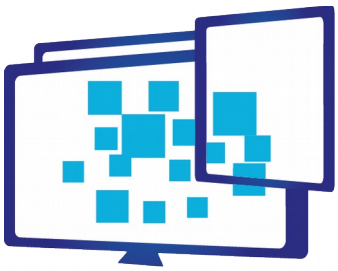


MediaScape

Dynamic Media Service Creation, Adaptation and Publishing on Every Device

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

WP3 – CONNECT – Multi-connection and authentication

Deliverable Number	D3.3
Title	Final Version of multi-connection mechanisms and multi-device authentication
Version	1.0
Date	23 Oct 2015
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	23Jun. 2015	Mikel Zorrilla (VIC)	Template and structure
0.2	21 Jul. 2015	Mikel Zorrilla (VIC)	Stucture changed
0.3	23 Sep 2015	Sean O'Halpin (BBC)	First draft
0.4	01 Oct 2015	Angel Martín (VIC), Benedikt Vogel (IRT), François Daoust (W3C)	Contributions on the different API documentation
0.5	10 Oct 2015	Sean O'Halpin (BBC)	Integration of all the document
0.6	11 Oct 2015	Njaal Borch (NOR)	Peer Review
1.0	23 Oct 2015	Sean O'Halpin (BBC)	Final version

Table of Contents

1. Executive Summary

WP3 CONNECT provides the services required to enable software agents running on a the same network to discover, query and communicate with other to enable them to use the capabilities of the host network and its connected devices to create a suitable platform for delivering MediaScape applications.

WP3 focuses on co-ordinating agents distributed over the same network to build services for interfacing to the devices and services for which they are responsible. WP4 and WP5, on the other hand, deal with co-ordinating *applications* distributed over the *Internet.*

This document describes the software components and APIs used and provided by the WP3 CONNECT layer and how they all work together.

We start with an overview that shows how the various parts fit together. We also explain the key existing technologies used: Multicast DNS and Network WebSockets.

The APIs supplied by WP3 and documented here are:

- *Discovery*, which determines the *capabilities* that represent what a device or service can do
- *Association* for dynamically pairing and co-ordinating agents
- *Cross Platform Authentication*, a protocol for creating an authenticated association between a user's online account and a device with limited input capabilities

The APIs documented here are the final versions of the APIs and protocols developed for discovery, association and authentication arising from the three tasks: WP3.1 Discovery and Association Mechanisms, WP3.2 Dynamic Pairing of Multiple Resources and WP3.3 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 system of software agents running on devices which work together to enable applications to provide those experiences.

WP3's goal 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.

All the participating devices and services in MediaScape are represented as *agents* which manage those devices (querying their capabilities and controlling them). This enables us to present a simple, uniform abstraction to other MediaScape components.

We needed to solve:

- how to find the agents managing devices and services on a local network
- how agents interact
- how to link agent actions on a device to a user account on the web

Each of these problems is composed of a number of subproblems to solve. For example, finding the agents which manage loudspeakers on a local network requires some form of discovery (who do you ask to find agents on your LAN?), how to specify who you want to talk to (names and addresses) and how to determine what each agent can do (capabilities). Once you've managed to find an agent, you then want to be able to send them commands and ask questions, controlling them remotely, which means you need to specify how agents communicate with each other. Finally, linking a press of a TV remote button to a user account on the web requires the ability to authenticate the association of a device with an account.

2.2. MediaScape Architecture

The WP3 stack is depicted schematically below:

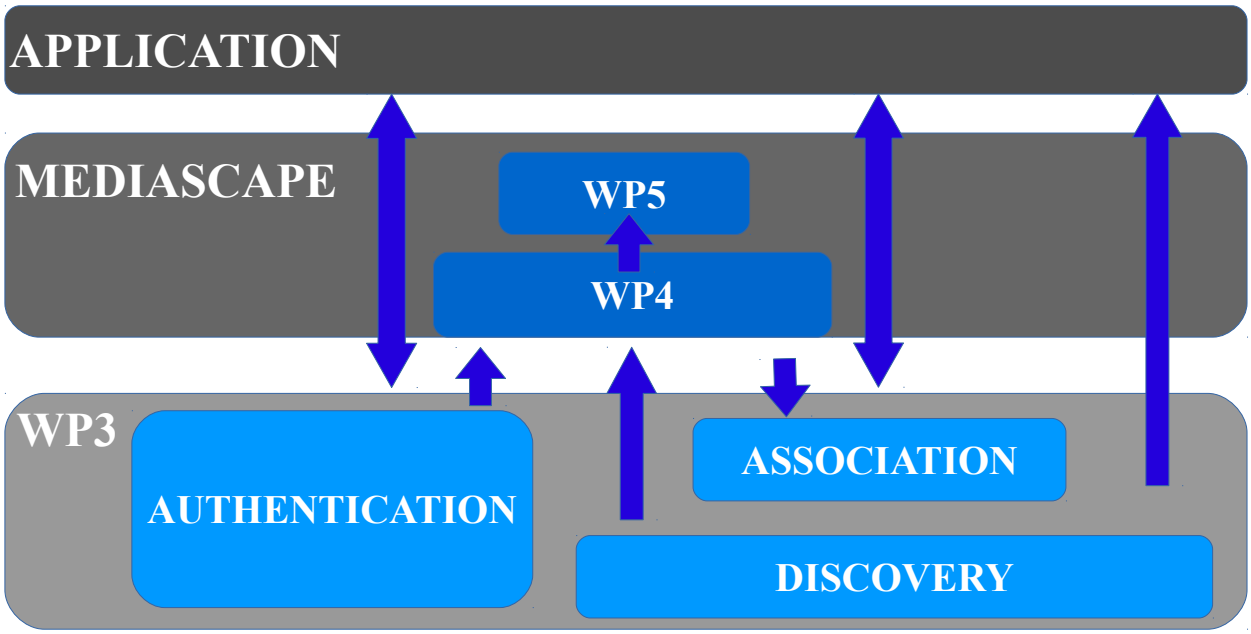


Figure 1: WP3 high level architecture

A MediaScape application uses these modules either directly or via the modules provided by WP4 and WP5. Discovery provides information to the upper layers, including presence information for pairing and the set of discovered capabilities for WP5's Adaptation module via the Shared State provided by WP4. 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 6.2 *MediaScape Device Capabilities*.

The *Association API* enables agents to invite other agents to join a session, which enables them to share state. This shared state can be used to implement the model discussed in D2.2 4.1.5 *Media Control as a Service.*

EBU *Cross Platform Authentication* (CPA) is an open, standard authentication protocol for IP-connected media devices 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.

We describe a simple media player API for controlling radios remotely and finally the various proxies and helpers we have implemented to fill some of the gaps in the coverage of the various technologies involved.

WP3 provides the application environment for WP4 and WP5. For WP4, it provides discovery of local nodes and the ability to query the agent's current state and to subscribe to local event streams. For WP5, it provides the information about capabilities and current state (communicated via the shared state provided by WP4) needed to provide adaptive user interfaces.

2.3. Approach

2.3.1. General challenges

The challenges facing WP3 are:

- 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
  - 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)

The challenges for WP3.3 *Authentication* have been both technical and organizational. The technical challenges have been to devise a protocol which is secure yet still straightforward enough not to put people off using it on limited input devices, to develop a reference implementation and to formally document the protocol. The organizational challenges have been to gain consensus among the task group on the scope and specification of the protocol, to shepherd the specification through the process of adoption by the EBU as a technical recommendation and to gain adoption by third parties.

2.4. Structure of the deliverable

This document describes the subsystems and APIs provided by WP3.

For each API or subsystem, we explain what it's for (*Goals*), what aspect of the WP3 architecture it is intended to provide or support (*Relation to the MediaScape Architecture*), what the API does and how it works (*Definition*). The API *Definition* contains examples, where to find the code, what license it is licensed under.

Finally, we discuss how this work can contribute to standardization.

3. APIs

The three primary APIs developed by WP3 are:

- The *Discovery API*[1] provides a framework for specifying, detecting and reporting device functions.
- The *Association API* is for dynamically pairing and co-ordinating agents to enable sharing state.
- *Cross Platform Authentication* explains the concepts, interactions and API for an OAuth 2.0 based protocol for securely associating a limited input device (such as an IP-connected radio) to an online user account.

3.1. Discovery API

"The goal of this task is to determine a mechanism to discover resources such as the different devices (their capabilities, location), the users, the services and the content... The output of this task will be the documentation of the developed discovery mechanism API and a reference implementation."

From the *Description of Work*

3.1.1. Functional Features

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).

The Discovery API creates a common interface to discover both:

1. Network interfaces and protocols the device is able to handle
2. The features or capabilities of a specific device

This is required in order to abstract the capabilities that are currently available, and enable future capabilities to be added, to make them all easier for developers to use.

Discovery of network interfaces and protocols is needed at the association stage and thereafter for communication. Other device features help the developer create the best user experience for the size and type of device.

This module is not strongly dependent on other MediaScape modules because it acts as a feature collector, and has passive behaviour, collecting information on demand and capturing the information at request time.

[1]This should probably be named the Capabilities API

In this implementation, an underlying RESTFul API uses various Discovery Agents in order to gather information not available from web browsers. The developer-focused API is a JavaScript library that can be included in a web application.

This approach has been taken for two reasons:

- Some capabilities are not available at all in browsers (e.g. APIs to network interfaces and protocols)
- Some features are inconsistently implemented across browsers, or vary between types of operating system.

A table is included in the Appendix 2 showing the levels of maturity for various browser features.

Get all information discovered

General pattern

The API pattern for the different target discovery technologies is:

```
mediascape.discovery.<function>("<target>", [parameters])
    .then(cbOk(returnedJSON), cbErr(returnedJSON));
```

Where `cbOK` is the function run in the event of success and `cbErr` the function run in the event of an error.

The result is JSON with this format:

```
{
        "functionName": [{
        "itemName1": {
                "itemValue1"
        },
        "itemName2": {
                "itemValue2"
        },
        …
```

```
        }]
}
```

Functions

To make it easier to get access to all the data collected by the Discovery Agents, general wrappers are available. These two functions return all the information discovered about the device, service or asset in a human-understandable way, performing calls to all the available Discovery Agents and producing an integrated result.

All Presence

```
mediascape.discovery.isPresent().then(cbOK(presenceJSON),cbErr(errorJSON));
```

This function provides a boolean result to get agent availability. For native Discovery Agents it performs a basic REST call and depending on the HTTP response (success 2xx or error 4xx/5xx), whether the agent is installed, deployed and running or not. All the agents installed on a device can be first checked for existence then probed for more detailed information.

All Extra

```
mediascape.discovery.getExtra().then(cbOK(extraJSON),cbErr(errorJSON));
```

This function returns additional information about the device, service or asset in a human-understandable way. Examples are below.

Discovery of device capabilities

For each individual target discovery technology, two equivalent functions are available:

isPresent

```
mediascape.discovery.isPresent("<target>").then(cbOK(presenceJSON),cbErr(errorJSON));
```

getExtra

```
mediascape.discovery.getExtra("<target>").then(cbOK(extraJSON),cb2Err(errorJSON));
```

<target> in this case can be any of the following:

Screen

- `IsPresent`: Presence of a screen, based on the Javascript screen object.
- `GetExtra`: details about the screen based on the screen object.

Geolocation

- `IsPresent`: presence of a geolocation sensor, based on the Javascript Geolocation API.
- `GetExtra`: user's location, using the positioning capabilities of their device, based on the Geolocation API.

Orientation

- `IsPresent`: presence of a device orientation sensor availability using the DeviceOrientation Javascript event.
- `GetExtra`: the user's device orientation using the DeviceOrientation event.

Camera

- `IsPresent`: camera availability on the user's device, based on the Javascript Media Capture and Streams API.
- `GetExtra`: details about the user's device's camera, based on the Media Capture and Streams API.

Vibration

- `IsPresent`: user's device vibration sensor availability, based on the Javascript Vibration API.

Battery

- `IsPresent`: user's device battery sensor availability, based on the Javascript BatteryStatus API.
- `GetExtra`: details about the user's device battery status, based on the Javscript BatteryStatus API.

UserProximity

- `IsPresent`: user's device proximity sensor availability, based on the Javascript UserProximityEvent event.
- `GetExtra`: details about user's device proximity sensor based on the Javascript UserProximityEvent.

DeviceProximity

- `IsPresent`: user's device proximity sensor availability, based on the Javascript UserProximityEvent event.
- `GetExtra`: details about user's device proximity sensor based on the Javascript UserProximityEvent.

Language

- `IsPresent`: user's device browser language availability, based on the Javascript Navigator object.
- `GetExtra`: details about the user's device browser language, based on the Javascript Navigator object.

DeviceType

- `IsPresent`: user's device type availability, based on the Javascript Navigator object.
- `GetExtra`: details about the user's device browser language, based on the Javascript Navigator object.

Connection

- `IsPresent`: user's connection type availability, based on the Javascript Navigator object.
- `GetExtra`: details about the user's connection type, based on the Javascript Navigator object.

Discovery of networking Discovery Agents

Most of the discovery protocols for networks use multicast and UDP, and are not available via web browsers, so we have used Discovery Agents (RESTFul wrappers around these protocols) in order to test for their presence in a Javascript framework. For these discovery protocols we are therefore therefore testing for the presence of these agents, and then using further functions to query them for more information.

General pattern

The general pattern for interacting with these agents via Javascript is as follows:

```
mediascape.discovery.<function>("<target>",[parameters])
    .then(cbOk(returnedJSON),cbErr(returnedJSON));
```

Where **`<target>`** refers the networking technology-related agent.

As before, the result of calling these functions is a JSON response with this format:

```
{
        "functionName": [{
        "itemName1": {
                "itemValue1"
        },
        "itemName2": {
                "itemValue2"
        },
        …
        }]
}
```

The complete set of Javascript functions defined for each Networking Discovery Agent are listed below.

Functions

- `IsPresent`: availability status of the agent. For native Discovery Agents It performs a basic REST call and depending on the HTTP response (success 2xx or error 4xx/5xx), whether the agent is installed, deployed and running or not. This way all the agents installed on a device can be first checked for existence then probed for more detailed information.
- `GetProfile`: information about the level of compliance with the standard interface or protocol or skills of the agent.
- `GetDevices`: launches the discovery messages and collects the reachable devices through a specific interface or protocol.
- `GetServices`: launches the discovery messages and collects the reachable services from the previously detected devices through a specific interface or protocol.
- `GetAssets`: launches the discovery messages and collects the reachable assets, or media resources, from the previously detected devices through a specific interface or protocol coming from internal and external devices.
- `GetExtra`: returns additional information about the device, service or asset in a human-understandable way that could be relevant for the user.
- `GetActions`: provides a set of controls that can be operated over the services or assets controlling media contents such as video/audio.
- `GetParameters`: provides the list of parameters that must be included to operate over the services or assets.

- `ConnectNWS` This function has been created just for Named Web Sockets. It creates the connection to the sockets and starts the connection and disconnection listeners.

3.1.2. Examples

Discovery of device capabilities

See if a screen is present, and return the dimensions if it is:

```
mediascape.discovery.isPresent("screen").then(
    mediascape.discovery.getExtra("screenSize"),
    function(data) {
      console.log("No screen available");
      return data;
    },
);
```

Result on success:

```
{"extra":[{"width":"1280","height":"1024"},{"screenX":"13.334","screenY":"10.666"}]}
```

Result on failure:

```
{"presence":false}
```

Discovery of networking Discovery Agents

Check for presence of UPnP SetVolume parameters:

```
mediascape.discovery.getParameters("upnp", 4, "RenderingControl", "SetVolume")
    .then(
      function(data) {
       console.log('Parameters Ok');
       console.log(data);
      }, function(data) {
       console.log('Parameters Error');
      }
);
```

Check for Bluetooth presence:

```
mediascape.discovery.isPresent("bluetooth").then(function(data) {
        console.log('Bluetooth Presence Ok');
        console.log(data);
}, function(data) {
        console.log("Bluetooth Presence Error");
});
```

3.1.3. Code and license

The code is available in GitHub:

https://github.com/mediascape/discovery-self

3.2. Association API

"The main goal of this task is to determine a solution for dynamic pairing of the resources for a specific session. Based on the discovery mechanisms developed in T3.1, this task will provide a contextual pairing of the different resources. It will manage the sharing of a single device through different users (e.g. a TV), multi-user sharing a single online experience through personal devices, etc. It will also consider the ubiquity of the users and devices to provide dynamic mechanisms for the pairing."

From the *Description of Work*

3.2.1. Functional Features

WP3.2 provides the 'glue' which connects the components of MediaScape together at the network level. WP3.2 provides the protocols and mechanisms by which agents communicate with each other, share information and enable remote control.

An important responsibility for WP3.2 is the provision of bridges to existing protocols such as UPnP and HbbTV, to enable devices using those protocols to act as agents in the MediaScape system. These helpers are documented fully in their repositories rather than here as they do not directly affect our APIs, protocol development or standards.

MediaScape targets applications that provide shared experiences across multiple devices. The Association API is responsible for establishing the communication mechanisms and protocols to perform the dynamic pairing of resources for a specific session on a local network.

It drives the bootstrapping of local multi-device sessions, connecting devices ready to bridge to an application instance. To achieve this, the MediaScape Association library must pair, automatically or on demand,

reachable devices through possible network interfaces and communication protocols. The idea is to have a common interface independent from the underlying technology, so that a developer can design and develop a service without considering what is happening behind the scenes.

As an illustration, we consider the case when the multi-device session involves two devices, although generalization to a larger number of devices can be done the same way or on a two-by-two basis.

The functional scenario is this:

1. The first device is running an app that can create a multi-device experience, which may be a regular Web app that the user is interacting with, or an autostart app triggered by a TV or radio channel.
2. After associating, apps running on both devices are from the same origin, so both devices are interacting with the same backend endpoint.
3. The two devices should then participate in the same multi-device experience, meaning that the two devices should be able to establish a communication channel and exchange messages.

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 functional features are:

- Device feature detection (using the discovery capabilities API described below)
- Start new association
- Join existing association
- Stop association
- Event handling

The API is asynchronous in order to prevent thread blocking, and consequent poor user experience, and so uses Javascript Promises to handle message deliveries and failures and provide a common format for callbacks. Event detection is a way to capture these events as they occur.

Association may be driven by a variety of bootstrapping methods: including in some circumstances automatically, or following a user interaction, via acoustic code (Audio frequency-shift keying), visual code (QR code), or pin entry or comparison.

As a security mechanism we assume that the devices are physically co-present. We also assume that the devices can access a suitable web page (which usually means that they are both able to access the internet, but does not require it, since one of the devices involved could host the page). We do not assume that they are on

the same local network, but more features are available if they are.

After association, the next step is up to the application developer. At this stage both devices are able to load a specific URL. This could be used to establish a shared application context using Work Package 4's SharedState backend; alternatively the first device might use that URL to send control commands to the second, for example using the Control API below.

General pattern

The pattern for the different target association technologies is:

```
mediascape.association.<function>("<target>", parameters]).then(cb(returnedJSON));
```

Returning (with Promises at runtime):

```
{"response":"functionValue"}
```

The complete set of association functions are listed below.

doAssociation

Each association technology has different requirements, so for each technology the parameters to this function call are different. For most of the technologies the requirements are sufficiently different that we have implemented separate calls for trigger and catcher.

```
mediascape.association.doAssociation("<target>",[parameters]).then(cb(sessionJSON));
```

For triggers, the `doAssociation` method returns the promise to the callback `cb(sessionJSON)`, where `cb(sessionJSON)` could mean:

- initiated a pairing process between two devices using the suggested method, process being in no particular state
- broadcast the provided URL using the suggested method
- found a secondary device that will load the URL

For catchers the promise returned by the "doAssociation" method is resolved on the Catcher when it has received a URL to launch.

stopAssociation

This function provides the ability to stop the association process. This function has been developed for the acoustic case, but could be used anywhere it is necessary.

```
mediascape.association.stopAssociation("<target>");
```

on

This function allows the developer to be notified of and react to events at different stages of the association process.

```
mediascape.association.on('event',cb(messageJSON));
```

The events currently defined are:

```
'shakeChange'
'startAudioProcessing'
```

Here's an example of using `on` to attach a handler to a 'shakeChange' event.

```
mediascape.association.on('shakeChange',function(data){console.log(data);});
```

On-demand Association

These are the APIs for mechanisms that need explicit user interaction to perform association.

In each case, for the trigger, the app must have previously obtained a groupID from the Mapping Service (see Work Package 4) to be broadcast and added to the base URL.

Synchronised action association

This approach uses the built-in sensors from mobile devices to solve the problem of device association. To use it, it is necessary to have a centralized server which manages all the association requests and processes.

Depending on the kind of device, the association will start with the end-user pressing a button or shaking the device. Once the user has shaken the device for certain amount of time, the association will detect and send an association message to the server. When the server detects that two devices have begun the association process during the same time and in the same area, it will associate both devices by sending the same sessionID to them.

This function uses accelerometer values, geolocation and clock to provide the information needed to disambiguate concurrent requests.

Trigger

```
mediascape.association.doAssociation("sync",url).then(cb(sessionJSON));
```

Catcher

```
mediascape.association.doAssociation("sync").then(cb(sessionJSON));
```

Parameters:

- `url`: full URL including the session to be employed in any device to join a shared experience on a Web application.

For the trigger the app must have previously obtained a groupID from the WP4 Mapping service to be broadcast and added to the base URL.

Acoustic pattern-based association

Audio frequency-shift keying (AFSK) is a modulation technique by which digital data is represented by changes in the frequency (pitch) of an audio tone, yielding an encoded signal suitable for transmission. It transmitted audio alternates between two tones: one, the "mark", represents a binary one; the other, the "space", represents a binary zero.

One of the most commonly examples of using AFSK is the earliest early telephone-line modems that used to use audio frequency-shift keying (AFSK) to send and receive data.

For association purposes, a trigger would create and play an acoustic pattern based on the complete URL or a temporary unique code, while other instances, catchers, record the environmental audio to detect it and decode to get the final URL.

This function uses HTML5 technologies to perform FSK encoding, playing a series of sounds, capturing audio with the microphone and decoding FSK codes.

Trigger:

```
mediascape.association.doAssociation("acoustic",url,short).then(cb(sessionJSON));
```

Catcher:

```
mediascape.association.doAssociation("acoustic",htmlElementID).then(cb(sessionJSON));
```

Parameters:

- `url`: full URL including the session to be employed in any device to join a shared experience on a Web application.
- `short`: if true, of shortened URLs
- `htmlElementID`: unique identifier of a div-like HTML tag element that will host the mic capturing solution for the Catcher.

For the trigger the app must have previously obtained a groupID from the WP4 Mapping service to be broadcast and added to the base URL.

QR Code

QR codes are two-dimensional barcodes which can encode lengthy pieces of text in machine-readable format. QR code readers are commonly available as apps for devices with cameras.

For association purposes, one device must be able to show the QR code on its screen, and the user takes a picture of the screen using a QR code scanner.

The device then processes the image using Reed–Solomon error correction until the image can be appropriately interpreted. The required data - in this case a URL - is then extracted from patterns present in both the horizontal and vertical components of the image.

This function uses HTML5 technologies to perform QR encoding, plato displaying an image, capture with the camera or decode QR codes.

Trigger

```
mediascape.association.doAssociation("qr",htmlElementID,url,short[,xsize,ysize]).then(cb
(sessionJSON));
```

Catcher

```
mediascape.association.doAssociation("qr",htmlElementID).then(cb(sessionJSON));
```

Parameters:

- `htmlElementID`: Unique identifier of a div-like HTML tag element that will host the QR code for the Trigger or the camera capturing solution for the Catcher.
- `url`: full URL including the session to be employed in any device to join a shared experience on a Web application.
- `short`: boolean indicate whether shortened URLs are used
- `xsize,ysize`: optional dimensions of the displayed QR code

This function returns a full URL including the session to be employed in any device to join a shared experience on a Web application.

Typing

The Trigger should display the returned URL to enable others type the URL in the preferred Web Browser. The Catcher simply opens the URL and no specific function is needed.

Trigger:

```
mediascape.association.doAssociation("text",url,short).then(cb(sessionJSON));
```

Parameters:

- `url`: full URL including the session to be employed in any device to join a shared experience on a Web application. This way, the trigger gets acceptation awareness while the catcher gets the redirecting URL
- `short`: boolean to indicate the use of shortened URLs

This function returns a full URL including the session to be employed in any device in order to join a shared experience on a Web application.

Automatic Association

Automatic association of devices may be possible if Named Web Sockets are supported.

The basic flow is as follows:

- A device connects to a web page
- The MediaScape Discovery API looks for Named Web Sockets on the device's local network
- If any are found, the trigger / catcher behaviour occurs depending on whether there are any other devices currently available.

The main function is the following one, which broadcasts the target URL in the trigger, and captures the URL in the catcher.

Trigger:

```
mediascape.association.doAssociation("nws", channel, url, beacon)
    .then(cb(sessionJSON));
```

Catcher:

```
mediascape.association.doAssociation("nws",channel).then(cb(sessionJSON));
```

Parameters:

- `url`: full URL including the session id required to join a shared experience on a web application. The trigger gets a notification of acceptance of association while the catcher gets a redirection URL
- `channel`: string - the name of the channel required in Named Web Sockets for communications on the local area network
- `beacon`: boolean flag for repeating the association message every second

3.3. Examples

Automated association example

The API detects the Named Websocket Proxy through the Discovery API:

```
mediascape.discovery.getPresence('nws');
```

The result is:

namedwebsocket proxy is accessible	`return '{"presence":true}'`
namedwebsocket proxy is not accessible	`return '{"presence":false}'`

If the proxy is not accessible, the system will halt the automatic association process and will invite the user to use an on demand mechanism.

If the Named Web Socket is present, we check to see if there is any other device connected to the network:

```
mediascape.discovery.getDevices('nws')
```

The result will be the list of devices connected in that moment in by the Named Web Sockets such as:

```
'{"devices":[{"id":1129747263621539},{"id":718561927787959}]}'
```

If no other device is connected, the result will be a list with just one device, the recently connected user device, and will be something like:

```
'{"devices":[{"id":1129747263621539}]}'
```

Once the joined device knows the participants of the network, there are two possible actions depending on the numbers of devices connected:

If the new device is the first and only connected:

- The trigger will create a new sessionId and will share through the Named Web Sockets.

If the new device is not alone and is the last connecting to the network:

- In this case, the catcher will wait till it receives a notification with the sessionId shared by the first device connected to the Named Web Sockets. Then, the system will take that URL and will use to connect to the SharedState.

If the developer prefers to associate in a on demand manner, she can call different methods to do so.

Code and license.

The code is available in GitHub: https://github.com/mediascape/association

Unless otherwise stated:

3.4. Towards standardisation

The Association API exposes a simple interface to bootstrap a multi-device session but relies on server-side components to emulate functionalities that are not available to the Web runtime (such as discovery) and on the presence of a MediaScape application running on the receiving device. MediaScape contributed to create and is now actively involved in the Second Screen Presentation Working Group[2] at W3C, which was precisely chartered to enable that bootstrapping directly within the Web runtime. The Presentation API[3] is the result of that on-going standardisation effort, with on-going implementations in Google and Mozilla in particular.

The Presentation API and the Association API are closely related. MediaScape developed a JavaScript polyfill[4] of the Presentation API that reuses some of the association mechanisms explored for the Association API (e.g. QR code, Physical Web), with a view to reporting feedback to the Second Screen Presentation Working Group. Details will be reported in the final standardisation report (D7.4 Final Standardisation Report).

However, the assumptions and goals of the Association API and of the Presentation API are slightly different in practice, resulting in different interfaces. Where the Association API provides a mechanism for the Trigger to invite nearby devices to join a multi-device session, usually by broadcasting a URL on a channel that other devices listen to, and let the application handle the communication logic across devices (typically through a SharedState as developed in Work Package 4), the Presentation API relies on the Trigger to discover nearby devices that it can take control of to launch the URL and establish a communication channel between them.

[2] https://www.w3.org/2014/secondscreen/
[3] https://w3c.github.io/presentation-api/
[4] https://mediascape.github.io/presentation-api-polyfill/

4. WP 3.3 Authentication

4.1. Cross Platform Authentication

As described in deliverable D3.2, section 6.1, in order to enable personalised services across different devices, those devices need to reference a shared identity, and to establish this shared identity we need authenticated association between a device and an online account. The Cross Platform Authentication (CPA) protocol offers a solution for devices with limited input and display capabilities, such as hybrid radios or IP-connected set top boxes with infra-red controllers.

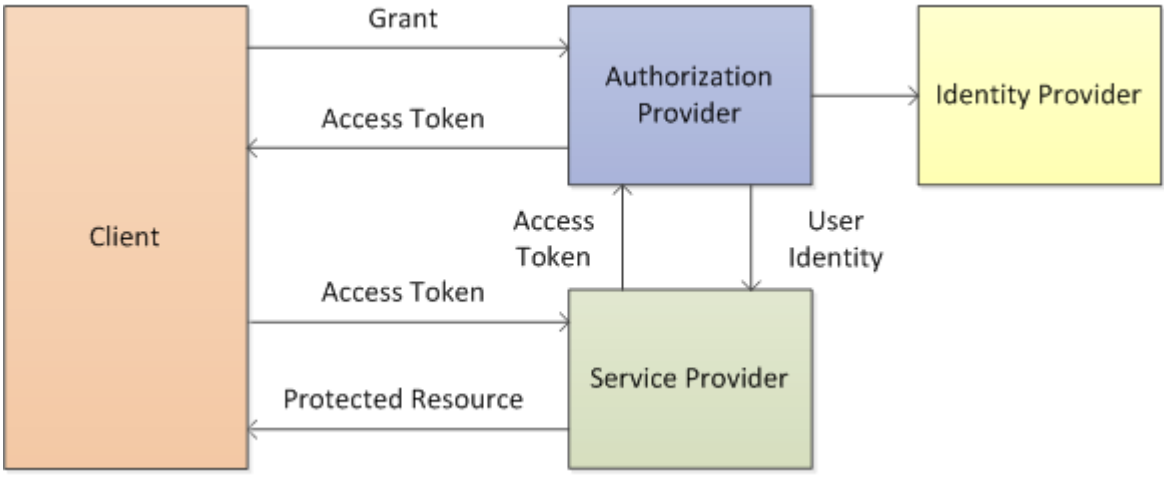
4.1.1. Functional Features

The CPA device flow specification[5] 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.

It does not deal with authentication of the end-user, i.e., verifying that the person authorizing access to a device is who they claim to be. There are other established protocols that can be used to do this, such as OAuth 2.0 and OpenID.

4.1.2. Functional Architecture

The diagram below shows the various components that participate in the Cross Platform Authentication protocol.



The Authorization Provider is a web service that manages client identities, the association of client identities with authenticated user identities, the issuing of tokens to clients and the linking of those tokens to the respective client identities.

The Service Provider is a web service that requires authorization to access its protected resources.

[5] EBU Tech 3366: The cross-platform authentication protocol https://tech.ebu.ch/files/live/sites/tech/files/shared/tech/tech3366.pdf

Authorization is achieved by the client presenting a bearer token when making requests, which the Service Provider verifies with the Authorization Provider. The Service Provider could be a service such as SharedState (WP4).

The Identity Provider is a service which authenticates a user identity, but how it does this is outside the scope of the CPA protocol.

The Client represents a single instance of a relation with an Authorization Provider, and is a consumer of protected resources at the Service Provider. Each Client has a unique identity provided by the Authorization Provider. A Client does not directly represent a user. Instead, it represents the use of a Service Provider by an application on a device. For example, it is better to think of the client identity as representing "My kitchen radio when listening to channel 1" than "Me".

4.2. API

4.2.1. CPA Client library

The cpa.js library implements the HTTP requests defined by the CPA protocol[6], and can be used from either client-side or server-side (Node.js) JavaScript code.

The library provides the following functions:

registerClient

Registers a client with the Authorization Provider.

```
registerClient(authProvider, clientName, softwareId, softwareVersion, callback);
```

Parameters:

- `authProvider` - the base URL of the Authorization Provider.
- `clientName` - the human-readable name of the client.
- `softwareId` - an identifier for the client software application.
- `softwareVersion` - a version identifier for the client software application.
- `callback` - function called on completion, successful or otherwise.

Callback parameters:

- `error` - on success, this value is null; on error, it is an `Error` object containing a descriptive error message.
- `data` - on success, this value is an object containing the information described below; on error, this value is null.

[6] EBU Tech 3366: The cross-platform authentication protocol
https://tech.ebu.ch/files/live/sites/tech/files/shared/tech/tech3366.pdf

- `client_id` - a unique identifier issued to the client by the Authorization Provider.
- `client_secret` - a shared secret value between the client and authorization provider.

requestUserCode

Associates a client with a user account. In response to this call, the Authorization Provider assigns a user verification code to the client, and returns a URL that the user should visit to authenticate themselves and input the user verification code.

```
requestUserCode(authProvider, clientId, clientSecret, domain, callback);
```

Parameters:

- `authProvider` - the base URL of the Authorization Provider.
- `clientId` - a unique identifier issued to the client by the Authorization Provider, returned in a previous call to registerClient.
- `clientSecret` - a shared secret value between the client and authorization provider, returned in a previous call to registerClient.
- `domain` - the domain name of the Service Provider.
- `callback` - function called on completion, successful or otherwise.

Callback parameters:

- `error` - on success, this value is null; on error, this parameter is an `Error` object containing a descriptive error message.
- `data` - on success, this value is an object containing the information described below; on error, this value is null.
  - `device_code` - the temporary device verification code.
  - `user_code` - the temporary user verification code, usually a short string of alphanumeric characters, which the user should enter after visiting the verification_uri.
  - `verification_uri` - The URL to be displayed to the user by the client.
  - `interval` - the minimum time the client should wait between making requests to obtain an access token, e.g., by calling `requestUserAccessToken`, in seconds.
  - `expires_in` - the length of time the `device_code` and `user_code` are valid, in seconds.

requestClientAccessToken

If the client does not want to associate itself with a user account, it can obtain an access token using the CPA “client mode”. This allows the service provider to identify the client, but the client is not linked to an authenticated user account.

```
requestClientAccessToken(authProvider, clientId, clientSecret, domain, callback);
```

Parameters:

- `authProvider` - the base URL of the Authorization Provider.
- `clientId` - a unique identifier issued to the client by the Authorization Provider, returned in a previous call to `registerClient`.
- `clientSecret` - a shared secret value between the client and authorization provider, returned in a previous call to `registerClient`.
- `domain` - the domain name of the Service Provider.
- `callback` - function called on completion, successful or otherwise.

Callback parameters:

- `error` - on success, this value is null; on error, this parameter is an `Error` object containing a descriptive error message.
- `data` - on success, this value is an object containing the information described below; on error, this value is null.
    - `access_token` - the access (or bearer) token value.
    - `token_type` - contains the value “bearer” to indicate that this is a bearer token.
    - `domain_name` - the name of the service provider suitable for display on the client device.
    - `expires_in` - the length of time the access token is valid, in seconds.

requestUserAccessToken

Requests an access token when the client is being associated with a user account, after a previous call to `requestUserCode`.

```
requestUserAccessToken(authProvider, clientId, clientSecret, deviceCode, domain,
callback);
```

Parameters:

- `authProvider` - the base URL of the Authorization Provider.
- `clientId` - a unique identifier issued to the client by the Authorization Provider, returned in a previous call to `registerClient`.
- `clientSecret` - a shared secret value between the client and authorization provider, returned in a previous call to `registerClient`.
- `domain` - the domain name of the Service Provider.

- `callback` - function called on completion, successful or otherwise.

Callback parameters:

- `error` - on success, this value is null; on error, this parameter is an `Error` object containing a descriptive error message.
- `data` - on success, this value is an object containing the information described below; on error, this value is null.
    - `access_token` - the access (or bearer) token value.
    - `token_type` - contains the value "bearer" to indicate that this is a bearer token.
    - `domain_name` - the name of the service provider suitable for display on the client device.
    - `expires_in` - the length of time the access token is valid, in seconds.

pollForUserAccessToken

Polls the authorization provider to receive a token for the user associated with this device, after a previous call to `requestUserCode`.

```
pollForUserAccessToken(authProvider, clientId, clientSecret, deviceCode, domain,
pollInterval, callback);
```

Parameters:

- `authProvider` - the base URL of the Authorization Provider.
- `clientId` - a unique identifier issued to the client by the Authorization Provider, returned in a previous call to `registerClient`.
- `clientSecret` - a shared secret value between the client and authorization provider, returned in a previous call to `registerClient`.
- `domain` - the domain name of the Service Provider.
- `pollInterval` - The time interval between requests, in milliseconds.
- `callback` - function called when the user has input the verification code, or if an error occurs.

Callback parameters:

- `error` - on success, this value is null; on error, this parameter is an `Error` object containing a descriptive error message.
- `data` - on success, this value is an object containing the information described below; on error, this value is null.
    - `access_token` - the access (or bearer) token value.
    - `token_type` - contains the value "bearer" to indicate that this is a bearer token.

- `domain_name` - the name of the service provider suitable for display on the client device.
- `expires_in` - the length of time the access token is valid, in seconds.

4.2.2. RadioTAG client library: radiotag.js

The radiotag.js library implements the HTTP requests defined by the RadioTAG protocol[7], and can be used from either client-side or server-side (Node.js) JavaScript code. The library provides the following functions:

tag

Creates a new tag, or bookmark, associated with a particular radio station.

```
tag(baseUrl, bearer, timeSource, time, accessToken, callback);
```

Parameters:

- `baseUrl` - the base URL of the Service Provider; the library appends the appropriate path to (e.g., /radiodns/tag/1/tag) create the full URL when making the request.
- `bearer` - a URI that identifies the radio station being tagged.
- `timeSource` - indicates the source of the time parameter, either “broadcast”, “user”, or “ntp”.
- `time` - the time of the tag.
- `accessToken` - the bearer token that identifies the client (and user) making the request, obtained from a previous call to either `requestClientAccessToken` or `requestUserAccessToken` in the cpa.js library.
- `callback` - function called on completion, successful or otherwise.

Callback parameters:

- `error` - on success, this value is null; on error, this parameter is an `Error` object containing a descriptive error message.
- `data` - on success, this value is an object containing the information described below; on error, this value is null.
  - `author` - the name of the service provider.
  - `title` - a title associated with the tag, such as the name of the radio programme.
  - `summary` - a brief description associated with the tag, such as artist and song title, or more information about the radio programme.

[7] RadioTAG Technical Specification, Version 1.0.0 Draft 7, June 2015 https://groups.google.com/group/radiotag-developers/attach/bcf47782ded6d432/rtag01_v100_draft_7.pdf?part=0.1&authuser=0

- `publishedDate` - the time of the tag.

getTags

Returns a list of all tags stored by the client, or user.

```
getTags(baseUrl, accessToken, callback);
```

Parameters:

- `baseUrl` - the base URL of the Service Provider; the library appends the appropriate path (e.g., /radiodns/tag/1/tags/) to create the full URL when making the request.
- `accessToken` - the bearer token that identifies the client (and user) making the request, obtained from a previous call to either `requestClientAccessToken` or `requestUserAccessToken` in the cpa.js library.
- `callback` - function called on completion, successful or otherwise.

Callback parameters:

- `error` - on success, this value is null; on error, this parameter is an Error object containing a descriptive error message.
- `data` - on success, this value is a JavaScript array of objects, each containing the information described below; on error, this value is null.
  - `author` - the name of the service provider.
  - `title` - a title associated with the tag, such as the name of the radio programme.
  - `summary` - a brief description associated with the tag, such as artist and song title, or more information about the radio programme.
  - `publishedDate` - the time of the tag.

4.2.3. Example

The client first tries to create a new tag, or bookmark, associated with a particular radio station. Because the client does not have an access token for the RadioTAG service, it passes `null` as the `accessToken` parameter.

```
var tagUrl = "https://radiotag.api.bbci.co.uk/tag";
var bearer = "0.c224.ce15.ce1.dab";
var timeSource = "user";
var time = new Date();

radiotag.tag(tagUrl, bearer, timeSource, time, null, tagCallback);
```

```
function tagCallback(err, tag) {
  var url;
  if (err.statusCode === 401) {
    url = err.authProviderBaseUrl;
  }
}
```

The Service Provider responds with a HTTP 401 “Unauthorized” error, and includes the Authorization Provider’s base URL in a WWW-Authenticate header. This is returned to the caller in `err.authProviderBaseUrl`.

```
var clientName = "Test Client";
var softwareId = "test_cpa_client";
var softwareVersion = "0.1";

cpa.registerClient(authProviderBaseUrl, clientName, softwareId, softwareVersion,
  registerClientCallback);

function registerClientCallback(err, client) {
  if (!err) {
    // Store client.client_id and client.client_secret
  }
}
```

Once the client has registered with the Authorization Provider and obtained its client ID and client secret, it can then start the process of association with a user account.

```
var clientId = client.client_id;
var clientSecret = client.client_secret;
var domain = "radiotag.api.bbc.co.uk";

cpa.device.requestUserCode(authProviderBaseUrl, clientId, clientSecret,
  domain, requestUserCodeCallback);

function requestUserCodeCallback(err, userCode) {
  if (!err) {
    // Ask the user to visit the URL in userCode.verification_uri
    // and input the userCode.user_code value.

    // store userCode.device_code and userCode.interval
  }
}
```

Now that the client has a device code it can poll the Authorization Provider to determine when the user has signed in and correctly entered the user code.

```
var clientId = client.client_id;
var clientSecret = client.client_secret;
var domain = "radiotag.api.bbc.co.uk";
var deviceCode = userCode.device_code;
var pollInterval = userCode.interval * 1000; // convert to milliseconds

cpa.device.pollForAccessToken(authProviderBaseUrl, clientId, clientSecret,
  deviceCode, domain, pollInterval, pollForAccessTokenCallback);

function pollForAccessTokenCallback(err, token) {
  if (!err) {
    // Store token.access_token for use with the Service Provider.
  }
}
```

Once the user has done this, the client receives an access token valid for use with the Service Provider. The client can then use the token to repeat its initial request.

```
var tagUrl = "https://radiotag.api.bbci.co.uk/tag";
var bearer = "0.c224.ce15.ce1.dab";
var timeSource = "user";
var time = new Date();
var accessToken = token.access_token;

radiotag.tag(tagUrl, bearer, timeSource, time, accessToken, tagCallback);

function tagCallback(err, tag) {
  if (!err) {
    // Store tag.author, tag.title, tag.summary, and display to the user
  }
}
```

4.2.4. Code and License

We have built reference implementations of the Authorization Provider, Service Provider, and Client, which are all publicly available on GitHub.

The Service Provider implements the RadioTAG protocol for bookmarking points in time in radio broadcasts and adding them to a user's playlist. To identify the client device and end user, RadioTAG uses bearer tokens obtained using Cross Platform Authentication.

The various software components are all available publicly in GitHub:

- Authorization Provider - https://github.com/ebu/cpa-auth-provider
- Service Provider - https://github.com/ebu/cpa-service-provider
- Demo Client - https://github.com/ebu/cpa-client
- CPA client library - https://github.com/ebu/cpa.js
- RadioTAG client library - https://github.com/ebu/radiotag.js

4.2.5. Towards standardisation

As part of research in Work Package 3, the BBC identified the need for an authentication protocol suitable for media devices, such as radios and TVs. Radios typically have small display screens, often limited to two rows of alphanumeric characters, and both radios and TVs are usually controlled using an infra-red remote control, which is difficult to use for text entry. These limitations mean that existing authentication protocols, such as OAuth 2.0, were not applicable to these kinds of devices.

Through the MediaScape project, the BBC was able to lead the development of the Cross Platform Authentication protocol, with a working group coordinated by the EBU. The Cross Platform Authentication protocol was released in September 2014 as an EBU recommendation. The CPA specification was submitted as a work item to ETSI in November 2015.

5. Abbreviations and Acronyms

API	↔	Application programming interface
BBC	↔	British Broadcasting Corporation
BR	↔	Bayerischer Rundfunk
CAS	↔	Central Authentication Service
CORS	↔	Cross Origin Resource Sharing
CPA	↔	Cross Platform Authentication
CSS	↔	Cascading Style Sheets
DIAL	↔	DIscovery And Launch«9»
DLNA	↔	Digital Living Network Alliance
DNS	↔	Domain Name System
DNS-SD	↔	DNS Service Discovery
DVB CM-COS	↔	Digital Video Broadcasting, Commercial Module (Companion Screen)
DoW	↔	Description of Work
EBU	↔	European Broadcasting Union
ETSI	↔	European Telecommunications Standards Institute
HbbTV	↔	Hybrid Broadcast Broadband TV«19»
HTML	↔	HyperText Markup Language
IETF	↔	Internet Engineering Task Force
IRT	↔	Institut für Rundfunktechnik
JS	↔	JavaScript
JSON	↔	JavaScript Object Notation
NOR	↔	Northern Research Institute (Norut)
OAuth	↔	Open Authorization«3»
P2P	↔	Peer-to-peer
PIN	↔	Personal Identification Number
RadioDNS	↔	RadioDNS is a industry standards body for hybrid radio
REST	↔	Representational state transfer
RTS	↔	Radio Télévision Suisse
SDK	↔	Software Development Kit
SOAP	↔	Simple Object Access Protocol
SSDP	↔	Simple Service Discovery Protocol«4»
TCP	↔	Transmission Control Protocol
TVP	↔	Telewizja Polska
UDP	↔	User Datagram Protocol
UC	↔	Use Case
UI	↔	User interface
UPnP	↔	Universal Plug and Play
VIC	↔	Vicomtech-IK4
VRT	↔	Vlaamse Radio- en Televisieomroeporganisatie
W3C	↔	World Wide Web Consortium
WP	↔	Work package
XML	↔	Extensible Markup Language