A brief introduction to xAAL v0.7-draft

Christophe Lohr Jérôme Kerdreux

December 10, 2019

Abstract

This document summary information which specifies xAAL such as defined by the IHSEV/HAAL team of IMT-Atlantique.

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 a device from vendor B (e.g. a lamp).

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

These points are detailed in the specification document of xAAL version 0.5-r2.[1], which is considered as the latest stable release. xAAL version 0.6 is a first attempt to introduce CBOR, but with some points to improve. xAAL version 0.7 aims to address them.

- xAAL architecture remain almost the same as in version 0.5-r2 except that Schema Repositories are withdrawn.
- xAAL schemas are still written in JSON, except that now data types are specified with the Concise Data Definition Language (CDDL, RFC 8610)[2] and no more with the JSON Schema[3] dialect. Moreover, data type definitions are gathered in a *data model* section of xAAL schema documents.
- xAAL version 0.7 proposes to use the Concise Binary Object Representation (CBOR, RFC 7049)[4] in place of JavaScript Object Notation (JSON, RFC 7159)[5] for messages.

The philosophy. xAAL is a distributed system based on "the best effort" principle. Each one does 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.

Following papers discuss main points of xAAL:

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.

[1]`http://recherche.imt-atlantique.eu/xaal/documentation/`
[2]`https://tools.ietf.org/html/rfc8610`
[3]`http://json-schema.org/`
[4]`https://tools.ietf.org/html/rfc7049`
[5]`https://tools.ietf.org/html/rfc7159`

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

A home automation facility is usually made of physical devices (sensors, actuators). The basic way for users to interact with a home automation facility is by pressing buttons or switches, that is to say activating dedicated physical sensor devices. Modern home automation solutions provide user interfaces such as mobile applications for smartphones tablets PC or voice interfaces. It is also common to provide some functionalities for scripting sequences of actions to be triggered automatically at a given time or to react to some expected event. There may also be some logging functionalities for monitoring parameters along the time, such as temperature in rooms, consumption of energy or water, so that users may reduce their consumption.

The xAAL system aims to address the typical uses and needs towards home automation systems.

Objectives:

- The system should be robust: the failure of a component should not prevent others from doing their job.
- The system should be lightweight, even in terms of communication. It should use existing communication networks available at home, without consuming too much bandwidth.
- The system should be flexible: adding or removing a component should not require long or complex configuration processes to make it work nor disturb the other components.
- The system should address security considerations, regarding users’ privacy, or third party’s threat to get control.
- Last, but not least, the system should handle interoperability issues, allowing a device from vendor A (e.g. a switch) to talk to a device from vendor B (e.g. a lamp), allowing consumers to buy devices they need without being enclosed into the market or the technology of the vendor of devices they already have bought.

2.1 Overview of 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.

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

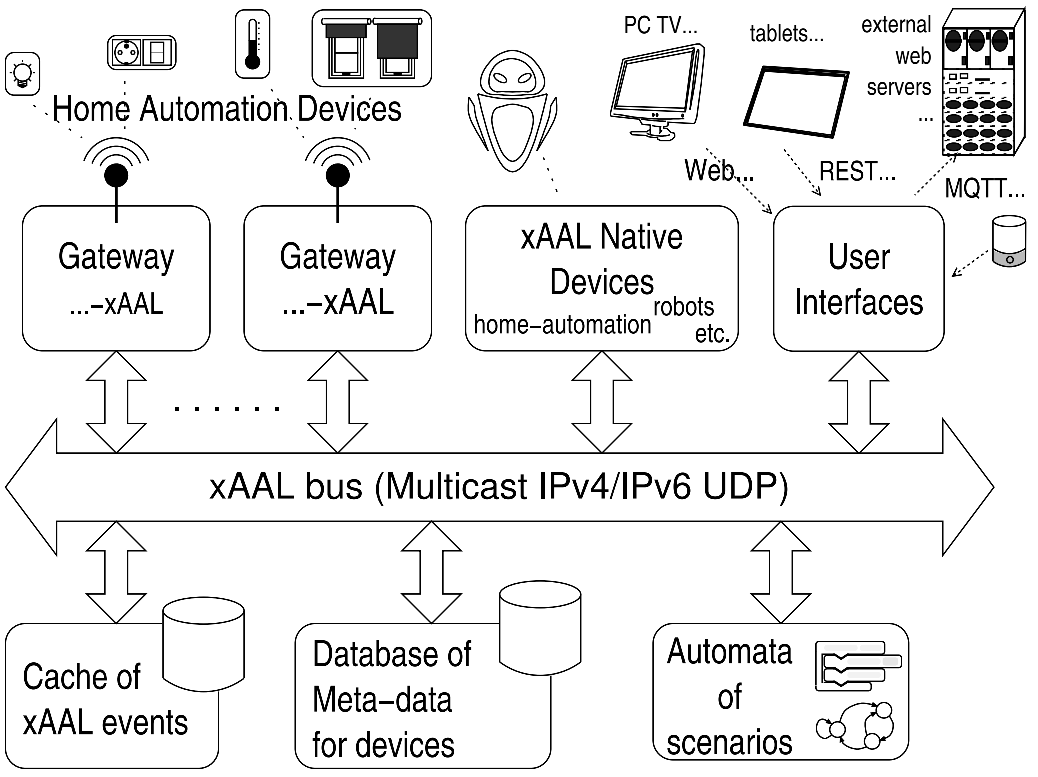

Figure 1: Functional Architecture of xAAL.

Gateways. In general, devices for home automation only support their own proprietary communication protocol. Therefore, elementary *gateways* 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 physical devices, addresses of devices, configuration, and the persistence of the configuration.

Gateways can be queried about the list of devices they manage.

Database of Metadatas. Each automation device within a facility is likely to be associated with a piece of user-defined 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 devices, 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 information associated with the identifier of a device, or conversely a list of devices associated with a given piece of information.

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

Such pieces of information are structured as key-value pairs. Keys and values are UTF8 strings. The xAAL specification does not define a normative list of pre-defined or well-known key entries. As a consequence, the meaning of a key makes sense only for the entity that write it in the database and for the entities which read it. This is the responsibility of those entities to agree on a common semantics to interpret information.

Cache. Unidirectional sensors are quite common: one cannot 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 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 not be accurate or synchronized well. This is the responsibility 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 inconsistencies.

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, consider a scenario to check for the presence of users in a room: it 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 services composition, etc.

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

2.2 Typical Work Flow

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 xAAL bus to be aware of the present devices;

2. Query devices about their description (model, manufacturer, URL, etc.);

3. Optionally, download non-standard schemas from the URL of the description of those non-standard devices, or else use standard schemas;

4. Query the database of metadata about devices to get the configuration of the installation (a friendly name to display, location and other key values associated with each device);

5. Query the cache for the latest known states of devices;

6. Dynamically build display screens and control interfaces for these devices.

2.3 Change in the xAAL Architecture

Compares to former versions, xAAL 0.7 has no more *Schemas Repository.* This xAAL device was used to provide xAAL schemas via the xAAL bus. Schemas are still there with xAAL 0.7 but are distributed by other means. Experiments on the long term have convinced us that there is not a real use of such a functionality (having schemas via the home automation bus itself instead of downloading from the Web).

Indeed, xAAL schemas are not of such a dynamic nature. They are downloaded and configured once. Standards schemas can be preconfigured, and non-standard schemas can be downloaded from the URL provided by non-standard devices in their description.

The last reason for this change is that, finally, schema files are not so used as-it by xAAL devices. Schemas are really important for developers to know formally what a device does. This is important for developers of the device described in the schema, and also for developers of other devices which interact with it. Schemas may also be used to generate code skeletons. As a consequence, the knowledge provided by schemas takes the form of software code within devices. One of the few cases where schema files are used as-is, this is by the generic user interface which is able to dynamically and automatically build interfaces on the basis of the xAAL schema files. To be honest, even if this generic user interface is completely functional, it is not so elegant neither ergonomic. It is probably not likely be used in real facilities, only for developers. To do the job, this generic user interface is able to download schemas files from the Web, it does not really need to get it from the xAAL bus.

Those discussions have led us to simplify the xAAL architecture and to withdraw the Schema Repository.

2.4 Communication Channel

The xAAL architecture is based on the many-to-many communication principles, concretely a communication bus. Any component can post messages (e.g. sensor data) to another, several, or all components without preliminary connection setup. This has many advantages over usual client-server communication principle. Point-to-point communications are avoided, which saves device memory resources. Indeed, a home automation component is usually linked to many others for things to work. Maintaining many permanent connections in parallel may be extremely heavy for constrained devices. A bus also brings some functional benefits. This 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.

For these reasons, the xAAL system has been designed on an IP *multicast bus* (IPv4 or IPv6).

2.4.1 WiFi and Multicast

WiFi weaknesses are out of the scope of the home automation study. However, this may have an impact on the xAAL architecture. This is why a discussion is welcome.

WiFi (IEEE 802.11) is a common use wireless technology for IP networks, also at home. It is widely used to provide Internet access to mobile devices (smartphones, tablets, laptops, etc.). It is also used for non-mobile devices (connected TV, video game consoles, IP CCTV cameras, etc.) when users are refractory to the installation of Ethernet cables.

Unfortunately, IEEE 802.11 protocols do suffer from design choices regarding the management of broadcast and multicast packets. Simply speaking, it has been chosen not to acknowledge broadcast and multicast packets at the radio level, contrary to unicast packets. Indeed, avoiding managing a list of nodes having not acknowledged packets greatly simplify the protocol implementation. Broadcast and multicast packets are sent only once, and at a lower rate to get a better chance to be received without error.

This point has been studied by research in wireless networks since the very first years of WiFi. Several proposals have been made, by modifying the radio MAC layer, by introducing a central coordinator, by replacing multicast by unicast either at the MAC layer or at the application layer, etc. [6] [7] [8] [9] [10]

Over the years, the WiFi Alliance has enhanced WiFi protocols, but with the objective to increase bandwidth, and for unicast. One may notice that the IEEE 802.11e standard proposes the Wireless Multimedia Extensions (WME). It provides basic Quality of service (QoS) features on which the first two classes of service among the four ones are dedicated to voice and video content. Classification is often performed on the basis of the TOS field of the IP header. WME is sometimes presented as a solution for multicast-over-WiFi issues, despite the mechanisms act at different layers and has concretely no impact. The confusion comes from the idea that multicast is dedicated to multimedia (e.g. TV diffusion over IP, which is actually one of the multicast uses cases among others), and enhancing multimedia do enhance multicast. This is not the case.

For twenty years, the Mboned IETF working group aims to facilitate deployment of multicast technologies for the Internet at large. This working group also consider the issues of deploying multicast on WiFi networks.[11] Several nowadays mitigation solutions are listed. None is perfect. The conclusion of this study is that “IEEE 802.1, 802.11, and 802.15 should be encouraged to revisit L2 multicast issues.”

Indeed, this situation contributes to the “Ossification of the Internet”: due to (technical and administrative) limitation of several infrastructure equipment, innovations tend to restrict themselves to a small subset of Internet services. To get a chance to get clients, one only uses TCP and Web communication technologies to implement innovations. On the other hand, equipment manufacturers take time to enhance their product to fully support Internet principles with the argument there is a few consumer request.

Note that the situation is probably similar when diffusing multicast packets over HomePlug-AV PowerLine Adapters. [12][13]

[6] `https://drakkar.imag.fr/IMG/pdf/perfAnomaly-infocom.pdf`

[7] `https://pdfs.semanticscholar.org/b348/624662cedc637b50eea6e28206d7e16dc71a.pdf`

[8] `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/dircast.pdf`

[9] `https://hal.inria.fr/inria-00084130v2/document`

[10] `https://link.springer.com/article/10.1023%2FA%3A1016631911947`

[11] `https://tools.ietf.org/id/draft-ietf-mboned-ieee802-mcast-problems-09.html`

[12] `http://www.plc.uma.es/articulos/Analysis_and_improvement_of_multicast_communications_in_HomePlug%20AVbased_in_home_networks_2014.pdf`

[13] `http://repositorio.upct.es/bitstream/handle/10317/4179/pjpe.pdf`

Is this severe? Well, this greatly depends on the quality of the WiFi Access Point in use. Let's consider a managed mode 802.11 network: several WiFi Stations (STA) connected to a WiFi Access Point (AP). If multicast packets are transmitted from an STA to the AP, packets are well received almost as good as unicast packets. However, if packets are transmitted from the AP to STAs, there are risks packets are lost, depending on the quality of the AP and the global situation. According to our informal and humble tests in lab with a small set of different AP, nowadays AP are not perfect but of a rather good quality. Moreover, probably thanks to the small size and their sporadic emission, xAAL messages are not really affected. However, there is no formal warranty at all.

Workaround: a multicast-unicast reflector. If network equipment of a home facility has severe issues regarding multicast, the xAAL infrastructure can be slightly modified. In such a situation, one may use a *Unicast-Multicast Reflector*. This application forwards messages from a multicast channel (e.g. the xAAL bus) towards connected UDP unicast clients, and vice versa.

This is the same approach as solutions developed in the middle of 90s for the MBone project [14] except that our reflector does not intent to manage RTP RTSP RTCP protocols. It works at the UDP layer.

To use it: first, put as many as possible xAAL devices on the wired side of your AP. On plain old Ethernet wires, multicast works well. Put also the reflector on the wired side. All those components can communicate with each other via the xAAL multicast bus, as usual. Then, xAAL devices that have to go on the wireless side of your AP should be configured to use the unicast IP of the reflector instead of the multicast IP of the xAAL bus (the xAAL network stacks accept a unicast peer). This increase delays and bandwidth consumption, but this works.

Note that such a reflector is not a xAAL component. One recommends not to use it, if possible. As an alternative, select a better quality WiFi AP, or use Ethernet.

3 The xAAL Devices Behavior and xAAL Schemas

xAAL devices are described by so-called schemas. Schemas are documents specifying attributes, notifications and methods API.

Note that the idea of specifying home automation devices by schemas is inspired by the UPnP approach.[15]. Thanks to that, UPnP entities could discover new entering devices, ways to interact with them, and possibly dynamically build user interfaces (well, for this latter point UPnP schemas usually pointed to some Microsoft DLL installers or Internet Explorer plugins to be installed on the user's PC, which is not very secure).

The other advantage of a formal specification of home automation devices is for software production. Having a clear API specification may help to automatically build skeleton of code. That was the case with the XDR (External Data Representation) for Sun's RPC (Remote Procedure Call) since the middle of 90s. Now there is a renewal with OpenAPI and other RESTful API description languages: during the first age of Web technologies, communications were rarely clearly specified since developers were used to manage both sides (the server code, plus the client in the shape of JavaScript code). Nowadays, with REST architecture principles, the web client is not necessarily the end-user web browser but sometimes a third-party application. Within the Web community, this revives needs towards clear API specifications.

The home automation community does have the same needs.

[14] `https://www.researchgate.net/publication/2855653_Multicast-Unicast_Reflector`

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

Objectives:

- xAAL devices should be specified in a clear way for others to know how to interact with them, what to provide, what to expect.

- Those specifications should describe devices in a generic and in an abstract way (e.g. all thermometers should have the same behavior, even if a manufacturer adds blinking nice LEDs on it or whatever)

- Those specifications should allow extensions to describe devices more sophisticated than the basic ones of the same family, without breaking the basic behavior one may expect for devices in this family, in a backward compatibility way.

- Those specifications should follow a strict syntax to be parsed by software for automatic treatments. (e.g. Building skeleton of software, or automatic discovering of peers API).

- The specification document should be readable by a human (well, with some software programming skills).

3.1 Definition of a device

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

dev_type: 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: `dev_type:`*`class.variant`*

  The first word refers to a class of device type (e.g. lamp, switch, thermometer, power plug).

  The second word refers to a variant of 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 to 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 variants 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 claim to be of the type `"any.any"`. Devices having no dedicated attribute, method or notification may use the *concrete* type `"basic.basic"`.

- For example, the pair `"lamp.any"` means all variants of the class `"lamp"`. This is also an abstract type. Remember that the concrete type `"lamp.basic"` is provided to describe basic features common to all lamps.

- We reserve a special name: `"experimental"`, for the class, as well as for variants. This designates concrete types. Associated schema, if written, should not be distributed outside the testing platform. (e.g. When someone makes a device but has not got a standard name for its type.)

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 anywhere. A device does not request any entity to get an address.
  Concretely, addresses may be assigned:

  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 is in principle very high probability of having no collision of UUIDs. However, it is technically possible to verify that a UUID is not already used on the bus by a kind of “Gratuitous ARP”, i.e. by `is_alive` requests to ensure that no one else already uses 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 schema.

A xAAL schema is a JSON object with a specific form that must validate given CDDL rules (*Concise Data Definition Language*, RFC 8610). See Appendix A.

The schema provides a bit of semantics for a device and describes its capabilities: 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 possible methods on this device (mechanism of request replies), a list of notifications that it emits spontaneously.

- A list of *attributes*. If the value changes, this spontaneously generates a notification to the bus.

  Each attribute is defined by:
  - A unique name which identifies it within the schema;
  - A type name relating to the data model section.
- A list of *methods*. Each method is described by:
  - A unique name which identifies it within the schema;
  - A textual description;
  - A list of `"in"` arguments to be filled by peers when invoking this method;
  - A list of `"out"` arguments returned by the device to peers;
    Each argument is defined by:
      A unique name which identifies it within this method definition in the schema;
      A type name relating to the data model section.
  - 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 `"out"` arguments included in the notification; Each argument is defined by:
      A unique name which identifies it within this notification definition in the schema;

A type name relating to the data model section.

- A *data model* section, that is to say a list of *data types* definitions. For each type name:
  - A textual description;
  - The unit, if any. This unit should be one of the IANA Sensor Measurement Lists (SenML) registry.[16] [17] If none is relevant, use a unit of the International Bureau of Weights and Measures. A standard unit allows automatic processing, data computation, or at least a consistent way for rendering by HMIs;
  - A type definition in the form of CDDL rules, for extra processing (to dynamically build software skeleton, a generic HMI, etc.).

3.3 Inheritance of schemas

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

1. A basic generic schema, named `basic.basic`, common to all existing devices in the world, that everyone has to implement. (See 4.)

2. A basic schema for every class (e.g. *lamp.basic*, *thermometer.basic*, *switch.basic*, etc.). Such basic schemas inherit from the generic schema, and extend it by defining basic functionalities shared by all device variants of the corresponding class. (e.g. lamps basically can do on/off)

3. Advanced schemas for more complex devices, by extending the basic schema of the corresponding class, and by defining new functionalities. (e.g. lamps basically do on/off, but some more sophisticated lamps are dimmable, others offer RGB control, etc.). All variants of a class must inherit the basic schema of the class. E.g. `lamp.dimmer` extends `lamp.basic` which extends `basic.basic`

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

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

Definition of the extension process: Let's consider a first schema which is extended by a second schema. The later express differences with the former. The extension process produces a new schema. Extending a schema has the following meaning:

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

[16] `https://www.iana.org/assignments/senml/senml.xhtml`

[17] Note that, contrary to SenML, within the xAAL context, the percent symbol `"%"` is used for values between 0 and 100, and not for values between 0 and 1.

- The latter schema may overload the definition of some existing objects, that is to say *attributes*, *methods*, *notifications*, and *data types*. An overloading means that such an object has the same name but 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*, *data types*) plus old ones that have not been overloaded.

4 The Basic Schema

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

Attributes

- *Attributes involved in the protocol*

  `dev_type`: string, the name of the schema to which the device obeys;

  `address`: byte string of 16 bytes, a UUID (the address of the device);

- *Attributes describing the device*

  `vendor_id`: string, the name or identifier of the vendor;

  `product_id`: string, an identifier of the product assigned by the vendor;

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

  `hw_id`: any type, some hardware identifier of the device (e.g. low-level addresses of the underlying protocol, a pairing code, a serial number, or any piece of information that may help to retrieve a device within a facility for maintenance);

  `group_id`: bytestring[16], a UUID shared by all devices 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 URL of a website with extra information

  `schema`: string, the URL to download the schema file in case if the device is of a non-standard `dev_type`;

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

  `unsupported_attributes`: array of strings (hopefully empty), with names of attributes of the schema that are actually not supported by the device for some (bad) reason.

  `unsupported_methods`: array of strings (hopefully empty), with names of methods of the schema that are actually not supported by the device for some (bad) reason.

  `unsupported_notifications`: array of strings (hopefully empty), with names of notifications of the schema that are actually not supported by the device for some (bad) reason.

The attributes of the `basic.basic` schema are mostly considered as *internal*, or dedicated to the description of the device, and are not likely to change along the life of the device. Unlike attributes of extending schemas, they must not be involved in the `attributes_change` notification nor in the `get_attributes` method described below, but via the `get_description` method.

Notifications

- `alive`: emitted when starting the device and then periodically emitted at a rate left to the discretion of the device. The notification message may contain a `timeout` parameter indicating to others when the next `alive` should arise.
- `attributes_change`: 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 may not be. This is normal.

  However, the generic schema defines this method with no attributes, since the default attributes defined above are relating to the description of the device, and do not characterize the real-time operation of a specific feature. Schemas extending `basic.basic` may overload the `attributes_change` notification according to the extra attributes introduced by the extending schema.
- `error`: issued when the device detects a major error or a failure.
  The notification may contain a `description` (a textual description of the error), and a `code` (the numeric code of error).

  Remember that xAAL is of the best-effort philosophy. Therefore, a wrong method invocation does not issue `error` notifications. (The called device does its best with what it received, and possibly change its attributes accordingly, but there is no one-to-one dialogue to explain mistakes to the caller.) Errors are issued only on major failure of the device.

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

Note: notification messages should be addressed to all.

Methods

- `is_alive`

  `dev_types` (in): array of *dev_type* 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-mentioned items, e.g. `["lamp.dimmer", "shutter.basic"]` if we are interested in that, etc.

  The target field of a `is_alive` request is often empty, i.e. broadcasted, but not necessary.

  A `is_alive` 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 extending schemas.

- `get_description`

  `vendor_id`, `product_id`, `version`, `hw_id`, `group_id`, `url`, `info`, `unsupported_attributes`, `unsupported_methods`, `unsupported_notifications` (out): see the meaning of the above list of attributes.

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

- `get_attributes`

  `attributes` (in): array of string, the name of 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 extending 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 extending schemas.

4.1 Composite devices

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

4.1.1 Gateways

A gateway is a piece of software and hardware 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 `dev_type`. They are called *embedded devices*.

Moreover, the gateway should present itself as an additional xAAL device, with its own address and a `dev_type:"gateway.basic"`, or any relevant 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).

Even if it is not recommended, 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 functionalities, 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 grouped. Within their description they have the same `group_id` value (a UUID in fact). They are called *grouped devices*.

A group just says that some xAAL devices are physically together, and only this. (Which is, however, a highly 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 functional 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 provide services on top of other devices. For instance, consider the case of the "smart control multi-plug socket". Each of the plug may be smart because it provides two services. First 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 other devices. It measures the power of something: possibly only one plug in the case of the previously mentioned "smart plug", or possibly a large set of several devices, a part of the power supplying network of the facility (e.g. all equipment 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 current version of the schema definition). This has no relation to the `group_id` introduced in the previous section.

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

4.2 Changes in xAAL Schemas

Schemas are written in JSON for now. This is still the case with xAAL version 0.7. The new version proposes to replace JSON by CBOR for messages, not for schemas. There are several reasons for this:

- First, JSON cover the need. As indicated, a schema tells the list of attributes, notifications and methods of the device, plus some extra information to describe the device (vendor, model, etc.). The expressiveness of JSON is enough for this.

- Then, a schema should be a document readable by a human (and possibly editable), since this brings some semantics that should make sense to users and developers. A binary document, such as CBOR, is only understandable by software. This is not desirable in the context of xAAL schemas.

Unfortunately, mixing JSON and CBOR may lead to uncomfortable situations that are investigated bellow.

4.2.1 Specifying types of attributes and parameters

Schemas specify the types of the attributes and parameters of devices.

To do this, for each attribute or parameter description, the former format of xAAL schemas included fragments of Json-Schema. [18] [19]. A Json-Schema is a kind of grammar that specifies the form that a JSON data must fit. Compares to alternatives (e.g. Json Content Rules (JCR)[20], JSON Constrained Notation (JSCN)[21], etc.), Json-Schema uses a pure JSON syntax. A Json-Schema specification is written in JSON. Thanks to this, the type of xAAL attributes and parameters (in Json-Schema) could be directly included in the xAAL schema.

The key point is that this describes a data presentation for data presented in JSON. But xAAL version 0.7 introduces messages in CBOR. Values of attributes and parameters are carried by xAAL messages, therefore encoded in CBOR. Json-Schema is inappropriate for describing a CBOR data model. The dedicate data-model language for CBOR is CDDL (*Concise Data Definition Language*, RFC 8610).

As a consequence, xAAL schema version 0.7 now describes the types of the attributes and parameters of devices by the means of CDDL rules.

4.2.2 Gathering Types Definitions in a Data Model Section

With the former versions of xAAL schemas, types of data used by attributes methods and notifications were specified 'in-line'. That is to say, the schema describes the first attribute by giving its name followed its type definition, then do the same for the next attribute. Then it describes a first method by giving its name and its list of parameters; and for each of these parameters it gives its name and its type definition.

As a consequence a schema may include several copy paste of data type definition. Typically, attributes definitions were duplicated into the standard `get_attributes()` method and into the `attributes_change()` notification. Those copy pastes made the schemas hard to read for a human.

Therefore, now xAAL schemas give a type name to each attribute and parameter, and gather type definition into a dedicated section called `data model`. This data model section is subject to extension mechanisms as for attributes, methods, and notifications.

4.2.3 On the Use of CBOR Tags

In CBOR, a data item can be enclosed by a tag to give it additional semantics. Such a tag gives an indication of how data should be interpreted. For instance, tag 0 says that a text string has to be interpreted as a date, tag 1 is for dates since POSIX Epoch (1970-01-01), tag 32 says that a text string is an URI, tag 37 says that a binary string is a UUID, etc. A small set of tags is part of the standard (RFC 7049); additional tags are defined by a dedicated IANA registry[22].

There are pro and cons on using CBOR tags. Tags values are carried with data, this consumes space. On the other hand, receiver gets in-line semantic indication to interpret data without having to check any side specification documents. This is particularly useful when CBOR is used for general (de)serialization in a programming environment: many CBOR libraries widely use tags to provide automatic casting of data into dedicated specific programming object types. Conversely, tags are somehow redundant with information provided within xAAL schemas.

Well, on the basis of these considerations, it has been settled that tags were not used for low-layers of the xAAL communication protocol (that is to say the Security Layer and the Application Layer as defined below). Indeed, these layers are of a small and rigid shape,

[18] `http://json-schema.org/latest/json-schema-core.html`
[19] `http://json-schema.org/latest/json-schema-validation.html`
[20] `https://tools.ietf.org/html/draft-newton-json-content-rules-08`
[21] `https://tools.ietf.org/html/draft-miller-json-constrained-notation-00`
[22] `https://www.iana.org/assignments/cbor-tags/cbor-tags.xhtml`

without any room for choice or interpretation of data. These data items are relating to the xAAL communication protocol itself.

However, data items relating to attributes, methods and notifications of xAAL devices are of more complex shapes. Therefore, xAAL schemas may use CBOR tags in their data type definitions; for instance, UUID, date, URI. Note that, if tags are specified in a xAAL schema, then they are mandatory into the *body* part of the Application Layer for corresponding actions. Tags management may lead to optional behaviors by decoders, but they are not optional in data items written in messages once they have been specified by a CDDL rule (as in xAAL schemas).

4.2.4 Guidelines for Writing Schemas

Writing Convention. Former versions of xAAL used the *Calmel Case*[23] writing convention for naming the elements of xAAL schemas. This choice was consistent with habits of JSON (and JavaScript) communities.

The new version of xAAL move from JSON to CBOR, leaving JSON communities. Main developers are closer to Python and C communities. One therefore decided to move to the *snake_case*[24] writing convention.

Action Verbs for Methods. A key to get an engaging API is to select a wording for elements which carry an intuitive and well-known meaning. This makes the talent of the designer. There is no universal rule. However, this sounds reasonable selecting action verbs to name methods, e.g. `turn_on()` and not just `on()`.

Similarly, programmers usually like *accessors* to *attributes*, i.e. `set_XXX() get_XXX()`. (Note that the latter may be redundant with the generic `get_attributes()` method of the `basic.basic` schema.)

Describes Functionalities from the Other Point of view. Amazon Alexa[25] and Google Home[26] propose means to design its own device by composing elementary so-called *capabilities* (Alexa) or *traits* (Google Home). E.g. for a color-changing lamp this means *power on-off brightness HSV color* etc.

Such APIs are oriented towards the description of products. For instance, an innovative startup may sell an all-in-one connected fan with an oscillating feature, a colored dimmable lamp, a camera and a thermometer. The API of such a device can be written by composing corresponding capabilities or traits. This is the responsibility of the client application to seek devices and to turn on the lamp of this fan if the user wants light.

Moreover, this may require sub-addressing. For instance, a weather station may provide both temperature and humidity measures, and be exposed with corresponding capabilities. However, if this weather station has two thermometers, the temperature capability has to be instantiated twice, with a sub-addressing to retrieve each of the two thermometers into the device. From the client application point of view, these two thermometers are not retrieved like other thermometers of the house.

As contrary, xAAL proposes schemas oriented towards the description of functionalities from the client point of view. Concretely, a mult-features product such as a weather station is split into several devices (thermometers, hygrometers, etc.). Each feature is implemented by a dedicated xAAL device. Such xAAL devices may then be grouped thanks to a *group identifier*.

[23]`https://en.wikipedia.org/wiki/Camel_case`

[24]`https://en.wikipedia.org/wiki/Snake_case`

[25]`https://developer.amazon.com/fr/docs/smarthome/get-started-with-device-templates.html`

[26]`https://developers.google.com/assistant/smarthome/guides/`

Client application may retrieve all thermometers in the same way whatever they are single thermometer or embedded into an all-in-one product.

5 The xAAL Communication Protocol

xAAL is a distributed system. Participating entities need to communicate with each other. The xAAL communicating protocol has been designed with following objectives in mind.

Objectives:

- The communicating channel should allow automatic discovering, allowing a new entity to join the group without much job to do, and allowing other entities to know that a new entity arrives.
- The communication stack should be simple and lightweight to implement in constrained devices, in a stateless manner, without having to maintain alive several communication channels in parallel.
- The communication should be lightweight in terms of messages size and number, close to the background noise of a usual home network.
- The communication should address security considerations, regarding users' privacy, or threat to get control.
- The communication should be simple to configure for the end user.

xAAL messages are carried by UDP multicast packets. xAAL messages are made of two layers: (i) a Security Layer with some clear fields mandatory for transport to receivers, followed by a ciphered payload; and (ii) an Application Layer which consists of the decrypted payload and containing all information for participating applications.

5.1 The Security Layer

The data of the Security Layer is the payload of the UDP multicast messages of the xAAL bus. This is a CBOR array of 5 fields:

- [0]:*version* - Unsigned int, of the value `7`. The version of the protocol. Other values should be rejected, the message is ignored.
- [1]:*seconds* - Unsigned int. First half of the timestamp. The number of seconds since the Epoch (1970-01-01 00:00:00 +0000 UTC).
- [2]:*microseconds* - Unsigned int. Second half of the timestamp. The number of microseconds since the beginning of above second.
- [3]:*targets* - A definite byte string build as the CBOR serialization of the array of destination addresses for the message. Note that xAAL device addresses are UUID (see Sec.3), here encoded as definite bytestring[16] without tags. A xAAL device receiving a message should accept it if its own xAAL address is present in the array of targets. An empty target array means a broadcast message. A target field containing an empty byte string is not allowed (the message should be ignored).

```
security_layer = [
  version : 7,
  timestamp_sec: uint,
  timestamp_usec: uint,
  targets : bytes .cbor ([ * (bstr .size 16) ]),
  payload : bstr
]
```

Figure 2: CDDL specification of the xAAL Security Layer

- [4]:*payload* - A definite byte string which is the ciphered *Application Layer* according to version 0.5-r2 principles (Poly1305/Chacha20, a symmetric key, a binary nonce build on the timestamp of messages, an acceptance window for the timestamp of messages, the target field covered by the cryptographic signature).

If a message includes other fields in addition to the above mandatory ones, the message may be accepted, but the extra fields must be ignored.
The above-described CBOR items must have no CBOR tags.
Figure 2 gives the CDDL specification of the xAAL Security Layer.

5.2 The Application Layer

The Application Layer is a CBOR array of size 4 or 5, depending if there is a *body* field or not:

- [0]:*source* - A definite bytestring[16]; the xAAL address (UUID) of the sender of the message.
- [1]:*dev_type* - A definite string; the schema name of the sender (The pair *classe.variant.*)
- [2]:*msg_type* - An usigned int of value: `0`:notify, `1`:request, `2`:reply. Other values must be rejected, the message is ignored.
- [3]:*action* - A definite 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.
- [5]:*body* - An optional field. If present, it must be a map. Keys of this map are definite strings, the names of parameters associated with the corresponding action according to the schema and their value. Depending on data model specified in the schema, values may have CBOR tags. Note that the body may be absent if the schema does not specify parameters for the corresponding action. Multiple identical keys are not allowed, the message should be ignored.

The above-described CBOR items must have no CBOR tags. However, value items within the *body* may be tagged, according to the corresponding schema.
Figure 3 gives the CDDL specification of the xAAL Application Layer.

5.3 The Ciphering Method

The security of the xAAL bus is ensured by:

- A symmetric key, pre-shared into all participating devices;

```
application_layer = [
  source : bstr .size 16,
  dev_type : tstr .regexp "[a-zA-Z][a-zA-Z0-9_-]*\\.[a-zA-Z][a-zA-Z0-9_-]*",
  msg_type : 0..2,
  action : tstr
  ? body : { * ( tstr => any ) }
]
```

Figure 3: CDDL specification of the xAAL Application Layer

- Poly1305/Chacha20 as the only cryptographic algorithm, and used according to RFC 7905 recommendations (i.e. with a 96 bits nonce and a 256 bits key);
- A binary nonce build as a timestamp since the Epoch (seconds (64 bits) + microseconds (32 bits));
- An acceptance window for the timestamp of messages;
- The list of targets in clear, but covered by the signature;
- A Security Layer in CBOR, as described above;
- An Application Layer in CBOR, as described above;
- The Application Layer is CBOR serialized to produce binary data, ciphered into a binary buffer which is placed into the Security Layer as a CBOR definite byte string.

Notes:

- The target field of the Security Layer is a CBOR definite byte string. This is not an array of UUIDs, this is the CBOR serialization of an array of UUIDs. This byte string may be 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).

Recommendations:

- To build the cryptographic key from a passphrase:
  The Poly1305/Chacha20 algorithm uses 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 versions are needed in a facility, it is recommended to use different xAAL buses (UDP ports).

Example. Figure 4 gives an example of a xAAL message (the Security Layer): a message to the target `8BCC7ED2-A6AC-4D83-A723-6ED3B168C51F`, with a ciphered payload.

Figure 5 shows the decoded payload (the Application Layer): the sender is the device `1ADFFD0D-67A6-415D-BC11-74C9CCB32EE9`, of type `thermometer.basic`, which replies about its attribute `temperature:18.0`.

The examples use the textual CBOR Diagnostic Notation.

```
[ 7,
  1572609657,
  519551,
  h'9F508BCC7ED2A6AC4D83A7236ED3B168C51FFF',
  h'BE67602B9DFC0EDA2CD59FA875109954190D11159C6D67B24CA50201EB09
    84FE782F8BCB4259CD38701027184C5959F080DAD013C7A584F44F7EAE52
    BAA212086AC467C6461AC866F5ECC13C2C5EFA4CD71BDE7987CE68D1E8F0' ]
```

Figure 4: Example of a xAAL message (*Security Layer*)

```
[ h'1ADFFD0D67A6415DBC1174C9CCB32EE9',
  "thermometer.basic",
  2,
  "get_attributes",
  {"temperature": 18.0} ]
```

Figure 5: The decrypted payload of a xAAL message (*Application Layer*)

5.4 Changes in the xAAL Communication Protocol

As in version 0.5-r2, the xAAL messages in version 0.7 are carried on an IP multicast bus (IPv4 or IPv6). Messages are still made of two layers: a *Security Layer* which encapsulates an *Application Layer*. Data fields, the associated semantic and behaviors are the same.

The novelty is that data are now serialized in CBOR (RFC 7049[27], and revision draft 7049bis [28]). For instance, instead of JSON objects, one may use CBOR map. Also, instead of using the textual representation of UUID (RFC 4122), one now uses its binary representation as a CBOR definite byte string of 16 bytes.

JSON was a fine message format for prototyping xAAL. As a textual format, this greatly has helped developers to diagnostic issues. Now xAAL has grown in maturity. Moreover, since xAAL version 0.5 have brought ciphering of messages, keeping a textual format have no more interest. CBOR brings compactness (xAAL messages are now about three times smaller), but its main advantage since xAAL has ciphered data is that CBOR can manage byte string properly (i.e. without base64 encoding onto a textual string, as JSON does).

[27] `https://tools.ietf.org/html/rfc7049`

[28] `https://tools.ietf.org/html/draft-ietf-cbor-7049bis-07`

CBOR has many other concurrent binary coding formats. Appendix E of RFC 7049 provides a brief comparison. Not surprisingly, this technical comparison is positive for CBOR. Another argument in favor of CBOR is that CBOR is a published standard.

Maps vs. Arrays. JSON is deeply linked to the JavaScript programming language and inherits of some of its concepts: object-oriented, dynamic typing, etc. Within the JSON community, data is frequently organized into so-called objects data structures (also called maps, associative arrays, or dictionaries, depending on programming language vocabulary). So that, each piece of data is associated with a name, which, if well chosen, may make sense in the head of other developers of the community, bringing a kind of intuitive semantic. This is one of the reasons why JSON is said a *self-describing* format. (The other reason is about the serialization/serialization point of view: thanks to punctuation marks and syntactic salt, the format of strings carries enough information to decode bytes into data types of usual programming languages without any other specification document.)

Within the CBOR community, habits slightly differ. Maybe due to its binary basement, developers prefer array structures, used as *struct* in C programming language. However, associative maps are also available: it is still useful to associate well-chosen names to data items. Moreover, CBOR provides another means to give semantic to data in a more formal way: data items may be tagged with a number which refers either a standardized tag[29] or an application-specific tag. (Note that CBOR is also said a *self-describing* format from the serialization/serialization point of view: the format carries enough information to decode bytes into data types of usual programming languages without any other specification document.)

In conclusion, xAAL version 0.7 chose to use array structures for its Security Layer and its Application Layer. There is no more keywords associated with data items for these two layers. Data items are provided in a strict order, in a strict way. Messages are even shorter, and this brings simplicity for receivers, without so much complexity for senders. Moreover, there is no CBOR Tags in these two low-level layers. xAAL nodes already must know those layers format, there is no need to carry tags for the meaning of fields.

Definite vs. Indefinite Data Structures. CBOR data containers (maps, array, strings, byte strings) are proposed in two variants: definite (with a predefined size), and indefinite (the container is open at the beginning, items follow in sequence, the container is closed on reception of a special end mark).

Both variants have pros and cons, with a balance between the complexity of the sender versus the complexity of the receiver. Indefinite data containers are fine for streaming data on the fly. This may also ease the job of the sender if it can't easily predetermine the size of data to send, without making two passes. On the other hand, indefinite data containers may bring more complexity for the job of the receiver which may manage this either by implementing streaming mechanisms (e.g. with specific call-back functions), or by allocating extra buffers. The situation is even more uncomfortable with indefinite strings and byte strings for which chunks may split data at any position.

So, to simplify the job of xAAL receiver, indefinite strings and byte strings are prohibited within the Security Layer and the Application Layer. A priory this choice does not increase so much the complexity in the sender for building messages since this two layer are of a well-known format.

[29] `https://www.iana.org/assignments/cbor-tags/cbor-tags.xhtml`

5.5 Measures on the Protocol

xAAL is deployed in our laboratory[30].

- There are 121 xAAL devices.
- An average of 4.82 messages per second, that is to say about one message every 25 seconds for a device.
- An average of 378 bytes pear message with xAAL 0.5, and an average of 128 bytes with xAAL 0.7, in other words: an average bandwidth of 5 bytes per seconds for a device.

Note that moving from xAAL 0.5 to xAAL 0.7 just changes message size, there is no impact on the global behavior nor on the number of messages.

Also note that the number of messages is impacted by timeout value for *alive* notifications and for *get_description* requests (usually performed by HMI applications). However, thanks to bus communication principles, a reply message to an application is used by other applications. Having multiple applications on the bus does not increase so much the number of messages.

5.6 Security Considerations

5.6.1 Local Communications

By design, home automation protocols over IP do benefit of the security offered by the home network, which is made of physical wires or of WiFi networks with WPA security. It is therefore at least as hard to get the control of a home automation from outside as to be able to join the home network. A protocol with messages in clear could have been fine.

However, main threats may probably not come from outside. Nowadays, end users (happy eye balls, RFC 8305) are used to bring many smart devices at home, to install many promising applications, embedding times-to-times suspicious code. If the home automation protocol does not have any security code, a malicious application could take control of it (e.g. deactivate the alarm, unlock the door for thieves). More probably, such a third part application could spy the home automation activity, monitor users' habits and preferences to feed ads and recommendation systems for consumers.

Such a situation is not desirable, or else, at least, with a clear and informed opinion of the end user and its agreement... This is why a certain level of security should be added to home automation systems. Messages should not be in clear, nor ciphered with a hard-coded static key; secret keys must differ from one home to another; secret keys must not be preconfigured by companies, but chosen by the end user.

In xAAL version 0.5 and Later, messages are ciphered using Poly1305/Chacha20 algorithm. According to experts, it is at least as much stronger than others (e.g. AES). It is now in the cipher suites for TLS (RFC 7905).[31] Due to its communicating bus principle, xAAL devices can't negotiate cryptographic parameters prior to sending useful data as with client-server communication principles (selecting a cipher algorithm, a nonce, Diffie-Hellman keys, etc.). xAAL messages must be self-content, also regarding security parameters. xAAL uses a pre-shared symmetric key. This is the only parameter that the end user has to configure. The nonce is a timestamp with a small acceptance window to allow clock drifts.

The security proposed by xAAL is a compromise. Of course, this can be criticized. But a higher security level will break simplicity and efficiency of xAAL. The security proposed by

[30]`https://www.imt-atlantique.fr/fr/recherche-et-innovation/plateformes-de-recherche/experiment-haal`

[31]`http://www.rfc-editor.org/rfc/rfc7905.txt`

xAAL is high enough to protect data carried by home automation communications, and fits requirements to address threats model one considers.

5.6.2 Cloud Communication

Within the home automation context, users are at home, and services are rendered at home by the home. One may wonder where to locate and execute service: at home or in the clouds. xAAL, and many others, propose to locate home automation services at home. This is a first step to address security and privacy issues. However, nowadays, consumers accept cloud solutions. This implies to trust involved companies, to give them control of our home, to give them a view of our activities at home, to believe in their ethic for today and for the future. Accepting this is a cost for the consumer. The counterpart is the promise to get a simple-to-use service.

Mobile Clients to Cloud Servers. The first promise is simplicity. Generally, 'smart devices' wording has to be read as 'devices having Internet access and connected to servers of the company in the clouds'. The consumer buy and install home automation devices (smart plugs, smart outlets, smart lamps, etc.). Those devices may embed a WiFi network interface, or may be connected via a dedicated radio protocol to a hub/gateway/box also present at home, which has a WiFi network interface and sometime an Ethernet one. There is a persistent connection between devices at home and external computers of the company: usually devices forward sensor measures and events in real time, and can be remotely controlled from those external computers. From its side, the consumer uses a mobile application (a dedicated one or a Web page) which is also connected to those external computers of the company. By this means, the user is informed of events at home and can send commands to its devices. Companies usually propose such services for free, or free for the first years. This process conforms to nowadays usual design of mobile applications: user applications are *clients* of *cloud servers*, servers that are retrieved thanks to a well-known DNS name (Domain Name System, the means to translate names into Internet addresses). This process is transparent to the user. In addition, the user has the same level of service whatever he or she is at home near devices, or outside. Controlling the house from outside by any member of the family is often presented as an advantage despite there is no clear demands from consumers.

Mobile applications are rarely designed to reach local servers. Indeed, few end-user services assume that there are servers of anything at home; everything is supposed to be in the clouds. One of the exceptions is for *multimedia drives* which allows users to store at home photos videos and music. Note that such services are now also competed by cloud services. These multimedia drives are based on protocols such as DLNA that offers discovering features: user applications can retrieve these drives on the local network and play the contents. xAAL also has a discovering feature by itself, which may be used by a xAAL mobile application. Or else, if the xAAL user interface / control point is a Web service in a box on the home network, it can register its well-known name to the local mDNS server thanks to the Home Networking Control Protocol (HNCP, RFC 7788[32] by the Homenet IETF working group). This provides a generic discovering mechanism for local services. The well-known name of such a xAAL box can then be retrieved locally by mobile applications, as usual, but at home.

Interoperability in the Clouds. Another promise is the interoperability in the clouds. Indeed, many home automation vendors propose to interconnect their solution with popular voice assistants (Google Home, Amazon Alexa, Apple Siri). These devices called 'voice assistants', 'smart speakers', 'connected speakers', but rarely 'connected microphones', are cloud services

[32] `http://www.rfc-editor.org/rfc/rfc7788.txt`

by themselves: the speech recognition (more or less automatic[33]) is mostly performed by external computers, not at home by the device itself. Thanks to programming API, but also to commercial agreements between companies, cloud computers of voice assistants are connected with cloud computers of home automation vendors. Such connected microphone records sounds at home all the time; when a sound matches a given keyword, the sound stream (or chunks) is pushed to computers of the company of the connected microphone; the sentence is analyzed by a speech-to-text process; if keywords associated with home automation are recognized then the text is pushed to computers of the home automation company; if some home automation commands are recognized then these computers send commands to devices at home to which they are permanently connected. As a result, the end user may tell requests to its house and see it in action, very impressive. This kind of universal interface is somehow a kind of interoperability in the case there are several home automation technologies at home. In fact, there is no real inter-operation between them, just a single user interface. For now, this is performed in the cloud, this is why such solutions are called 'interoperability in the cloud'. In fact, this could also be performed locally at home. Speech-to-text is a complex process, and is a precious added value for companies which manage it. On-line (cloud) solutions seems to be of better quality than off-line (local) solutions. But, depending on the user's real needs and feeling, local solutions are good enough for simple interaction (recognize keywords), as for home automation services. Moreover, there is no convincing evidence that speech-to-text algorithms executed nowadays by cloud computers could not be performed by local devices. Companies promise continuous improvements of their secret algorithms by collecting in the clouds voice and sentences of their consumers, probably by unsupervised learning algorithms, without ground-proof and labeled data.

To sum up, performing home automation 'in the clouds' raise many safety, security, and ethical issues. There is no real technical obstacles to perform this completely locally, at home, without any leak of information in the clouds or to third parties. xAAL has been designed with this idea in mind.

5.7 Alternative Home Automation Protocols over IP

The xAAL communication protocol was inspired by UPnP, xAP, xPL and others.

5.7.1 UPnP

Universal Plug and Play (UPnP) brings advanced concepts and designs for device-to device communications on, for instance, home networks. Functions are well defined, devices are formally described by schemas, the system allows discovering, etc. The main criticism is about its heaviness: it is based on HTTP, SSD, XML, SOAP, ZeroConf. Participating nodes often need all of these layers, each action requires several TCP connections. Nowadays, UPnP remain in use for smart TVs to display videos from home media servers, via the UPnP DLNA sub-profile (Digital Living Network Alliance)[34]. The home automation UPnP profile has never really been used in products, except for proof of concept demos.

5.7.2 xAP and xPL

The eXtensible Automation Protocol (xAP)[35] is a pragmatic ad hoc simple textual home-automation protocol which uses IPv4 broadcast for message passing between senders and re-

[33] `https://www.bbc.com/news/technology-49502292`
[34] `https://www.dlna.org/`
[35] `https://www.xapautomation.org/`

ceivers. xAP has also the notion of schemas, which defines messages format classes, and somehow device types. Main criticism is about its ad hoc nature: schemas are not so formal, and it is concretely rather difficult to interoperate with a device that one has not coded by oneself. The other difficulty is about its addressing principles by 'logical name', 'device id', 'instance id', 'sub-addressing endpoints', which are commonly replaced by the wildcard mark `'*'`. The last surprising point is about the architectures: hosts are communicating with each other by IPv4 broadcast, but software on the same host are locally connected via a UDP unicast socket to a 'hub' component in this host which is in charge to forward those UDP packets to other hosts by broadcast.

The xPL protocol[36] is a fork of the xAP protocol. It defines itself as a 'glue' to tie together home automation technologies. It tried to do things in a more formal way: the UDP port in use was claimed and officially registered by IANA, schemas were written in XML, unfortunately without clarifying behavior of devices. Finally, it suffers from the same criticisms as xAP.

xAP and xPL are ten year old protocols and are now closed projects. However, they have participated founding basements of home automation protocols over IP in terms of need expression and in terms of solutions basis for interoperability: distributed systems, device-to-device communications, schemas for devices' definition, lightweight communication stack...

5.7.3 MQTT-like Protocols

There are still recent proposals in the open-source community for simple ad hoc home automation protocols using the home IP network, such as MQTT-UDP[37].

The Message Queuing Telemetry Transport (MQTT[38]) is publish/subscribe messaging transport protocol designed for machine-to-machine (M2M) communication: sending commands to a set of devices, collecting data from sensors (e.g. measuring particulates and dust of town), etc. MQTT is a one-to-many communication system, based on a *star* architecture: a central node (called broker) is in charge of forwarding published messages towards subscribers. Messages are organized by topics, topics are somehow the addressing system.

The idea of MQTT-UDP is to reuse MQTT principles in a lightweight manner by replacing the broker by UDP IPv4 broadcast. The result is an ad hoc pragmatic transport protocol for smart home applications. The functional definition of implied components and specifications of their API is out of the scope of MQTT-UDP.

5.7.4 Aqara

Aqara is home automation company of Lumi United Technology.[39] Aqara products are generally sold by Xiaomi. Some product variants may be purchased directly to Aqara, with almost same shape and functionalities.

Technically, Aqara devices are connected to an Aqara Hub at home via the Zigbee radio protocol.[40]. This hub is a small home automation gateway with a WiFi interface for Internet connection, and which is in charge of forwarding home devices messages to and from Aqara cloud servers via a non-published but verbose UDP protocol. Cloud services allow consumers to get control of their home devices via a mobile application, retrieve monitored activities, and to set up automation scripting. Some automation (e.g. alarms) are executed locally by the hub, other more complex are executed by cloud services. Aqara cloud platform provides a

[36]`http://xplproject.org.uk/`

[37]`https://mqtt-udp.readthedocs.io/`

[38]`http://mqtt.org/`

[39]`https://www.aqara.com/en/home.html`

[40]`https://www.aqara.com/eu/smart_home_hub.html`

documented REST API for third-party applications (e.g. for compatibility with Google Home, Alexa, and others), or for open-source applications.[41]

Some models of Aqara Hubs may be configured to activate a home network interface in addition to the cloud connection.[42] This allows user applications controlling home devices directly, without using cloud services. This local communication uses two UDP sockets: multicast and unicast. The multicast channel is used: (i) for discovering (for applications to discover hubs present on the home network), (ii) for heartbeat messages (the hubs and their sub-devices regularly announce that they are alive, plus a battery level, plus a changing security token), and (iii) to sends report messages (i.e. notifications of events and sensors changes, from the hubs to users' applications). The UDP unicast channel is for command control from users' applications to the hubs. These messages indicates: (i) the sub-device id (which seems to be stable and in relation to some Zigbee address of the physical device), (ii) the *model* name of the sub-device (this model name is specific to the Aqara or to the Xiaomi products), and (ii) a set of *parameters* relating to the indicated model name. These parameters are documented on Web pages of Aqara.

Aqara messages are in JSON format, in clear text. Note that command messages include a security key which is computed by concatenating the gateway key with the device token and then ciphering this with the AES-CBC 128 algorithm, initialized with a predefined vector. The gateway key is randomly generated by the Aqara cloud service for the first time the gateway is installed by the consumer. This key is stable and is supposed to be secret. Mobile applications get it from the cloud after user login. Devices tokens are publicly available in heartbeat messages and change every 10 seconds. Aqara messages are not ciphered nor signed, this security key just brings a certain level of confidence in the fact that command messages are sent by a regular application.

To sum up, the Aqara local protocol has some pro and cons. Having two sockets makes the communication protocol more cumbersome. On the other hand, providing some functionalities on unicast is a pragmatic workaround for weakness of WiFi regarding multicast. Devices are described by a model name, with an effort to document it. Unfortunately, there is no attempt to organize those devices' descriptions in terms of family of devices with shared functions or so. For now, there are about twenty device models. One may wonder about the evolution of the products list. The security of the local communication protocol is rather poor, but at least it exists, which is a rather rare in this domain. But the main criticism is about the privacy concerns caused the cloud services.

5.7.5 Tuya

Tuya is a business-to-business IoT company[43]. It does not sell home automation by itself (or very few), but provides services for other companies to propose their own devices to consumers. Tuya has about 75 major companies, and about 93 000 small or medium companies as clients, over 200 countries, for about 90 000 compatible products. [44] [45] Tuya is one of the preferred technical solutions for low-cost IoT devices. An extremely large number of home automation and IoT products use the Tuya proprietary technology as backend. Everything is 'Tuya-compatible'. Tuya therefore present itself as an interoperability solution. Well, a dominance industrial position can hardly be seen as a real interoperability solution.

The service provided by Tuya is in three parts. The heart of its service is a cloud platform: compatible devices can connect to it, send data to it, and receive commands from it. The

[41]`http://docs.opencloud.aqara.com/en/development/cloud-development/`
[42]`http://docs.opencloud.aqara.com/en/development/gateway-LAN-communication/`
[43]`https://en.tuya.com/`
[44]`https://docs.tuya.com/en/iot/introduction-of-tuya/introduction-of-tuya`
[45]`https://www.journaldunet.com/ebusiness/internet-mobile/1440892-tuya-smart-le-geant-derriere-100-mil`

second part is a generic mobile application for consumers to get control of their device via the Tuya platform (the *Tuya Smart Life App*), as well as REST API for companies to develop their specific mobile application on top the Tuya platform. The third part of Tuya services is to provide a code for developers of companies, to be embedded in their devices. For this, developers are invited to enroll the Tuya program, to describe their device (functions, data type, commands, etc.). In return, Tuya provides keys to access the platform and piece of programming code to be embedded in a dedicated Tuya electronic chips with has integrated WiFi module. Companies develop their product around the chip and the code provided by Tuya. The code opens a TCP socket from the device at home towards the Tuya cloud platform. Users data goes on the Tuya cloud platform. The Tuya cloud platform control Tuya devices at home, send commands, update firmware remotely, etc.

Tuya is a very popular solution with many devices sold. As a consequence, it has been widely studied, via reverse-engineering processes due to the lack of open documentation, either for checking its security[46][47], either to develop alternative compatible applications[48].

Tuya devices communicate with their servers via HTTP (or HTTPS, depending on the Tuya version) for configuration and key exchange. Then command control of devices is performed with MQTT, whose messages may be encrypted using AES-128 (depending on the Tuya version). Security is performed by a symmetrical key which is configured by the Tuya company and pushed to users' devices and application. In parallel to cloud communications, Tuya devices accept local communications with a similar shape of protocol, allowing mobile applications of the user to control devices locally without a go-and-back in the clouds. Messages are in JSON format. Parameters are passed as a map of registers named `"1"` `"2"`, etc. whose meaning is not specified and corresponds somehow to what the developer has set up the device functions. There is no formal specification of devices. Tuya is widely deployed, but suffer from safety, security and privacy issues.

5.7.6 CoAP

The Constrained Application Protocol (CoAP[49], RFC 7252[50]) is a popular machine-to-machine protocol. It is based on the REST[51] model, adapted for small devices. It is a client-server architecture (i.e. many-to-one), and use HTTP-like methods such as GET, PUT, POST, and DELETE. It is resource-oriented. For instance, a thermometer has the temperature value as a resource. The server address and the resource name compose the URI managed by CoAP. CoAP is much lightweight than the HTTP protocol. It is based on UDP. The payload is in XML, JSON, or CBOR format. Packets may be ciphered with DTLS[52]. CoAP support multicast (without ciphering) for one-to-many communications, for instance to send a command towards several devices. CoAP was designed for IoT use cases, to manage devices in large area networks.

CoAP could also have been a good candidate for home automation and for the xAAL system. Unfortunately, its client-server approach does not fit the many-to-many communication needs. xAAL proposes a full distributed architecture for home automation. Well, CoAP could have been diverted to do the same: if all nodes are both client and server, one gets a multi-server architecture which is somehow a distributed architecture. With a bus communication, a thermometer post one message when its temperature changes. With a client-server communi-

[46]https://www.heise.de/newsticker/meldung/Smart-Home-Hack-Tuya-veroeffentlicht-Sicherheitsupdate-4292
html

[47]https://techsecurity.news/tag/tuya/

[48]https://github.com/codetheweb/tuyapi

[49]http://coap.technology/

[50]https://tools.ietf.org/html/rfc7252

[51]https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[52]https://tools.ietf.org/html/rfc5238

cation, the thermometer expects request messages then send reply messages, or posts messages to previously subscribed clients (that have to be kept in memory). For home automation, a bus communication is more efficient than client-server communications even if this could work.

The main obstacle in using CoAP is for the definition of a security mechanism for many-to-many communications. CoAP uses DTLS (TLS for datagrams) which require a preliminary negotiation stage prior to send data: to agree on a ciphering algorithm, session keys, nonce, etc. This is feasible for client-server communication, not for bus communications: each one post its messages on the bus without any preliminary knowledge of others. This is why a dedicated communication protocol has been designed within the xAAL system.

5.7.7 OpenHAB

The Open Home Automation Bus (OpenHAB[53]), ported by the OpenHab Fundation[54] is a popular open-source software solution for home automation. This is a central solution (e.g. to be installed in a home automation 'box'). This centralizes in a single software several home automation functions: user interface, scripting features for automation, and gateways towards many physical devices and technologies. To address interoperability issues, this software supports multiple vendor specific home automation systems by the means of add-ons (named *bindings* in the OpenHAB vocabulary). For now, add-ons are developed by the open-source community. Home automation companies are then invited to provide their add-ons for their devices (similar than PC drivers for peripherals). Due to this technical detail, this is still the OpenHAB name and logo which is displayed on the box, not the ones of home automation companies. This can be a real obstacle to the adoption of this approach.

The software is developed with the Java language, and use a OSGi architecture (Open Services Gateway initiative, now OSGi Alliance[55]). Natively, Java VM does not provide technical means for dynamic libraries loading. OSGi is an answer to cover the need. OSGi proposes a component architecture. Components, named *bundles*, can be dynamically loaded to extend the software, and are communicating with each other via a software bus. OpenHAB specializes this bus with *channels*; this bus is the *B* of OpenHAB. OpenHAB is not a network protocol. However, some bundles propose a REST API to remotely control the box from iOS, Android, or Web applications.

5.7.8 Confluens

The Confluens project[56] aims to address home automation interoperability. It is both a software solution and a company created by home automation companies: CDVI (access control systems), Delta Dore (heating control devices), Hager, Legrand, Schneider-Electric (electrical equipment) and Somfy (shutters control automation). Technically, this is an inter-box communication system. The idea is that a consumer may have several products at home from different companies, each one managed by the home automation box of the corresponding company. So, there may be several home automation boxes at home. The boxes may share information and work together. Communication is based on the MQTT protocol, inside the home network. A discovery mechanism allows joining new boxes and to elect the box which is to play the role of the MQTT broker. Communications are securized by TLS (Transport Layer Security[57]). The Confluens company is the certificate authority, signs the TLS certificates, and manage their deployment on partners' boxes.

[53]`https://www.openhab.org/`

[54]`https://www.openhabfoundation.org/`

[55]`https://www.osgi.org/`

[56]`http://confluens.io/`

[57]`https://tools.ietf.org/html/rfc8446`

A home automation box is a centralized solution. Confluens proposes a multi-box approach. This is less distributed than the xAAL approach, but this is a first step. The MQTT protocol is a machine-to-machine protocol designed for the global Internet, and may be heavy to be used inside a home network, a local area network. The pyramidal signing mechanism of TLS' certificates can't allow third parties to enter the loop without prior agreement between companies. The security is in the hands of companies and not in end users' ones. At the opposite, xAAL allows users to manage their privacy.

Confluens is now an ended project. No product was sold with this technology. According to last news from involved companies, the code is willing to be distributed to the open-source community.

A CDDL Rules for xAAL Schemas

```
; Definition of Schemas for xAAL version 0.7
; Copyright Christophe Lohr IMT Atlantique 2019
; Copying and distribution of this file, with or without modification, are
; permitted in any medium without royalty provided the copyright notice
; and this notice are preserved.  This file is offered as-is, without any
; warranty.


schema = {
  ; The name of the device schema, i.e. the dev_type
  title: dev_type,

  ; A short description in natural language
  description: tstr,

  ; IETF BCB47 language tag of descriptions
  lang: tstr,

  ; URI (rfc3986) pointing to a more comprehensive documentation
  documentation: tstr,

  ; URI (rfc3986) pointing to the original version of this schema
  ; i.e. before any extention process
  ref: tstr,

  ; License of the the original schema file itself
  ? license: tstr,

  ; The schema name which is extended by this one
  ? extends: dev_type,

  ; List of attributes managed by the device
  ? attributes: { + identifier => type_name },

  ; Methods supported by the device
  ? methods: { + identifier => method },

  ; Notifications emitted by the device
  ? notifications: { + identifier => notification },

  ; Specifications of data mentionned in the schema
  ; Typically: attributes, parameters of methods and notifications
  ? datamodel: { + identifier => datadef }
}



; Format of the name of a schema in the form "foo.bar"
dev_type = tstr .regexp "[a-zA-Z][a-zA-Z0-9_-]*\\.[a-zA-Z][a-zA-Z0-9_-]*"


; Format of names for attributes, methods and notification
```

```
identifier = tstr .regexp "[a-zA-Z][a-zA-Z0-9_-]*"

type_name = identifier


; Definition of a method
method = {
  ; A short description in natural language
  description: tstr,

  ; List of input parameters
  ? in: { * identifier => type_name },

  ; List of output data
  ? out: { * identifier => type_name },

  ; List of device attributes that may be modified while invoking the method
  ? related_attributes: [ * identifier ]
}


; Definition of a notification
notification = {
  ; A short description in natural language
  description: tstr,

  ; List of output data
  out: { * identifier => type_name }
}


; Definition of a data
; Used by device attributes, methods and notifications parameters
datadef = {
  ; A short description in natural language
  description: tstr,

  ; Unit, according to the IANA Sensor Measurement Lists (SenML) registry
  ? unit: tstr,

  ; Formal descrtiption in CDDL (rfc8610)
  type: tstr
}
```