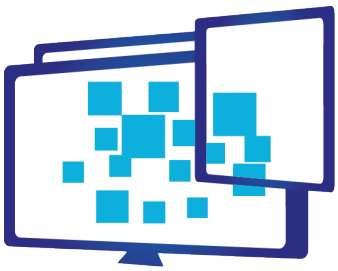

MediaScape
Dynamic Media Service Creation,
Adaptation and Publishing on Every Device

Instrument: Collaborative Project (STREP)
Call Identifier: FP7-ICT2013-10
Grant Agreement: 610404

WP3 – CONNECT – MediaScape multi-connection mechanisms and multi-device authentication

Deliverable Number	D3.4
Title	Multi-connection mechanisms and multi-device authentication
Version	1.2
Date	26 April 2016
Status	FINAL

Restricted

Project Partners: VIC, IRT, NEC, BBC, W3C, NOR, BR

Contributors: **VIC, IRT, BBC, W3C**

Document Control

Version	Date	Author	Modifications
0.1	23 Jun. 2015	Mikel Zorrilla (VIC)	Template and structure
0.1	21 Jul. 2015	Mikel Zorrilla (VIC)	Structure changed
1.0	19 Nov, 2015	Sean O'Halpin (BBC)	First draft
1.1	10 Mar, 2016	BBC	Added content
1.1	10 Mar, 2016	Mikel Zorrilla (VIC)	Formatting and minor changes
1.2	26 Apr, 2016	BBC	Updated following peer review

Table of Contents

1. Executive summary

The goal of WP3 is to bootstrap the environment which enables MediaScape applications. This entails finding suitable devices on the home network on which to deliver the application experience, finding out their capabilities, communicating with them, and enabling other components of MediaScape to control them.

This deliverable documents the work done under WP3 over the course of the MediaScape project in the following tasks:

- T3.1 Discovery and Association Mechanisms
- T3.2 Dynamic Pairing of Multiple Resources
- T3.3 Multi-device Authentication

We cover the research and exploratory work done, the prototypes we built to assess both existing technologies and our own, overviews of the APIs and libraries we have developed, and our findings and conclusions.

For T3.1, we have investigated technologies for discovering devices on a local area network and querying device capabilities, as input to the media adaptation developed in WP5. For T3.2 we have looked into several approaches to pairing and control of local devices, with a focus on browser-based solutions that preserve end-user privacy and security. We have also developed an API that allows a MediaScape application on one device to invite another device into the same application session, using a selection of pairing technologies. For T3.3 we have developed a new OAuth 2.0-based protocol for user authentication suitable for media devices that have limited input capability, such as connected TVs and radios.

For detailed descriptions of the APIs provided by the libraries and protocols developed within WP3, please refer to D3.3 *Final version of multi-connection mechanisms and multi-device authentication*.

2. Introduction

2.1. Background

One of the main goals of MediaScape is to explore the potential of coordinated user experiences across a number of devices. For this, we need to create a distributed system of software agents that work together to enable applications to provide those experiences.

The goal of WP3 is to bootstrap the environment which enables that platform. This entails finding suitable devices on the home network on which to deliver the application experience, finding out the device capabilities, and enabling other components of MediaScape to control them. From the Description of Work, WP3 “define[s] the mechanisms that will enable multiple devices to discover services, pair with each other and establish a unified, authenticated, networked application context for the distributed applications developed in later work packages”.

At the architectural level, we have decided to interact with devices via software agents which manage those devices (querying their capabilities and controlling them). This enables us to present a simpler, uniform abstraction to other MediaScape components.

Based on an analysis of our use cases, described in section 2.2, we have identified the need to solve the following three problems:

- how to find the agents managing devices on a local network
    - discovery (how to lookup agents)
    - naming (how to name/identify each agent)
    - capabilities (how to determine what each agent can do)
- how to use / control them remotely (i.e., device pairing)
    - how agents communicate with each other
    - how a user can use them
- how to link agent actions to a user account (authentication and authenticated association)

2.2. Scenarios and requirements

Requirements were driven by the use cases described in D2.1, and the use cases that specifically inform WP3 are (from D2.1 Section 5.5 *Requirements Summary*):

- 5.1.1 UC1: Single user with two devices and extra info
- 5.1.4 UC4: TV and newspaper
- 5.1.5 UC5: Market place
- 5.1.6 UC6: Second user pairing
- 5.1.8 UC8: Audio exchange and synchronisation

- 5.2.1 UC9: Starting a quiz
- 5.2.3 UC11: Quiz with a group of friends
- 5.3.1 UC12: Radio discovery via browser
- 5.3.2 UC12B: Radio control from a tablet / smartphone
- 5.3.4 UC14: Multi-radio control and context
- 5.4.1 UC16: Personalised TV programme guide
- 5.4.2 UC17: Sharing TV viewing information in a social group
- 5.4.3 UC18: Sharing a link to a live TV programme
- 5.4.4 UC19: Controlling a TV from a web page: play on-demand media
- 5.4.5 UC20: Controlling a TV from a web page: change channel
- 5.4.6 UC21: Commenting alongside a live TV programme
- 5.4.7 UC22: Commenting alongside an on-demand TV programme

We can group the requirements arising from these use cases by function as follows:

Device discovery

- the PC discovers a connected radio (5.3.1, 5.3.2, 5.3.4)
- a new radio in the kitchen is discovered (5.3.4)
- the web page is able to discover a TV device belonging to the authenticated user (5.4.4, 5.4.5)

Service discovery

- service related to TV programme (5.1.1, 5.1.4, 5.1.8)

Capability discovery

- find a device capable of playing radio (5.3.1)
- find TV (5.4.4, 5.4.5)

Device pairing / association

- pairing TV and tablet (5.1.1)
- tablet controls TV experience (5.1.5)
- new device associates with the TV and with the service (5.1.6)
- there is an association between devices (5.3.1)
- Andy associates and uses now the new radio and the smartphone (5.3.4)
- remote control (5.4.4, 5.4.5)

Local network discovery, pairing and control

- should work on home LAN without Internet connection (5.3.2, 5.3.3, 5.3.4)

User authentication

- Peter and Eddie are authenticated in the quiz so they can find friends and can be found (5.2.3)
- the Web application requires the user to sign in to view personalised content (5.4.1, 5.4.2, 5.4.3-7)

By grouping these functional requirements by domain, we can derive several broad categories of system requirements.

From the requirements listed above under **Device discovery**, **Service discovery**, and **Capability discovery**, for devices to be able to detect each other automatically and dynamically, and find each other by function, we can derive the need for a service discovery protocol that incorporates presence, addressability and capabilities.

From the requirements under **Device pairing / association** we derive the need for a communication protocol that enables agents to send commands to each other (remote control), query each other (find out what programme is playing) and to raise events (notifications).

From the requirements under **User authentication**, and throughout the use cases wherever the term 'personalized' is used, we need the ability to link a client device to a user identity, hence authentication.

2.3. MediaScape architecture

From the WP3 point of view, a MediaScape application runs via one or more agents. An agent is a software component that is able to participate in MediaScape applications, discover and communicate with other agents on the networks, and mediate access to the capabilities of the underlying devices on which the agents run.

As described in D3.2, "MediaScape prefers web browser based agents where possible. First, because native agents developed on top of specific OS SDKs, and hardware APIs must be implemented and built for each kind of target device and OS, requiring continuous upgrades as the OS updates or a new model of the device comes out. Second, most Web browser agents are based on W3C standards or drafts, widening the range of suitable devices and increasing future applicability."

The WP3 stack is depicted schematically in Figure 1.

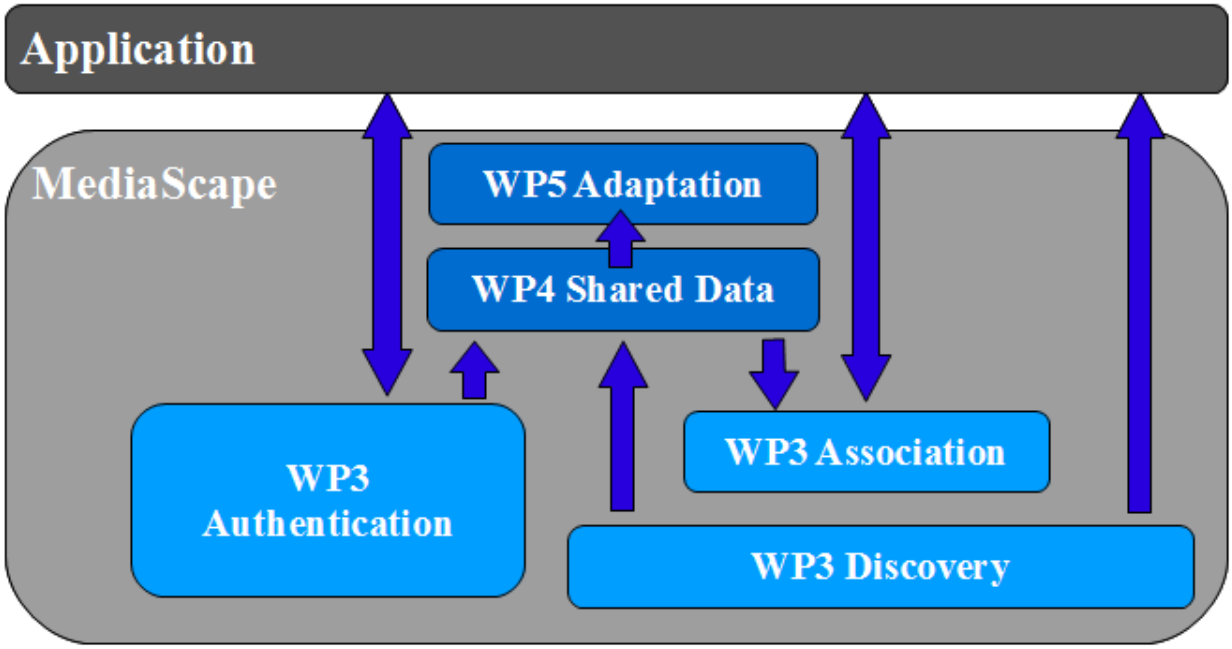

Figure 1. WP3 high level architecture

Discovery provides information on the presence of local nodes to the upper layers of WP4 and WP5. For WP5, it also provides information about capabilities and current agent state (communicated via the shared state provided by WP4) needed to provide adaptive user interfaces. Authentication provides authorization tokens for the application and to grant session access by WP4.

The *Discovery API* provides the means to query and represent the capabilities of devices as discussed in D2.2 section 6.2 *MediaScape Device Capabilities*.

The *Association API* enables agents to invite other agents to join a MediaScape application session, which allows them to share state, providing the basis for distributed applications.

EBU *Cross Platform Authentication* (CPA) is an open, standard authentication protocol for IP-connected media devices based on OAuth 2.0 [1]. It was developed largely under the auspices of the MediaScape project. This protocol provides a common basis for device manufacturers and content providers to offer personalized services and can be implemented by IP-connected media devices with limited display and input capabilities.

2.4. Multi-connection mechanisms and multi-device authentication approach

Our overall approach to WP3 has been to explore the issues raised, assess the suitability of the technologies identified as relevant by the review of the state of the art (D2.2 sections 4.3 *Discover and Publish a Service* and 4.4 *Authentication*), to identify the gaps in the current state of the art, and to devise possible ways of filling those gaps.

For our work on discovery and pairing, we did not attempt an a-priori specification of the architecture of WP3. First, we needed to investigate existing or proposed standard technologies to determine which could satisfy all the requirements of the target MediaScape scenarios. We did this in two ways: a literature review and then by creating technical prototypes.

At the start of the project we conducted a literature survey of existing device discovery and pairing technologies [2]. The goal was to provide an overview of some of the previous work in this area, in order to focus our efforts. We started from the state of the art as described in the MediaScape Description of Work, and covered discovery techniques used by UPnP, mDNS, and various applications that use them, such as Chromecast, Airplay, as well as more experimental techniques. The literature survey also describes related techniques used in NFC and Bluetooth, and various approaches for out-of-band linking of devices, with links to various W3C and other relevant specifications.

After this, we implemented technical prototypes using different candidate technologies to better decide which technologies to adopt. These are also described in this document.

The technical evaluation and prototyping work done in year one led in year two to:

1. The development of a set of loosely-coupled JavaScript APIs being developed to abstract over several discovery, pairing, and association technologies, as well as integration with components from WP4, such as the Shared State. These are demonstrated in the prototypes developed for WP6, which combine and integrate various technologies from each of WP3, WP4, and WP5.
2. Contributions to standardisation efforts in the area of device discovery and pairing at W3C, notably the Presentation API within the Second Screen Presentation Working Group. The research and JavaScript APIs developed by MediaScape served as input for discussions in that group.

In contrast, our work on authentication has taken a different path due to the fact that we were able to build on the extensive prior work in this area done by the BBC on RadioTAG [3]. Because we had already been through the exploration and prototyping stages, we were able to work directly on developing a protocol specification (Cross Platform Authentication [4]) and the supporting reference implementations.

These approaches have allowed us to experiment with and iterate APIs, and then build the application prototypes in Work Package 6.

2.5. Structure of the deliverable

This document describes the investigations, prototypes, and technologies we have built to explore specific aspects of the problem space covered by WP3.

As work progressed during the course of the project, it became apparent that there was a significant overlap between tasks T3.1 Discovery mechanisms and T3.2 Dynamic pairing of multiple resources. For example, some of the technologies we have worked on, such as Named WebSockets and the W3C Presentation API, encompass discovery, pairing, and inter-device communication.

In this deliverable, we therefore present the technologies we have investigated and developed in the following structure:

- Section 3 discusses detection of device capabilities, of individual devices, as well as other devices discovered and available on the network. This is implemented in the Discovery API. This relates to task T3.1 Discovery mechanisms.
- Section 4 presents several approaches to discovery, pairing, and communication with devices on a local area network, with a particular focus on browser-based approaches. These all relate to both T3.1 Discovery mechanisms and T3.2 Dynamic pairing of multiple resources.
- Section 5 presents the Association API, an abstraction over several different ways of sharing MediaScape application session URLs between devices. This relates to task T3.2 Dynamic pairing of multiple resources.
- Section 6 describes the approach to user authentication in MediaScape applications, in particular the Cross Platform Authentication protocol that we have developed to address the particular needs of connected TV and radio devices. This relates to task T3.3, Multi-device authentication.
- Section 7 describes other technologies that are useful for multi-device Web applications, but do not fit neatly into one of the WP3 sub-task headings.

Each section starts with an overview of the problems we have been trying to solve, the technologies we investigated and an explanation of our approach. For each technology or prototype, we explain its purpose (Goals), what aspect of the WP3 architecture it is intended to explore (Relation to the MediaScape Architecture), any relevant standards, what the component does and how it works (Definition), and lastly an overview of the implementation (Implementation). Details of any APIs exposed are available in D3.3 *Final version of multi-connection mechanisms and multi-device authentication*.

3. Device capabilities

3.1. Discovery API

The Discovery API was developed by Vicomtech-IK4.

3.1.1. Goals

In order for MediaScape applications to make use of the facilities provided by a multi-device environment, it is essential to make the agents managing those devices aware of the context in which they operate.

To achieve this, agents need to be aware of the features provided by the devices they manage and of the presence of neighbouring agents. These discovery operations cannot always be directly performed from the web browser. For certain features and devices, we need agents that can retrieve the target information through a specific application stack.

The devices targeted by MediaScape are highly heterogeneous. For any group of devices, some will support UPnP and some mDNS for network discovery and control, and some will only have proprietary APIs. Each will have different individual features suitable for applications of various types (for example large screen size, camera, proximity detection). It is very important from the perspective of a MediaScape application developer to have a common API that abstracts the underlying technologies and internal workflows.

Discovery activities can be grouped into two categories:

- Device capabilities that expose the available features of a specific device
- Devices or services reachable through a specific protocol or communication interface

This section describes our API for interfacing with discovery agents and controlling those agents.

3.1.2. Relation to the MediaScape architecture

The Discovery API provides several fundamental aspects of WP3's contribution to the MediaScape architecture:

- T3.1 Discovery: implements one of the underlying existing discovery mechanisms (SSDP) and provides HTTP and JavaScript interfaces
- T3.2 Pairing: Proposes communications methods, message formats and control protocols

3.1.3. Relevant standards

These are the most relevant standards and draft specifications used for the implementation of the Discovery API:

- W3C Geolocation API [5]
- W3C Screen Orientation API [6]
- W3C Vibration API [7]

- W3C Battery Status API [8]
- W3C Proximity Events [9]
- W3C Network Information API [10]
- W3C Media Capture and Streams [11]
- Universal Plug and Play (UPnP) [12]
- RFC 6762 Multicast DNS [13]
- RFC 6763 DNS-Based Service Discovery [14]
- RFC 6455 The WebSocket Protocol [15]

3.1.4. Definition

The aim of the Discovery API was to define a universal and open interface to detect new agents in the same local area network (LAN) or in a co-located area. This involves two distinct forms of discovery:

- The discovery of MediaScape devices and services on a LAN, using technologies such as mDNS and UPnP SSDP [16].
- The discovery of an agent's particular capabilities, such as screen size, and availability APIs for geolocation, camera, etc. These attributes enable adaptation of the application or media to the characteristics of participating devices. See D5.4 for more information on adaptation.

The discovery operations cannot always be directly performed from web browsers. For some specific features, the need for agents that retrieve the desired information through a specific application stack is fundamental. The implementation of web browser Agents and Native Agents solve the problem of information acquisition.

- Web Browser Agents: W3C specification based JavaScript functions that use the abilities of the web browser to know and operate its underlying hardware and software stack.
- Native Agents: These use native code for discovery and introspection and publish the results in a uniform format over an HTTP interface.

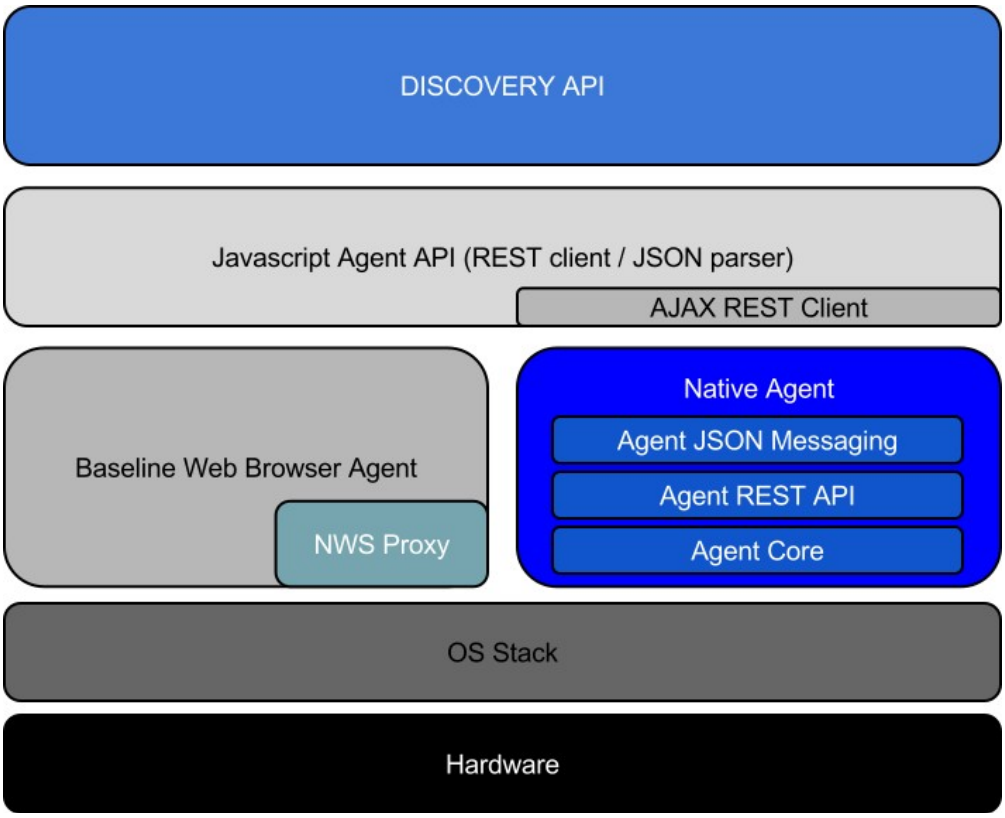

Figure 2. Discovery API participants

The MediaScape JavaScript Discovery API hides the complexity of the HTTP interface. This way, the developer does not need to take care of HTTP interactions, and just needs to execute JavaScript calls.

The Discovery API interfaces with underlying Agents, native or web-browser based. In order to create a uniform JavaScript interface, we have defined a common REST web service for local native agents. The results are formatted as JSON. This way, the Discovery API just has to perform some HTTP requests to collect information from native agents.

3.1.4.1. Service and device discovery

The following technologies discover the list of devices or services that are in the LAN or in a co-located area:

- **Named WebSockets** [17] An association technology that provides device pairing in an automatic manner. Before making an association, it is important to detect if there are any devices available.
- **UPnP** In the context of TV, It is important to provide a system to get devices that offer UPnP services in the LAN, because most of the services related to TV are published using this technology.
- ***Bluetooth*** In the context of mobile devices, most peripheral devices use this technology for pairing. Those peripherals, e.g., headphones or speakers, in

combination with the mobile device provide the user with an immersive experience and therefore a better user experience.

3.1.4.2. Device capabilities

The following capabilities are the most relevant for MediaScape applications. These capabilities provide useful information to enable content adaptation, and thereby improved user experience:

- ***Screen*** The detection of device screen size is essential to adapt application components to the screen. This knowledge permits the best fit of those components to the different devices in an automatic manner.
- **Geolocation** In terms of TV context it is important to define a device's location to discriminate between the places where a content is private and where is public. Furthermore, for the Shake & Go association system (see section 5.1 on the Association API) it is necessary to use geolocation as an element of the pairing process.
- ***Orientation*** Mobile devices switch content orientation from landscape to portrait thanks to the detection of orientation changes of device's accelerometer sensor. Application developers have to detect this type of changes to adapt the content that is being displayed.
- ***Camera*** One of the most frequently used association systems, QR codes, requires the device to have a camera. See section 5.1 on the Association API.
- ***Vibration*** Use of the vibration sensor combined with notifications can improve application usability, which is why awareness of this sensor availability is important.
- ***Battery*** The knowledge of the battery level provides to the developer the ability to create notifications to prevent unexpected shut-downs because of low battery. Providing this information improves the user experience.
- ***User proximity*** Some mobile devices contain a user proximity sensor that detects if the user is close to the device. This sensor provides to the system a signal to detect if the application has to be stopped because the user is taking a call, or because there are objects that do not allow the screen to be seen.
- ***Language*** The detection of language is a very important aspect to have in mind. It will allow the system to adapt the content, such as subtitles or other text, to the same language as the browser.
- ***Device type*** In the development of the adaptation algorithm, we found the need to provide the device type as an input, e.g., whether the device is a smartphone or HbbTV, etc. Identifying the device type allows adaptation of the content, and the activation of device-specific features or components.
- ***Connection type*** In the development of the WP5 adaptation algorithm, we found the need to obtain information about the type of connection the device uses to connect to the Internet. The connection type allows the adaptation algorithms to define the adequacy and suitability of components used in the web application to

devices. If the connection is not fast enough to play video in the most appropriate way, the algorithm can avoid playing the video, or play it on another device inside the LAN that is associated with the same application session.

3.1.5. Implementation

Two different implementations have been developed on top of the library.

The first, a *discovery-self* implementation has been made to discover all the capabilities and features available by the Web browser.

The second, a *discovery-multi* implementation, for use in multi-device applications, where all the devices in the same session are aware of the other devices and their capabilities. All the different instances of the application share a context, where each individual browser uploads and updates the discovered capabilities and features that are available by the Web browser. This way, each device is aware not only of its own capabilities but also the capabilities of all the devices in the same session.

3.1.6. Links

Discovery API: discovery-self implementation
https://github.com/mediascape/discovery-self

Discovery API: discovery-multi implementation
https://github.com/mediascape/discovery-multi

3.2. Android REST service

This prototype was developed by Vicomtech-IK4.

3.2.1. Goals

The main idea of this prototype was to create a UPnP server that was accessible via JavaScript. For this we decided to use a RESTFul server and agent concept. The RESTFul server provides information about Bluetooth devices and information about device capabilities via a native implementation. It also generates Android notifications.

Note that another approach to discovery and control of UPnP devices is presented in section 4.1 *SSDP Service*.

3.2.2. Relation to the MediaScape architecture

The Android REST Service provides several fundamental aspects for the discovery of Android device capabilities in WP3's contribution to the MediaScape architecture:

- T3.1 Discovery: implements a REST service for the discovery of Android devices capabilities accessible by JavaScript APIs through AJAX technology.
- T3.2 Pairing: Provides the installation and configuration for Named WebSocket proxy [18] in Android devices.

3.2.3. Relevant standards

The Android REST service implements the following standards:

- Universal Plug and Play [12]

- Bluetooth

3.2.4. Definition

The Android REST service provides a MediaScape agent for getting UPnP services, detecting Bluetooth devices, querying device capabilities, and creating Android notifications.

3.2.5. Implementation

The implementation of an Android service for obtaining device information, and detecting other devices and services through the web browser is a double challenge. Firstly to collect native information, and secondly to provide a way to make this accessible to the web browser.

We have implemented a daemon that works in the background and starts when the operating system starts.

The collection of device capability information is straightforward, because Android provides native APIs to collect that information. Android also provides an API to use the Bluetooth stack, but in case of UPnP services we used a specific library called Cling [19].

The issue of making that information available is solved by creating a RESTful service by using the Restlet library in combination with AJAX technology. The Android RESTful service works as an intermediary service between a JavaScript API and the device operating system.

Furthermore, It was necessary to implement an automatic way to install a Named WebSocket Proxy. The Android RESTful service does the installation and device configuration in rooted devices.

3.2.6. Links

- Android REST service

  https://github.com/mediascape/discovery-self/tree/master/complements/discovery-agent-REST

4. Discovery and pairing

Discovery and communication with local devices is not yet a standard feature in Web browsers. In an ideal world, this document would describe client-side libraries that could be used directly in web pages, which would be acting as "polyfills", i.e., JavaScript libraries that mimic a future standardized API.

In practice, native (as opposed to browser-based) components are almost always necessary because of the requirement to bridge existing protocols, as outlined in D3.2. This can be because UDP-based networking protocols are not supported by browsers at all, which prevents the most common protocols (such as mDNS and SSDP) from being interfaced in JavaScript. Also, bridges to older components of all kinds will usually require some sort of native component, because either the interfaces to these components are not HTTP-based (e.g., many examples of media-playing software have socket-based interfaces only), or they do not support cross-domain requests, so even HTTP calls (e.g., to RESTful services or SOAP services) won't be possible within a browser. The only case where proxies or native components are not required is where the functionality already exists in the browser.

We have taken two slightly different approaches to this problem:

- Build a separately-installable native proxy or "Agent" between the JavaScript component and the network that sits either on the network or on the device. Examples of this are the SSDP Service (section 4.1), the Named WebSocket proxy (section 4.2), the Presentation API, where we have built a polyfill implementation (section 4.3), and the Chrome app mDNS proxy (section 4.4).
- Build a more integrated native proxy, so that the proxy and JavaScript client library are installed at the same time. The second approach makes installation simpler, but also more tightly coupled. An example of this is the RESTful Discovery API (section 3.2).

4.1. SSDP service

This prototype was developed by IRT.

4.1.1. Goals

One of the principles of MediaScape is the requirement to interface with existing standards such as UPnP [12] and DIAL [20] which are based on SSDP [16]. An important responsibility for T3.2 is the provision of bridges to existing protocols such as UPnP and DIAL, to enable devices using those protocols to act as agents in the MediaScape system. DIAL is interesting since it could be used for launching Applications on HbbTV 2.0 devices.

The SSDP Service is intended to enable interoperation with existing consumer media devices supporting the UPnP or DIAL protocol. UPnP and DIAL uses SSDP as discovery mechanism. After a device is discovered by SSDP all the communication for DIAL or UPnP can be done on the client side. The prototype described here is a preliminary step towards realising this goal.

In this section, we describe a prototype of a UPnP Control Point which runs in a state of the art web browser and enables access to UPnP devices via a proxy service.

The prototype demonstrates access to the UPnP stack for device discovery and device control from a web browser, using the draft W3C Network Service Discovery API [21]. An example application of this is the ability to display a list of media files residing on a UPnP Media Server in the browser and select one to play on another UPnP renderer on the local network.

In addition, the SSDP Service makes it possible to discover HbbTV 2.0 devices and launch HbbTV applications on the device from a web browser.

4.1.2. Relation to the MediaScape architecture

The SSDP Service sits between a MediaScape application and existing consumer media devices supporting the SSDP protocol, acting as a translator between the two protocols. As such, it is a potential participant in any scenario that requires access to devices and their capabilities.

The prototype SSDP Service specifically investigates the following:

- discovery (T3.1)
- pairing and control of devices (T3.2)

The proxy component acts as the MediaScape agent for all the SSDP devices on the LAN. This could be any DIAL device like a HbbTV 2.0 device or a UPnP devices.

4.1.3. Relevant standards

The SSDP-Service implements the following standards, and draft specifications:

- Simple Service Discovery Protocol/1.0 [16]
- Universal Plug and Play [12]
- DIAL - DIscovery And Launch [20]
- Cross-Origin Resource Sharing, W3C Recommendation 16 Jan 2014 [22]
- Network Service Discovery, W3C Working Draft 20 Feb 2014 [21]

4.1.4. Definition

Running a UPnP Control Point or a DIAL controller inside a modern web browser requires us to solve two main problems:

1. How to discover devices on the local network from the browser environment
2. How to communicate with the discovered devices on the local network

Direct communication between a web page served from the Internet and a device on the home LAN isn't permitted by the Same-Origin-Policy because UPnP devices don't support the Access-Control-Allow-Origin (CORS) header [22]. Even software implementations like XBMC or UPnP Monkey don't currently support CORS headers. This is a hard issue to resolve because classic UPnP is now quite disjoint from current web technologies where the use of CORS is becoming standard practice. It's not clear when (if

ever) off-the-shelf UPnP devices will implement CORS and even if new devices do implement it, the vast number of existing devices already in homes don't support it. So, we need to find a solution that works with devices that do not support CORS.

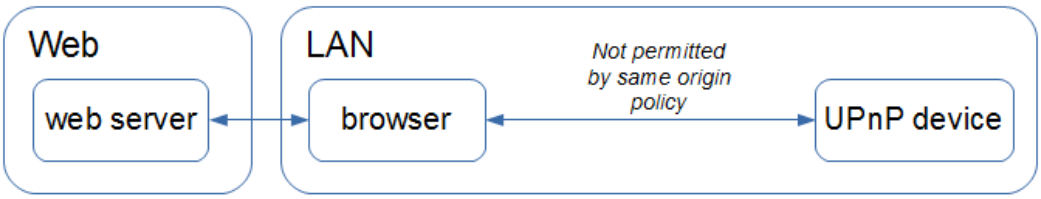

Figure 3. Web pages loaded from the web are not allowed to communicate with devices on the LAN

These two issues can be solved by using a proxy running on the local network which provides the following two services:

1. The service proxy provides information which describes all DIAL/UPnP devices in the current local network in JSON format (including the CORS header). This information is used to produce a polyfill for the W3C Network Service Discovery API.
2. The service proxy enables sending HTTP POST requests by bypassing the Same-Origin-Policy for UPnP/DIAL devices which don't support the CORS header. This is done by POSTing a JSON file to the proxy implementing the CORS header which contains:
    - SOAP header (as per the UPnP specification)
    - the device URL
    - XML message (as per the UPnP specification)

The proxy accepts this POST request, reformats it into a valid UPnP request, sends it on to the target device then forwards the response from the device to the requesting browser.

In effect, the proxy acts as the MediaScape agent for all the UPnP or DIAL devices it bridges to.

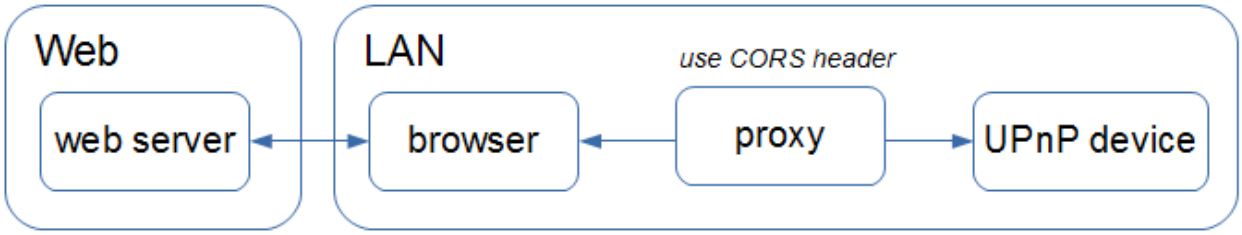

Figure 4. A proxy enables access to devices which do not implement CORS

4.1.5. Implementation

Figure 5 below shows the workflow of the SSDP Service which can be used from a modern web browser running a JavaScript application hosted on a third party web server

somewhere on the web. The web server is not part of this image.

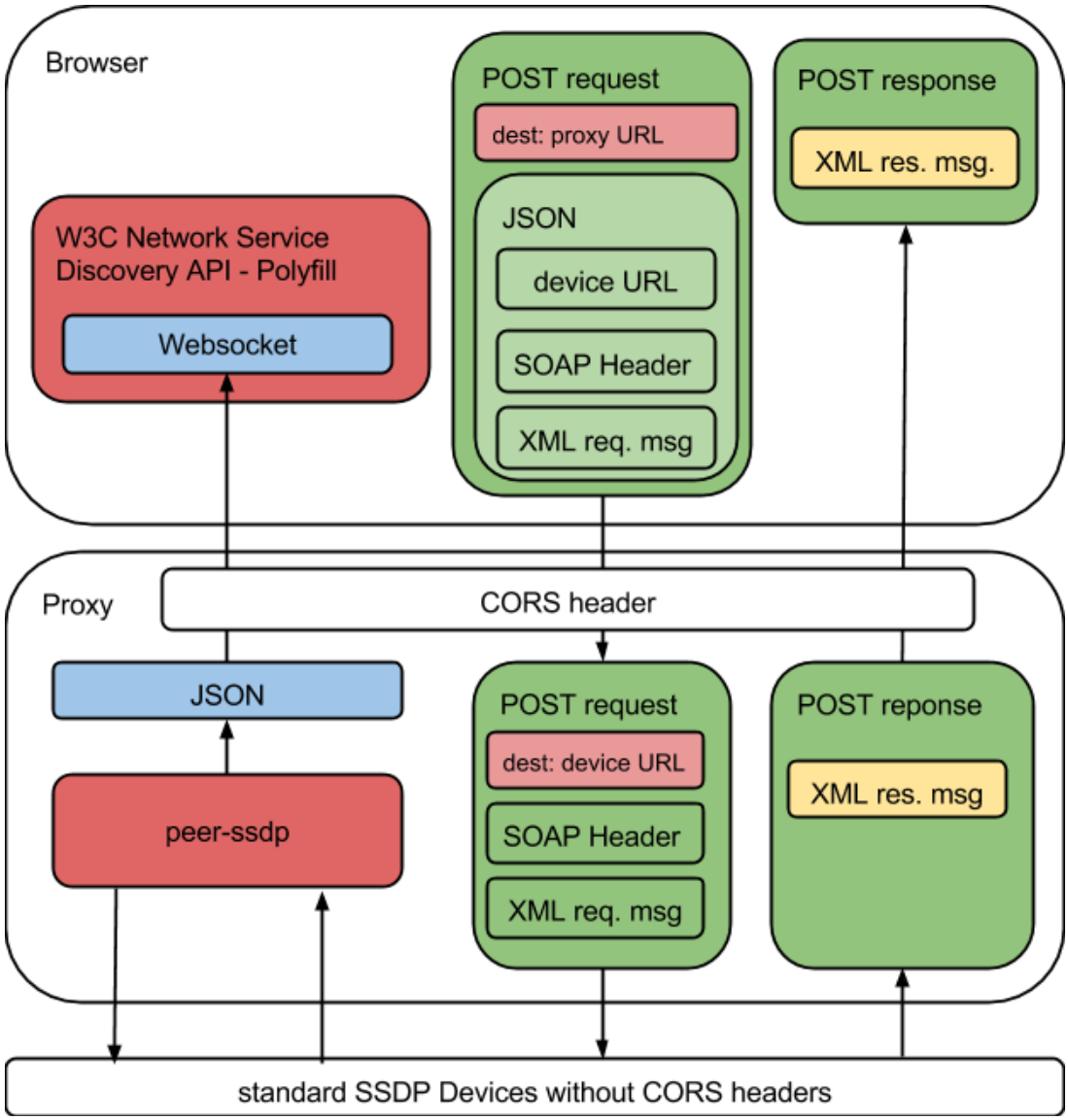

Figure 5. Workflow of the SSDP service

It should be noted that this solution has the disadvantage that any web page can access the proxy and can control any UPnP devices in the local network. This issue is inherent in the current W3C Network Service Discovery API specification [21], which we discuss further in the context of the Presentation API in section 4.3.1.

The prototype is based on a demo from Opera [23] which demonstrates their implementation of the W3C Network Service Discovery API in an experimental version of the Opera browser. Some small changes were made to use this demo with the described proxy solution.

Our JavaScript polyfill for the W3C Network Service Discovery API does the job of

discovery implemented in native code in the experimental Opera browser. The only change to the Opera demo code required was to send commands to the proxy instead of direct to the devices, because of the CORS issue.

The polyfill library implements only those parts of the W3C Network Service Discovery API needed to run the Opera demo. The library connects via WebSockets to the SSDP service. For every discovered SSDP device it receives a JSON message, which contains all information about the discovered SSDP device on the local network. The SSDP device discovery is done by the peer-ssdp framework [24], which implements resource discovery and announcement over SSDP (resource announcement and discovery, description, control, event notification, and presentation).

Our prototype shows that it is feasible to bridge between a Web application and UPnP or DIAL devices. However, the current specification of the Network Service Discovery API raises some serious security and privacy issues [25, 26].

While we can reuse the work we've done in interfacing to the proxy from JavaScript, the fact that work has stopped on the Network Service Discovery API means that other approaches to discovery and communication between Web pages and devices must be considered - for example, the Presentation API described in section 4.3 or the declarative approach described in section 4.4.

4.1.6. Links

- SSDP Service

  https://github.com/mediascape/SSDP-Service

- MediaScape Web view

  https://github.com/mediascape/MediaScapeWebView

4.2. Named WebSocket proxy

This prototype was developed by Vicomtech and BBC.

Named WebSockets[1] are a way for web pages and applications to create a communication channel on a local network based on a channel name and the use of WebSockets. This technology was developed by Rich Tibbett, the editor of the W3C Network Service Discovery specification, when work on that specification stopped.

Named WebSockets can create ad-hoc or full-duplex broadcast channels. They use a peer-to-peer model and so are particularly relevant for use on a local network, since there may be no server available locally to act as a relay.

Named WebSockets are not currently available on devices, so we have deployed an intermediary server, a Named WebSocket Proxy, through which browsers can connect to each other. The Proxy performs the following three functions:

- Broadcast to its channel peers.
- Discover other channel peers.

[1] Named WebSockets are also known as Network WebSockets. In this document we use the term Named WebSockets throughout.

- Establish communication channels between its own channel peers and other channel peers.

4.2.1. Goals

One of the purposes of Mediascape is to perform a solution for dynamic pairing of resources in a specific application session. Named WebSockets create encrypted automatic communication channels through web pages and applications. Furthermore, through these channels it is possible to discover and connect peers that share a common channel name in the local network, besides the messaging. After analysing association technologies we concluded that messaging through Named WebSockets offers a great solution to the problem of sharing the URL for automatically bootstrapping different devices in the same session.

Rich Tibbett has also made available a Named WebSockets Proxy, a proxy server that implements the Named WebSockets protocol. One potential use for this proxy we found, apart from the WebSocket connector, was as an association message receiving notification generator. The necessary existence of one proxy on each device made it the perfect daemon for generating notifications.

4.2.2. Relation to the MediaScape architecture

Named WebSockets provides a fundamental system for automatic association:

- T3.1 Discovery: serves as a way to detect devices connected within the same WLAN/LAN through a Named WebSocket Proxy in a web socket channel.
- T3.2 Pairing: Provides message exchange, session URL, between two devices connected through a WebSocket channel.

4.2.3. Relevant standards

Named WebSockets uses the following standards for discovery and message exchange:

- RFC 6762 Multicast DNS [13]
- RFC 6763 DNS-Based Service Discovery [14]
- RFC 6455 The WebSocket Protocol [15]

4.2.4. Definition

A web page or application can create a new Named WebSocket by choosing a channel name (any alphanumeric name) via any of the available Named WebSocket interfaces. When other peers join the same channel name then they will join all other peers in the same Named WebSocket broadcast network and send each other an association URL. Once the URL is received by a Named WebSocket Proxy, it sends a native OS notification (either Linux or Android), which alerts the user. The user can then choose to join the shared application session by responding to the notification.

4.2.4.1. Integration of the Named WebSocket proxy with the REST service

Two important parts of the Discovery API (section 3.1) and Association API (section

5.1) are the Named WebSocket Proxy and Android RESTful service. The Named WebSocket Proxy allows the discovery and association of different devices connected on the same LAN, while the Android RESTful service allows the discovery of capabilities of mobile devices with Android OSs.

In huge projects less is more. With that aim in mind, we have tried to optimize the installation of the REST service in Android devices and Named WebSockets Proxy binary as much as possible. If the user installs as many things as possible in one, it will be a better user experience than if the application stalls and requests user input. This way the user can concentrate on using the application, instead of losing interest in the application because of the awkwardness of installation. Furthermore, it is not easy for the general user to install and execute external binaries not implemented in Java. To resolve those problems, we integrated the Named WebSockets Proxy binary into the REST service and launch execution of the proxy from an Android service.

4.2.5. Implementation

We have modified Rich Tibbett's code to fit with MediaScape's requirements. With the new code each proxy detects the URLs received by the proxies and sends to the rest of the peers instead to send even to himself. With the received URLs the proxy creates a native notification in Android and Linux operating systems to facilitate the consume of the users. To prevent overwhelming the user with notifications we have also developed a mechanism to notify each URL only once until the proxy is restarted.

We have also developed a pure JavaScript implementation of a Named WebSocket Proxy using Node.js.

4.2.6. Links

- Original Named WebSocket Proxy

  https://github.com/namedwebsockets

- Mediascape Version of Named WebSocket Proxy

  https://github.com/mediascape/discovery-self/tree/master/complements/namedwebsockets-proxy

- Android RESTful service integrated with the Named WebSocket

  https://github.com/mediascape/discovery-self/tree/master/complements/discovery-agent-REST/discovery-agent-REST-android

- Pure JavaScript Named WebSocket Proxy implementation

  https://github.com/mediascape/named-websocket-proxy

4.3. W3C Presentation API

The prototype described in this section was developed by W3C.

4.3.1. Goals

When MediaScape started, the expectation was that local network discovery would eventually be exposed to Web applications. Ongoing efforts to standardise the Network

Service Discovery [21] specification seemed to illustrate that trend. This did not happen. In practice, these efforts stalled out of security concerns [25, 26]: most devices that advertise themselves on the local network through mDNS or SSDP consider that the local network is to be trusted and do not take any sort of measure to protect themselves against external attacks. Such devices should not be exposed to arbitrary Web origins.

Similarly, the discovery process exposes too much information about the devices available on the network, including the ability to list these devices in the first place, as well as information (e.g. the URL) about the local network.

Asking the user for permission works up to a point, but users are unlikely to understand the security implications when they grant permission to discover devices on the network.

The Network Service Discovery specification was updated to address some of these issues, in particular to restrict discovery to local services that support Cross Origin Resource Sharing (CORS) and were thus designed to be accessed by web applications. However, the main issue remained and the general consensus progressively shifted against allowing web applications to list nearby devices.

The Network Service Discovery specification also required web developers to understand the different protocols and message formats to use to communicate with discovered services afterwards, be it through XML messages sent over an XMLHttpRequest or JSON messages passed over a WebSocket connection.

The development of the Presentation API [31] was initiated at W3C end of 2014, supported by MediaScape, out of a desire to take a different approach to discovery and pairing to solve these issues. The development of the Presentation API is still on-going as of early 2016, with early implementations in Firefox and Chrome.

4.3.2. Relation to the MediaScape architecture

The Presentation API explores both the discovery (T3.1) and pairing and control (T3.2) aspects of the MediaScape architecture. The promise of the API being implemented across Web browsers makes it an attractive technology to bootstrap multi-device sessions without having to rely on some third-party proxy service, in particular.

4.3.3. Relevant standards

The Presentation API is currently a Working Draft document from the W3C Second Screen Working Group, and should become a Web standard by end of 2016.

4.3.4. Definition

The workflow of the Presentation API is the following:

1. The web application requests display of some web application (URL) on a second screen.
2. The user agent handles the discovery in the background, using mDNS, SSDP or some other discovery protocol.
3. The user agent prompts the user to select one of the devices discovered.

4. The user agent establishes the connection with the selected device and serves as proxy for subsequent exchanges of messages between the web applications running on both devices.

The immediate benefits of this solution are that the requesting web application never sees the list of devices discovered by the user agent. It does not even know how they were discovered, and does not need to know how to send commands to the other end; it merely speaks to another web application, exchanging messages as if it were using a WebRTC data channel or a WebSocket communication channel.

In fact, hiding anything related to discovery to the requesting web application also enables the so-called one user-agent (1UA) scenario where the user agent renders both the requesting web application and the presented web application, and simply streams the result to the second screen. In other words, this makes it possible to “discover” and use screens attached through a simple HDMI cable or through a Miracast connection. The 1UA scenario is important for browser vendors, although not a priority for MediaScape.

The Presentation API approach comes with a number of drawbacks from a MediaScape perspective, though:

1. It essentially turns discovery into a black box handled by the user agent, and each user agent will support its own set of discovery protocols.
2. It mandates the establishment of a peer-to-peer communication channel between devices, again based on protocols that will vary from user agent to user agent. The notion of Shared State explored in MediaScape in WP4 relies on a more usual client-server architecture and does not need peer-to-peer communication channels.
3. It is restricted to the establishment of one connection at a time with a second screen found through discovery. Research in MediaScape suggests that invitation-based mechanisms (such as QR codes, acoustic codes, broadcast of URLs through Physical Web) is also a valuable mechanism to bootstrap a multi-device session.

In WP3, MediaScape explored the limits and possible extensions of the Presentation API through the development of a polyfill written in JavaScript.

4.3.5. Implementation

MediaScape developed an initial version of a Presentation API polyfill, at first to explore the API itself. This exploration led to the development of the HTML Slidy Prototype described in D6.2 Initial Testing Report, which allowed to provide early feedback to the Second Screen Working Group and improve the draft specification accordingly.

During the second year, as the divergences of approach between MediaScape’s expressed needs and the directions followed by the Second Screen Working Group became clearer, the project iterated on this polyfill to integrate invitation-based mechanisms presented in this document, typically QR Codes and Physical Web.

The main goal was to show how the Presentation API may be built on top of other mechanisms if it did not mandate the establishment of a communication channel, but rather provided an option to let the requesting web application handle the communication

between the devices that join the session, as done in MediaScape. This work was presented to the Second Screen Working Group during one of its face-to-face meetings in Sapporo, Japan, end of October 2015.

The polyfill shows the usefulness of building additional mechanisms into the Presentation API. However, it also shows the limits of that approach. From a technical perspective, the Presentation API is designed to establish one connection at a time, but more devices may join the session when an invitation-based mechanism gets used. The interfaces defined in the Presentation API would need to evolve to cover that possibility.

More importantly, invitation-based mechanisms change the boundaries of the API in terms of privacy: by definition, the goal of these mechanisms is to invite other devices to join a multi-device session by advertising the URL of the Web content to load, and that URL needs to contain some sort of Session ID for devices to actually join the same session. The user may not want to see the URL that she is willing to present exposed to everyone. Also, devices that receive the invitation do not need to authenticate themselves a priori. The authentication of the users and devices is up to the application. The user does not even get to select which device is able to join the session. In other words, invitation-based mechanisms are prone to eavesdropping.

In contrast, discovery-based mechanisms do not expose the URL and may prompt the user to select a device from a list before the connection with that device is established. The user could be warned about these potential issues when the application starts using the Presentation API, but the workflow is not the same: in one case, the user chooses a device, implicitly agreeing to sharing information with it; in the other case, the user chooses a mechanism with fuzzy privacy implications.

This essentially suggests another approach where discovery- and invitation-based mechanisms are exposed under different interfaces. The project notes that Mozilla is investigating a similar idea in their FlyWeb proposal [27], which leaves discovery to an interface that mimics the Presentation API, while applications need to register themselves to send invitations. The project also investigated the compatibility of the Presentation API with HbbTV 2.0 [28] in collaboration with the GLOBAL ITV EU project. Some changes are required in HbbTV to expose the communication channel created by the Presentation API to the application running on the TV, but the interfaces are well aligned otherwise.

4.3.6. Links

- Presentation API polyfill implementation

  https://github.com/mediascape/presentation-api-polyfill

- HTML Slidy prototype

  https://github.com/webscreens/slidyremote

4.4. Control radio from a browser

This prototype was developed by the BBC.

4.4.1. Goals

This prototype implements Use Case 12 Radio discovery via browser (Scenario 5.3.1 in

D2.1). Our goal was to prototype a simpler, more lightweight protocol for discovering and controlling media devices than existing established standards such as UPnP, that can be integrated into web browsers without compromising end user privacy.

To recap the original use case:

Prerequisites

- Radio station website with MediaScape functionality
- One or more radio are connected to the network at home (Wi-Fi, Ethernet), which are discoverable and controllable using MediaScape defined mechanisms
- Laptop is connected to the same network (Wi-Fi, Ethernet)

Use Case

- Andy browses the website of a the Pulse radio station on a PC
- Andy gets a notification, telling he can use the MediaScape enabled radio to listen to the radio station
- Andy accepts and the MediaScape enabled radio starts to play

Requirements

- Device discovery: The browser on a PC discovers a connected radio
- Device pairing / association: There is an association made between the browser and a particular radio device
- Device notifications: Andy receives a notification in the browser saying that there is a radio available
- Capability discovery: The browser determines that the radio can accept audio media URLs to play

We also implement the control volume function specified in UC12B Radio control from a tablet / smartphone (D2.1 5.3.2).

The purpose of the prototype is three-fold:

- to try the user experience to see if it works
- to understand the implementation details of the problem
- to explore the security implications of the various approaches

4.4.2. Relation to the MediaScape architecture

The radio control prototype explores both the discovery (T3.1) and pairing and control (T3.2) aspects of the MediaScape architecture.

In this prototype, a Chrome App [29] performs the function of the MediaScape agent.

4.4.3. Relevant standards

This prototype makes use of the following standardized technologies:

- RFC 6762 Multicast DNS [13]

- RFC 6763 DNS-Based Service Discovery [14]
- RFC 6455 The WebSocket Protocol [15]

4.4.4. Definition

Analyzing the use case, we can see that there are two different ways we could implement it: either by streaming the audio from the laptop to the device, or by telling the radio device to play the stream itself. We chose the latter of these approaches, in keeping with the overall goals for T3.2, and developed a technical prototype, intended to explore how media devices can be controlled programmatically and externally over the network.

With this approach, the initiating device (laptop, tablet, smartphone) need only send a control message to the device, rather than having to maintain a constant streaming connection and so can be freed up to do other things.

We considered two approaches to the question of discovery and control. Firstly, implementing a central broker service on the local network that would discover and control devices. Secondly, allowing the browser to perform discovery and control itself.

In the first approach, the broker service would provide an HTTP or WebSocket API to allow any web browser to discover and control devices without the browser having to be modified or have custom extensions. The broker would itself have to be discovered, possibly by existing at a well-known hostname and port on the network. This “discovery bootstrap problem” arises for any method relying on standard TCP/IP-based web protocols.

In the second approach, a browser extension uses a network discovery protocol to discover MediaScape compatible devices and provides an API to JavaScript applications running in the web context to enable them to interact with those devices.

There are various advantages and disadvantages to each. The broker approach raises security questions about how the broker service could be trusted and whether only one of these broker services could exist on the network. The in-browser approach requires the browser itself to be modified or extended to perform discovery and expose that functionality to developers.

A significant benefit of the browser-based approach is that it does not require a specific application to be running on the network for users on the network to be able to control devices. A user may join a network using a MediaScape enabled browser and control any compatible device on the network.

4.4.4.1. Security model

Security is especially important when giving a web page access to control things on your network. Both because of the potential for a malicious page to do something unwanted with your devices but also because a page could gather information about your context and send that information back to the Internet, breaching your privacy.

The diagram below shows various methods of allowing service discovery along a spectrum from allowing total control of network primitives like TCP and UDP sockets on the left, which is potentially the most insecure, to more secure methods on the right.

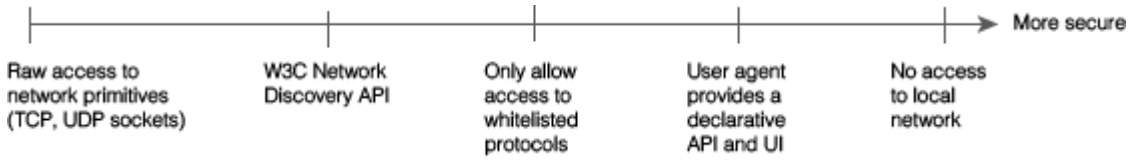

Figure 6. Service discovery security spectrum

For this prototype we have explored what we believe is a more secure but more restrictive model than that proposed by the Network Service Discovery API. In the approach described in this section, developers declare their stream URLs in HTML and the user agent is then responsible for the service discovery and control. This means that the presence of devices discovered on the local network is not exposed to web pages displayed by the browser. New standards such as W3C Custom Elements [30] would allow these elements to be exposed to developers in a way that can be styled and controlled to match each web site.

4.4.5. Implementation

The use of Chrome Extensions [32] and Apps [29] allow us to prototype these use cases and to discover what could be built into the browser and how it should be exposed to developers.

The radio control prototype is composed of three components:

- The **Discovery Helper App** runs in a privileged context and is responsible for discovering services on the network (using DNS-SD) and passing a list of these services to the extension. A "service" consists of basic information such as the network address of the device and a friendly name. The Discovery App does not display any information to users.
- The **Discovery Extension** runs in the restricted web context. It receives the list of services from the Discovery Helper App. The extension displays the list of service to the user in a pop-up when the MediaScape button is pressed. The Extension also detects playable streams embedded in the web pages by looking for elements and shows a list of devices that the stream can be played on and deals with communicating with the device over the network to tell it to play the stream.
- The **BBC Radio Extension** detects when a BBC web page is visited by the user and overrides the functionality of the "Listen Live" button. It injects the stream URL into an HTML element which can be detected by the Discovery Extension. In a MediaScape-enabled world, this extension would not be required as compliant web pages would already contain the required metadata.

Figure 7 below shows the interactions between the three components.

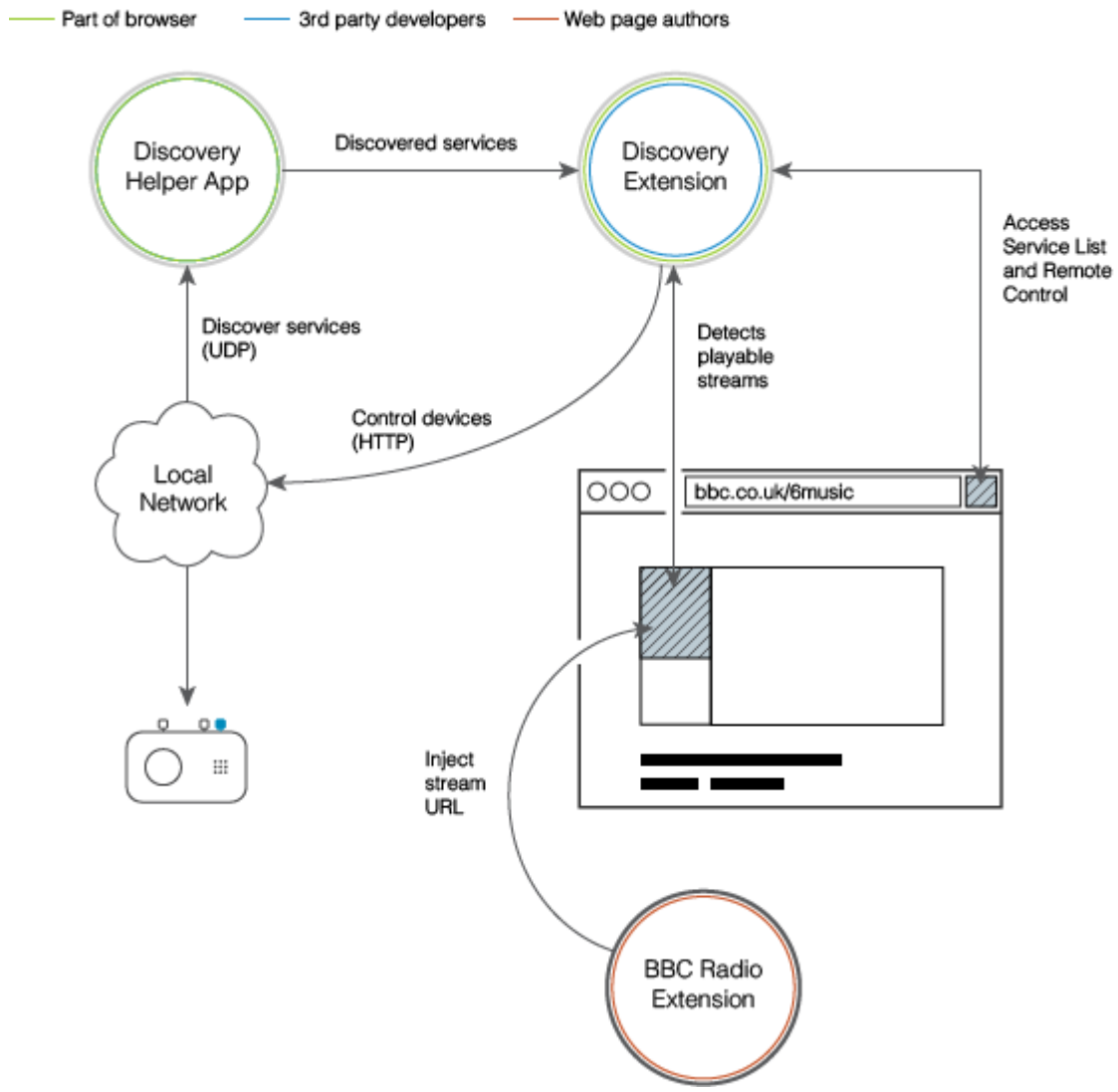

Figure 7. Discovery Chrome extension and helper app

The diagram has been annotated to shows which parts of the application could be:

- made part of the browser (the DNS-SD service)
- created by 3rd party developers (remote control)
- created by web page authors (embedding the MediaScape control widgets)

The radio device itself features a custom HTTP-based protocol for controlling its features (play, pause, volume, radio station selection). The device advertises availability of this protocol using the _mediascape-http._tcp service record.

The prototype is implemented using the Chrome platform which provides APIs that extend the capabilities of a web browser, in particular the ability to run a privileged 'Chrome App' which has access to otherwise restricted machine capabilities such as raw TCP and UDP socket communications.

The three components implement the following functionality:

1. **Discovery Helper App** This is a Chrome App [29] since it requires access to UDP sockets. It contains no user interface and is responsible for implementing the DNS-SD protocol and querying the network for available services implementing the _mediascape-http._tcp protocol. Communication from the app to the extension (2) is done using messages and events via the SendMessage API [33].
2. **Discovery Extension** A Chrome Extension [32] that provides a remote control interface and "send stream to device" functionality.

   Remote Control provides a user interface (UI) to:

   - list discovered MediaScape compatible devices
   - allow the remote control of a selected device using the custom HTTP-based command protocol

   Send stream:

   - monitors pages that are browsed to and finds HTML elements with the attribute data-mediascape-playable-stream=<URL>. The extension then provides a UI to play the stream on a selected device
3. **BBC Radio Extension** A Chrome Extension that detects the "Listen Live" button on BBC Radio web pages, extracts the stream URL and adds it to the button as the data-mediascape-playable-stream attribute.

Together these three components show the user available devices on their local network, allow the user to control compatible devices in a browser window and also send streams from BBC pages to the device. Once the stream is playing on the radio device (our physical radio prototyping platform), it can be controlled via the remote control.

4.4.6. Conclusion

We have produced a usable prototype implementation that shows discovery and control of a radio device from a web page, one of the scenarios we identified in D2.1 (section 5.3.1 and 5.3.2). The prototype implements a possible technical solution and demonstrates the user experience. Having a real implementation to use and experience has confirmed the potential usefulness of the scenario.

On the technical side, the prototype has shown that mDNS is a viable technology for local network discovery, that Chrome is a good platform for prototyping, and that there are important security issues to address when enabling control of local devices by web pages. Our approach of using a data-* attribute [34, section 3.2.5.9] gives the user agent a way to identify playable audio (or video) media URLs in a web page, and allow them to be played on local devices, without exposing any details of the presence of those devices to web pages.

On the user experience side, it has raised some interesting questions about mental models of distributed media control. These models can potentially become confusing when that control can be initiated by web pages, which are transient and can be shut easily.

4.4.7. Links

- Controlling a radio device with MediaScape (blog post)

  http://www.bbc.co.uk/rd/blog/2014/09/control-radio-mediascape
- Discovery Chrome extension

  https://github.com/mediascape/discovery-extension
- Discovery helper Chrome app to bridge to DNS-SD

  https://github.com/mediascape/discovery-helper-app
- BBC Radio Chrome extension to rewrite pages

  https://github.com/mediascape/discovery-bbc-radio

4.5. Physical Web

This prototype was developed by W3C.

4.5.1. Goals

The Physical Web project [35] is an experimental open-source project started by Google that aims at using URLs for locally relevant interactions. It defines a number of mechanisms by which these URLs are broadcast to local devices, which may then choose to follow these URLs where a Web-based user interface for these interactions is expected to be available.

4.5.2. Relation to the MediaScape architecture

The Physical Web project explores both the discovery (T3.1) and pairing and control (T3.2) aspects of the MediaScape architecture.

The goal of the Physical Web project of associating nearby devices via sharing of a URL is very cleanly aligned with the approach to device association that MediaScape has investigated and developed.

4.5.3. Relevant standards

The Physical Web project was initially focused on Bluetooth Low-Energy [36] (also known as Bluetooth 4.0+). It now also supports broadcasting URLs over mDNS and UPnP.

4.5.4. Definition

The expected workflow of a Bluetooth LE broadcast in the Physical Web project is:

- any device with a Bluetooth LE emitter can use the beacon format defined by the project to emit an URL,
- nearby devices that have a Physical Web scanner [37] (either as a native app, or integrated directly in the browser as in e.g. recent versions of Chrome on Android) can detect that URL and offer to the device user to open it, presenting it with useful metadata (e.g. title of the page, final URL in case of redirects).

4.5.5. Implementation

The MediaScape project built a prototype that demonstrates how a MediaScape device

could invite other devices to join a multi-device experience by broadcasting the URL using the Physical Web.

The prototype focuses on the usage of Bluetooth LE as the carrier for the URL broadcast, since local network based advertizing was explored separately.

In the MediaScape prototype, the first device to connect to a multi-device app (e.g. a TV) emits the URL on Bluetooth LE (using the beacon format defined by the Physical Web project) when the user requests to add more devices.

Any device with a Physical Web scanner in range of the Bluetooth transmission will then prompt the user to open that URL. As long as that URL contains sufficient information to join the existing session, any device where the user follows that URL will then become part of the multi-device application.

To make it possible for a browser to broadcast a URL over Bluetooth, two approaches have been explored:

- making the Bluetooth beacon broadcast one of the options for an extended version of the Presentation API as described in section 4.6; in that model, when the user chooses to share the app with other devices, the browser offers among the possible available devices an option to invite any Bluetooth LE enabled device nearby. If the user picks that option, the browser then uses the URL passed as parameter to the Presentation API as the URL broadcasted in the beacon.
- relying on a Bluetooth LE API implemented in browsers; the Web Bluetooth Community Group in W3C [38] has already developed an API allowing to connect with and collect data from Bluetooth Low Energy devices. From a Bluetooth perspective, this means the browser acts as a central device and collects data from peripheral devices. To use the browser to broadcast a URL, the browser would need to act as a peripheral device — adding that capability is in the longer term roadmap of the Web Bluetooth API.

Longer term, the usage of a Physical Web URL beacon for connecting devices hints at further advantages: the Web Bluetooth API roadmap includes bundling the take up of a broadcast URL with a hand-over permission to use data emitted by the peripheral device, thus establishing a potential continuous communication channel between these devices.

4.5.6. Links

The prototype is available on GitHub at:

https://github.com/dontcallmedom/mediascape-physical-web

5. Association mechanisms

MediaScape targets applications that provide shared experiences across multiple devices. Therefore mechanisms are needed by which client devices can join as participants in a shared MediaScape application session. In this section we describe how MediaScape agents join an application session, potentially making use of the Shared Context and Shared State systems from WP4, as well as Adaptation from WP5.

5.1. Association API

The Association API was developed by Vicomtech-IK4.

5.1.1. Goals

Task 3.2 *Dynamic pairing of multiple resources* is addressed by the development of an Association API. MediaScape targets applications that provide shared experiences across multiple devices. The Association API is responsible for establishing the communication mechanisms and protocols to perform dynamic pairing of resources for a specific session on a local network, such as the URL to access an application session (e.g., a WP4 Shared State). It drives the bootstrapping of local multi-device sessions, connecting devices ready to bridge to an application instance.

5.1.2. Relation to the MediaScape architecture

The Association API allows a MediaScape application session URL to be shared, allowing one participating device to invite another device, possibly belonging to another user, into the session. The API therefore acts a bridge between devices and the WP4 Shared Context and Shared State systems. This relates to Task 3.2 *Dynamic pairing of multiple resources*.

5.1.3. Relevant standards

The Association API provides an abstraction over the following standardized technologies:

- Presentation API, W3C Working Draft, 13 Oct 2015 [31]
- ISO/IEC 18004:2015, Information technology - Automatic identification and data capture techniques - QR Code bar code symbology specification [39]

5.1.4. Definition

In the Association API there are two main roles:

- **Trigger** The application instance that wants to share its session to invite others to join the same experience.
- **Catcher** The application instance that wants to join with others to get the same experience.

The selected technologies should have triggers able to automatically detect the catchers, as much as possible, with minimum interaction from the user.

In developing the Association API, we analyzed several possible technologies that

could be used for associating different devices, based on different needs:

- Named WebSockets
- Acoustic Codes
- QR Codes
- Shake & Go
- Presentation API

The Association API must pair, automatically or on demand, reachable devices through possible network interfaces and communication protocols. The idea is to have a common interface and uniform API independent of the underlying technologies, so that a developer can design and develop a service without considering what is happening behind the scenes.

5.1.5. Implementation

In the following subsections we describe how the Association API is implemented, using the various technologies we have developed or investigated.

5.1.5.1. Named WebSockets

Named WebSockets provides the most automatic system of association technologies implemented in the Association API. This is because as soon as one device connects to a LAN where a device is executing a MediaScape application, it receives an association notification. Depending on the needs of the application, instead of just sending a notification, the association can be totally automatic, but for the security of user we decided to use Android notifications.

For example, consider the case where a group of people are associated to a MediaScape application and a new person arrives and connects to the LAN, but this person does not want to associate with the rest of the devices in the application. If the system automatically decides, this person will have to associate the device, but with the notification system the user is able to reject the association process.

The implementation of the association based on Named WebSockets is inspired by the development made by Rich Tibbett and published in the GitHub repository https://github.com/namedwebsockets/networkwebsockets. For the use of this technology, a proxy that acts as an intermediary in establishing communications is needed [18].

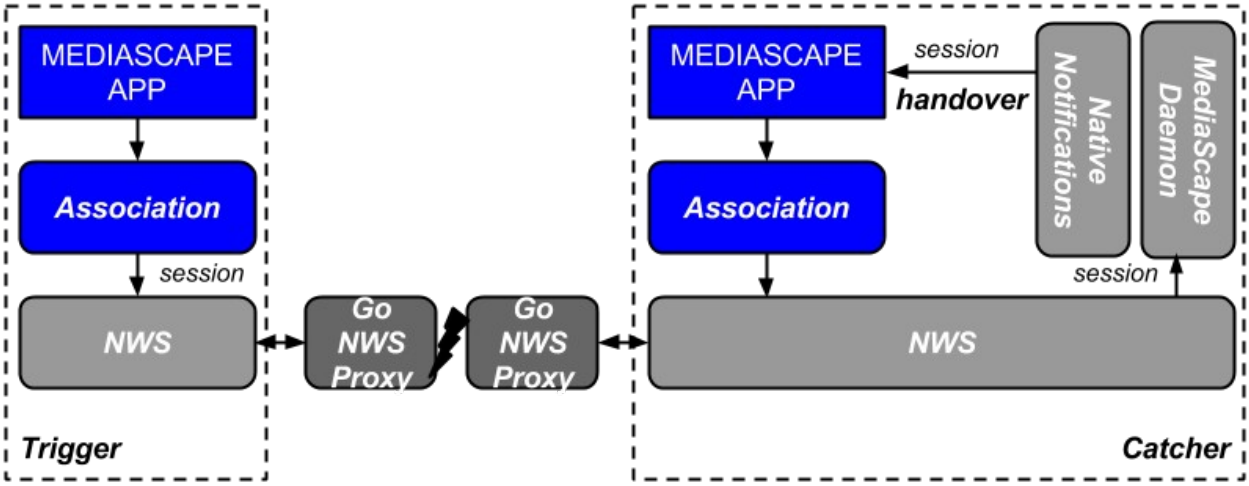

Figure 8. Association API using Named WebSockets

5.1.5.2. Acoustic codes

The use of acoustic codes is becoming more and more popular. While we were developing our association system, Google announced an app for the Chrome browser that allows synchronization of all the computers with that Chrome app executing [40]. One computer generates an acoustic tone from a URL to share via the speakers, and the other computers catch the message using their microphones.

The process implemented by Google is very similar to the concept that we planned at the beginning of the development. The idea was to codify a string into a "song" of tones of different frequencies. Each frequency tone encodes a letter that both the sender and the receiver recognize. The development is based on initial work done by BBC and improved by adding additional security in the emission and reception of the signals, thanks to sending each tone of the signal for a longer time and a fast tracking in a shorter period.

This acoustic system tries to solve the association in the case where a Named WebSockets Proxy or camera for scanning a QR code are not available or cannot be used. The aim of this method of association is to provide to the developer a complementary way of association.

The implementation is based on JavaScript Web Audio API [41] through the AudioContext interface. To obtain the different frequency sounds we have combined the OscillatorNode interface and the AudioContext Gain Control.

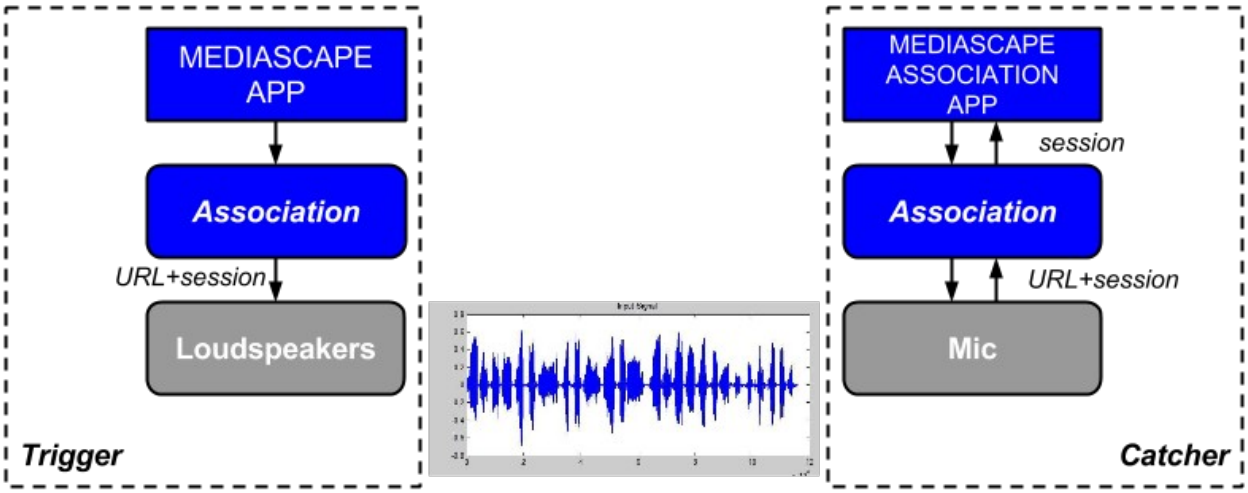

Figure 9. Association API using acoustic tones

Trials to improve the acoustic code

During the implementation of the acoustic association system we realized that for some people the sound of the tones can be a bit unpleasant. For this reason we decided to try to soften the sound by camouflaging the tones with a musical song. However, the technology used and the sensors of the devices were not prepared to admit the mask and the unmask of the tones into a song.

Also, during testing, to improve reliability and enhance the algorithm, we tried using different base FSK frequencies, and changed the bandwidth spectrum division. We also increased the redundancy rate, but we could not avoid the problems generated by the high levels of ambient noise.

5.1.5.3. QR codes

In recent years the use of QR codes has become more widespread, for uses such as tracking parts in vehicle manufacturing, commercial tracking, entertainment, transport ticketing, and product marketing.

From the point of view of MediaScape, the most important use of this kind of labeling is the encoding of website URLs, because the aim of association is the sharing of the URL that provides access to web-based application sessions.

Furthermore, most smartphones have at least one camera sensor installed, which those devices can use to acquire information for further processing. There are some mobile apps that perform this operation automatically: as soon as the camera scans a QR code the app is able to decode the message, be it a vCard contact, a simple text message, or a URI.

There are JavaScript libraries [42, 43] that allow scanning of QR codes and return the decoded information, as well as libraries for creating QR codes. These libraries have enabled us to develop a lightweight implementation of the trigger and catcher.

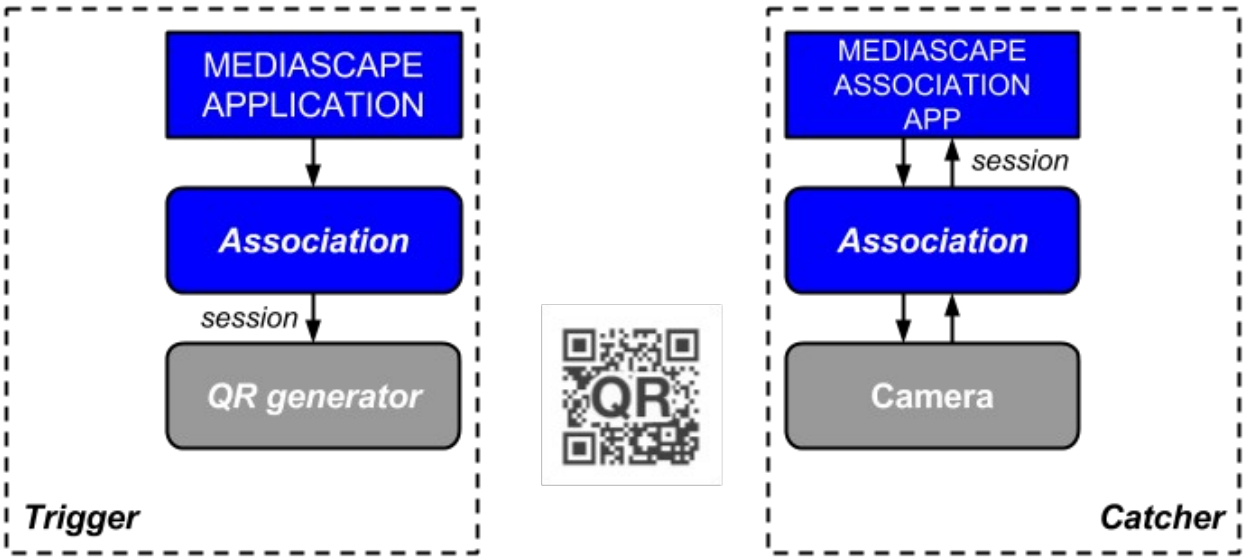

Figure 10. Association API using QR codes

5.1.5.4. Shake & Go

Apart from the most popular technologies mentioned before, we saw the need for a new association method for devices without audio or video sensors, or where the Named WebSocket proxy cannot be used. The aim of the "Shake & Go" method is to provide a complementary association method, based on a timestamp and the geolocation of the devices.

The following diagram shows the architecture of this system.

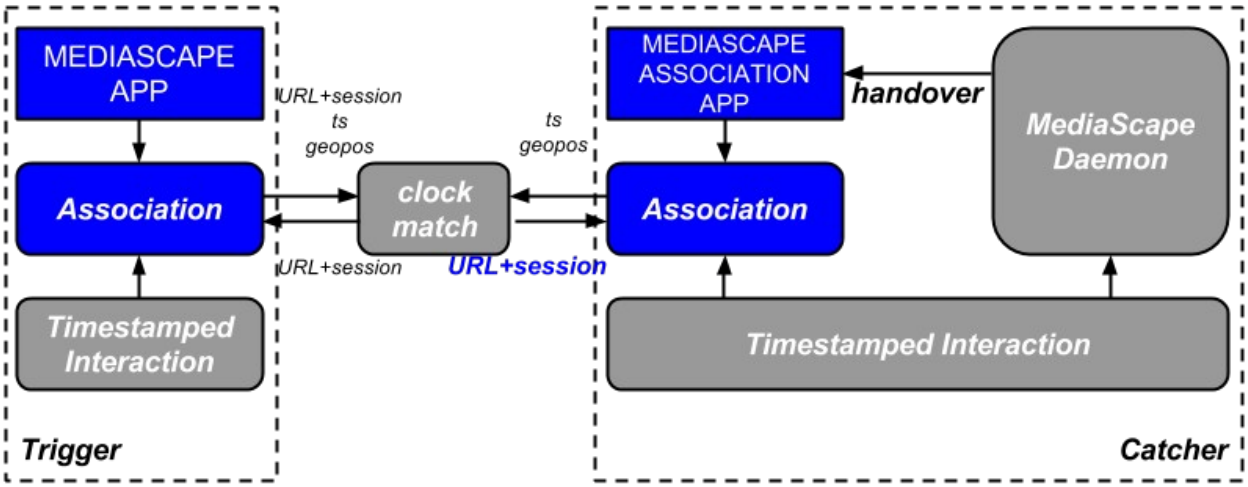

Figure 11. Association API using Shake & Go

Each device creates a request with the timestamp and its geolocation, and sends this information to a central server that centralizes all the requests.

To detect two devices that want to associate, the server discriminates the requests depending on the timestamp information. But the timestamp is not enough to determine which two requests are intended for the same session, because a popular application could generate many such requests at the same time. For those cases we determined that a good differentiator could be geolocation.

Based on the trials that we have done, we can say that the combination of those two parameters seems strong enough to detect the requests of two devices that want associate in the same session.

At the beginning of the Shake & Go association process we thought about several alternatives for obtaining a timestamp from the different devices. The first and most obvious option was to use the system time of each device. The problem with this is that this time is not synchronized between devices. In some devices the time is taken from the Internet, and in others it is set by the user. Another possibility was the use of NTP for JavaScript, but it supposed a continuous synchronization of the time and it is impossible to perform for an API that just is called when necessary. Finally, to obtain a satisfactory timestamp that could be available for any kind of device and browser and that maintains synchronization, we used a third-party server that provides time information [44]. This way we ensure that the same time source is used by each device, and when any device starts the association process the timestamps will be based on a synchronous time.

Trials to improve the Shake & Go process

There are plenty of ways to launch an application on mobile devices, and as we explained before in Shake & Go association process we selected the shake of the devices. Once the launch of the application has been done, the way to associate the devices is based in a timestamp and the geolocation of the devices.

Instead of using that information for the association of devices on Shake & Go processes, because of a fault in some devices' GPS sensors, we also tried to track movement patterns to detect devices that made the same pattern. This way it is easier to detect and distinguish devices that are together and want to associate with the same session. The tracking of device movement has to be done using Web languages like JavaScript and HTML. A problem we found in practice was that each accelerometer had a different calibration, which made it difficult to track all the movements and design a conclusive method for distinguishing pairs of devices in the incoming requests.

In addition, this association method still has open questions, like the control of the speed, the scale or the rotation of the drawing pattern, which arise because the patterns produced by different people can vary significantly.

5.1.5.5. Presentation API

The W3C Second Screen Presentation Working Group has developed the Presentation API, as described in section 4.3. The aim of this API is to enable web content to access external presentation-type displays such as projectors or connected TVs, available to the Web and takes into account displays that are attached using wired (HDMI, DVI, or similar) and wireless technologies (Miracast, Chromecast, DLNA, AirPlay, or similar) and use them

for presenting web content. In the present section we describe the integration of the Presentation API into the MediaScape Association API.

We have focused in particular on the association of devices connected to a Chromecast. As there are currently no browser implementations of the Presentation API, we have integrated a polyfill for this API into the Association API. The polyfill implements the Presentation API in the Chrome browser, with Chromecast devices. In future, when the Presentation API is implemented more widely, we expect to see interoperability between browsers and devices from different vendors.

5.1.6. Links

An implementation of the Association API can be found in the public MediaScape GitHub repository at https://github.com/mediascape/association.

6. Multi-device authentication

To enable a user to access a personalized service across multiple devices, those devices need to reference a shared user identity. To establish this shared identity, we need to create an authenticated association between each device (or, more precisely, the MediaScape agent running on the device) and an online user account. To determine what the user is allowed to do, we need authorization.

There are many existing standards for authentication and authorization to choose from, none of which is immediately suitable for our use cases without some modification or extension, mainly because they generally assume that the device to be authenticated or authorized comes equipped with a screen and a keyboard.

From our review of authentication methods, in D2.2 *State of the Art and Initial Architecture*, section 4.4, we concluded that OAuth 2.0 [1] provides the best toolkit to base an authentication and authorization architecture on, though the lack of any single definitive set of protocols means we are forced to design our own protocols using the elements of OAuth 2.0 rather than simply adopt an existing set.

The Shared Data and Mapping Service implementations from WP4 both use OAuth 2.0 for authentication and authorization. However, OAuth 2.0 does not include an authorization flow suitable for media devices with limited input capabilities, such as connected TVs and radio receivers, nor for devices with limited display capabilities, such as radios. This gap is addressed by the Cross Platform Authentication (CPA) protocol, which we describe in section 6.1.

6.1. Cross platform authentication

6.1.1. Goals and purpose

Our review of authentication methods showed that there was no obvious candidate for authenticated association on the full range of devices used to consume media, in particular connected TVs and radios. To address this problem, the BBC, with the EBU and other partners, have developed a protocol based on OAuth 2.0 (called “Cross Platform Authentication” or CPA). This specification was published as EBU Technical Recommendation 3366 in September 2014 [4].

Version 1.0 of the CPA protocol deals with devices with limited input and display capabilities, such as hybrid radios or IP-connected set top boxes with infra-red controllers. The protocol will be further developed to support more capable devices, such as a tablet with a browser, and to support conferring identity from a more capable device to a less capable one, for example to enable using a smart phone to authenticate a headless device.

The protocol was developed to meet broadcasters’ requirements:

- Single sign-on across multiple services using a common authorization service.
- Client mode, so that devices work "out of the box" without user registration.
- User mode, to associate a device with a user account.

The CPA protocol forms the basis of identity management within MediaScape, in particular the concept of the client identity on the device and how that is associated with an online user identity.

Since publication of the EBU Technical Recommendation, we have worked to develop an authorization flow suitable for mobile devices, and we have submitted the specification to ETSI for standardization.

6.1.2. Relation to the MediaScape architecture

The Cross Platform Authentication specification delivers a key element of the MediaScape architecture: the ability to associate requests made to web APIs with a user account. This enables a wide range of desirable features, such as the ability to store information centrally and access it from different devices over the Internet, personalization of data feeds, and sharing between people.

CPA introduces and defines the concepts of client identity, service provider, authorization provider and identity provider, laying out a clear conceptual foundation for the application of authenticated identity and authorization throughout MediaScape.

By using OAuth 2.0 as the basis for our protocol, we have made it possible to add authentication and authorization to any web request from any component with minimal adaptation.

The diagram in Figure 12 below shows the main parties involved:

- **Client** Represents the use of a service provider by an application on a device. This could be a MediaScape agent (as described in section 2.3).
- **Service Provider** An online service which requires authorization to access its protected resources. In MediaScape this could be a web service such as the WP4 Shared State.
- **Authorization Provider** Manages client identities, the association of client identities with authenticated user identities, and the issuing of access tokens to clients.

Also shown, but not itself a participant in the CPA protocol is the **Identity Provider**, which authenticates the end user and provides user identities to the Authorization Provider.

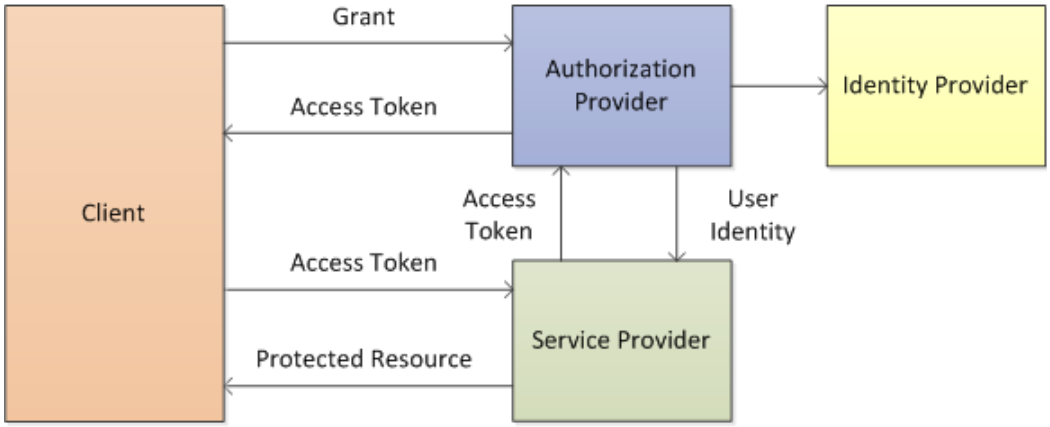

Figure 12. Cross Platform Authentication architecture

6.1.3. Relevant standards

The Cross Platform Authentication protocol has been published as an EBU Recommendation [CPA], and at the time of writing the specification has been submitted to ETSI for standardisation.

The protocol is based on the following existing standards:

- RFC 6749 OAuth 2.0 [51]
- RFC 6750 OAuth 2.0 Bearer Token Usage [45]
- OAuth 2.0 Device Profile (draft) [46]
- OAuth 2.0 Dynamic Client Registration (draft) [47]
- RFC 2818 HTTP over TLS [48]

6.1.4. Definition

The CPA device flow specification defines the protocol between a client device and an authorization provider by which a client device acquires an authorization token it can use to access protected resources and by which the service provider can identify the client device and an associated user account.

While CPA can be used more generally, the focus of the specification is on IP-connected media devices with limited input and display. Because such a device is generally not capable of running a fully capable HTTP user agent such as a browser, the user must have access to another computer which can run such software to complete the authorization.

CPA optionally supports single sign-on between multiple services that share a common Authorization Provider. Depending on business rules reflecting the relationship between service providers, the Authorization Provider can:

- Require the user to log in and enter a pairing code.
- Require the user to log in and give confirmation, without entering a pairing code.

- Automatically grant an access token, without requiring any user action.

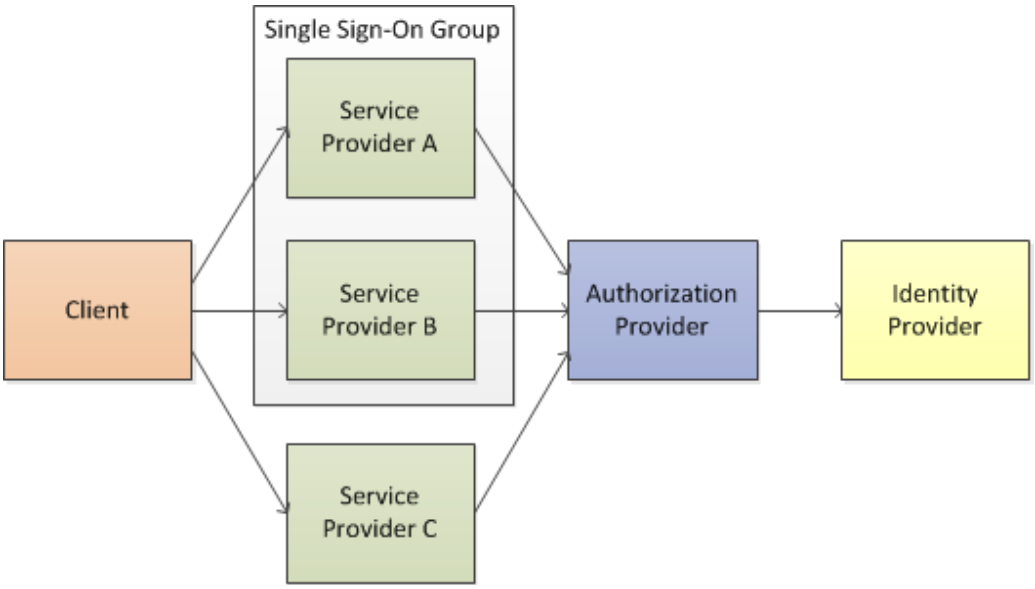

Figure 13. Single sign-on groups

Figure 14 below illustrates the relationships between users, clients, and devices. In summary:

- One or more **Users** interact with a device.
- The **Device** may implement a separate user profile for each User.
- A user profile on a device may be associated with one or more **Authorization Providers**.
- A **Client** represents the association between a device and a single Authorization Provider. The relationship is identified by the client_id and client_secret issued to the Client by the Authorization Provider.
- A Client has an access token for each Service Provider.

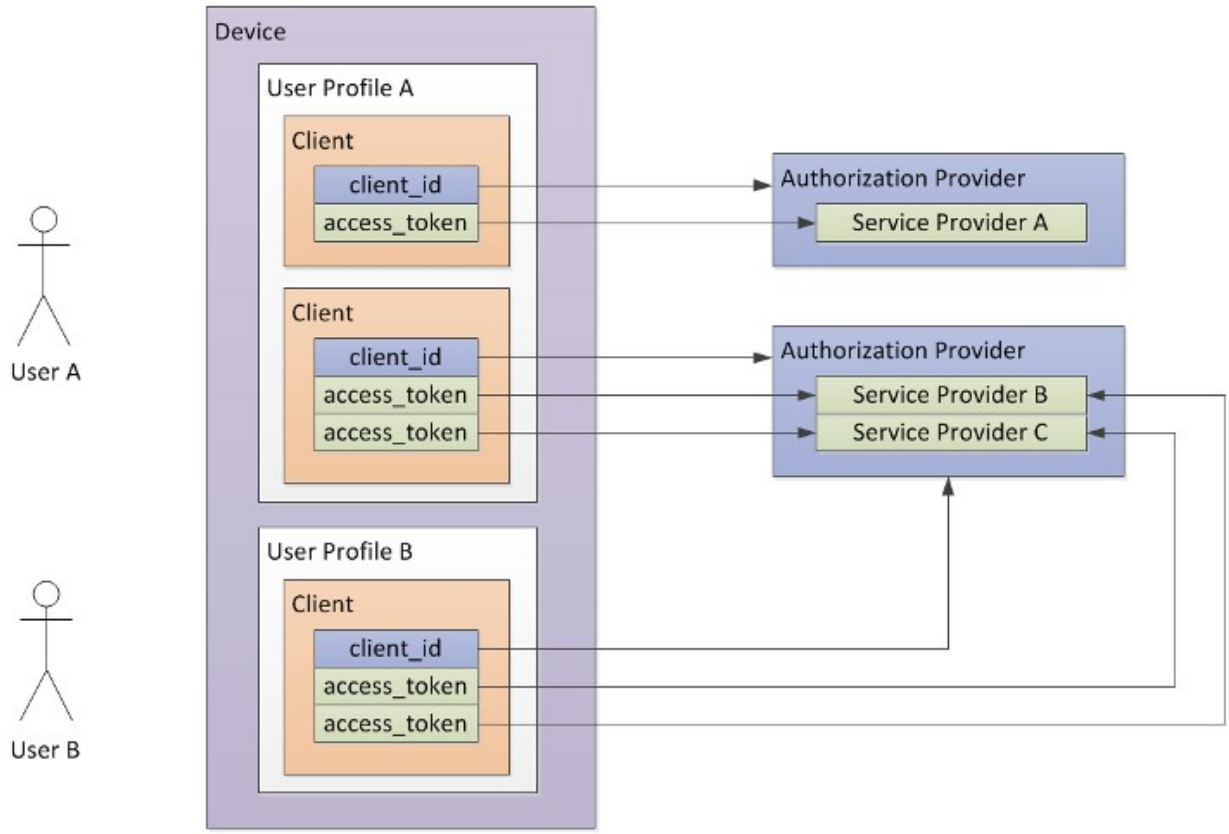

Figure 14. Relationship between entities in the CPA protocol

For details of the protocol itself, please refer to [4], and the description of the client libraries in D3.3, section 4.2.

6.1.5. Implementation

Alongside development of the Cross Platform Authentication protocol itself, we have built reference implementations of all the necessary software components. These are all written in JavaScript, using Node.js for the server-side components.

In addition, we have published an interactive tutorial, which includes a virtual machine setup with all the components installed and configured. These are intended to demonstrate how the protocol works in practice, and to help guide implementers.

6.1.6. Links

- Authorization provider reference implementation

  https://github.com/ebu/cpa-auth-provider

- Service provider reference implementation

  https://github.com/ebu/cpa-service-provider

- Client reference implementation
  https://github.com/ebu/cpa-client
- Interactive tutorial
  https://github.com/ebu/cpa-tutorial

7. Other related technologies

This section describes other related technologies that are useful for multi-device applications, but are not directly related to the WP3 tasks of discovery, pairing, association, or authentication.

7.1. W3C Push API

7.1.1. Goals

The W3C Push API [49] is a recent addition to the Web Platform which allows servers to wake up a Web application remotely via the transmission of a push signal to the host device.

These push notifications have traditionally been used by native applications to inform the user of time-sensitive news (e.g. time bound sales, breaking news). Their emerging availability in Web browsers [50] is based on the new background activities enabled by Service Workers [51].

Push notifications provide another useful mechanism to enroll devices in a MediaScape multi-devices experience.

7.1.2. Relation to the MediaScape architecture

The Push API explores both the discovery (T3.1) and pairing (T3.2) aspects of the MediaScape architecture. It has the advantage of running directly in Web browsers that implement the API (hopefully all of them will as the specification matures on the standardisation track) and could be used to bootstrap multi-device sessions in cases when all devices have already loaded the underlying application at least once.

7.1.3. Relevant standards

The Push API [53] is a W3C Working Draft developed by the Web Platform Working Group at W3C. The draft should be published as a Web standard once its APIs have stabilized and once it has been implemented across browsers.

7.1.4. Definition

The overall flow of that enrolment with push notifications is as follows. We assume the user has used the said application on multiple devices at some point; the application also knows these devices belong to the same user (e.g., because the user has logged in on these devices);

- on each of these devices, the user has previously accepted to receive push notifications from the application;
- now the user is active on the said application in one of these devices; the application, having determined the user owns multiple compatible devices, offers to switch to a multi-device mode by prompting the user with the list of devices the application knows of;
- the user picks the devices they would like to add to their current experience;

- the application then sends a push notification to all the selected devices; the user only has to activate these notifications, and each device is now up and running with the app correctly configured.

These push notifications also provide a great way to bootstrap multi-user experiences: assuming the application knows that two users are related, the application can use the same mechanism to offer to invite another user to the application, even if that user is not currently logged in or interacting with the application at that moment.

7.1.5. Implementation

The Push API is still a fairly recent technology. MediaScape reviewed it but did not implement a prototype, in particular due to the late availability of Web browsers that supported it. Google Chrome was the first browser to ship with an implementation of the API, now followed by Firefox.

7.1.6. Links

Push API, W3C Working Draft, 8 Dec 2015, http://www.w3.org/TR/push-api/

8. Conclusions

The goal of WP3 is to bootstrap the environment which enables MediaScape applications. This entails finding suitable devices on the home network on which to deliver the application experience, finding out their capabilities, communicating with them, and enabling other components of MediaScape to control them.

In this report we have described how devices can discover and communicate with each other, how devices can join a MediaScape application session. We have described our investigations into discovery and control of media devices on a local area network, and how this can be achieved from web browsers while preserving end-user privacy and security. We have also described how personalization is enabled by associating media devices with a user profile, and how users can authenticate through media devices with limited input capability.

The **Discovery API** enables both discovery of local devices, as well as introspection of device capabilities, providing information that can be used to adapt media content to the characteristics of those devices. The SSDP Service shows how existing protocols for discovery and control of media devices can be integrated into MediaScape applications.

The **Association API** provides an abstraction that allows a MediaScape application on one device to invite another device into the same application session, using a selection of pairing technologies, including Named WebSockets, acoustic codes, QR codes, vibration patterns ("Shake & go"), and the W3C Presentation API.

The **Presentation API** allows a web browser to discover a second screen available on the network, launch a Web URL on that screen, and establish a communication channel with it.

The **Cross Platform Authentication** protocol provides a secure mechanism that allows media devices to be associated with a user account, enabling personalized content to be delivered to devices. The protocol is based on OAuth 2.0, but designed for use with devices having limited input capability, and also addresses broadcaster requirements, such as single sign-on and a mode allowing personalization without requiring user authentication.

The Presentation API and Cross Platform Authentication form part of the contribution to standards from the MediaScape project, and our libraries and APIs are published as open source software.

9. References

[1] The OAuth 2.0 Authorization Framework, Oct 2012, http://tools.ietf.org/html/rfc6749

[2] Discovery and Pairing Literature Review for MediaScape, https://github.com/bbcrd/device-discovery-pairing/blob/master/document.md, last visit: 18 Jan 2016

[3] RadioTAG is a pairing and audio bookmarking protocol for IP-connected radios developed by the BBC as part of the RadioDNS family of specifications, https://github.com/bbcrd/radiotag-doc, last visit: 23 Mar 2015

[4] EBU Tech 3366: The Cross Platform Authentication Protocol, Version 1.0, Sept 2014, https://tech.ebu.ch/docs/tech/tech3366.pdf

[5] Geolocation API Specification, W3C Editors Draft, 11 Jul 2014, https://dev.w3.org/geo/api/spec-source.html

[6] The Screen Orientation API, W3C Working Draft 23 Dec 2015, https://www.w3.org/TR/screen-orientation/

[7] Vibration API, W3C Recommendation 10 Feb 2015, https://www.w3.org/TR/vibration/

[8] Battery Status API, W3C Candidate Recommendation 09 Dec 2014, https://www.w3.org/TR/battery-status/

[9] Proximity Events, W3C Working Draft 03 Sep 2015, https://www.w3.org/TR/proximity/

[10] Network Information API, Living Document, W3C Editor's Draft 19 Dec 2015, https://w3c.github.io/netinfo/

[11] Media Capture and Streams, W3C Last Call Working Draft, 14 Apr 2015, https://www.w3.org/TR/mediacapture-streams/

[12] Universal Plug and Play, http://www.upnp.org/, last visit 27 Jan 2016

[13] Multicast DNS, RFC 6762, Feb 2013, https://tools.ietf.org/html/rfc6762, last visit 27 Jan 2016

[14] DNS-Based Service Discovery, RFC 6763, https://tools.ietf.org/html/rfc6763, last visit 27 Jan 2016

[15] The WebSocket Protocol, RFC 6455, https://tools.ietf.org/html/rfc6455, last visit 27 Jan 2016

[16] Simple Service Discovery Protocol/1.0, https://tools.ietf.org/html/draft-cai-ssdp-v1-03, last visit 27 Jan 2016

[17] Network Web Sockets, Living Specification, Unofficial Draft, 27 Nov 2014,

http://namedwebsockets.github.io/spec/, last visit: 26 Jan 2016

[18] Named WebSockets Proxy, https://github.com/namedwebsockets/cmd, last visit: 18 Jan 2016

[19] Cling Java/Android UPnP library and tools, http://4thline.org/projects/cling/, last visit 15 Feb 2016

[20] DIAL - Discovery And Launch http://www.dial-multiscreen.org/, last visit 9 Mar, 2016

[21] Network Service Discovery, W3C Working Draft 20 Feb 2014, http://www.w3.org/TR/discovery-api/

[22] Cross-Origin Resource Sharing, W3C Recommendation 16 Jan 2014, https://www.w3.org/TR/cors/

[23] Network Service Discovery API Support in Opera, Daniel Davis and Rich Tibbett, 23 Oct 2012, https://dev.opera.com/articles/network-service-discovery-api/, last visit: 26 Jan 2016

[24] Node.js Client and Server module implementing the Simple Service Discovery Protocol SSDP, https://github.com/fraunhoferfokus/peer-ssdp, last visit: 11 Mar 2016

[25] https://lists.w3.org/Archives/Public/public-web-and-tv/2014Jun/0005.html, last visit: 23 Mar 2015

[26] https://lists.w3.org/Archives/Public/public-web-and-tv/2014Apr/0055.html, last visit: 23 Mar 2015

[27] Mozilla's FlyWeb proposal, https://wiki.mozilla.org/FlyWeb, last visit 01 Feb 2016

[28] HbbTV 2.0 Specification, ETSI TS 102 796 V1.3.1, http://www.etsi.org/deliver/etsi_ts%5C102700_102799%5C102796%5C01.03.01_60%5Cts_102796v010301p.pdf

[29] What are Chrome Apps?, https://developer.chrome.com/apps/about_apps, last visit 12 Feb 2016

[30] Custom Elements, W3C Editor's Draft http://w3c.github.io/webcomponents/spec/custom/, last visit: 16 Jan 2015

[31] Presentation API, W3C Working Draft, 13 Oct 2015, https://www.w3.org/TR/presentation-api/, last visit: 26 Jan 2016

[32] What are Chrome Extensions?, https://developer.chrome.com/extensions, last visit 12 Feb 2016

[33] Message Passing, https://developer.chrome.com/extensions/messaging, last visit 12 Feb 2016

[34] HTML5, A vocabulary and associated APIs for HTML and XHTML, W3C Recommendation, 28 Oct 2014, https://www.w3.org/TR/html5/

[35] Physical Web, https://google.github.io/physical-web/, last visit 15 Jan 2016

[36] Bluetooth Low-Energy Core specification, version 4.2, 2 December 2014,

https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=286439&_ga=1.130486732.277404890.1455034074

[37] Physical Web Browsers, https://phy.net/physical-web-browsers/, last visit: 15 Jan 2016

[38] Web Bluetooth Community Group, http://www.w3.org/community/web-bluetooth/, last visit 28 Jan 2016

[39] ISO/IEC 18004:2015, Information technology - Automatic identification and data capture techniques - QR Code bar code symbology specification

[40] Tone: An experimental Chrome extension for instant sharing over audio, http://googleresearch.blogspot.co.uk/2015/05/tone-experimental-chrome-extension-for.html, last visit: 18 Jan 2016

[41] Web Audio API, W3C Working Draft, 8 Dec 2015, http://www.w3.org/TR/webaudio/

[42] QRCode.js, https://github.com/davidshimjs/qrcodejs, last visit: 18 Jan 2016

[43] WebCodeCam jQuery Plug-in, https://github.com/andrastoth/WebCodeCam, last visit: 18 Jan 2016

[44] TimeAPI server, http://www.timeapi.org/utc/now.json

[45] The OAuth 2.0 Authorization Framework: Bearer Token Usage, Oct 2012, https://tools.ietf.org/html/rfc6750

[46] OAuth 2.0 Device Profile, Jul 2010, expired Oct 2015, https://tools.ietf.org/html/draft-recordon-oauth-v2-device-00, last visit 26 Jan 2016

[47] OAuth 2.0 Dynamic Client Registration Protocol, 21 Apr 2005, https://tools.ietf.org/html/draft-ietf-oauth-dyn-reg-28, last visit 26 Jan 2016

[48] HTTP Over TLS, RFC 2818, May 2000, https://tools.ietf.org/html/rfc2818, last visit 26 Jan 2016

[49] Push API, W3C Working Draft, 8 Dec 2015, http://www.w3.org/TR/push-api/

[50] Can I use... Support tables for HTML5, CSS3, etc. (Push API) http://caniuse.com/#feat=push-api, last visit: 15 Feb 2016

[51] Service Worker, W3C Working Draft 25 Jun 2015, https://www.w3.org/TR/service-workers/