A brief introduction to xAAL v0.4-draft Rev 2

Christophe Lohr Jérôme Kerdreux Philippe Tanguy

April 14, 2015

Abstract

This paper gives a brief description of xAAL such defined by the IHSEV/HAAL team of Telecom Bretagne and its status in 2015.

Be aware that this is an ongoing work.

1 Motivation

Many vendors offer home-automation solutions. Small devices are plugged everywhere at home. They talk to each other using dedicated link layers and protocols (powerline channels, radio channels, etc.). There are almost as many home automation protocols as manufacturers (or alliances of manufacturers). The consequence is a strong issue about interoperability: how to make a device from vendor A (e.g., a switch) talking with a device from vendor B (e.g., a lamp)?

Solutions exist to address the interoperability issues (the state-of-the-art is not presented in this document). All solutions play around the idea of a so-called *gateway*: a box (a small computer) is equipped with two or more modules, each module talking a given home-automation protocol; then a piece of software over those modules make it possible to forward messages between protocol A and protocol B. Moreover, those boxes embed additional functions to make all of this usable: HMI (via Web and/or smartphones), configuration facilities, connectors to extra services in the cloud, etc. Such solutions are however partial from the interoperability point of view since those boxes are hardly able to talk to each other.

Based on these facts, we decided to reorganize and formalize architectures of those boxes, which are in fact all composed of the same functionalities. So, we propose to split functionalities into well defined functional entities, communicating to each other via a messages bus over IP. Each entity may have multiple instances, may be shared between several boxes, and may be physically located on any box. As a result, the xAAL system will be interoperable by design (hopefully).

2 General Architecture

The xAAL system is based on an IP *multicast bus* (IPv4 or IPv6) called the xAAL bus. Devices send messages to the bus to announce themselves (*alive* notification) or changes of theirs attributes, listen to requests (possibly in a broadcast way) and send replies according to their API specified in *schemas*.

A bus allows the *discovery*: when a new component appears in the installation, it announces itself. All other entities can then take it into account. Similarly, when a new component enters, it can query the bus to discover the other components already present. This greatly facilitates the configuration, allows dynamicity as well as the evolution capacities of the system.

There are several advantages to the use of a IP multicast bus:

- Point-to-point communications are avoided which saves devices memory resources;
- Changes on the available home-automation devices are easily taken into account by allowing the devices to announce themselves when entering the xAAL bus.

Messages sent/received to/from the xAAL bus are intended to manage or act on home-automation devices. Therefore, a set of functions should exist and interact with the bus in order to allow this. The required functions are described below and extensions are also proposed, i.e., functions that are not necessary for xAAL to be fully operational but that are interesting for end users. Each of the functions may be replicated, provided by one or several hardware devices, and should be able to send messages over the xAAL bus.

2.1 Functional Architecture

Figure 1 shows the general functional architecture of the xAAL system.

Native Equipments. Some home automation devices (sensors, actuators) can communicate natively using the xAAL protocol.

Gateways. In general, the home automation devices only support their own proprietary communication protocol. Therefore, *gateways* will have the responsibility to translate messages between the manufacturer protocols and the xAAL protocol.

At this level it is assumed that, within an installation, there are as many gateways as manufacturer protocols used in the building.

According to the technologies and the manufacturer protocols, each gateway will handle the following issues: pairing between the gateway and devices, addresses of devices, configuration, and the persistence of the configuration.

Note that it may be interesting to be able to query a gateway about the mapping between xAAL addresses and manufacturers addresses of devices. The mapping information given by the gateway may be useful for the configuration of the database of metadata (described below). Indeed, during the configuration, the installer usually notes the manufacturer physical address of devices with their location in the house. To facilitate the configuration of this database (location information associated with the xAAL address of devices), it may be interesting, at the beginning, to look at the manufacturer identifier or address of the devices.

Schemas Repository. The home automation devices are all described by a *schema* (the nature of the equipment, how to interact with it). These schemas are listed in one or more *schemas repositories*.

Because schemas repositories are xAAL devices, they are present on the xAAL bus. Other entities query them via this bus to get schemas describing other devices (whenever those devices are active or not on the bus).

One queries registry on a schema name. If known, it returns the content of the schema.

In a given installation, there may be zero or more schemas repository talking on the xAAL bus.

The schema definition of a device must be especially careful. A number of general purpose schemas have been specified by our “xAAL International Bureau of Standardization” ☺. More specific schemas may be further defined by the manufacturers themselves.

Therefore there possibly can be conflicts: given a schema name, several registries on the bus could return schemes having divergent descriptions.

Then, it is necessary to plan a mechanism for conflict resolution: either in a static way (where an administrator invalidate or delete conflicting schemas in a register), or dynamically

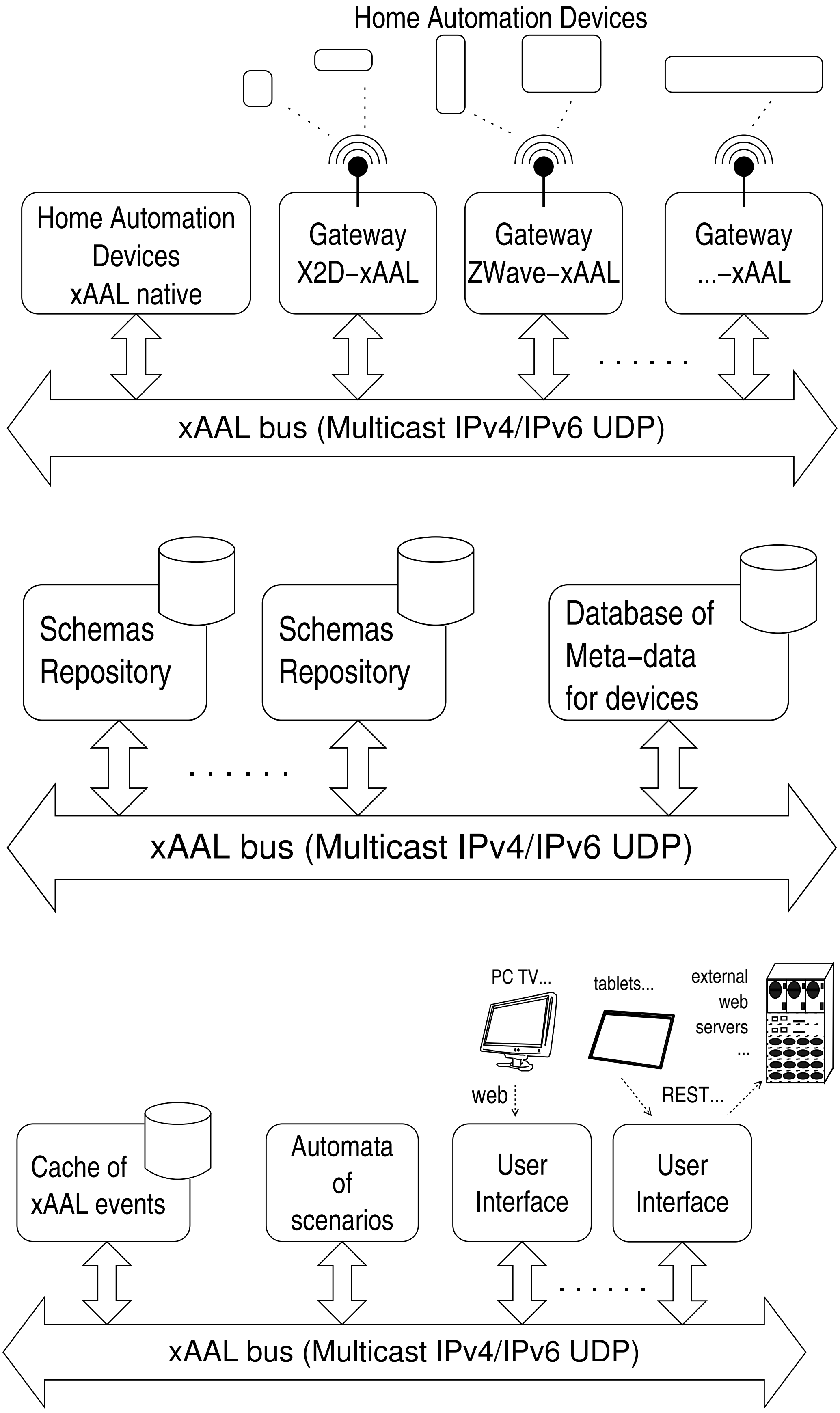

Figure 1: Functional Architecture of xAAL.

(where schema repositories spread priority information in addition to schemas; letting receivers to short schemas). This point has not yet been decided in the current specification.

However, because those schemas are locally available, this allows easing maintaining the consistency of a home-automation facility (at least locally).

The updating of schemas is, a-priori, a maintenance operation. However, the dynamicity is also possible. For instance, when a schema repository discover a new device with a not-yet-known schema, it can ask him its description, look at an url in this description, and try to download the schema content from this url.

Database of Metadatas. Each automation device within a facility is likely to be associated to a piece of information: at least location information (e.g., declare that the equipment typed as a lamp and having address X is located in the kitchen), and possibly a symbolic name (e.g., device having address X is called "lamp#1").

All this information is grouped in a *database of metadata*. For openness and extensibility (as flexibility and not in the sense scalability), this information is stored as a set of tags.

This database contains somehow the configuration of home automation installed.

The database of metadata is present on the xAAL bus. Other xAAL devices can query it via this bus to obtain tags associated to the identifier of a device, or conversely a list of devices associated with one or more tags.

- For each device, the database is fed with information: its xAAL address and a set of tags. (Note that the type of the device may be a tag.)
- Information can be added, updated or deleted.
- One can question it on tags associated to a device.
- One can question it on devices associated on some tags.
- One can question it on existing tags.
- One can question it on known devices.
- Note that a device may be known by the database without being actually present on the bus. (The presence is managed in a different way.)

There should be at least one database of metadata on the bus. There could be several.

The creation of this database must be established with the greatest care. It will thus be advisable to implement the classic mechanisms of organization of tags (or quality of data): check inconsistencies, synonyms, ambiguities, provide facilities renaming tags, etc.

Note that structuring the information with tags is different from a structure by key-value. For instance with tags one just records "living-room" instead of "location: living-room". This implies that the semantic is given by the value of the tag itself and not by a set of predefined keys. There are benefits in doing this so. For instance, the classic trap is to specify location information in two places, for example indicate a *friendly name* tag "living-room lamp", then add a *location* tag "living-room" (by chance in this case, information is redundant but convergent...) This notion of friendly name can be found in UPnP... In fact this mechanism is a source of problems: either it is diverted from its initial role to put location information as in the previous example, or it is left to the manufacturer value (how many DLNA TV or smartphones have for friendly name *Media Server*?...)

Cache. Unidirectional sensors are quite common: one can not question them, they just send their information in a sporadic way. (E.g., A thermometer which sends the temperature only if it changes.)

So, there should be at least one cache on the xAAL bus that stores this information so that other entities can query it whenever necessary.

Such a cache should store at least the notifications sent by the devices. One can further wonder about the relevance of recording messages of type request/reply.

As in any caching mechanism, it is necessary to associate a timestamp to cached information. When another entity on the xAAL bus asks the cache for information, it also gets the age of the cached information, and decides by itself if it is good enough or not. This is not the responsibility of the cache to manage the relevance of data according to their age, this is the responsibility of the client that makes the request.

Again, inconsistencies may arise if two caches return information that has the same timestamp but divergent values. We assume that this phenomenon is a priori very rare. Consequently, we do not intend to define a particular mechanism to manage inconsistencies. Only the client entity is in charge of sorting received information according to its own criteria.

Automata of Scenarios. Scenarios are advanced home automation services like for example: start a whole sequence of actions from a click of the user, or at scheduled times, or monitor sequences of events and then react, etc.

To do this, xAAL proposes to support it by one or more entities of the type *automata of scenarios*.

These automata of scenarios are also the right place to implement virtual devices: for example a scenario could aggregate and correlate a variety of events from real devices, and then synthesizes information such as "presence" and notify it on the bus, in order to be used by other entities. By proceeding in this way, this scenario should appear by itself as a device on the bus, with its address and its schema. This scenario is a kind of virtual device.

As usual, there may be inconsistencies, conflicts or deadlock between scenarios (one turns on the light, the other off). This is not the xAAL specification itself which is going to prevent it. Conflicts are supposed to be treated as far as possible off-line when editing scenarios. The specification does not require the use of a model-checker based on colored timed Petri network or anything like this... It remains a research subject, certainly very interesting, but outside the framework of specifying xAAL. (If anyone wants it, it can introduce a monitoring device, which is itself a kind of engine of little particular scenarios.)

User Interfaces. One or many user interfaces are provided by specific entities connected to the xAAL bus. This can be a real hardware device with a screen and buttons; or a software component that generates Web pages (for instance) to be used by a browser on a PC, a connected TV, or whatever; or software that provides a REST API for mobile applications (tablets, smartphones), to an external server on the cloud for advanced services, to an MQTT server, or to offer features for service composition, etc.

Even there, it seems natural to be able to arrange several user interfaces (smartphone, tablet, remote control, etc.) within a home automation system.

Synthesis. Such a functional architecture allows managing the dynamic aspects of the infrastructure (modularity, scalability, adaptation, etc.). Thus, advanced functions such as HMI or automata of scenarios can adapt themselves automatically to the infrastructure and to its modifications (if equipment enters / leave, if one fails or is replaced, etc.).

The typical work flow of an HMI could be:

1. query the bus to be aware of the present devices;
2. query devices about theire description (model, manufacturer, URL, etc.);
3. query the schema repository for the type of these devices, the type of data from the sensors / actuators, the possible actions;
4. query the database of metadata about devices to get the configuration of the installation (a friendly name to display, location and other tags associated with each device);
5. query the cache for the latest known states of such equipment;
6. dynamically build display screens and control interfaces for these devices.

2.2 Hardware Architecture

The interest of such a functional distributed architecture is to allow greater freedom regarding the placement and grouping functions on hardware.

For example:

- A manufacturer of automation equipment could sell a *box* that would include: the gateway for its own home automation protocol, the repository of schema for equipment of its brand, a cache for equipment managed by the gateway, some automata of scenarios to implement virtual devices, and a small metadata database for its own devices to record their location.
- A provider of automation services could sell a *box* (different from and complementary to the first one) that would include: a database for location of devices, automata of scenarios, web interfaces, REST connectors to communicate with servers in the cloud to offer enriched services, etc.
- A provider of virtual services (i.e selling only service and no equipment) could provide an application for smartphone or tablet that embarks: a user interface, an automaton of scenarios, a connector to its external servers for advanced services, etc.
- A provider of services dedicated (for instance) to assistance for seniors, telemedecine, etc. could offer its box (yet another one) that would include: a gateway for specific equipment (medical, remote alarm, etc.), automata of scenarios for monitoring&alert, an interface to its external servers, etc.
- A manufacturer of innovative connected objects (Internet of Things) could sell these connected objects which would embark: the sensor/actuator component itself speaking natively in xAAL, the repository of schema describing the object itself, automata of scenarios, an interface to its external servers, etc.
- Services or scenarios a bit complicated (e.g., based on actigraphy, or composition of adaptive services, etc.) could be made of automata of scenarios, linked to a cache, linked to a database of metadata.

The reliance on IP as a means of interconnection and this lightweight protocol allows, hopefully, flexibility and better interoperability between these different actors who have nothing else than xAAL message to disclose to competition.

The really hard point is the definition of schemas and the rigor with which implementations comply with.

3 The Transport Layer

It uses IP multicast (IPv4 and/or IPv6). Thus, it is in UDP.

- The multicast IP address has to be defined.
- The port number has to be defined. We reserve a range of ports numbers.
- If there are several "home networks" in the building, they use different ports.
- Messages may have a size limited by UDP (i.e., 65507 bytes in IPv4, and more than 65535 with IPv6 jumboframes). IP fragmentation operates normally. (As opposed to approaches such as xPL that limit to the MTU (1500 bytes) and propose *continue* flags. IP fragmentation already handles this very well.) Remains the problem that some PICs do not necessarily manage well the IP fragmentation (e.g., the basic IP stack for Arduino). We only recommend trying to be limited to 1500 bytes. If services are willing to exchange larger amounts of data between devices, they are asked to leave the automation bus (which is not made for it) in favor of another transmission channel (e.g., A TCP connection), that they can advertise via the xAAL bus.
- The IP configuration of devices is outside the scope of xAAL. Equipment can use a static IP configuration, DHCP, the IPv6 Router Advertisement, Zeroconf, a Link-Local address (169.254.0.0/16, fe80::/64), etc.

  Note however that thanks to the use of a multicast bus, the source address of the transmitters doesn't really matter (and is actually not used in our case). Some equipment might be configured with a phony hard-coded IP address...
- Note for developers: please avoid horrible stuff that have been seen with implementations of xPL or xAP buses (often in Java or .Net) regarding the use of a "hub": that piece of software that repeats messages from bus to applications running the same machine. This is useless on almost all current operating systems: applications on the same machine may without problem have their own socket on the multicast bus, as long as dedicated socket options are used (typically `SO_REUSEADDR` and `IP_MULTICAST_LOOP`)...
- The choice of IP multicast IP almost imposes the UDP protocol. As UDP does not manage the detection of packet loss and retransmissions, there could have been more comfortable to choose TCP. For people not convince by the UDP protocol:

  In the context of a network at home, it can be assumed that packet loss, though still possible, is a priori rather rare.

  By design, xAAL has little requirements regarding error processing (cf. 8.3). From this point of view xAAL seems rather robust.

  Using TCP to manage packet loss only shifts the problem to the protocol stack (and in a heavy way: it is systematic for all messages, and the TCP automata TCP consumes resources). With xAAL, if one need error processing for some scenarios (which, finally, is not necessarily systematically required) this will be at the services layer (located on equipment which generally has comfortable resources).

4 The Data Presentation Layer

It is a question of deciding if the data transmitted by the xAAL bus are rather of textual nature or either of binary nature.

This point was the object of a debate within the team, where two schools of thought wonder.

4.1 The *Binary Protocol* approach

The main motivation is to stay at a low level: a protocol which can be coded within a PIC (e.g., in traditional C), rather than a sophisticated middleware coming from the PC world and web applications.

It does not necessarily mean compact or compress messages, but simple messages easy to write and parse (by cast to C data structures), self-sufficient, in a constrained format, with a constraint variability, a predictable size, etc.

More concretely we would decide to only manipulate 32-bit words, aligned on 32-bit (i.e., with padding if necessary).

The disadvantage is that the debugging stage of messages is more complicated than a textual approach, because a validation tool for analyzing the exchanged frames is needed. The other drawback and that it is too far from trendy techno (around Web), and it will be a priori more difficult to entice a community.

Finally, this is not the approach that was chosen by the team for the implementation of xAAL. However, a draft of specification has been defined and is available on request, this is the *version 0.1-bin*.

In the future it might be interesting to look at the Mihini/M3DA protocol to do this...

4.2 The *Text-Based Protocol* approach

The advantage of a text-based protocol is that developers only need to check the frames exchanged.

The disadvantage is that the messages are more complex to build, and even more complicated to parse.

The adopted compromise was to choose Json as exchange format of data. Many libraries are available in many development environments.

Furthermore, it is rather close to the Web World, and as the last generations of IT specialists were massively and firstly formed to the technologies of the Web, we can hope for a better membership with this public, as well as with certain public of decision-makers...

This approach was experimented in the previous implementations of xAAL made by the team (*version 0.1-json*). The specifications in the following document are based on a text protocol approach and extend to describe the current *version 0.3*.

An interesting variant would be to use certain aspects of CoAP[1] (or a subset)... To consider for the future... even if the proximity to the Web World appears nevertheless of a limited technical interest...

5 Definition of a device

A *device* has:

- a `devType`: references the *schema* and defines the type of the device. It is hard-coded into the device.
    - This schema identifier is a string consisting of a pair of words separated by a dot.
        - The first word refers to a class of device type (e.g., lighting, heating, multimedia, etc.).

[1] `https://datatracker.ietf.org/doc/draft-ietf-core-coap/`

    - The second word refers to a type in a given class (e.g., within the lighting class one may have an on-off lamp, a lamp with dimmer, a mirror ball, etc.). The second word may also refer a schema extension by a manufacturer (cf. 6.1).
  - The identifier `"any.any"` is reserved and refers to all types of all classes. This is a *virtual* type; no one can clame to be of type `"any.any"`. Devices having no dedicated attribute, method or notification may use the type `"basic.basic"` which is a *concret* type that simply inherit of `"any.any"`.
  - For example the pair `"lamp.any"` means all types of the class `"lamp"`. This is also a *virtual* type; if needed, the *concret* type `"lamp.basic"` is provided.
  - We reserve a special name: `"experimental"`, for the class, as well as the type. This are concrete types to which may belong devices. Associated schema, if written, should not be distributed outside the testing plateform. (I.e. When someone makes a device but has not got a name for our xAAL International Standardization Office.)

- An `address`: a device ID, unique on the bus.

  - The device identifier is a UUID (RFC 4122): the textual representation of a 128-bit random number.
    For instance: `634c4471-bc3e-47f2-8cd3-eadfc08c6c6a`
  - Address `00000000-0000-0000-0000-000000000000` (*uuid*0) is reserved.
  - Addresses are self-assigned. There is no "naming service" nowhere. A device does not request any entity to get an address.
    Well, concretly, how is assigned an address?
    - Either, this is hard-coded in the factory.
    - Auto-generated (random) at the time of installation. However, it is recommended that this address remain persistent: i.e., please save it (if possible) during power breaks.
    - There are in principle very high probability of having no collision of UUIDs. However, it is technically possible to verify that an UUID is not already used on the bus by a kind of "Gratuitous ARP", i.e., by `isAlive` requests to ensure that no one else already use this address.
    - But certainly not assigned by a bus supervisor/coordinator or something like that!

- An equipment may consist of several devices. This is typically the case of a gateway. (E.g. An equipment which is going to make the bridge between our xAAL bus and X10, Zigbee, X2D, Z-Wave, HomeEasy, KeeLoq, etc.) But it may also be the case of a small weather station with several sensors that measures indoor/outdoor temperature, humidity, etc.

  - Such equipment is then called a *composite device*,
  - and each of its components is an *embedded device*.
  - The composite device announces (possibly) itself on the bus with its own address and its own schema name. Others can interact with him specifically (e.g., for its configuration, consult battery level, etc.). It also announces the list of its embedded (their address).

  - Each embedded is also announced on the xAAL bus like an ordinary bus-native device, with its own address and its own schema name. Embedded may be question about their parent, i.e., the composite to which they are linked. (Note: the real bus-native devices have as parent *uuid*0).
  - A composite device can also choose to be not visible, do not announce itself, nor indicates that it has embedded devices, nor that those embedded devices have it for parent.

6 Defining a Schema (the type of device)

Each device is typed, i.e., described by a *schema*. Those schemas are inspired by the UPnP approach.[2]. A xAAL schema is a Json object with a specific form that must validate a given *Json Schema*[3].

- The schema provides a bit of a semantic to a device and describes its capabilities: a list of possible methods on this device (mechanism of request-reply), a list of notifications that it emits spontaneously, and a list of attributes that describe the operation of the device in real time (the device announces change of values on the bus).
- A list of *methods*. Each method is described by:
  - a unique name which identifies it within the schema;
  - a textual description (see later how to handle the language and issues of internationalization);
  - A list of arguments, each argument is defined by:
    - a name uniq within the method definition;
    - a direction, a string among `in out inout`;
    - the unit (in the sense of the International Bureau of Weights and Measures...); a string to be displayed in HMI;
    - a textual description;
    - a type definition (according to Json Schema dialect) used for extra processing (e.g., to dynamically build an HMI).
  - A list of related attributes that may be affected by the method (e.g., to be refreshed in an HMI after the method call)
- A list of *notifications*. Notifications are messages that the device emits by itself on the bus. These notifications are defined by:
  - a name
  - a description
  - a list of variables (name, unit, type)
- A list of *attributes*. If the value changes, this spontaneously generates a notification to the bus. Each attribute is defined by:
  - a name

[2] `http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf`, chapter 2.

[3] `http://json-schema.org/`

  - a description
  - the unit
  - the type

6.1 Inheritance of schemas

There is a notion of inheritance between schemas. A schema can extend an existing schema. We define the first 3 levels of this genealogy:

1. A generic schema common to all existing devices of the world that everyone has to implement.

2. A specific schema for every class (e.g., *lamp*, *thermometer*, *switch*, etc.). Such specific schemas thus inherit from the generic schema.

3. A specific schema for each type that inherits from a class schema, which indicates the level of functionality (e.g., a lamp type can be on/off, some are of a trigger type, and others with dimmer...)

4. Thereafter, manufacturers of home automation equipment will naturally define their own schemas among their product range. However, it is forbidden for these manufacturer schemas to be defined as level 1 schemas (the generic schema is the only one), or as level 2 schemas (class schemas). Schemas from manufacturer are necessarily extensions of schema from level 2 or higher. About the naming of schema (the pointed pair described in 5), the second word is then left with the discretion of the manufacturer. (Manufacturers frustrated by this situation are cordially invited to discuss with our Standardization Office if they wish to introduce new schema at level 2 or 3.)

Note: please think, while listing the APIs of devices described by these schemas, that some devices will not be able to listen to the bus and will only emit notifications changes of their attributes.

Semantic of a schema extending another schema:

- The new schema may introduce new *attributes*, *methods*, and/or *notifications*. The result of the *extend* operator is a new schema that contains all *attributes*, *methods*, and *notifications* of the former schema, plus the new ones introduced by the latter schema.

- The new schema may overload the definition of some existing *attributes*, *methods*, and/or *notifications*. An overloading means that objects has the same name and a different definition. In such a case the resulting schema contains all new objects plus old ones that has not been not overloaded. In other words, the new definition *replace* the old one, without any attempts to merge them.

6.2 Naming of Schemas

The naming of schemas has to be done carefully in order to maintain as much as possible the consistency of things in the minds of everyone... In too many systems (e.g., xAP xPL), the name of devices and their typing is regularly misused to put concepts that do not have their place in it.

What is a schema. Two things: It is hoped that the schema name refers above all to the functionality of the device, means to interact with it. Secondary, this name must also give an idea of the nature of the sensor.

What is not a schema.

- The name is not supposed to refer to a technology of manufacturer (and even less to a specific model).
- The schema name must not refer to a given use: a door contact switch (that transmits "open/closed" notifications) can be used both for safety (on windows or on the main door) as well as for actigraphy (on kitchen drawers); in the same way a thermometer (that sends notifications of temperature) may be dedicated to a weather station or to the management of heating. So there are no reasons to have categories like *security* or *hvac* (Heating, Ventilation and Air-Conditioning)! These are chapters in a catalog of sale of a manufacturer, not categories of functions of home automation equipment. If wanted, the role associated with a sensor (security, heating, etc.) is recorded as a tag in the metadata database.
- One shall avoid carefully junk categories such as the "software" type (that already is seeing!). Obviously there will be a number of software components. Also, some equipment will be purely software (e.g., an application such as a web-radio on a PC). The fact that the component is in soft rather than in hard has no interest from the xAAL point of view.

7 Generic Schema

This is the basement of every schema. All other schemas have to inherit from it somehow.

- Attributes:
  - `devType`: the name of the schema to which the device obeys;
  - `address`: a string, an UUID (the address of the device);
  - `parent`: a string, an UUID (the address of the parent in the case of a composite, or *uuid*0 if not);
  - `busAddr`: a string specifying the IPv4 or IPv6 address of the xAAL bus used by the device. Well, this value is obvious since one is talking to the device via this bus;
  - `busPort`: an unsigned integer (well, limited to 65535), indicating the UDP port of xAAL bus;
  - `hops`: an 8-bit unsigned integer indicating the number of hops used for sending multicast packets;
  - `ciphers`: a array of strings among `"none" "md5" "sha256" ...`, the list of supported ciphers.
  - `key`: a string, the cryptographic key for encrypted communications.
  - Note: the above attributes are mostly dedicated to the configuration of the device. They are not supposed to be involved in the `attributesChange` notification nor in the `getAttributes` method described bellow.
- Notification:

  - `alive`: emitted when starting the device and then periodically at a rate left to the discretion of the device. The notification message contains no parameters, the *body* is empty. (See the definition of messages below.)
  - `attributesChange`: emitted at every change of one of the attributes. The body of the message contains only attributes which changed.

    A schema gives the list of all possible attributes that may appear within this notification message. So, in a given message, some of those attributes may be present, some other not. This is normal.

    However the generic schema defines this method with no attributes, since the default attributes defined above are dedicated to the configuration of the device, and do not characterize the real-time operation of a specific feature.
  - `error`: issued when the device detects an error.
    - `description`: the textual description of the error
    - `code`: a numeric code of the error.

    This is intended to be overridden in the definitions of specific schemas.

- Methods:
  - `isAlive`
    - `devTypes` (out): an array of *devType* strings, giving names of the schema of devices that should wake up.
      One may have `["any.any"]` to wake up everybody, or `["lamp.any" ]` to wake up all lamps, or `["lamp.basic", "lamp.dimmer"]` to wake up just those types of lamp, or `["lamp.dimmer", "gateway.zwave"]` if we are interested by that, etc.

    There is no response messages addressed specifically to the sender of this request. Instead of this, recipients(s) must respond as much as possible by an `alive` notification addressed to all.
  - `getDescription`
    - `vendorId` (out): string or number assigned by our office?
    - `productId` (out): string
    - `version` (out): string
    - `parent` (out): address of the parent if any, or *uuid*0.
    - `url` (out): string, the url of a web site with extra information (and possibly the content of the schema to download).
    - `info` (out): string, any additional info if any about this device. (E.g., on the thermometer of a weather station, this may indicates that that this is the *indoor thermometer*.)
    - `unsupportedMethods` (out): an array of strings (hopefully empty), with names of methods of the schema that are actually not supported by the device for some (bad) reason.
    - `unsupportedNotifications` (out): an array of strings (hopefully empty), with names of notifications of the schema that are actually not supported by the device for some (bad) reason.
    - `unsupportedAttributes` (out): an array of strings (hopefully empty), with names of attrbutes of the schema that are actually not supported by the device for some (bad) reason.

- `getAttributes`
  - `attributes` (in): an array of string, the name of the wanted attributes. If the array is empty or if this parameter is absent within the request, this means one wants all attributes to be returned.
  - `<key-values of attributes>` (out): Attributes actually returned. The generic device defines this method with no attributes. However this method will be overridden in the definition of specific schemas of each device in order to list all attributes that could be returned by this method.
  - Note that this is not mandatory to return all requested attributes. Peers should not make any assumption on this.
  - The reply may indicate specific error codes: for instance a device may be not fully initialized at startup, or may having detected that its physical sensor is not responding; in such a case the device may reply to an early getAttribute request by returning the corresponding attribute but while adding in the error section that the corresponding value is suspicious/stale/cached...
  - Attention: devices do not return the attributes `devType`, `address` that are anyway in the header, nor the `parent` which is returned by the `getDescription`.
- `setBusConfig`
  - `busAddr` (in/out): a string specifying the IPv4 or IPv6 address of the bus on which xAAL is supposed to switch.
    If the returned address is not the sent one that means that the device could not be set with this address (in short, there is an error, and an error notification).
  - `busPort` (in/out): an unsigned integer (well, limited to 65535), indicating the port on which the xAAL bus is supposed to switch.
    Similarly, if the value returned is not the one that had been sent, is that there was an error.
  - `hops` (in/out): an 8-bit unsigned integer indicating the hops used for sending multicast packets.
    A priori we put 1 by default, but it seems normal to leave the option of setting it too.
- `getCiphers`
  - `supportedCiphers` (out): an array of strings among `"none" "md5" "sha256"` etc.: the list of ciphers supported by the device.
  - `enabledCiphers` (out): an array ov ciphers that are actually enabled.
- `setCiphers`
  - `ciphers` (in/out): an array of strings among `"none" "md5" "sha256" ...` plus keywords `"all"`: the list of ciphers to supported now (used to enable/disable some ciphers).
    The array returned is the list of ciphers that were actually enabled. (It may be different of the requested if some are not handled by the device.)
- `setKey`
  - `key` (in): string; This is the easiest way to configure the cryptographic key used for encrypted communications. Since allowing changing the security key by network may introduce security issues, some device may decide to always ignore this request or report an error...

```
{
 "header":
  {
   "version": "0.4",
   "source": "bd081741-ceb6-41eb-87e0-6628d4959657",
   "targets": [ "83c84e87-a230-4245-8829-f788fa6365a0" ],
   "devType": "hmi.basic",
   "msgType": "request",
   "action": "getAttributes",
   "cipher": "none",
   "signature": ""
  }
}
```

```
{
 "header":
  {
   "version": "0.4",
   "source": "83c84e87-a230-4245-8829-f788fa6365a0",
   "targets": [ "bd081741-ceb6-41eb-87e0-6628d4959657" ],
   "msgType": "reply",
   "devType": "thermometer.basic",
   "action": "getAttributes",
   "cipher": "none",
   "signature": ""
  },
 "body":
  {
    "temperature": 9.3
  }
}
```

Figure 2: Example of xAAL messages

8 Definition of a message

A message consists of a *header* and a *body*. The header is mandatory, while the body is optional.

Figure 2 gives an example of xAAL messages.

8.1 Header

The header of a message includes:

- `version`: The version of the protocol. This document describes version 0.4.
- `source`: Address of the sender of the message.
- `targets`: An array of destination addresses of the message.
  Note that the array may be empty, which means the message is of broadcast type. The receivers (thus everybody) are invited to consider this message. Notification messages are often of broadcast type, but not necessarily. In contrast, response messages are addressed specifically to the address of the requester.
- `devType`: The schema name (the pair *classe.type*) of the sender.
- `msgType`: The type of the message among `request reply notify`. Thus, typically, a message `request` is followed by a `reply` if it is necessary to return values to the requester (or a broadcasted notification if it is more useful from the application point of view.) Note that the specification does not define any sequence number or nor id for request-response... Inconsistencies may happen; it is discussed below.
- `action`: The name of the action brought by the message from the list of *methods* and *notifications* described in the schema whose name is indicated in the message.
- `cipher`: The safety mechanism used in the message. So far the `none` mechanism has been defined...
  In the future we may consider signing/ciphering mechanism using well-known algorithms (HMAC SHA AES...), with a key shared across the bus.
  One can also consider asymmetric encryption such as PGP (or SSH); the private key of each device being hard-coded, and public key written on a sticker on devices and shared in the bus (and signed by the owner of the installation)...
  See below for a discussion on the subject.

- `signature`: the signature of the message by the selected signing algorithms...
- `timestamp`: an positive, the date of the message in seconds since 1970-01-01 00:00:00 UTC. This parameter is optional and is dedicated to security to avoid the replay of messages. Devices may consider an acceptance window, e.g., 5 seconds.

8.2 Body

The body contains parameters and values for queries, responses and notifications, if needed.

The body is optional: it depends on what has been defined in the schema for the considered action.

8.3 Cases of error

The spirit of the xAAL specification is primarily to focus on interoperability. It is guided by the will to propose, not to constrain... Consequently xAAL has very few safeguards against the failure. It is the responsibility of developers of xAAL components to do the best they can.

The specification says nothing about the lack of response. On the one hand, since it is UDP, the query as the answer may be lost. On the other hand, there are devices that only know how to emit but not to listen to on the bus. The specification does not impose any timeout mechanism: some device does not necessarily have a clock at hand and can manage responses to requests in a circular buffer (and if a reply arrives so late that we forgot question, too bad)...

Another argument concerning the non-response to a request: in the case of a non-unicast query (i.e., actually a broadcast one or on an overall class type), it seems natural that devices unable to honor the request may choose to remain silent rather than solicit the bus with error messages.

In some cases one may wants to ensure that messages are not lost (e.g., alarm, urgency). Since the loss is not supported by the xAAL protocol layer itself, this can be supported by the application above. Several strategies can be considered:

- After an action request to a device, the requester questions the target device to see if there has been a change in its attributes;
- The API of a request includes an ad-hoc parameter (e.g., `requestId`) that the device has to repeat in its answer;
- etc.

The cases of error or inconsistency are possibly many. The specification does not impose a specific behavior. So, in case of inconsistent message, duplicates, divergent answers or notifications, *the behavior is unspecified.* (Please do not make any assumption on error recovery.)

One might want to declare nevertheless that, faced with a poison message, there is the obligation to reject it, to ignore it. Unfortunately, for some equipment, it is not possible to undo what it started to do (not enough memory or partial action). In short, the specification imposes nothing. If the error can be recovery, fine, do it! Otherwise, don't worry! The developers have the responsibility to be careful, and do the best.

For example, in some cases it is relatively easy to decide:

- A device receives a request but the name of the method does not exist in the schema. It can easily identify the problem and ignore the request.

- A device receives a request but some parameters are missing, or of the wrong type. Depending of the context, it may have started to partially fulfill a number of tasks for the requested method and may end up in trouble. Maybe, it cannot undo what he began. Therefore, no one can impose it rejects the message. If he does, that's fine. If not, too bad. At the most, it sends a relevant error code.

- Conversely, if there are missing returned values in an answer, or of the wrong type, the device that initiate the request may have started to partially process the response... It is asked to at its best.

- If the schemas are written carefully, they can provide a number of messages error. But this is the responsibility of schemas, not of the xAAL bus.

Software developers must be aware that xAAL is not a *transactional* system nor a *remote procedure* call system. (Well, not quite.) xAAL is much more lightweight. It is a basically stateless system. One sends a request to a device; some time later one may get an answer (or not); from time to time the device may detect an error (based on a faulty request or something else) and notify others... There is actually no formal relation between requests, answers and errors. As a consequence, there is no reason (or few) to wait for an answer or an error message after sending a request. Do not implement state-machines that keep the past in memory and expect something specific in the future. So, software developers should implement basic reactive systems, without states (or very lightweight).

9 Security

In preliminary work, we have not proposed a model for security mechanism. Today, security seems to be required. This section draws some perspectives.

Let us consider an infrared remote control: the RC5 link (or RC6) is not encrypted. To hack the transmission between the remote and the TV, it is necessary that the neighbor can aim with its own remote control through the window of the living room. These are the walls of the room which insure the safety of the transmission, not the protocol itself.

It is the same for IP protocols: it is necessary to be already present on the network to be able to attack it. It is necessary to be physically plugged to wire with Ethernet; WiFi connections have (usually) WPA keys that one have to know; remains powerline adapters for which encryption if available but rarely enabled (or with the predefined key)...

And finally, to contrast, think that the firewall configuration of the home network can be bad, or a PC on the network can be compromised by virus...

So, one can do something...

9.1 Need for security

When one talk about security, one must first define against what one need protection. In our case, it is not a question of dealing with cases of failure, but rather cases of malice.

Among the classical precepts for security (integrity, confidentiality, availability, non-repudiation, authentication), what does one need?

At least, it is necessary to ensure the authentication of messages sender, that messages are not corrupted, and that replay can not be done.

Remains the issue of confidentiality: a priori this does not necessarily matter if the neighbor can listen (as long as he cannot inject false messages). However, this can be annoying for certain use-cases: this can be a problem if others may be informed that the alarm control is switched off, or when the door is unlocked...

On the other hand, it is necessary to keep in mind that one may have very simple devices and that encryption mechanisms are resource-intensive.

In addition, to implement this it is necessary to go through a configuration stage (to set up encryption keys). This can be complex. (One can have a look at automatic configuration mechanisms for WiFi key, such as WPS, or the association-button for HomePlug AV)

9.2 Proposals for security

- `"cipher":"none"`: it is necessary to keep the possibility of communicating without encryption, just for all those devices that are too light to embed encryption mechanisms and configuration keys.

- `"signature":xxx`: a signature mechanism with a conventional hash function (MD5 SHA1 SHA-256 SHA-512 Blowfish...) with a shared key. This ensures authentication and integrity. On the other hand it implies to manage the configuration of device to set up this key.

  Using a pre-shared key seems acceptable in the context of a home network: it is the choice of WiFi WPA/WPA2 Personal and of HomePlug AV Powerline.

  Another approach is to use asymmetric key mechanisms: the private key being hardcoded into the device, its public key is written on a sticker on the device, with a mechanism for sharing a signature of these keys by others (like PGP).

  Nevertheless, to avoid replay it must be added in the signed message a little something that makes it unique... For this, a timestamp can be added (RFC3339). Or, the schema of the device can provide a mechanism for challenge-response: first ask the device for a cookie, and then return it in the next message by signing it. Such a mechanism has its right place in the schema of the device, not in the xAAL protocol itself.

- Confidentiality: it implies encrypting the information in the message. There are two possible approaches:

  - This can be specified not by xAAL but in the schema of the device that needs it: one exchanges a public *data* parameter, but the value is encrypted.
  - Another approach would be to redefine the message format by adding a new body block which will be fully encrypted.

At the moment, all this is still embryonic. However, the general idea is that security must be possible but optional.

9.3 Preliminary tests

Nowadays a signing mechanism has been implemented and tested.

It use HMAC SHA256 on messages, with the `"signature"` parameter previously padded with zeros. The signature is then encoded in hexadecimal. Then the signature is replaced in-place within the message. (I.e., without reusing a json library since some library may reorder json parameters or add spaces.)

The key is computed from a passphrase using PBKDF2 HMAC SHA256 with 4096 iterations and the null-terminated string "xAAL" as salt.

A timestamp is added. A typical acceptance window is of 5 seconds.

It looks fine. However this adds complexity: devices must define a list of supported ciphering methods, a list of enabled ciphering methods to accept messages, the ciphering method used to send messages, and an acceptance window regarding timestamp of received messages... This experiment leads us to think that only one security method should be specified, not several at a time.

This experiment was only about signing. Future experiments will consider encryption, probability with Poly1305/ChaCha20 and the timestamp reused as the nonce... However, early adopters of xAAL should be aware that the message structure may be rearranged to support the encryption...

10 Best Practices

The xAAL specification in itself allows numerous of things, good and bad... Along preceding pages some suggestions of good practice were discussed. They are grouped here.

10.1 Composite Device

In the home automation domain, it is common to find devices which include several functions in one unit: e.g., thermometer, door opening sensor, pilot wire (this is a real case).

With xAAL this is organized around the notion of *composite device*. (See it as a way to manage multiple inheritances...) This device is then going to expose as many xAAL devices as elementary functions. In the example here: a device with the thermometer schema, a device with the door opening schema, a device with a pilot wire schema.

Furthermore, the device will exhibit itself with the composite schema, whose sole function is to list its embedded devices (give addresses and schema names). The fact of exposing itself as a composite is not mandatory, it is only recommended: it does not provide any functionality from the home automation point of view, but it allows expressing the hardware architecture, which can aid in the diagnosis and maintenance.

N ote: The schema of a composite device may indicates the list of the required schemas of embedded devices by using the optionnal key `used_embedded`. This is at the schema level, as additional information (compares to the run-time level when a composite is questioned about its embedded).

Fore instance, the schema of a specific weather station (e.g. called "weatherstation.foocorporation") may have `"used_embedded": "thermometer.basic", "thermometer.basic", "hydrometer.basic"`. This means that a device of the type "weatherstation.foocorporation" embeds two thermometers (e.g., an indoor thermometer and an outdoor thermometer) and an hydrometer.

10.2 Gateway

Let us summarize the role of a *gateway*:

- Transform xAAL messages into messages of a home-automation protocol of a manufacturer, and vice-versa;
- Manage the pairing between itself and manufacturer devices;
- Manage the configuration of these devices (e.g., Z-Wave being bidirectional, it is the so-called "Z-Wave base" which pushes certain parameters towards devices: here the xAAL gateway plays then the role of the Z-Wave base, so it manages these Z-Wave parameters but does not exhibit them on the xAAL bus);

- Expose the devices of the manufacturer on the xAAL bus with their xAAL address and their xAAL schema;

- Do the mapping between the xAAL addresses and the manufacturer addresses;
  Note that this mapping can be useful in the configuration of the database of metadata. (For instance when the power is reconnected after installing some automation devices: all will appear in a mess on the bus... It is thus necessary to identify which is which.) The gateway can therefore be asked on this mapping. For each device, the mapping is typically a pair: the xAAL address, manufacturer address. From one manufacturer to another, the notion of address may differ (e.g., some add the notion of group). However, thanks to the flexibility of Json, the address format for each manufacturer will be defined later. (At least it is a dictionary that contains the parameter `protocol` whose value names the protocol of the manufacturer, followed by physical address of the device according to the protocol of the manufacturer.)

- Do the translation between the functions of manufacturer devices and methods described in xAAL schemas;
  This translation is supposed to be fairly straightforward, since the gateway is almost transparent in this point of view.

By design, a gateway appears as a composite device. This is a somewhat more sophisticated than the simple composite device previously described, but it is a device that exhibit itself as such on xAAL bus.

There is the case of gateways that handle manufacturer devices that are themselves composite.

Let us take the case of the «Door / Window Sensor 3 in 1 FIB-FGK-101».[4] This Z-Wave equipment does: door/window opening detection, thermometer, digital data relay (to connect an add-on). The Z-Wave gateway will therefore exhibit on the xAAL bus:

- A door opening detection device (with a schema "opening detection" and its own xAAL address);
  This device has for *parent* the composite device described below.

- A thermometer device (with the schema "thermometer" and his own xAAL address);
  This device has for *parent* the composite device described below.

- A device "digital data" (with the schema "digital data" and his own xAAL address);
  This device has for *parent* the composite device described below.

- A composite device (with the schema "composite" and his own xAAL address);
  This composite device has for *embedded* the previous three devices.
  Moreover, it has for *parent* the gateway described below.

- A gateway device (with the schema "gateway" and its own xAAL address);
  This composite device has for *embedded* the previous four devices.
  It has no *parent*.

[4] `http://www.planete-domotique.com/autres/par-technologie/zwave/capteur-d-ouverture-3-en-1-fibaro.html`

10.3 Virtual Device

In the functional xAAL architecture, the automata of scenarios are an ideal place to implement "intelligent" home-automation services. For example:

- When an order is received, trigger a whole series of actions (e.g., "evening of TV".)
- Or conversely, by correlating a variety of events, synthesize information (e.g., "Presence of a user" or "Water tap left open").

In so doing, the automata appears then itself on the xAAL bus as being a device. It is a *virtual device*, but a device with its own methods, notifications, attributes: in brief, is defined by a schema and other entities can interact with it via the xAAL bus.

The fact that it is virtual has no impact from the xAAL point of view. It is a device like others.

10.4 Device Overloading another one

A home automation system may have multifunctional devices.

Let us consider for example the "Micro-module Z-Wave Switch FGS-211". [5] This equipment has a size studied to be inserted into a wall outlet and is able to control any electrical device (lamp, fan, shutters, etc.).

This type of micro-module, more and more common, provides complex functionality. This is what says the catalog about this one:

- On/off relay controlled by Z -Wave or connected buttons;
- Can use 1 or 2 buttons (the second optional button can control one or more other modules associated with Z-Wave);
- The buttons can be bistable (on/off) or monostable (pulse pushbutton);
- Report state via Z-Wave.

Therefore, the Z-Wave gateway that supports it will exhibit on the xAAL bus the following devices:

- A device of type relays with state feedback (with the schema "relay", and its own address) corresponding to the electrical device connected to it and thus controlled by the micro-module (but at this level it does not say if the controlled electrical device is a lamp, a fan, a shutter or something else);
- A device of type button (either with the schema "bistable", or either with the schema "monostable" according to the chosen configuration), and correspondent in the second optional button (since the first button is used for the manual command of the relay);
- And possibly a composite device embarking the two previous ones.

However, one misses the information of configuration saying that behind this module this is a lamp which is controlled and not a fan nor a roller blind.

To do this, one will define a new xAAL entity, a new device that will expose the "lamp" schema and that will do the translation between actions on this "lamp" device and actions on the "relay" device described above. This new device is a virtual in a way, but mostly it just

[5] `http://www.domotique-store.fr/domotique/modules-par-types/micro-modules/19-fibaro-fgs-211-relay-switch-module-micro-module-zwave-switch-relais-simple-on-off.html`

overloads the generic device. So, this can be called an *overloading device.* (And from the xAAL point of view it has as parent the preceding generic device.)

Note that since it is the gateway that exposes this new overloading device, this gateway is not completely transparent in our architecture: it must be configured to set up how generic devices are overloaded. We believe that these configurations should be located in the gateway and not in (for example) in the metadata database. Indeed, it is the setting of inner workings of the device, not an attribute which just tag it.

Strictly speaking, a gateway that will handle a generic micro-module should therefore expose on the xAAL bus two devices: the generic one plus the overloading one. However, for reasons of simplicity for implementation (and also because we know how are the developers), it is quite acceptable that the gateway makes the economy of the generic device and only expose on the xAAL bus the overloading device.

10.5 Hidden Device

Some devices are not intended to be presented in user interfaces. This is the case of gateways, composites, automata of scenarios, metadata databases, caches, interfaces themselves, generic devices that are masked by an overloading device, etc.

One should not make assumption of which device should be hidden or not. This property (if needed) should be an explicit tag within the metadata database. (Possibly with a default value according to the class diagram, but a tag that can be changed later.)

User interfaces know all devices (hidden or not), but choose to display only those that are not hidden unless the user explicitly requests it (like in a file manager on your PC you can choose "show hidden files").

This information (to specify that a device is hidden or not) could be just local to the HMI and be managed locally inside the HMI. However, we think this is specific to the automation system and must be stored in the metadata database and available to all.

11 Acknowledgments

We wish to thank Maria Teresa Segarra, Corinne Le Moan and Florian Le Nestour for their active participation in the work on xAAL.

A Draft of Schemas

Here are some general ideas on the content of schemas of devices.

```
Notes:
- a:attribute
- m:methode
- n:notification


any.any (the generic schema)
=======
a:devType
a:address
a:parent

n:alive()
n:variablesChange(out v:+)
n:error(out description, out code)

m:isAlive(in devTypes)
m:getDescription(out vendorId, out productId, out version, out parent, out url,
                 out unsupportedMethods, out unsupportedNotifications,
                 out unsupportedAttributes)
m:getAttributes(in attributes, out a:*)
m:getBusConfig(in/out busAddr, in/out busPort, in/out TTL)



lamp.basic (extends any.any)
==========
a:light(bool)
m:on()
m:off()


lamp.dimmer (extends lamp.basic)
===========
a:dimmer(float, precent)
m:dim(in/out float level)


lamp.rgb (extends lamp.queryable)
========
a:level {red, green, blue} (float, precent)x3
a:mode [fixed,shuffle]
m:dim(in/out float red, green, blue)
m:setMode(fixed|shuffle)



shutter.basic (extends any.any)
=============
a:position (open|closed|opening|closing)
```

```
m:up()
m:down()
m:stop()


door.basic (extends any.any)
==========
a:position (open|closed)


locker.basic (extends any.any)
============
a:position (locked|unLocked)
m:lock()
m:unLock()


mediaplayer.basic (extends any.any)
=================
a:activity (play|pause|stop)
a:destination (address of the mixer)
m:play()
m:pause()
m:next()
m:previous()
m:stop()


mediaplayer.spotify (extends mediaplayer.basic)
===================
v:url
m:playUrl(url)


audiomixer.basic (extends any.any)
================
a:source(address of the player)
a:volume(percent)
m:setSource(addr)
m:up
m:down
m:mute
m:unmute


composite.basic (extends any.any)
===============
a:embedded


gateway.basic (extends composite.basic)
```

```
=================
a:address_mapping
```