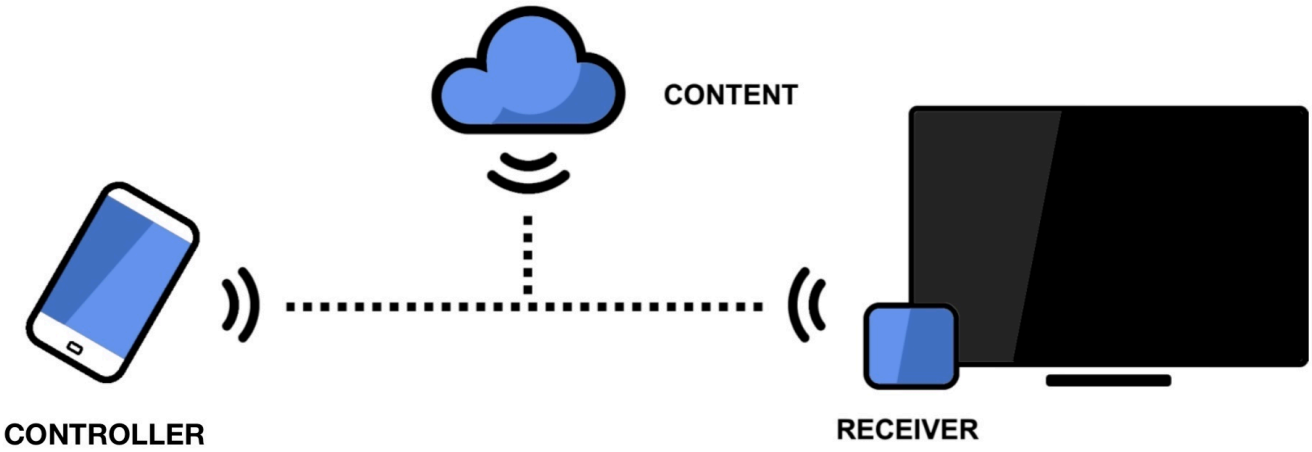
OCAST PROTOCOL V1

http://ocast.org

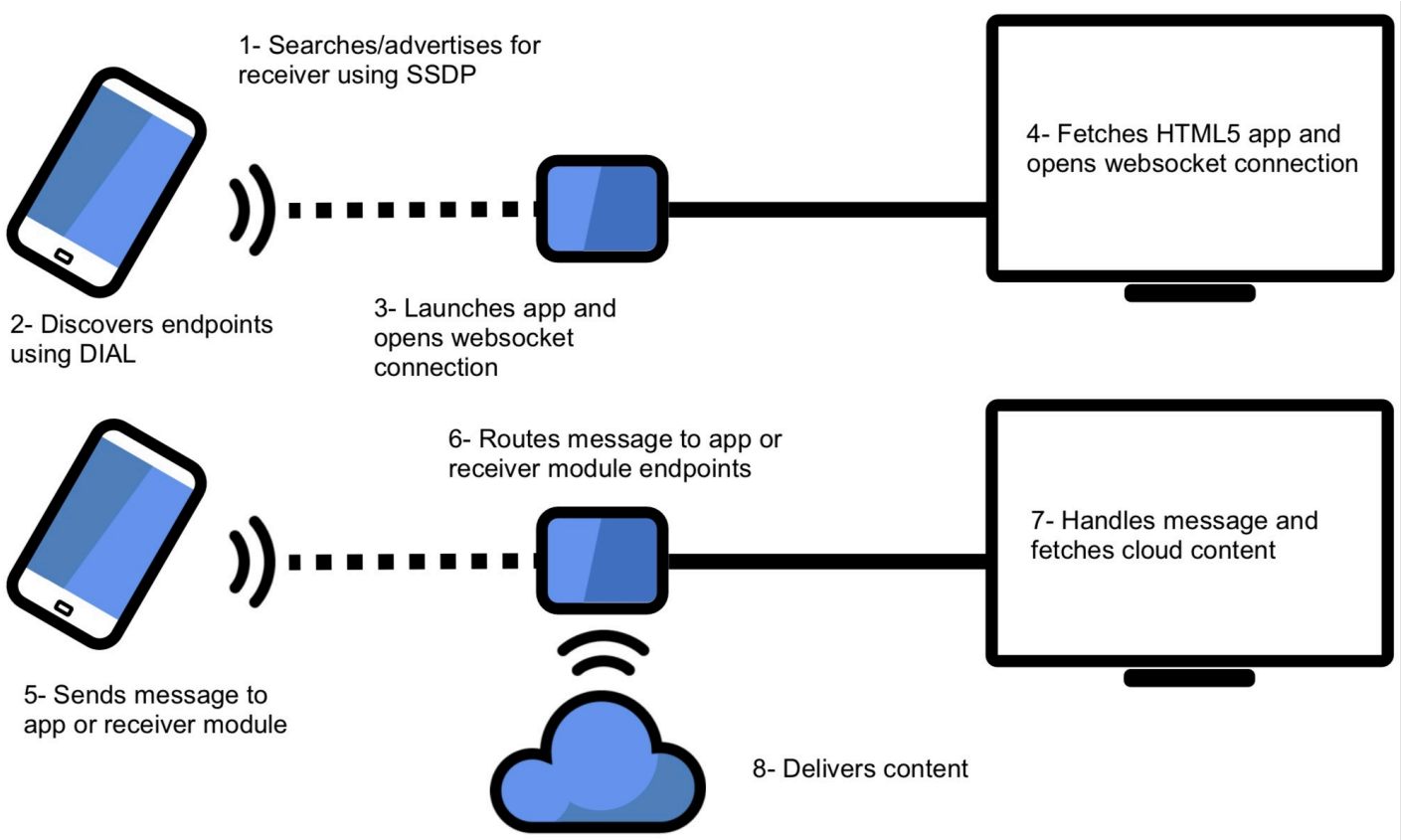
OCast is an ***open source cast*** solution that lets mobile phone users control – via native *controller* apps – the multimedia webapps they watch on their TV sets – via hardware *receiver*: dongles, sticks, set-top-boxes, consoles, etc. OCast is not about mirroring screens but controlling the overall multimedia experience (including audio and video streaming) using the *controller* as a remote for the *receiver*, seamlessly and securely.



- **Controller**: in most case the sender of commands, typically a smartphone application with the OCast SDK
- **Receiver**: the device that is controlled to play HTTP Web content with the OCast Web SDK and the Router component.
- **Router**: part of the Receiver that is responsible to handle WebSocket connections and routes messages to the good component(s)
- **DIAL** spec 2.1: http://www.dial-multiscreen.org/dial-protocol-specification/DIAL-2ndScreenProtocol-2.1.pdf?attredirects=0&d=1

End-to-end service

The goal of OCast protocol is to define the API between the controller and the receiver. It is based on a **websocket** connection between the receiver websocket server and the client controller.



Major elements

This document describes the various aspects of the **OCast Protocol v1**:

1. The **discovery** procedure, especially the elements that exposed the OCast service
2. The **device layer protocol** for controllers and receivers
3. The **application layer** that need to be implemented by the OCast Mobile SDK on the Controller and the OCast Web SDK on the Receiver (formerly the OCast WebApp API and OCast Media API), and the parts that need to be implemented by the OCast Mobile SDK on the Controller and the setting component on the Receiver (formerly the OCast Settings API)
4. The **security** aspects.

1. OCast Discovery

The basis of the OCast architecture is **DIAL**. Service discovery will allow the handset app locate and identify the OCast Receiver in the home network. It will rely on SSDP as in DIAL and mDNS.

SSDP

Somme specific parameters will enable the application to get the port information to use for the websocket connection.

Simple Service Discovery Protocol (SSDP) version 1.1 will be used, similarly as it is used by DIAL. SSDP Discovery must be based on DIAL specification 2.1.

Note on MSEARCH

From the Controller side, the first step for the service discovery must consist in the App sending a M-SEARCH request over UDP to the IPv4 multicast address 239.255.255.250 and UDP port 1900 including the Search Target header (ST) with the following value: urn:cast-ocast-org:service:cast:1.

From the Receiver side, the response to the M-SEARCH request must be conformed to the DIAL specification. Moreover, headers must be modified in order to present the OCast service as follow.

MSEARCH reponse header name	MSEARCH reponse header value
LOCATION	`http://IP_ADDRESS:HTTP_PORT/dd.xml`
ST	`urn:cast-ocast-org:service:cast:1`

```
HTTP/1.1 200 OK
LOCATION: http://IP_ADDRESS:HTTP_PORT/dd.xml
.....
ST: urn:cast-ocast-org:service:cast:1
USN: uuid:XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
```

Note on DeviceDescription XML

Device Description is retrieved on the URL in the M-SEARCH response LOCATION header. The response of this request must include a header 'Application-URL', specifying the IP address and port to be used for the HTTP Service Interface, but also a device description XML. Properties of this XML have to follow the definition in the following table. The fields in *italic* must be changed according to the OCast Receiver which is used.

Device Description property name	Device Description property value
friendlyName	*FRIENDLY_NAME*
manufacturer	*MANUFACTURER_NAME*
modelName	*MODEL_NAME*

For example:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Application-URL: http://IP_ADDRESS:HTTP_PORT/apps/
Content-Length: CONTENT-LENGTH
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0" xmlns:r="urn:restful-tv-org:schemas:up
np-dd">
    <specVersion>
    <major>1</major>
        <minor>0</minor>
    </specVersion>
    <device>
    <deviceType>urn:schemas-upnp-org:device:tvdevice:1</deviceType>
    <friendlyName>FRIENDLY_NAME</friendlyName>
    <manufacturer>MANUFACTURER_NAME</manufacturer>
    <modelName>MODEL_NAME</modelName>
    <UDN>uuid:XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX</UDN>
    </device>
```

The uuid must be 128bit long and generated according to the rfc4122 (https://www.ietf.org/rfc/rfc4122.txt).

DIAL AdditionalData

AdditionalData must be added to each Application XML Description of the GET /apps/<*APPID>* DIAL response to expose WebSocket URL and WebSocket protocol version.

AdditionalData name	AdditionalData value
X_OCAST_App2AppURL	`wss://IP_ADDRESS:WS_PORT/ocast`
X_OCAST_Version	`VERSION`

Ocast app2appurl tells to controller where to find the Ocast websocket server.

Ocast version tells to controller what kind of data it would find in the device layer and as well, in the application layer. It is a global version protocol. It may change according to this document.

```
<service xmlns="urn:dial-multiscreen-org:schemas:dial" xmlns:ocast="urn:cast-ocast-o
rg:service:cast:1" dialVer="2.1">
    <name>APPID</name>
    <options allowStop="false"/>
    <state>running</state>
    <additionalData>
        <ocast:X_OCAST_App2AppURL>wss://IP:4433/ocast</ocast:X_OCAST_App2AppURL>
        <ocast:X_OCAST_Version>1.0</ocast:X_OCAST_Version >
    </additionalData>
    <link rel="run" href="<RUN_URL>"/>
</service>
```

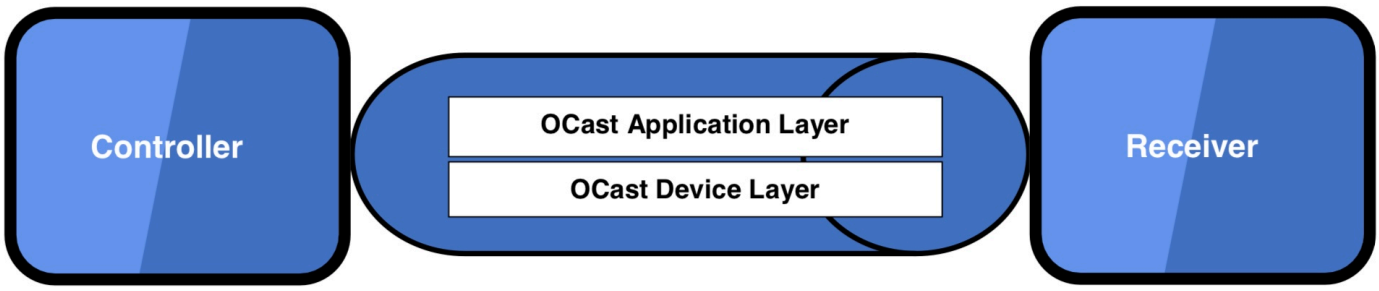
Service Ports

Port Name	Port Value	Protocol
SSDP WebServer	8008	HTTP
DIAL WebServer	8008	HTTP
OCast Secure WebSocket	4433	WSS

2. OCast Protocol

Overview

The OCast Protocol is designed to transport messages from one controller to a receiver and vice-versa. Therefore, a websocket is used as a communication channel. Messages will be composed of commands, events and replies. The OCast Protocol contains two layers:



OCast Device Layer

The OCast Device Layer describes the data transported in the Application Layer. The aim of this Device Layer is, for the Controller and Receiver, to be able to directly route application data to the right component. The behavior of data routing is described below.

OCast Device Layer JSON format:

```
{
"dst" : "string",
"src" : "string",
"type" : "string",
"id" : "long",
"status" : "string",
"message" : { object }
}
```

id	description
dst	Identify which component is addressed Destination identifier is defined as this: **browser / controller's uuid or * / settings**. It must be set by the sender of the message to address the component it wants to communicate with.
src	identify which component sent the message Source identifier is an enum of **browser / controller's uuid / settings**. It must be set by the sender to identify which component has sent the message. Controller's uuid is a unique identifier provider by each controller to identify themselves on the receiver (see identification routing procedure on next point). If the receiver wants to event to all controllers, it sets a wildcard as the destination identifier value.
type	what kind of message is transmitted Message type is an enum of **command / event / reply**. In correlation with the destination identifier, the Request name allows the message to be identified as event against reply for a controller, or command against event for the receiver.
id	Autoincrement id for each session (lifetime of the websocket) Message identifier is an auto increment long value that identifies each command in a session. It allows a controller to validate that its command had been treated by receiving a reply with the same message identifier, or later, an event with the same identifier for long time response. (-1 for error)
status	Determine whether or not the message transport failed on device layer (only for reply) Message status is only applicable for reply message type. It provides information on the transmission of the message. In case of a transport error, the value tells what kind of error occurs, otherwise 'ok' is set. See transport errors values in the following table.
message	information describing the application layer

Cases of transport errors:

Error	Message returned
JSON malformatted (json parse failure)	{'dst': null, 'src': null, 'type': 'reply', 'id':-1, 'status': 'json_malformat', 'message': { } }
Field missing (Mandatory field : dst, src, type, id, message)	{'dst': null, 'src': null, 'type': 'reply', 'id':-1, 'status': 'missing_mandatory_field', 'message': { } }
Malformatted Value (Mandatory value : (type not in enum (command/event/reply)) && dst not null && src not null)	{'dst': null, 'src': null, 'type': 'reply', 'id':-1, 'status': 'missing_mandatory_value', 'message': { } }
All other errors	{'dst': null, 'src': null, 'type': 'reply', 'id':-1, 'status': 'internal_error', 'message': { } }

In best effort, the router component must try to fill the dst, src and id field in the reply.

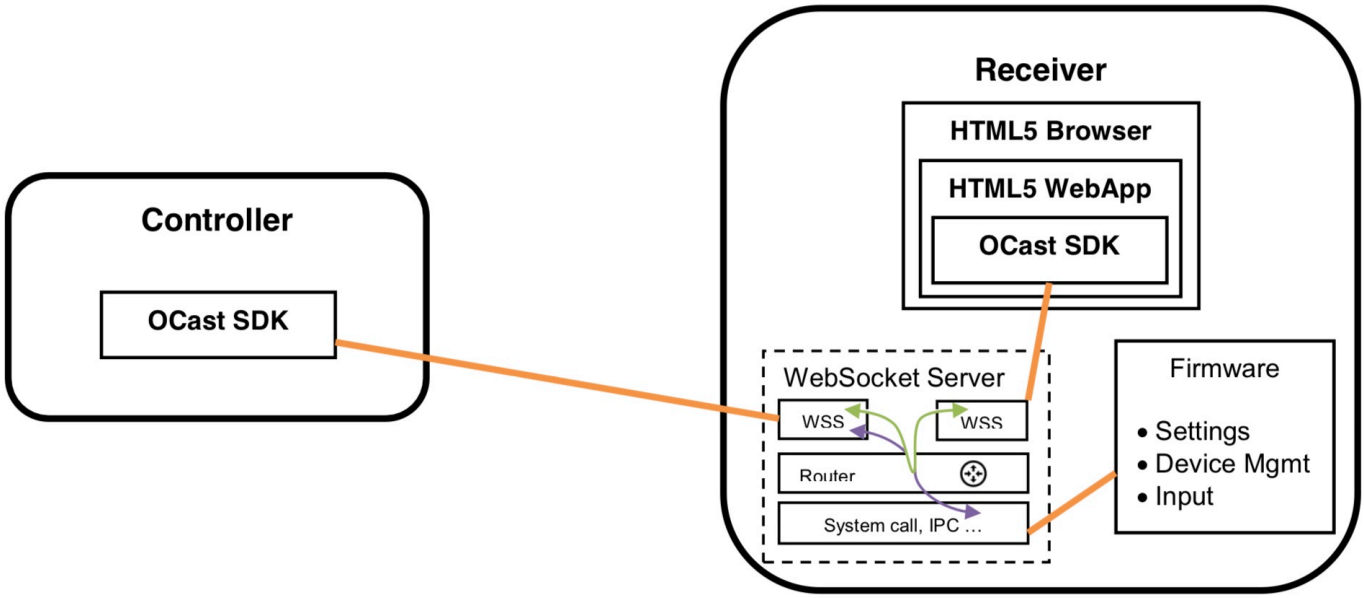
Status Value	Meaning
ok	No errors were found while processing the command
json_format_error	There is an error in the JSON formatting
value_format_error	This is an error in the packet, typically caused by a malformatted value
missing_mandatory_field	This is an error in the packet, typically caused by missing a field
internal_error	All other cases
forbidden_unsecure_mode	Packet has no right to access the required destination or service.

WebSocket Server (Message Router)

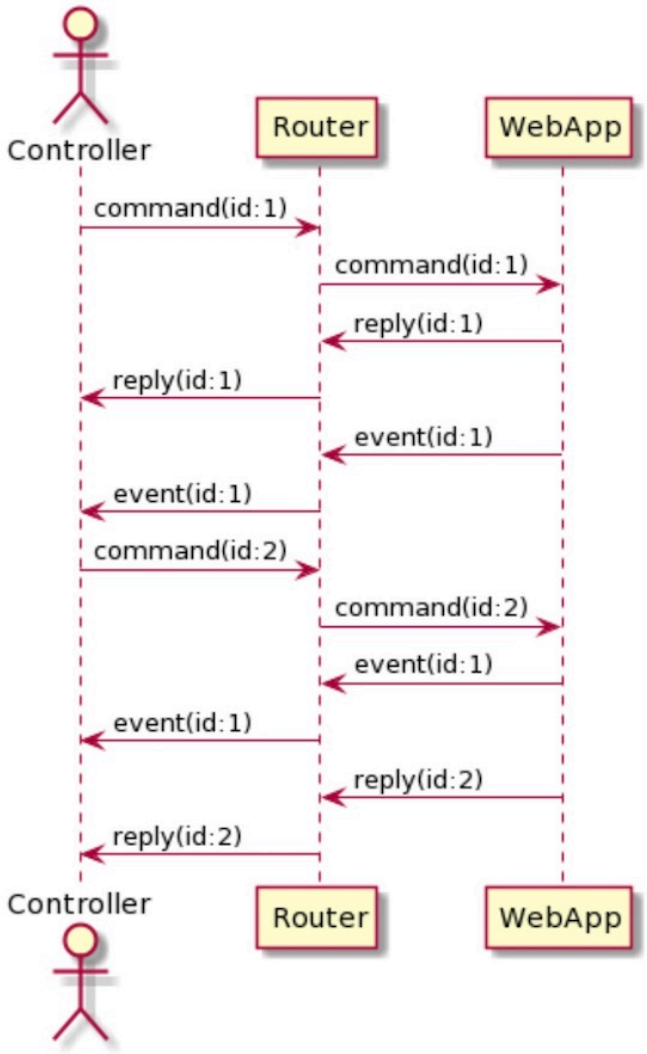
The WebSsocket Server is the end point of multiple WebSockets.

- From Controller to Receiver (1Way TLS WebSocket)
- From Receiver to WebApp

It is designed to route message from one socket to one or more socket(s):



For each message sent by the controller, a reply is sent by the receiver on the same socket. Event can be delivered on the same socket and between command and reply. Each command and reply is identified by an identifier to be re-associated by the sender of the command and the receiver of the reply.



Routing procedure

On the router implementation, a table should be maintained in order to associate which WebSocket is associated with which component identifier.

For instance, the component identifier **browser** must always be associated with the localhost WebSocket. Each controller must be identified by a **uuid** (unique universal identifier) which is generated by the controller. The uuid must be 128bit long and generated according to the rfc4122 (https://www.ietf.org/rfc/rfc4122.txt).

If a WebSocket is disconnected, the table must be cleaned by removing the corresponding row.

The **settings** component is a special identifier that may not be associated with a WebSocket. Therefore, when a message is addressed to the settings identifier, the router should call the specific system component and wait for a response from this component. The settings component is responsible of formatting a response following this documentation.

3. OCast Application Layer

The Application Layer is contained in the message json object. It is described by a service identifier which indicates what application is addressed, and a data object that represent Application dependent API.

This lets custom API to be implemented for partners for instance.

Message name	Message value	Required
service	string	V
data	object	V

The service field must a bundle identifier string (dot separated string) such as 'org.ocast.media'.

Basic implementation (example for mediaplayer implementation):

```
{
    (….)
    "message" : {
        "service" : "bundle_id",
        "data" : {
            "name" : "message_name",
            "params" : {
                "name": "value", …
            },
            "options": {
                "name": "value", …
            }
        }
    }
}
```

Application, identified by the service value, is responsible of parsing the data object and managing a response if any.

OCast WebApp API

In order to find out when the launched WebApp is available and ready to be accessed by the websocket, an event must be waited.

service: ***org.ocast.webapp***

Event	Params	Description
connectedStatus	`{ status: string }`	Send the WebApp status (connected, disconnected)

OCast Media API

Casting a media on the default Application receiver is based on the OCast Media API. It defines methods for loading, launching, and managing the playback of an HTTP HAS or HTTP Progressive DL media.

service: ***org.ocast.media***

Command	Params	Description	Reply
prepare	`{ url : url,` `frequency : uint,` `title : string,` `subtitle : string,` `logo: url,` `mediaType : string,` `transferMode : string,` `autoplay : bool }`	Prepare a play -frequency : in sec, 0 = no event -mediaType : { audio, image, video } - transferMode : { buffered, streamed }	`{ code : int }`
track	`{ type : string,` `trackId : string,` `enable : bool}`	Change a track of the current playback. Type is 'text' for subtitle, 'audio' for audio, 'video' for video track. Only one track of each type can be enabled at a time.	`{ code : int }`

play	`{ position : long }`	Play from an optional millisec position or from start	`{ code : int }`
stop	`{}`	Stop the current playback	`{ code : int }`
resume	`{}`	Resume a current playback.	`{ code : int }`
volume	`{ volume : float }`	Change the volume (between [0-1])	`{ code : int }`
pause	`{}`	Pause the current playback.	`{ code : int }`
seek	`{ position : long }`	Seek to the position (in millisecond)	`{ code : int }`
getPlaybackStatus	`{}`	Get the status of the current playback.	`{ code : int, volume : float, mute : bool, state : enum, position : long, duration : long }`
		Get	`{ code : int, title : string, subtitle : string, logo : url, mediaType : { audio, image, video }, subtitleTracks : [{ language : iso639-1/2, label : string,`

getMetadata	`{}`	metadata for the current playback.	`enable : bool,`
			`trackId : string}],`
			`audioTracks: [{`
			`language : iso639-1/2,`
			`label : string,`
			`enable : bool,`
			`trackId : string}],`
			`videoTracks: [{`
			`language : iso639-1/2,`
			`label : string,`
			`enable : bool,`
			`trackId : string}] }`

mute	`{ mute : bool }`		`{ code : int }`

Event	Params	Description

playbackStatus	`{ volume : float,`
	`mute : bool,`
	`state : enum,`
	`position : long,`
	`duration : long }`
		Sent at a rate defined by the frequency parameter of the prepare method.

metadataChanged	`{ title : string,`
	`subtitle : string,`
	`logo: url,`
	`mediaType : { audio, image, video },`
	`subtitleTracks : [{`
	`language : iso639-1/2,`
	`label : string,`
	`enable : bool,`
	`trackId : string }],`
	`audioTracks: [{`
	`language : iso639-1/2,`
	`label : string,`
	`enable : bool,`
	`trackId : string}],`
	`videoTracks: [{`
	`language : iso639-1/2,`
	`label : string,`
	`enable : bool,`
	`trackId : string}] }`
		Sent each time a metadata changed on the current playback.

OCast Settings API

For managing device identifier and receiver status, especially FW upgrade, a dedicated API is available.

service: ***org.ocast.settings.device***

Command	Params	Description	Reply
getUpdateStatus	`{}`	Retrieve the FW upgrade status	`{ code : int, state : string, version : string, progress : int }`
getDeviceID	`{}`	Retrieve the unique ID of the device (serial number)	`{ code : int, id : string }`

Event	Params	Description
updateStatus	`{ state : string, version : string, progress : int }`	Sent each sec for downloading, otherwise when state changes.

OCast SDKs

OCast SDKs take care of all low-level communication mechanisms between senders and receivers, including the discovery (based on Dial standard), two-way communication (over websockets), security (white-list based filtering), etc. This helps developers focus on high-level messaging integration in their apps.

The **controller** mobile apps just need to integrate the relevant SDKs:

- OCast-iOS SDK (Swift & Obj-C) for iOS devices
- OCast-Java SDK for Android devices.

The hardware **receiver** just needs to run html5 webapps on a server, with OCast-JS SDK installed

The receiver **emulator** OCast-Receiver-Emulator SDK can be installed on any Linux-based machine: Mac, PC or Raspberry PI 3 (model B).

4. Security aspects

All basic exchanges are made through a one way TLS websocket.

5. About

OCast protocol, this documentation and the Open Source SDKs are brought to you by **Orange** teams.