The Smart Device Specification for Remote Labs

Christophe Salzmann, Sten Govaerts, Wissam Halimi, and Denis Gillet
School of Engineering
Ecole Polytechnique Fédérale de Lausanne (EPFL), Station 9, 1015 Lausanne, Switzerland
Emails: {christophe.salzmann, sten.govaerts, wissam.halimi, denis.gillet}@epfl.ch

***Abstract*—This paper presents the Smart Device specification to interface with remote labs. To encourage the broader sharing of remote labs, the Smart Device paradigm decouples the client from the server and provides well-defined interfaces between client and server. Such Smart Device services are exposed on the Internet and enable interoperability with client applications, other Smart Devices and external services (e.g. a booking service). This papers presents the extensible and platform-agnostic specification of the Smart Device services and internal functionalities. The Smart Device specification contains sufficient service metadata to enable the automatic generation of basic client applications. The specification is illustrated through an example and first implementations of the specification are presented.**

Keywords*—*Remote labs, Smart Device, Specification, Remote control, Web Service, Websockets

I. Introduction

The Smart Device paradigm originates from the RFID and sensor world, where one adds information to a static sensor to enhance its functionality. Thus, instead of a thermometer just returning a voltage, a sensor provides additional information such as the sensor ID, a timestamp or a data range. Thomson [1] defines that smart objects connected to the Internet need some or all of the following capabilities: *i)* communication, *ii)* sensing and actuating, *iii)* reasoning and learning, *iv)* identity and kind, and *v)* memory and status tracking.

We extended Thomson's proposition to support more complex devices that are using web-based technologies, namely to support remote labs [2]. We used this paradigm to specify on one hand the remote lab interfaces exposed on the Internet and on the other hand its internal functionalities [3]. Since the Smart Device interfaces are well-defined, a Smart Device becomes interoperable with other Smart Devices, external services and client applications. Such interoperability fosters reuse of applications and external services, and can provide extra functionality to any Smart Device (e.g. booking and authentication), simplifying the development of remote labs. The specification is designed to enable any client application developer to easily interface with a remote lab. Moreover, the specification of the services is machine readable, enabling the automatic generation of a skeleton of the client application. The actual implementation of the specification, as well as the remote lab software and hardware implementation, is left to the lab owner's discretion.

This paper presents the Smart Device specification together with an example and multiple software packages demonstrating implementations on different software and hardware platforms. The specification uses open protocols, is easily extensible and makes use of a slightly modified version of the Swagger [4] web service description language to support WebSockets. Note that the specification was first documented in deliverable D4.1 of the European FP7 project, Go-Lab [5].[1]

This paper is organized as follows: first we summarize the Smart Device as a paradigm for remote labs. Then, we discuss the architecture and interoperability features enabled by the Smart Device. The next section is dedicated to describing the Smart Device specification for remote labs in detail. Examples and extensions are provided in the last section.

II. Smart Devices Paradigm

The Smart Device paradigm revisits the traditional client-server architecture, on which many remote lab implementations rely. The main differences between existing implementations and the Smart Devices' are first the complete decoupling between the server and the client, and second the server representation as a set of well-defined services and functionalities that enable interoperability [3], [6], [7]. Similar approaches were proposed at the sensor/actuator level to enable the plug and play mechanism for Smart Electronic Transducers which provides electronic data sheets describing themselves [8]. This paper proposes a specification that handles the interaction between clients and servers at the service level.

The decoupling removes the umbilical cord between the client and the server so that they can live their own separate lives. While in a traditional client-server architecture [9], the server and client share a specification that is often uniquely used by them. On the contrary, the Smart Device paradigm defines one common specification that is shared by all Smart Devices. This reuse of a common specification and the client-server decoupling alleviates most of the problems developers are facing when the client application needs to be adapted to new OS/platforms, or if the client application is to be integrated in other environments such as learning management systems (LMS), or simply if additional features are added to the server. Furthermore, interoperability with, and reuse of existing applications and services becomes possible when labs share a common specification.

Smart Devices mainly provide web services to access sensors and actuators. Traditional solutions often provide a monolithic interface without the possibility to specifically access a given sensor or actuator [10]. The Smart Device specification fully describes the Smart Device from a client point of view by specifying only the interfaces, not the inner working of the lab, which is left to the lab owner's discretion. The Smart Device specification is agnostic about the server-side hardware, but re-engineers the software component by adding 'intelligence' to handle complex tasks through the API.

[1] The Go-Lab project, http://www.go-lab-project.eu

There is no assumption regarding the communication channels for Smart Devices [11]. The Internet is the de facto choice for online labs [2], [12]. In addition, open Web technologies enable a broader compatibility and adoption, while proprietary technologies break the core ubiquitous access requirement.

The Smart Device may not necessarily provide a User Interface (UI), but often proposes a minimal client UI. Thanks to the interoperability provided by the Smart Device specification, client applications can be developed to operate with different Smart Devices promoting reuse. Due to their ubiquity, web browsers are the preferred environment to render the client UI. There is often a direct relation between the Smart Device sensors and actuators, and the client app rendering their information. For example, an oscilloscope app renders the voltage evolution measured by a sensor of the Smart Device. In general, the Smart Device paradigm defines an ideal autonomous device which provides internal functionalities and that can be accessed through well-defined services.

III. Smart Devices for remote labs

A generic Smart Device can already be seen as an autonomous online lab. On the other hand, it does not target a specific purpose and therefore the expected requirements may not be satisfied. The principal aim of remote labs is to represent its partial or full state , at the client side, and to enable real-time interaction. For example, it could be implemented in the form of a simple oscilloscope depicting the temporal evolution of a given sensor or a full 3D representation of the system. Interacting with the physical lab by directly controlling actuators or indirectly through a supervision stage (local controller or other logic) should also be possible. When considering remote labs, the client side that renders the server information needs also to be taken into account. Remote lab client applications are typically running in a Web browser. This specific choice of open Web technologies enables a broader compatibility and favors adaptation as well as adoption. Proprietary technologies (e.g. Java or Flash) should be avoided since they limit the ubiquity of the solution. The Smart Device paradigm enables the rethinking of such an interface into a Web 2.0 interface.

The Smart Device provides interfaces to remote labs for clients and external services through well-defined *services* and internal *functionalities*. A precise definition of these services and functionalities permits the decoupling between the client and the server. Some of these services and functionalities are meant for the client application, while others are meant for the Smart Device. The Smart Device's additional intelligence and agility mainly comes from these internal functionalities. The services and functionalities definition enables anyone to design his/her own interface for accessing the Smart Devices for any remote lab.

A service represents, for instance, a sensor or an actuator exposed to the outside world (e.g. a client) through the API. Services are fully described through metadata, so that a client can use them without further explanation. A functionality is an internal behavior of the Smart Device. There may be communication between internal functionalities and client applications or external services through Smart Device services. While the required services are fully specified, the functionalities are only recommended and best practice guidelines are provided.

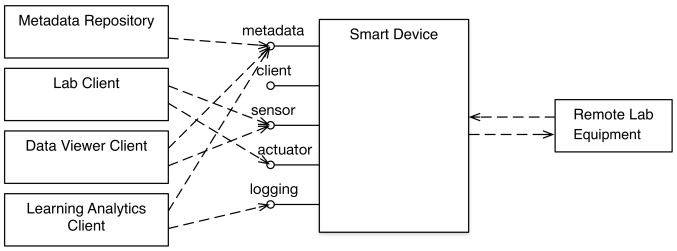

Fig. 1. UML Component diagram of different clients making use of the most common Smart Device services (arrows represent calls).

For example, imagine an actuator service that enables the client application to set the voltage of a motor, and a functionality that checks if the maximum voltage is not exceeded. The actuator service is well described by the Smart Device metadata (see Subsection V-C). The internal validation is left to the lab owner's discretion, since it will be mainly ad-hoc. Still, such a mechanism has to be implemented to ensure the protection of the server and the connected equipment.

The Smart Device specification (see Section V) defines the communication and interfaces between the client and server, and sufficient information is provided to generate client applications or reuse existing client applications. Since the specification is common to many Smart Devices, client apps are not tightly coupled to one server, encouraging interoperability and reuse.

IV. The Smart Device Architecture

The Smart Device specification provides a set of well-defined interfaces that enable communication between the remote lab, external services and applications. Figure 1 illustrates a basic architecture with interaction examples that abstract the implementation of a remote lab, by providing a set of required and optional interfaces. The specification does not define the communication between the Smart Device and the Remote Lab equipment in Figure 1. The communication on the left side of Figure 1 is what the Smart Device specifies, namely the protocols and data formats of the interfaces of the Smart Device (i.e., the 'metadata', 'client', 'sensor', 'actuator' and 'logging' interface in Figure 1). For instance, a metadata repository can retrieve the metadata of any Smart Device, index it and provide a lab search engine. Because the interfaces are well-defined, client apps can be reused among Smart Devices. For example, one Data Viewer Client or Learning Analytics Client could retrieve data from any Smart Device and present it to the user. Additionally, a metadata format that describes the Smart Device, its functionalities and its services is specified. Section V will elaborate on this metadata and each service and functionality in detail. Below, we will discuss how Smart Devices enable interoperability in the Go-Lab infrastructure.

1) The Smart Device in the Go-Lab Infrastructure: As described above, the well-defined interfaces of the Smart Device, ensure that a client app and a service can communicate with any Smart Device. This section will discuss such a concrete scenario with the Go-Lab platforms [13] that interact with the Smart Device. Of course, any other service, platform or client could make use of these interfaces to create features beyond what is presented below. The Go-Lab overview component

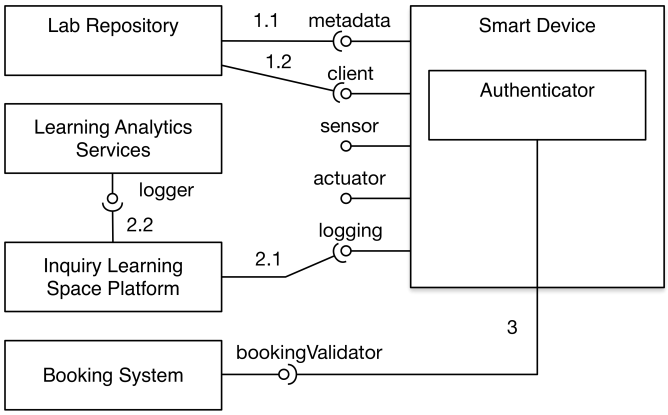

Fig. 2. UML Component diagram of the interactions between different Go-Lab services and the Smart Device.

diagram is shown in Figure 2. It depicts: (1) the Lab Repository [13], which is a portal where teachers can find online labs and resources to use in combination with these labs in their courses; (2) the Inquiry Learning Space (ILS) Platform [13] that provides a collaborative editor to assemble a learning activity for students with the Lab Repository resources; (3) the Learning Analytics Services [14], which are collecting tracked user activities and analytics results; and (4) the Booking System that provides a common UI for teachers to reserve remote labs. In addition to enabling user interaction with the remote lab equipment, the Smart Device enables the following features in infrastructures such as the Go-Lab infrastructure:

a) Publishing labs on the Lab Repository: A lab owner can publish any lab on the Go-Lab Lab Repository [13][2], which provides a searchable catalogue of online labs. If a lab supports the Smart Device specification, its metadata can be retrieved and parts of the lab registration form can be automatically completed. Additionally, the client apps to control the lab can be added automatically, see step 1.1 & 1.2 in Figure 2. The annotated lab metadata can then be exploited for search, but also to support the learning analytics services.

b) Tracking user activity: The Smart Device contains a user activity logging service that enables the delivery of learning analytics. Step 2.1 shows how the Inquiry Learning Space (ILS) Platform [13] retrieves Smart Device user activity logs and passes them to the Learning Analytics Services where the user activity is stored and can be further analyzed.

c) Booking a lab: The Smart Device itself does not necessarily contain a booking mechanism, but can use existing booking mechanisms. When booking is required, a user retrieves an authentication token from the Booking System with which she can authenticate to the Smart Device. The Smart Device only contains logic to validate tokens. Step 3 illustrates that the Smart Device has an Authentication component that validates tokens with the Booking System.

Note that the above features will only be available if the corresponding Smart Device services are implemented. Publishing and retrieving lab metadata will work for any Smart Device because the metadata service is required, but the other features depend on optional services. In Section V, we will further elaborate on the optional and required Smart Device services.

[2]Golabz, http://www.golabz.eu

V. The Smart Device Specification

This section presents selected parts of the Smart Device specification in more detail. The complete Smart Device specification is available at https://github.com/go-lab/smart-device-metadata/raw/master/smart-device-specification/Smart_Device_specification.pdf.

First, the communication protocol and the terminology used are described. Then, we will elaborate on the Smart Device well-defined services and internal functionalities.

A. Data Transfer Protocol

The goal of the Smart Device is to enable access to remote laboratories via the Internet. The targeted client application is a Web enabled client, which can run on a tablet. We rely on open, standardised Web protocols to provide the data transfer between the Smart Device, external services, and applications to avoid dedicated plug-ins or customer lock-in. Typically, widely used candidates are HTTP and recently WebSockets. The problem with most HTTP-based Web Services is that they follow a synchronous request-response schema. Hence, data can often only be 'pulled' from the server, and the server cannot initiate a 'push' of information to the clients. However, remote laboratory experiments, often require asynchronous data transfer, e.g. a lengthy experiment should be able to push its results to the clients upon completion. HTTP solutions are often inefficient, e.g. via long polling [15].

WebSockets [16] on the other hand are asynchronous by nature and allow both pushing and pulling. This provides a bidirectional, full-duplex communication channel. Although WebSockets are a recent technology, they are supported by all modern browsers[3]. Since WebSockets suppport both push and pull technologies efficiently and often with less programming effort than HTTP-based services, the Smart Device specification uses the WebSocket protocol. Only the metadata service that defines the other services (see Subsection V-C) will be provided via HTTP GET to enable easy text retrieval.

B. Terminology and Concepts

The following terminology and concepts are used:

- The terms *sensors* and *actuators* reflect the travelling direction of information relative to the Smart Device. For example, a sensor enables the reading of a thermometer. An actuator enables the setting of a value, e.g. setting a motor voltage.
- Sensors and actuators can be *physical* (temperature sensor), *virtual* (computed speed derived from a position measurement) or *complex*, i.e. an aggregation of sensors/actuators (the front panel buttons of an oscilloscope or a 3D accelerometer).
- Both sensors and actuators can be configured, see the metadata service in Subsection V-C.

[3]Can I use Web Sockets?, http://caniuse.com/websockets

C. Metadata Service

The metadata service is a required service that is at the core of the interoperability provided by the Smart Device specification. The requirements of the metadata are:

- Describe the lab (e.g., the contact person and the goals), which can be useful to allow automatic indexing by search engines (see Subsection IV-1).
- Describe the integration with external services (e.g., authentication with a booking service)
- Describe the concurrency mechanisms (e.g., are lab observations allowed, while someone is doing an experiment?)
- Describe and define the provided services (e.g., specify the service requests and responses formats)
- Be easily extensible to enable adding extra services

First, we survey different Web service description languages and highlight our choice. Afterwards, the metadata design choices and the metadata for the services is described, and how metadata can be added for additional services.

1) Comparison of Web Service Description Languages: Several options to describe Web service specifications have been surveyed with the goal not to reinvent the wheel, but to use open, robust and complete specifications. Furthermore, some specifications already allow the automatic generation of client applications. Since no Web service description languages specific to the WebSocket protocol were found, SOAP and REST-based description languages were considered.

One of the most popular Web service description languages is WSDL[4], which originally strongly focuses on SOAP, and provides support for REST since version 2.0. However, currently limited software is available for WSDL 2.0[5]. Other description languages are dedicated to RESTful services. WADL [17] can be considered as the REST equivalent of the original SOAP-only WSDL. RSDL[6] is more focused on the structure of the Web service URIs. While RAML[7] relies on markdown and JSON Schema[8].

Since all above mentioned languages were hard to use WebSockets with, we have opted for Swagger v1.2[9]. Swagger is a JSON-based description language meant for RESTful APIs, but it was easily extensible to WebSockets, while conserving all of Swagger's features. Since Swagger aims to describe web services for both humans and computers, it strongly focuses on automatically generating user interfaces, which is one of our goals. Using JSON Schema, Swagger specifies the data format of requests and responses. Due to its large and growing list of supporting software, Swagger is growing in popularity. The specification is open and the community is currently finalising an updated version. In the remainder of this section, we will elaborate on how we have applied and extended Swagger for the Smart Device Specification.

2) Smart Device Metadata Design Choices: Based on the requirements elicited above, the following main design choices were made:

- *Sensor & actuator metadata service:* The metadata that describes the available sensors and actuators is provided by separate services. In this way a developer of a simple Smart Device needs just to edit a few lines of metadata and does not need to add complex descriptions and models of actuators and sensors. The Smart Device software packages provided by Go-Lab (see Section VII) already implement these services, so the developer can just edit this implementation, which also keeps this metadata very close to the actual sensor and actuator implementation.
- *Service names:* Each service requires a method name, and each request and response of a service needs to pass this method name (e.g. the service for the sensor metadata is called 'getSensorMetadata'). By passing this name, a WebSocket can be reused (channeled) by different services since the requests and responses can be identified by method name. Additionally, the method names are used to control access to services.

3) General Smart Device Metadata Specification: The official Swagger RESTful API documentation specification can be found on https://github.com/wordnik/swagger-spec/blob/master/versions/1.2.md. The Swagger specification is typically split over multiple files per service and served in the path of a REST service. Since WebSockets are not hierarchically organized in different URLs, we have opted to provide one specification file, containing the general metadata and all service-specific metadata.[10] This section will introduce the general structure of the adapted Swagger file. However, code samples and exact field names are omitted for brevity, but are available in the full specification.[11] The metadata consists of six parts:

a) Swagger-Related Metadata: Swagger requires to declare the version of Swagger and the API. The version of Swagger should not be changed by the developer.

b) General Metadata: These default Swagger fields provide information about the lab, such as the name, a short description, a contact person, and licensing information.

c) API Metadata: The root URL path of the Smart Device services is described and all services are defined. Each service will be described from Subsection V-F to V-J.

d) Authorisation Metadata: Swagger supports common REST-based authentication and authorisation mechanisms, e.g. OAuth. All these mechanisms can be used in the Smart Device. For instance, in the Go-Lab booking system, we are using a token-based authorisation, which can be modeled with Swagger's `apikey` type since the booking token is a sort of temporary API key for the duration of the booking.

[4]Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl

[5]Web Services Description Language – Wikipedia, http://en.wikipedia.org/wiki/Web_Services_Description_Language

[6]RESTful Service Description Language (RSDL), http://en.wikipedia.org/wiki/RSDL

[7]RESTful API Modeling Language (RAML), http://raml.org/

[8]JSON Schema specification – JSON Schema: core definitions and terminology json-schema-core, http://json-schema.org/latest/json-schema-core.html

[9]Swagger website, http://swagger.wordnik.com/

[10]Metadata specification examples for Smart Devices are available on GitHub: https://github.com/Go-Lab/smart-device-metadata

[11]The full Smart Device specification is available at https://github.com/go-lab/smart-device-metadata/raw/master/smart-device-specification/Smart_Device_specification.pdf.

e) Concurrent Access Metadata: We have extended Swagger to model the concurrency models of remote labs. Different concurrency schemes exist and it is up to the lab owner to decide on an appropriate scheme. One can interact with a lab in a synchronous or asynchronous way. In a synchronous lab, the users are interacting directly with the experiment and are aware of actions of other concurrent users. When in the asynchronous mode, the user typically prepares an experiment, submits it, waits to get results back, and is not aware of other users. The rest of this metadata is for synchronous labs, since asynchronous labs can deal internally with concurrency issues. Typically two concurrency schemes are possible: 'concurrent' and 'roles'. Either users are allowed to use the experiment at the same time or different user roles control the access. Each role has a name, can declare which services will be accessible for a user with that role and a mechanism to select the role. Different mechanisms have been identified to switch roles:

- *Fixed role:* The user cannot be promoted from one role to another, e.g. the teacher can control the remote lab but the students can only observe.
- *Dynamic role:* The role can change during the session, e.g. a user observing can later control.
- *Race:* The first user who accesses when nobody is using it, gets control. When occupied, the user has to retry.
- *Queue:* Upon access, the user is added to a first-come-first-served waiting queue.
- *Interruptor:* The user can abort the session of another user and take control of the Smart Device.

f) Models: This section lists all data models in JSON Schema used in the service requests and responses.

4) Service Metadata Specification: This section discusses how a service can be added as a JSON object in the API metadata on a high level (for details, refer to the full specification). Optionally, new data models need to be declared in the models section. However, we have tried to design the specification so that for simple Smart Devices, developers do not need to learn how to describe a service in Swagger. The specification provides reusable service metadata descriptions and models for the sensor, actuator and logging services.

A new API object needs to contain the path, description, and also an optional 'protocol' field that the Smart Device specification has been extended to support the WebSocket protocol. Then a list of all operations of the service is specified and its response messages that describe the error messages (relying on HTTP status codes [18]). Each operation can specify the protocol method, in case of WebSockets this is typically 'Send', and one can define the type of WebSocket: text or binary. Binary WebSockets can make the transmission of binary data much more efficient, e.g. for video streaming. Additional documentation can be provided in the 'summary' and 'notes' fields. Next, the service arguments and results can be configured using JSON Schema primitives[12], or the ID of a model from the models metadata section. One can also model the response format using any Internet Media Type [19], e.g. for a service that returns images. The service input arguments are typically represented as a data model. Simple request models are provided, but more complex models can be defined when needed. More information on adding a new service can be found in the Swagger specification [4], the JSON Schema specification and the available GitHub examples which illustrate how we have extended Swagger.[11]

D. Sensor Metadata Service – getSensorMetadata

As mentioned, the sensor and actuator metadata are provided via separate services and not in the metadata description itself. In this section we will elaborate on the sensor metadata.

The service is called 'getSensorMetadata', and can be called like most Smart Device services with a JSON object by specifying the 'method' field, and an optional authentication token in case booking is required. As mentioned before this method field enables the reuse of one WebSocket to channel multiple services. The service returns an array describing each sensor exposed to the outside world. Each sensor contains:

- *The ID* to identify the sensor, e.g. '3D-acc'.
- *The full name*, e.g. '3D acceleration'.
- *The description*, e.g. 'the robot arm 3D acceleration'.
- *The WebSocket type* is 'text' or 'binary' (e.g. for video).
- *The response type* of the sensor service for the sensor defined as an Internet media type [19], e.g. a webcam sensor using JPEG compression uses `image/jpeg`.
- *The measurement value array* will contain a single value for a simple sensor like a thermometer, but for a complex sensor like an accelerometer, the array contains for example 3 elements for the X-Y-Z acceleration. Values are described with a name and unit. Since the set of possible units is almost infinite, we recommend to use the SI units [20] and the SI derived units.[13] Optionally, a last measured time stamp and a range minimum, maximum and iteration step of the range in which the values safely operate, can be added. Furthermore, for continuously measured values the frequency at which the measurement is updated can be provided in Hertz (s^{-1}).
- *The configuration parameters* can be used to adjust the sensor when requesting a sensor value (see Section V-F). Each parameter has a name and data type as a JSON Schema primitive, array or data model for complex parameters, e.g. to configure the video resolution.
- *The access mode* describes how the sensor can be accessed, e.g. some sensors can be measured once (pull) while others provide a continuous data stream (push or stream). For 'push' sensors, one can specify the nominal update interval and whether the measurement frequency can be modified by the user.

Both sensors and actuators can be configured, which means that the information can be sent and received even for the sensor. For example, the image resolution of a webcam sensor can be configured. Similarly, for actuators some aspects may be set through configuration (e.g. the gain of a power amplifier could be configured), while the actual value is set through the actuator value itself (see Subsection V-G). Typically sensors and actuators are rarely configured.

[12]JSON Schema specification – JSON Schema: core definitions and terminology json-schema-core, http://json-schema.org/latest/json-schema-core.html

[13]SI Derived Units – Wikipedia, http://en.wikipedia.org/wiki/SI_derived_unit

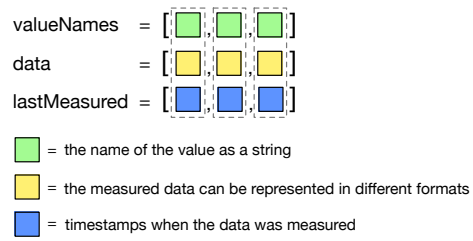

Fig. 3. Sensor and actuator data structures.

Streaming video of the experiment is an essential service that a Smart Device should provide through a sensor. We recommend that such sensor treats the video image as an encoded image, for example JPEG encoded. Using JPEG encoding results in binary data which either should be transmitted through a binary WebSocket (recommended), or it is BinHex'ed prior to sending it using a textual WebSocket. If further processing is required at the client side, a pixmap (pixel array) could be used, this at the cost of being 10% to 90% larger in size [21].

E. Actuator Metadata Service – getActuatorMetadata

As mentioned, the actuator metadata is also provided via a service, named 'getActuatorMetadata'. The service is very similar to the sensor metadata service, so we will only discuss the difference in the service response: *the input type* expresses what data type can be used for a specific actuator in the actuator service. By default this is JSON, but it can be set to any Internet Media Type [19]. This replaces the response type of the sensor metadata service.

F. Sensor Service – getSensorData

The sensor and the actuator data services are at the core of the Smart Device interaction and both are quite similar. They handle the main data exchange between clients and the Smart Device. Both services in combination with their metadata services enable developers to create apps that can adapt to different Smart Devices, enabling app reuse and interoperability. Similarly, different apps could be developed for a Smart Device. For example, for a Smart Device that provides a temperature measurement every second, one app could just update a text field, while another app could visualize the temperature evolution over time. This difference in app functionality requires absolutely no change on the Smart Device services. Furthermore, using the sensor metadata service, these two proposed apps could be made interoperable and reusable with any Smart Device.

Different sensors and actuators exist:

- *Real:* represents a physical sensor, e.g. a thermometer.
- *Virtual:* represents a computed sensor, e.g. a speed measurement derived from a position measurement.
- *Complex:* represents the aggregation of sensors/actuators, e.g. buttons on the front panel of an oscilloscope.

The data structure returned by a sensor or sent to an actuator may vary depending on the number of values and the measurement data structures. The data structure (see Figure 3) contains three fields to enable flexible data representation. In the 'valueNames' field, the names of the sensor or actuator measurement value are listed as returned by the sensor or actuator metadata services (see Subsection V-D). Then, the actual data for each value is listed. Finally, the optional 'lastMeasured' array contains the timestamps when a value was measured. This timestamp array should not be included when sending data to set an actuator. The data as well as the 'lastMeasured' timestamps are listed at the same array index as the value name, as indicated by the dashed lines in Figure 3. The elements in the data array can be in different formats: (1) *a single value*, e.g. temperature; (2) *an array of values* representing a set of single values over time, e.g temperatures over the last minute; (3) *aggregated values* representing a sensor or actuator that returns multiple values, e.g. a 3D accelerometer; (4) *an array of aggregated values* representing a set of aggregated values over time, e.g. the 3D acceleration over the last minute; and (5) *complex data structures* are used when sensors and actuators require input and output not definable with primitive variables or arrays, e.g. for complex JSON objects or binary data. This data representation was chosen, because flat array based data can be more efficient to process than complex data structures interleaved with timestamps.

As an example of a complex data structure, a webcam can be modelled as a single value sensor that returns a compressed image, as an array of values based on the image bitmap or as a binary value with JPEG encoded data. The choice between the three representations is up to the lab owner.

A request to the getSensorData service is more complex than the previous services due to possible authentication, concurrency and configuration settings. Optionally, an access role from the concurrency role list (see V-C3e) can be passed. If no accessRole is available, the Smart Device can decide the role. The Smart Device will decide whether these rights can be granted and reacts accordingly.

The getSensorData service will return the data in the above described data format (see Figure 3) together with the method name, sensor ID and access role to foster possible WebSocket reuse. This is in case the user has the controller role. But when the user is an observer and does not have access to the measured data the service can optionally provide extra waiting information that can be used to display how long the user has to wait and how many people are in front of her (e.g. the queue size, position and waiting time left). Furthermore, the sensor configuration might be used (e.g. for a video sensor), if it is described in the sensor metadata. For example, this can be very useful to adapt to the client screen size and network speed by reducing the transmitted image resolution and compression (if configurable). Similarly, the data transmission pace could also be controlled. If the user temporarily needs to throttle the video stream, the client can ask the Smart Device to reduce the number of images sent per second by setting the update frequency (see Subsection V-D). The sending may even be interrupted by setting the update frequency to 0 Hz. It is up to the application developer to take advantage of these features.

G. Actuator Service – sendActuatorData

The actuator service is very similar to the sensor service (see Subsection V-F). The main difference with the sensor service is the fact that the *sendActuatorData* service allows

the user to actually set the desired actuator value. Meaning that the data model of Figure 3 is sent in the request.

The internal functionality of the Smart Device should first validate the sent value (see V-K0c) prior to applying it to the actuator itself. While sensors often do not have concurrency issues, the actuator may also be controlled by another client concurrently and its access needs to be moderated. Various schemas can be implemented by the lab owner to internally manage the actuator access (see V-C3e). In the following examples, we will assume one of the most common scenarios: a user can either control the lab or can observe what others are doing. Given that the user has a controller role, the actuator may set the value and acknowledge the actuator change by returning the set values in the payload of the response. The payload is optional and the format is not specified. As a good practice we recommend to return the data of the actuator in the same format as the request data format. This returned actuator data in the payload can be used to update the client application UI with the actual value. The client can assume that the actuator has fulfilled the request when no errors are returned. If the actuator is currently in use, a more specific payload, detailing some information regarding the time the user has to wait prior to control the actuator, similar to the example in Subsection V-F.

Furthermore, a user with the 'interruptor' role can abort the actuator control of current user. The way the conflict is resolved and the policy to grant this role is defined by the lab owner and/or the client application.

H. User Activity Logging Service – getLoggingInfo

The optional user activity logging service returns logged user actions or lab status info in the ActivityStream 1.0 JSON format[14]. The ActivityStreams format is a JSON-based specification to describe a sequence of user actions with a timestamp and it is often used in social media platforms. To retrieve a continuous stream of real time user activities of the Smart Devices, the getLoggingInfo service can be called with an optional authentication token to validate access (which is recommended due to the privacy sensitive data).

I. Client Application Service – getClients

This optional service provides links to the client applications to operate the Smart Device. The client technology is not strongly specified. The Go-Lab project advocates OpenSocial gadgets [22], since they effortlessly run on the Go-Lab ILS platform [13]. Upon sending a request to the getClients service, a client app list will be returned, with for each item a type that specifies the kind of application and a url. The current version of the Smart Device specification contains the following extensible list of types: 'OpenSocial Gadget, 'W3C widget', 'Web page', 'Java WebStart' and 'Desktop application'.

J. Models Service – getModels

This optional service can provide several models of the physical lab (i.e. the instrumentation) and its theoretical background. For instance, a 3D graphical model of the lab instrumentation can enable a client app to generate a GUI with a 3D scale object that students can manipulate. With a mathematical model of the experiment, a client app can be built with a local simulation. This can provide an interactive simulated version of a remote lab that can be used by students when the lab is already in use (i.e. to provide a better observer mode). Due to the wide range of existing formats to express graphical and theoretical models (e.g. VRML[15], X3D[16] & MathML[17]), we do not limit the specification and leave the model language choice up to the lab owner.

K. Functionalities – Best Practices:

Internal functionalities are implementation suggestions for the Smart Device. They are provided as best practices, since the implementation of these functionalities are often ad-hoc and strongly related to the connected equipment.

a) Authentication functionality: The Smart Device does not need to contain a booking system. It can make use of an external booking system, such as the Go-Lab booking system (currently under development). When a user reserves a lab, the Go-Lab booking system provides an authentication token. At the booked time the user can connect to the Smart Device with this authentication token. The Smart Device then contacts the booking system to validate whether the user is currently allowed to access the Smart Device. Thus, integrating the booking service in the Smart Device requires little effort, compared to providing its own authentication and booking mechanisms.

b) Self and known state functionality: The precise implementation of this recommended functionality is left to the lab owner's discretion. This functionality ensures that the remote lab is reset to a proper state after an experimentation session is completed or a system outage occurred, so that the next user can properely use it. Since remote experiments are supposed to be conducted from faraway, nobody is expected to be around the experiment to put it back in a known state. Thus, the system should be as autonomous as possible, which implies an adequate and defensive software and hardware design that is able to adapt to 'any' situation. We suggest to implement the following procedures in the Smart Device: (1) automatic initialization at startup, (2) reset to a known state after the last client disconnects, and (3) potentially hardware calibration.

c) Security and local control: This functionality is recommended and its implementation is left to the lab owner's discretion. At all time the security of the server and its connected equipment must be ensured. All commands should be validated before being forwarded to the connected equipment. This step may require the addition of a local controller to track the connected equipment's state, e.g. a speed increase may need to follow a ramp before being applied to a motor. Users often try to take the system to its limits, i.e. not only the physical limit of a given sensor/actuator, but also signal patterns on a sensor over time may also need to be considered. Since the actuators may be connected to the Internet, it is essential to validate all applied values and to consider potential external constraints. The lab owner should implement the following procedures in

[14]The ActivityStreams specification is available at http://activitystrea.ms/specs/json/1.0/

[15]Virtual Reality Modeling Language (VRML), http://gun.teipir.gr/VRML-amgem/spec/index.html

[16]X3D, http://www.web3d.org/standards

[17]MathML, http://www.w3.org/Math/

the Smart Device: (1) *value validation* before applying data to actuators, and (2) *actuator state validation* to check if the command to be applied is safe.

d) Logging and alarms: This functionality logs session and lab information, as well as user interactions. In case of problems, alarms may be automatically triggered by this functionality. Since a Smart Device will be typically online unattended for an extended period of time, it is essential to monitor it and have a method to perform post hoc analysis. The user action should be logged, and can be made accessible via the user activity logging service (see Subsection V-H). But extra information should also be logged, e.g. the system state and the environment (e.g. room temperature). Note that some sensors may be available internally to the Smart Device, but not necessarily accessible via the sensor service. We suggest to track the following information: (1) user actions, (2) the complete system state, and (3) its environment state. Additionally, by definition the Smart Device is connected to the Internet and has no knowledge of its clients. Proper action is required to prevent abuse. A firewall or a DMZ[18] may protect it from attacks. While some hostile actions may be reduced using such mechanisms, the Smart Device should add internally additional measures: (1) validate the requests sent by clients, (2) throttle continuous requests of a malicious client, and (3) log all Internet connections for later analysis. If an unexpected event occurs, its potential danger should be assessed by the Smart Device and an alarm may be triggered.

e) Local simulation: When the experiment is busy or unavailable, a local simulation might be a useful alternative for waiting users. The simulation data could be read or modified through virtual sensors/actuators. A mathematical model describing the physical equipment can be made available to the client via the models service, which the client developer can use to simulate the hardware. Such simulations can require computational resources unavailable at the client. However, this computation can be done server side and the results can be sent to the client using virtual sensors and actuators.

VI. A Detailed Smart Device Example

This section illustrates how a Web client interacts with a simple Smart Device, with one sensor and one actuator. Both the Smart Device and the Web client are available on GitHub[19]. The full JSON messages are omitted for brevity, but similar examples can be found in the full specification.[11] The first step taken by the Web client is to ask the Smart Device about its general capabilities using the metadata service. This is done with an regular HTTP GET request to `http://serverIP/metadata`. The Smart Device returns JSON containing the metadata (see Figure 4). Then, the client requests the available sensors from the Sensor Metadata Service. This request is performed via a WebSocket. A JSON object containing `{"method": "getSensorMetadata"}` is sent to the server. Upon which the Smart Device replies with another JSON object containing an array of available sensors `{...["sensorID":"discPos", ...]}` and related information such as range, etc. The next step

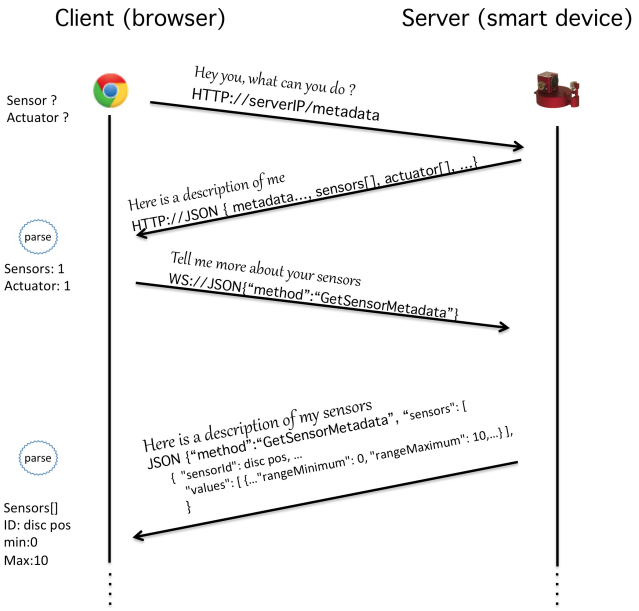

Fig. 4. The web client asks the Smart Device about the available sensors.

is to ask about available actuators with a similar request (see Figure 5). The Smart Device replies that there is one actuator: a motor, with `"actuatorID":"motor"`, `"rangeMinumum":"-5"` and `"rangeMaximum":"5"`. The client app has now enough information to build a basic UI. In this case two UI fields: one to display the `discPos` sensor value and one to set the motor actuator value.

The fields of the generated skeleton UI need to be populated with the data coming from the Smart Device. In other words, we need to tell the Smart Device to start sending measured values to the client via a WebSocket. This is done by sending the request `{"method": "getSensorData", "sensorID":"discPos",...}`. The Smart Device will start pushing the measured values continuously to the client (see Figure 6). The client application needs to parse the received JSON objects and update the sensor field in its UI with the received value.

When the user modifies the actuator value in the client UI, a WebSocket request is sent to the Smart Device with the new actuator value, `{"authToken":"42FE36", "method": "SendActuatorData", "actuatorID":"motor", "values": [...]}`. This request carries an authentication token, which will be used by the Smart Device to verify that access to the actuator is granted to the client application (e.g. based on a lab booking of a user at a given time). To control the access to the actuator, the Smart Device will contact the booking service with the provided token. If the booking service confirms the token, the new actuator value will first be internally validated (e.g. within a specified range), and then applied to the motor. If the token is invalid or if the value is out of range, the value will not be applied to the motor and an error message may be returned to the client application.

Upon completion of the remote experiment, the client closes the WebSocket connections. Internally, the Smart Device should go back to a known state and wait for the next user

[18] Demilitarized Zone (DMZ), http://en.wikipedia.org/wiki/DMZ_(computing)

[19] https://github.com/go-lab/smart-device/tree/master/Desktop/Simple-examples

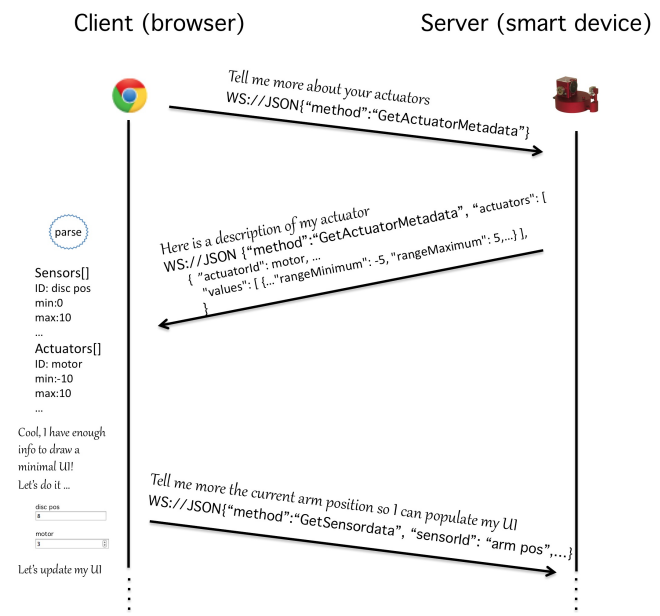

Fig. 5. The web client asks the Smart Device about the available actuators.

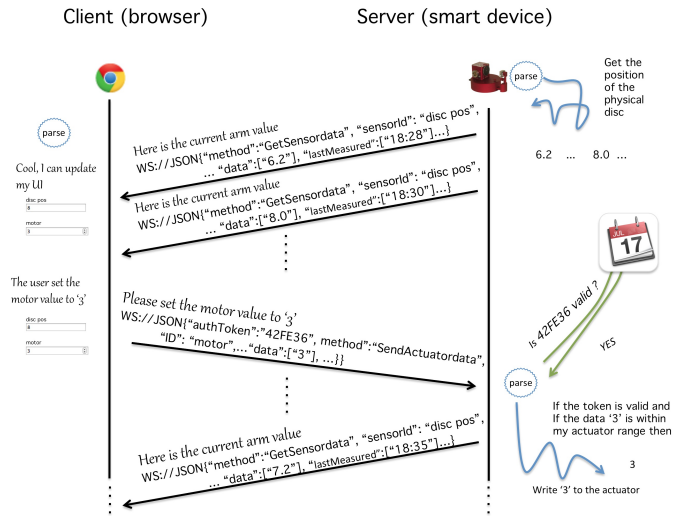

Fig. 6. The Smart Device pushes the measured values to the client. It also receives and validates the actuator value prior to apply it to the motor.

to connect, e.g. set the motor voltage to '0' to save energy.

VII. Implementation Examples

To illustrate that the Smart Device specification is software and hardware platform agnostic, we have implemented the specification on various platforms and with different programming languages. These examples are publicly available on GitHub[20].

A. LabVIEW

The LabVIEW examples are designed for both desktop computers and embedded hardware, i.e. the National Instruments myRIO. LabVIEW is a development environment well known by many lab owners. A complete implementation is available[21] and executable on Windows, OSX and Linux.

B. Javascript

The Javascript examples are designed for small embedded computers such as the Raspberry Pi or BeagleBone Black. They rely on Node.js and Socket.IO to implement the services. Hardware access is possible by either using on-board pins or by interfacing other I/O modules via USB (e.g. Arduino).

C. The Smart Gateway

In some situations, it will not be possible to modify the server of already existing labs, e.g. due to the lack of resources. In this scenario, a Smart Gateway [23] that lies between the client and the remote lab does the necessary translation to make the remote lab behave like a Smart Device from a user point of view. This translation is performed by the Gateway4Labs[22], a software orchestrator that relies on plug-ins to adapt the different existing labs to the Smart Device specifications. The Smart Gateway internal definition is beyond the scope of this paper, but further reference can be found in [23].

VIII. Standardization Efforts

To encourage and strengthen the adoption of the proposed Smart Device specification, several partners of the Go-Lab project are involved in the IEEE Working Group P1876[23] on Networked Smart Learning Objects for Online Laboratories. This group is sponsored by the IEEE Education Society. The standardisation work is at an initial phase, however several meetings were held to define the standard at three levels: a pedagogical level, a service level, and a communication protocol level. The pedagogical level describes how to package resources in a standardised way and how to enable their integration in learning environments (e.g. LMS, MOOC platforms or social media platforms). The service level standardizes how clients communicate with a remote lab. The abstraction layer provided by the Smart Device specification was well received as a proposal and has the potential to become the seed of the final IEEE specification, still to be drafted and finalised. Finally, the communication protocol level standardizes the way all the loosely coupled services and platforms supporting the usage of remote labs could interoperate. Several Smart Device services can enable such interoperability, with for example a booking system or learning analytics services. Due to the early stage of this standardization effort, it is hard at the time of writing to assess the impact of the Smart Device specification on the finalized standard. However, we believe that the Smart Device characteristics are essential for the standard.

IX. Conclusion

In this paper, we presented the detailed Smart Device specification for remote experiments. We first summarized the Smart Device paradigm and its application to remote labs. From a client or external service point of view, the Smart Device

[20] https://github.com/go-lab/smart-device

[21] https://github.com/go-lab/smart-device/tree/master/Desktop

[22] https://github.com/gateway4labs

[23] IEEE Working Group P1876, http://ieee-sa.centraldesktop.com/1876public/

is described through well-defined services and functionalities. Services permit to access the inputs and outputs of the Smart Device, such as sensors and actuators. Functionalities refer to provided internal behavior such as range validation for an actuator. The main goal of this paper is to define the services and functionalities of a Smart Device using Swagger, a JSON-based description language. This specification is sufficiently detailed, thanks to the properties of Swagger, that a code skeleton for the client application can be machine generated without additional information from the lab owner. Furthermore, this shared specification enables a complete client-server decoupling by enabling interoperability, thus allowing the integration of Smart Devices in any environment, OS or device. Additionally, we have shown that implementing the specification is feasible by providing several examples and templates for developers to get started. In the future, we plan to develop more Smart Device enabled remote labs to further assess the power of the specification. Some technical assumptions are made when considering the client application for remote labs. The first one implies that the client resides typically in a recent Web browser that runs on a tablet, this implies a plug-in free solution. In addition the means to exchange information between the client and the server is made using JSON encoded messages that are transmitted using asynchronous WebSockets. Finally, the proposed specification is open and can be extended at will.

ACKNOWLEDGMENT

This research is partially funded by the European Union in the context of Go-Lab (grant no. 317601) project under the ICT theme of the 7th Framework Programme for R&D (FP7). We would like to thank Anjo Anjewierden, Lars Bollen, Augustín Caminero, Manuel Castro, German Carro, Gabriel Díaz, Danilo Garbi Zutin, Miguel Latorre, Irene Lequerica Zorrozua, Pablo Orduña, Antonio Robles, Elio San Crístobal, and Simon Schwantzer (in alphabetical order), for their input during the numerous discussions leading to this specification.

REFERENCES

[1] C. W. Thompson, "Smart devices and soft controllers." *IEEE Internet Computing*, vol. 9, no. 1, pp. 82–85, 2005.

[2] C. Salzmann and D. Gillet, "From online experiments to smart devices," *International Journal of Online Engineering (iJOE)*, vol. Vol 4, no. SPECIAL ISSUE: REV2008, pp. 50–54, 2008.

[3] C. Salzmann and D. Gillet, "Smart device paradigm standardization for online labs," *IEEE EDUCON Education Engineering 2013*, pp. 1217–1221, 2013.

[4] "Swagger RESTful API specification." [Online]. Available: https://github.com/wordnik/swagger-spec/blob/master/versions/1.2.md

[5] T. de Jong, S. Sotiriou, and D. Gillet, "Innovations in stem education: the go-lab federation of online labs," *Smart Learning Environments*, vol. 1, no. 1, p. 3, 2014. [Online]. Available: http://www.slejournal.com/content/1/1/3

[6] D. Gillet, T. de Jong, S. Sotirou, and C. Salzmann, "Personalised Learning Spaces and Federated Online Labs for STEM Education at School: Supporting Teacher Communities and Inquiry Learning," in *Proceedings of the 4th IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 2013, pp. 769–773.

[7] M. Tawfik *et al.*, "Laboratory as a Service (LaaS): a Novel Paradigm for Developing and Implementing Modular Remote Laboratories," *Int. Journal of Online Engineering*, vol. 10, no. 4, pp. 13–21, 2014.

[8] IEEE, "IEEE Standard for a smart transducer interface for sensors and actuators wireless communication protocols and transducer electronic data sheet (TEDS) formats," *IEEE Std 1451.5-2007*, 10 2007.

[9] X. Chen, G. Song, and Y. Zhang, "Virtual and remote laboratory development: A review," *Proceedings of Earth and Space 2010*, vol. 1, no. 1, pp. 3843–3852, 2010.

[10] C. Salzmann and D. Gillet, "Remote labs and social media:agile aggregation and exploitation in higher engineering education," *IEEE EDUCON Education Engineering 2011*, pp. 307–311, 2011.

[11] D. Cascado *et al.*, *Standards and Implementation of Pervasive Computing Applications*. John Wiley & Sons, Ltd, 2011, pp. 135–158.

[12] M. Auer, A. Pester, D. Ursutiu, and C. Samoila, "Distributed virtual and remote labs in engineering," in *Industrial Technology, 2003 IEEE International Conference on*, vol. 2, Dec 2003, pp. 1208–1213 Vol.2.

[13] S. Govaerts *et al.*, "Towards an online lab portal for inquiry-based stem learning at school," in *Advances in Web-Based Learning ICWL 2013*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 8167, pp. 244–253.

[14] T. Hecking *et al.*, "A flexible and extendable learning analytics infrastructure," in *Advances in Web-Based Learning – ICWL 2014*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8613, pp. 123–132.

[15] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins, "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP," RFC 6202, Internet Engineering Task Force, Apr. 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6202.txt

[16] "Websocket specification." [Online]. Available: http://tools.ietf.org/html/rfc6455

[17] M. J. Hadley, "Web application description language (WADL)," Sun Microsystems Inc., Tech. Rep., 2009. [Online]. Available: http://java.net/projects/wadl/sources/svn/content/trunk/www/wadl20090202.pdf

[18] "Hypertext transfer protocol (HTTP) status code registry (RFC7231)," IETF, Tech. Rep., 2014. [Online]. Available: http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

[19] N. Freed, M. Baker, and B. Hoehrmann, "Media types," Tech. Rep., 2014. [Online]. Available: http://www.iana.org/assignments/media-types/media-types.xhtml

[20] B. N. Taylor and A. Thompson, "The international system of units (SI)," National Institute of Standards and Technology, NIST Special Publication 330, 2008.

[21] B. Furht, "A survey of multimedia compression techniques and standards. part i: JPEG standard," *Real-Time Imaging*, vol. 1, no. 1, pp. 49–67, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1077201485710054

[22] M. Marum, "Opensocial 2.5.1 specification." [Online]. Available: https://github.com/OpenSocial/spec

[23] P. Orduña *et al.*, "Generic integration of remote laboratories in public learning tools: organizational and technical challenges," *Proceedings of the IEEE 2014 Fontiers in Education Conference*, Oct. 2014.