriented towards broadcast services, being based around concepts of "channels" and "programmes". There are no obvious advantages to using this format, and several disadvantages.

- BBC iPlayer RSS feeds [15] return scheduling information in a non-standardised XML format loosely similar to the ontological structure used by /programmes. The format is designed for on-demand sources and has no obvious support for scheduling information. Since it has no support for scheduled services or logical channel numbers it was not directly usable for our purposes.

- The now-discontinued BBC Web API [16] returned metadata in both TV Anytime and its own "Simple XML" format. The information conveyed is similar to that in TV Anytime, and the two formats are convertible, but the simple format is significantly flatter and less normalised. It is somewhat similar to the format used for RadioDNS, but even simpler. It lacks any concept of genre tags or logical channel numbers, however.

- The DVB SI Event Information Table (EIT) format [17] is a binary standard used for the transmission of EPG information in DVB broadcasts. EIT is a very common way for DVB set-top boxes and PVRs to obtain EPG metadata, and hence it was considered beneficial to pick a metadata standard for UC to which EIT could easily be converted. (As a binary format, we ruled out adoption of EIT itself, for the reasons explained in 5.7). EIT has been standardised by ETSI in EN 300 468. Of itself this format was not appropriate for our needs, but was borne in mind as the probable source for a lot of metadata made available via the API.

The key issues that we considered when trying to pick a format for schedule metadata were verbosity (and its implications on resource consumption), ease of SAX parsing (the most desirable XML parsing option on resource-limited platforms), breadth of adoption or degree of standardisation, and similarity to other standards with which compatibility is desirable. A first attempt involved the adoption of a subset of TV Anytime extended to support channel numbers, but we found the resulting XML to be too verbose and difficult to parse. Ultimately, we decided to define our own metadata format, with a structure designed to be concise and easy to parse, but with a data model that mapped well to EIT, TV Anytime and the BBC /programmes service.

For details of the final format, see [1].

6.6.1 The API Extension Mechanism

This solution was implemented in preference to an even simpler scheme, in which remote clients or interactive apps opened TCP sockets directly, and the UC API was used simply for exchanging discovery information about the additional service, such as its URL. We considered that the benefits of implementing communication as an extension of the UC API (the guarantee that communicating devices are mutually authorised to exchange information by the user if the server implements the security scheme described in section 6.4), the built-in mechanism for real-time event notification (if the server implements the mechanism described in section 6.1), and the ability for clients and servers to deal with IP address reassignment elegantly (particularly if the client implements the automatic discovery mechanism described in section 6.2) outweighed the reduction in implementation overhead

on the behalf of server implementors. In coming to this decision, we took into account the fact that a means of communicating information from interactive applications to the UC server would still need to be implemented, and that the server implementor would presumably still want to limit communication with remote clients to trusted interactive applications alone.

The possibilities of using the API extension mechanism as a way of making individual applications more accessible are perhaps even more important than its potential for enabling multi-device "orchestrated media" applications, although doing so is likely to make it harder to create truly universal standalone accessible clients.

6.6.2 Application Lifecycles

Because the API extensions implemented by particular applications are completely undefined, a general purpose client is not possible. Instead, the concept of an "app-manager" client application is foreseen. Such an application could connect to a UC server and then look on the Internet for other client software capable of interacting with each interactive app installed on the box. This software could, for example, take the form of web pages or widgets launched from a browser within the app-manager, Java classes containing custom clients for each API extension that are downloaded and then instantiated directly by a Java app-manager, or standalone applications that the app-manager identifies in a platform application store. If appropriate, the authentication credentials that allow the app-manager to connect to the server could be transferred automatically to the clients that it invokes.

A feature of the UC design is that clients have full control over their own lifecycles. This is an absolute requirement on the client platforms that lack multitasking capabilities, and a significant advantage on all battery powered client platforms, since requiring a client to listen permanently or poll repeatedly for invocation requests (using a long- or short-polling mechanism) would have a significant effect on resource consumption, particularly battery life. This being the case, if multiple devices must communicate to present a shared experience to one or more users via the UC API, it may well be the case that each client implementation must be invoked manually be a user before it can detect and join the experience.

The flexibility of this approach is considerable, but not total.

7 Conclusions

The Universal Control API has been designed to be a good API for creating remote user interfaces running on PCs and mobile phones and in web browsers, controlling set-top boxes and similar devices on the local network. It has been designed with accessibility use cases foremost, but with the intention of enabling all kinds of user interface to be designed, for the wider population as well as those with specific accessibility requirements. It has also been designed to enable new kinds of user experience, with synchronised content playing on remote devices, and applications on client and server devices communicating to create new kinds of user experience that span multiple devices within the home.

References

[1] J. Barrett, M. Hammond, and S. Jolly. The Universal Control API v.0.6.0. TBC 0, BBC R&D, 2011.

[2] A. van Kesteren. Cross Origin Resource Sharing. Technical report, W3C, 2009. `http://www.w3.org/TR/access-control/`.

[3] A. Mason and R. Salmon. Factors Affecting Perception of Audio-Video Synchronisation in Television. October 2008. 125th Convention of the Audio Engineering Society paper 7518.

[4] Internet Protocol. RFC 791, DARPA Internet Program, 1981.

[5] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291, IETF, 2006.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, 1999.

[7] Representational State Transfer. Technical report, wikipedia.org, 2011. http://en.wikipedia.org/wiki/Representational%20State%20Transfer.

[8] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, May 2007. http://oreilly.com/catalog/9780596529260.

[9] B. Woodcock and B. Manning. Multicast Domain Name Service. Technical report, watersprings.org, 2000. http://www.watersprings.org/pub/id/draft-manning-dnsext-mdns-00.txt.

[10] E. Rescorla. Diffie-Hellman Key Agreement Method. RFC 2631, IETF, 1999.

[11] EBU/ETSI JTC Broadcast. Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems("TV-Anytime"); Part 3: Metadata; Sub-part 1: Phase 1 - Metadata schemas. Technical Specification TS 102 822-3-1, ETSI, 2010. http://webapp.etsi.org/WorkProgram/Report_WorkItem.asp?WKI_ID=33465.

[12] BBC Programmes. Technical report, British Broadcasting Corporation, 2011. http://www.bbc.co.uk/programmes.

[13] RadioDNS Website. Technical report, RadioDNS, 2011. http://radiodns.org.

[14] The XMLTV file format. Technical report, XML TV Project, 2011. http://wiki.xmltv.org/index.php/XMLTVFormat.

[15] iPlayer RSS Feeds. Technical report, British Broadcasting Corporation, 2011. http://www.bbc.co.uk/iplayer/feeds.

[16] The BBC Web API (beta). Technical report, British Broadcasting Corporation, 2011. http://www0.rdthdo.bbc.co.uk/services/api/.

[17] EBU/ETSI JTC Broadcast. Digital Video Broadcasting (DVB); Specification for Service Information (SI) in DVB systems. European Standard EN 300 468, ETSI, 2010. http://pda.etsi.org/exchangefolder/en_300468v011101p.pdf.

Appendix A Example Client-Server Interaction

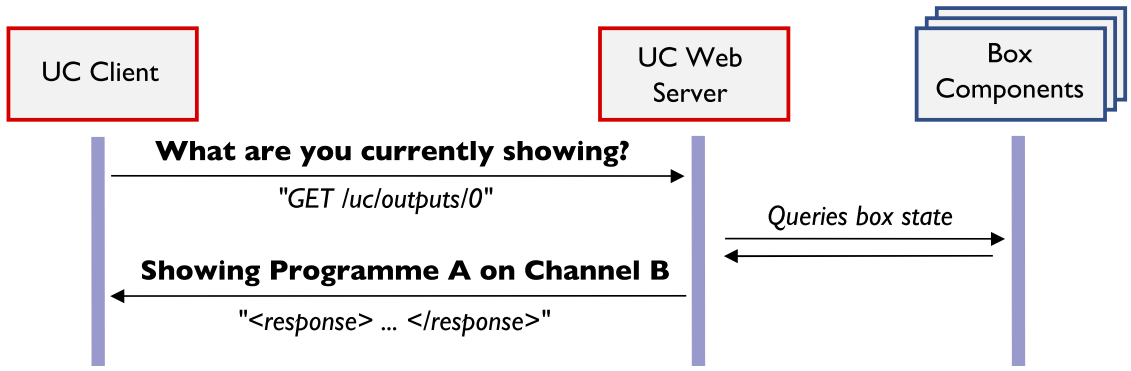

Figure 3: A sequence diagram showing how a UC client and server interact to obtain information about the current programme and channel to which the box is tuned.

The above diagram illustrates an example sequence of interactions between a UC client and a UC server running (in this case) on a television or set-top box. In this example, the client and server interact to obtain information about the television channel currently being presented by the box.

In the interaction, the client first requests information from the server about the state of one of the box's "outputs". An output represents a part of the box that presents media to the user. In the case of a television, this could be the television's screen; in the case of a set-top box, this could be its HDMI socket. The server queries the internal components of the box (such as its tuner) for the information necessary to fulfil the request, and then returns to the client a "representation" of the state of the resource (an XML document) containing (perhaps amongst other information) information about the channel and programme currently being presented.

Figure 4 shows how a client might request that the box present a different television channel. To do so, the client sends back to the server a new version of the resource representation that it retrieved previously, in which the channel and programme have been changed. The server interprets this as a request to change the channel being presented on the output to the one specified in the client request, and makes a corresponding request to the internal components of the box to change channel.

If and when the channel successfully changes (or if another change occurs that would change the information contained within a resource representation), the server notifies the client (using the notification mechanism described in section 6.1) that the state of the resource has changed. The client then repeats its request for information about the state of the output, and the response from the server includes the information that the channel has been changed to that requested by the client.

Appendix B Technology Support on Key Platforms

The following tables summarise the research carried out into the capabilities of key platforms during the design of the UC specification. It should not be regarded as authoritative—each cell in the table contains the results of a reasonably exhaustive but necessarily time-limited research exercise carried out by the project team during the first few weeks of the project, in July/August 2009. It is included as an explanation of the design decisions taken. Where the information stated is particularly uncertain (perhaps because the information was inferred rather than directly available), this is indicated with a question mark.

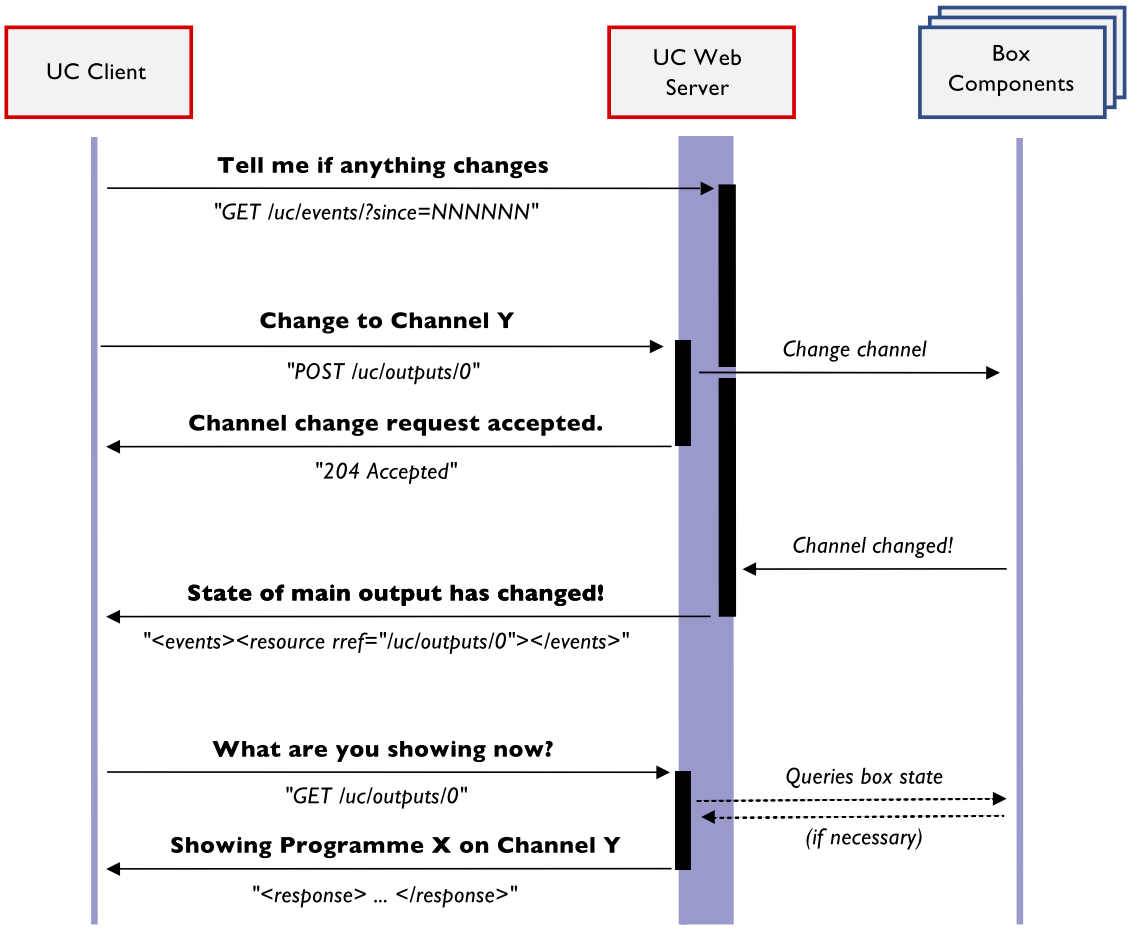

Figure 4: A sequence diagram showing how a UC client and server interact to request a change to a different channel, and then to notify the user that the requested channel change has completed successfully.

	Mobile Platforms						PC Platforms		
Technologies	J2ME	WebOS	Android	S60 native	iPhone	HTML/JS	Java	C(++)	Python
TCP/IP (client)	Supported in MIDP 2.0 but not MIDP 1.0	Applications may be limited to HTTP.	yes	yes	Yes	no	Yes	Yes	Yes
TCP/IP (server)	Supported in MIDP 2.0 but not MIDP 1.0	Applications may be limited to HTTP.	yes?	yes	Yes, more fully featured support via library	no	Yes	Yes	Yes
HTTP (client)	Yes (HEAD, GET, and POST only)	Yes	Yes	Yes	Yes (all verbs) easier with a library	Yes (standard only requires HEAD, GET, and POST, but some implementations support PUT and DELETE)	Yes (all verbs)	Yes (all verbs)	Yes (all verbs)
HTTP (server)	possible in MIDP2.0				No libraries.	No	Yes	Yes	Yes
XMPP	JXA is a J2ME Jabber client. It appears to be a student project that is not being maintained actively.	No known libraries. Applications may be limited to HTTP.	The Smack library has been ported to Android.	No known native libraries. A standard c++ library may be portable, particularly to s60r5 phones.	The XMPP framework library has been ported to the iPhone. It probably doesn't support any RPC stuff.	xmpp4js is a Javascript library—no support for Jabber-RPC or XMPP-SOAP extensions though. "strophe" is an alternative that looks less mature.	Smack is a Java library—no support for Jabber-RPC or XMPP-SOAP extensions.	Several libraries are available, such as gloox.	Several implementations available, eg the Kamaelia-based "headstock".

Technologies	J2ME	WebOS	Android	S60 native	iPhone	HTML/JS	Java	C(++)	Python
ICE	No	no	no	no	Yes	no	Yes	Yes	Yes
Jabber-RPC[a]	no	no	no	no	no	no	Groovy implementation available	no known implementations	probably
XML	yes (model, push, pull)—see also JSR-172, which mandates a SAX parser	presumably	yes (model, push)	yes (model, push)	yes (push), and expanded capabilities via third-party libraries	yes (model, push (via library), and in some implementations E4X	yes (everything)	yes (model, push)	yes (model, push)
wbXML	Apparently, via kxml. May be feature-limited.	Unknown	Apparently, via kxml. May be feature-limited.	May be possible, using libwbxml	Unknown	No	Apparently, via kxml. May be feature-limited.	Yes—via libwbxml	Immature support available via pywbxml
XML-RPC[b]	Yes, via J2MEpolish or kXML-RPC	unknown	yes, via librar(y/ies)	no(?)	kind of	yes, via several libraries	yes, via library	yes, via libraries	yes
JSON	yes, via library	probably	yes	no, although a standard c++ library might be portable	yes, via library	yes	yes, via libraries	yes, via libraries	yes

[a]Because Jabber-RPC is essentially XML-RPC over an XMPP connection, it is often possible to implement a Jabber-RPC server or client using an XMPP library and an XML-RPC library. See "Programming Jabber" (O'Reilly) for some examples of this.

[b]XML-RPC and JSON-RPC are little more than a bit of standardisation for implementing RPC by passing specially-formatted XML/JSON messages between client and server, so a lack of pre-existing client libraries is (slightly) less troubling than for heavier-weight protocols.

Technologies	J2ME	WebOS	Android	S60 native	iPhone	HTML/JS	Java	C(++)	Python
JSON-RPC[7]	no?	unknown	no?	no?	no?	yes, via library	Yes, via (server-side) library		Yes
SOAP	a subset is supported if client implements JSR-172; libraries also exist	unknown	May be possible with library	yes	Google provide a WSDL to Cocoa native code converter tool.	probably	yes	yes	Yes
GENA	One closed source implementation, only over Bluetooth, and not WiFi.				No known implementations.				An incomplete implementation
SSDP	Not possible	no?	known to be possible, but no known library	known to be possible, but no known native library	known to be possible, but no known native library	No	Yes (as part of UPNP control point implementation; may be separable)	Yes (Linux, Windows).	Coherence implements support; separability unclear
other UPNP components[a]	"UPnP Mobile control", a closed-source client, Bluetooth-only. A second client from "adoubleu.de" may not support GENA.			known to be possible, but no known native library	known to be possible, but no known native library. Some Applications are available.		yes (c++, client), yes (C, linux library), etc	yes (details uncertain)	
Bonjour	Not possible	Probably not	Yes	Yes	Yes	No	Yes	Yes (Windows, Mac, Linux	Yes

[a]UPNP uses SOAP over HTTP for control, and GENA for notifications, so see those lines for hints.