A brief introduction to xAAL v0.5-draft rev 2

Christophe Lohr Jérôme Kerdreux

October 9, 2018

Abstract

This document summaries information which specifies xAAL such as defined by the IHSEV/HAAL team of IMT-Atlantique (ex-Telecom Bretagne) in spring 2017.

Simply speaking, this is an attempt to gather into one document the main elements of xAAL version v0.4-r3 and xAAL version v0.5-r1. Motivations, arguments, alternatives, and discussions have been withdrawn. This document describes how is xAAL, just that.

Be aware that this is an ongoing work.

1 Introduction

The xAAL system is a solution for home-automation interoperability. Simply speaking, it allows a device from vendor A (e.g., a switch) to talk to device from vendor B (e.g., a lamp).

The xAAL specification defines: (i) A functional distributed architecture; (ii) A mean to describes and to discover the interface and the expected behavior of participating nodes by so-called schemas; and (iii) A secure communications layer via an IP (multicast) bus.

The xAAL systemp is based on "the best effort" principle. Each one do its best according to its capacity for things to go well. There is no warranty. There is no quality requirement to expect/provide from/to others. Main points of xAAL where discussed in the following papers:

1. C. Lohr, P. Tanguy, J. Kerdreux, "xAAL: A Distributed Infrastructure for Heterogeneous Ambient Devices", Journal of Intelligent Systems. Volume 24, Issue 3, Pages 321–331, ISSN (Online) 2191-026X, ISSN (Print) 0334-1860, DOI: 10.1515/jisys-2014-0144, March 2015.

2. C. Lohr, P. Tanguy, J. Kerdreux, "Choosing security elements for the xAAL home automation system", IEEE Proceedings of ATC 2016, pp.534 - 541, Jul 2016, Toulouse, France.

2 The xAAL architecture

The xAAL system is made of functional entities communicating to each other via a messages bus over IP. Each entity may have multiple instances or zero, may be shared between several boxes, and may be physically located on any box.

Figure 1 shows the general functional architecture of the xAAL system in a typical home-automation facility.

2.1 Typical xAAL nodes

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

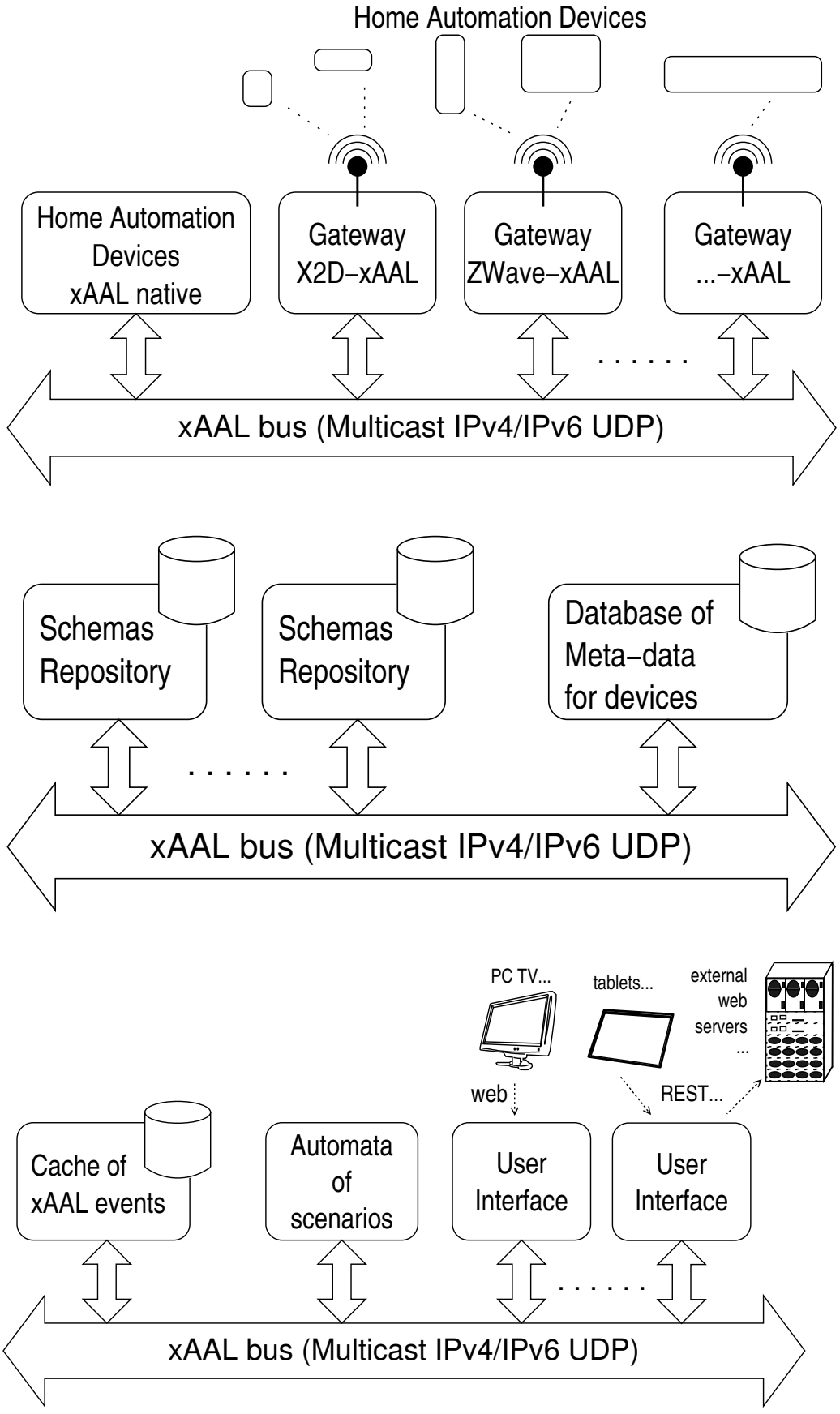

Figure 1: Functional Architecture of xAAL.

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.

Within an installation, each manufacturer protocol may be served by a dedicated gateway.

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.

Gateways can be queried about the list of managed 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 a repository 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. Therefore, there possibly can be conflicts: given a schema name, several registries on the bus could return schemes having divergent descriptions. This is the responsability of the callee to solve conflicts according to its own startegy, for its own purpose. The xAAL specification does not enforce any rule for this.

Database of Metadatas. Each automation device within a facility is likely to be associated to a piece of information: for instance some location information (e.g., declare that the equipment typed as a lamp and having address X is located in the "kitchen" or in the "bedroom of Bob"), possibly a friendly name to be displayed to the end-user (e.g., device having address X is called "lamp#1"), or any useful piece of information for users regarding devices on its facility (e.g., pronunciation of the nickname of the device for a voice interface, or whatever).

All this information is grouped in a *database of metadata*. This database contains somehow the configuration of home automation installed from the end-user point of view.

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, if needed.)
- 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 on the xAAL bus is managed differently.)

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

Such tags are structured as key-value pairs. Keys are UTF8 strings. The xAAL specification does not define a normative list of predefined or well-known keys entries. As a consequence, the meaning of a tag just make sense for the entity which write it in the database and for the entities which read it. This is the responsability of those entities to agree on a common semantic to interpret those tags.

Cache. Unidirectional sensors are quite common: one can not question them, they send their information sporadically. (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. Note that even if such caching feature is implemented by the gateway software itself, the cache is seen as a dedicated xAAL device.

Such a cache should store at least the values of attributes carried by notifications sent by the devices. It can also stay informed by monitoring request/reply messages between devices about values of attributes.

As with 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. Moreover, the clock used by caches to set such timestamps may be not accurate or well sychronized. This is the responsability of clients to deal with that.

Again, inconsistencies may arise if two caches return information that has the same timestamp but divergent values. This phenomenon is a priori very rare. However, the xAAL specification does not enforce any rule to solve incosistencies.

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.

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 microphone which performs voice recognition; 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.

Within a home automation system there may be one or many user interfaces.

2.2 Summary

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.

3 The xAAL devices behavior and schemas

3.1 Definition of a device

A *device* has a `devType` and an `address`.

devType: This 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.).

  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, an RGB lamp, etc.). The second word may also refer a schema extension by a manufacturer (cf. 3.3).

- The keyword `"any"` is reserved and acts as a wildcard for requests. So that the identifier `"any.any"` is reserved and refers to all types of all classes within request messages. This is a *virtual* type. It is not allowed to defines a schema named `"any.any"`. No one can clame to be of type `"any.any"`. Devices having no dedicated attribute, method or notification may use the *concret* type `"basic.basic"`, simply.

- For example the pair `"lamp.any"` means all types of the class `"lamp"`. This is also an abstract type. If needed (to implement a simple basic lamp), the concret type `"lamp.basic"` is provided.

- We reserve a special name: `"experimental"`, for the class, as well as the type. These are concret 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.)

Address: The device ID, unique on the bus.

- This device identifier is a UUID (RFC 4122), a 128-bit random number.

- Addresses are self-assigned. There is no naming service nowhere. A device does not request any entity to get an address.
  Concretely, addresses may be assigned with thereby:

  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.

3.2 Definition of a schema, the type of device

Each device is typed, i.e., described by a so-called *schema*. An xAAL schema is a JSON object with a specific form that must validate a given *JSON Schema*[1].

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;
  - 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*. Each notification is described by:
  - A unique name which identifies it within the schema;
  - A textual description;
  - A list of information elements; Each information element is defined by:
    A name
    A description
    The unit
    The type
- A list of *attributes*. If the value changes, this spontaneously generates a notification to the bus.

  Each attribute is defined by:

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

A name
A description
The unit
The type

3.3 Inheritance of schemas

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

1. A generic schema common to all existing devices of the world that everyone has to implement. (See 4.)

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 dimmable, and others offers RGB control, etc.)

4. Thereafter, manufacturers of home automation equipment will naturally define theirs own schemas among theirs products range. However, manufacturer must not define theirs own schemas as level 1 schemas (the generic schema is the only one), nor as level 2 schemas (class schemas). Schemas from manufacturer are necessarily extensions of schema from level 2 or higher.

   While naming such schemas, manufacturers may use the second word of the pointed pair as their own discretion. This is theire responsability to choose a name that may not conflict with existing ones. However, the schema name should refer above all to the functionality of the device and means to interact with it; the name should also give an idea of the nature of the device.

Semantic. Extending a schema has the following meaning:

- 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 objects, that is to say *attributes*, *methods* and/or *notifications*. An overloading means that such an object has the same name and a different definition. There is no attempt to *merge* the old and the new object definition. The resulting schema contains all new objects (*attributes*, *methods*, *notifications*) plus old ones that has not been not overloaded.

4 Generic Schema

This is the basement of every schema. All other schemas must inherit from it somehow. It is named `basic.basic`.

Attributes

- *Attributes involved in the protocol*

  `devType`: the name of the schema to which the device obeys;

  `address`: a string, an UUID (the address of the device);

- *Attributes describing the device*

  `vendorId`: string, the Id of the vendor assigned by the xAAL bureau (via the dedicated IANA Registry);

  `productId`: string, an Id of the product assigned by the vendor;

  `version`: string, version or revision of the product assigned by the vendor;

  `hwId`: untyped, some hardware Id of the device (e.g., low-level addresses of the underlaying protocol, a pairing-code, a serial number, or any piece of information that may helps to retrieve a device within a facility for maintenance);

  `groupId`: a string, an UUID shared by all deviced belonging to the same physical equipment (e.g., each plug of a multi plug outlet, each thermometer and hydrometer of a weather station, etc.);

  `url`: string, the base url of a website with extra information (and possibly the content of the schema to download on <`url/devType`>);

  `info`: string, any additional info if any about this device that should make sens for the end-user; (E.g., on the thermometer of a weather station this may indicates that this is the *indoor thermometer* or the *outdoor thermometer*, or on a plug belonging to multi plug outlet this may indicate the position of the plug.)

  `unsupportedAttributes`: 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.

  `unsupportedMethods`: 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`: 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.

- *Attributes for accessing the bus*

  `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.

The attributes of the `basic.basic` schema are mostly considered as *internal*, or dedicated to the configuration of the device. Unlike attributes of extended schemas, they must not be involved in the `attributesChange` notification nor in the `getAttributes` method described bellow. They are managed via specific ways.

Notifications

- `alive`: emitted when starting the device and then periodically emitted at a rate left to the discretion of the device. The notification message contains no arguments. (I.e, the *body* part of the message is empty; see the definition of messages below.)

- `attributesChange`: emitted at every change of one of the attributes (except those belonging to `basic.basic`). 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. Schemas extending `basic.basic` may overload the `attributesChange` notification according to the extra attributes introduced in the extended schema.

- `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.

Note: notifications messages must be addressed to all.

Methods

- `isAlive`

  `devTypes` (in): an array of *devType* strings, giving names of the schema of devices that should wake up.

  One may have:

  an empty array or `["any.any"]` to wake up everybody, or

  an abstract type, e.g.,`["lamp.any" ]` to wake up all lamps, or

  a specific type, e.g. `["lamp.basic", "lamp.dimmer"]` to wake up just those types of lamp, or

  an array of above mentionned items, e.g. `["lamp.dimmer", "shutter.basic"]` if we are interested by that, etc.

  An `isAlive` request must not cause any response message. Instead, recipients(s) of the request must respond as much as possible by an `alive` notification (addressed to all).

  This method must not be overloaded by extended schemas.

- `getDescription`

  `vendorId`, `productId`, `version`, `hwId`, `groupId`, `url`, `info`, `unsupportedAttributes`, `unsupportedMethods`, `unsupportedNotifications` (out): see the meaning in the above list of attributes.

  This method should not be overloaded by extended schemas. In case if this method is overloaded, the above listed arguments must remain.

- `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, all attributes should be returned.

  `<key-values of attributes>` (out): Attributes actually returned. The schema `basic.basic` defines this method with no attributes. However, this method may be overridden in the definition of extended schemas.

  It is not mandatory to return all requested attributes. Peers should not make any assumption on this.

  Reminder: devices do not return the attributes defined by the `basic.basic` schema, only attributes introduced by extended schemas.

4.1 Composite devices and equipments

This section discusses the case of equipments that are composed of a set of several elementary devices. Those cases arise in different circumstances for which the xAAL solutions differs. Here are some best-practices guidelines.

4.1.1 Gateways

A gateway is a piece of software and hadware which make the bridge between the xAAL bus and a proprietary home-automation protocol used by some devices deployed in a facility (e.g., X10, Zigbee, X2D, Z-Wave, HomeEasy, KeeLoq, etc.).

The gateway presents one xAAL device on the xAAL bus for each proprietary home-automation device it manages; each one with its own xAAL address and its own `devType`. They are called *embedded devices*.

Moreover, the gateway should presents itself as an additionnal xAAL device, with its own address an a `devType:"gateway.basic"`, or any relevent extended type of gateway. The gateway maintains the list of its embedded devices and can be questioned about it. Some advanced gateways may also provide information about the status of managed devices (e.g., if the gateway know that certain devices are defective or dead).

Eaven if it is not recommanded, a gateway can also choose to be not visible on the xAAL bus, does not announce itself, nor indicates that it has embedded devices.

4.1.2 Group of devices

A physical equipment may consist of several devices. For instance consider the case of a "smart control multi plug socket", or a small weather station with several sensors that measures indoor/outdoor temperature, humidity, etc.

In such a case, the equipment is seen as a set of different xAAL devices, each one with its own functionnalities, attributes, etc. From a logical point of view (the xAAL point of view), this equipment is divided into several xAAl devices. However, those devices that belong to the same physical equipment are groupped. Within their descritption they have the same `groupId` value (an UUID in fact). They are called *groupped devices*.

A group just tells that some xAAL devices are physically together, and only this. (Which is however a high valuable information for maintenance.) This does bring no indication about the fact that some of those xAAL devices could be in relationship from a logical or functionnal point of view or to provide some service. (This is handled within the definition of the schema of the devices, if needed.)

4.1.3 Devices using others

A device may provides services on top others devices. For instance consider the case of the previous "smart control multi plug socket". Each of the plug may be smart because it provides two services. Fisrt of all this is a basic plug with a remote controlled on/off. This service is implemented by the xAAL device `powerrelay.basic`. Moreover, those smart plug may embed power-consumption monitoring services. Such a service is implemented by the xAAL device `powermeter.basic`.

By design, the xAAL device `powermeter.basic` provides its service on top of others devices. It measures the power of something: possibly only one plug in the case of the previousely mentionned "smart plug", or possibly a large set of several devices, a part of the power supplying network of thefacility (e.g., all equipments of the kitchen, or of a given floor, etc.). The definition of the `powermeter.basic` device includes an attribute to list measured devices (called `power` in the curent version of the schema definition). This has no relation with the `groupId` introduced in the previous section.

There is the same consideration with the xAAL device `weatherforcast.basic`. Its schema definition includes an attribute to list other devices actually used to compute its forcasts (thermometers, hydrometers, etc.). This has no relation with the `groupId` usage, which only means that those thermometers, hygrometers and forecast algorithms may belong (or not) to the same physical box (the weather station).

5 The xAAL communication protocol

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

Note: Due to the "experimental" status of this protocol, no IP multicast address (neither IPv4 nor IPv6) and no UDP ports range have been yet be claimed to IANA registries.

Messages carried by the xAAL bus are in JSON format (rfc7159).

xAAL messages are made of two layer: a *security layer* which encapsulate an *application layer*.

5.1 The Security Layer

The data of the security layer is the palyoad of the UDP multicast messages of the xAAL bus. This is a JSON object whose fields must be:

- `version`: The string `"0.5"`. (The version of the protocol.) Others values should be rejected.

- `targets`: A string build as the JSON serialization of the array of destination addresses for the message. Remeber that xAAL addresses are UUID. The textual representation of UUID (RFC 4122) must be used. An xAAL device receiving a message shoud accept it if its own xAAL address is present in the array of targets. An empty targets filed means a broadcast message. A targets filed containing an empty string is not allowed (the message should be rejected).

- `timestamp`: An array of two (and exactly two) integers: the first number is the number of seconds since the Epoch (1970-01-01 00:00:00 +0000 UTC), and the second number is the number of microseconds since the beginning of this second.

- `payload`: A string build as the base64 encoding (rfc4648) of the ciphered *application layer*.

If a message include other fields in addition to the above mandatory ones, the message may be accepted, and the extra fields must be ignored.

5.2 The Application Layer

The application layer is a JSON object whose fileds must be: a `header` (see bellow), and an optionnaly `body`. Other fileds must be ignored.

The header. This is a JSON object whose fileds must be:

- `source` (string): Address of the sender of the message according to UUID textual representation.
- `devType` (string): The schema name (the pair *classe.type*) of the sender.
- `msgType` (string): The type of the message among `"request" "reply" "notify"`.
- `action` (string): The name of the action brought by the message from the list of *methods* and *notifications* described in the considered schema. The considered schema is the one of the sender for reply or notify messages, and the one of receiver(s) for request messages.

Other fileds must be ignored.

The body. It contains parameters, values and information elements 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.

5.3 The Ciphering Method

The security of the xAAL bus is ensured by:

- A symmetric key, pre-shared into all participating devices;
- Poly1305/Chacha20 as the only cpryptographic algorithm, and used according to RFC7905 recomandations (a 96bits nonce, a 256 bits key);
- A binary nonce build as a timestamp since the Epoch (seconds (64bits) + microseconds (32bits));
- An acceptance window for the timestamp of messages;
- The list of targets in clear, but covered by the signature;
- A security layer in JSON;
- An application layer in JSON;
- The application layer is serialized into a string, ciphered, encoded in base64, and placed as a string into the security layer.

Notes:

- The `targets` field of the security layer is a string. This is not an array of UUIDs, this is the JSON serialization of an array of UUIDs. This string may bee seen as a buffer of bytes and can directly be used as the *public additional data* for the Poly1305/Chacha20 algorithm, to be covered by the cryptographic signature.

- The binary nonce (96 bits) to be used with Poly1305/Chacha20 is composed of the seconds and microseconds (in this order): first a 64 bits big-endian unsigned integer (seconds), followed by a 32 bits big-endian unsigned integer (microseconds).

- The encrypted payload is encoded in base64. Due to its heritage from the email tradition, the base64 encoding may insert line breaks every 72 chars, this is unnecessary here.

Recommendations:

- To buid the cryptographic key from a passphrase.
  The Poly1305/Chacha20 algorithm use a binary key on 256 bits. A fine way to select a *"good"* key is to build it from a passphrase using a cryptographic hashing algorithm.

  It is proposed to use the dedicated function provided for this purpose in the reference Chacha20 library (the *sodium* library), and derived libraries:

  - function: `crypto_pwhash_scryptsalsa208sha256()`
  - for the salt: a buffer of zeros
  - for the *opslimit*: `crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_INTERACTIVE` (512k cycles)
  - for the *memlimit*: `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_INTERACTIVE` (16 Mbytes)

- To choose a window of acceptance for the timestamp.
  An acceptance window of two minutes should be fine.

- To have several keys on the same bus.
  If several key or xAAL version are needed in a facility, it is recommended to use different xAAL buses (UDP ports).

Example. Figures 2 and 3 give an example of an xAAL message (the security layer) with its decoded payload (the application layer).

```
{
  "version": "0.5",
  "targets": "[ \"174255ad-e2a1-41e9-bf23-8dbcdfc812d4\" ]",
  "timestamp": [ 1439824426, 467313 ],
  "payload": "8sbrvczRc5NpWRpG+vwro...mlkKeeSAUc5Dj9SKoe82="
}
```

Figure 2: Example of an xAAL message (*securitiy layer*)

```
{
  "header": {
    "source": "06b71935-c5bc-4b09-8f2b-dae3d3b8ce77",
    "devType": "thermometer.basic",
    "msgType": "reply",
    "action": "getAttributes"
  },
  "body": {
   "temperature": 33.0
 }
}
```


Figure 3: The decrypted payload of an xAAL message (*application layer*)