A brief introduction to xAAL v0.7-draft rev 2*

Christophe Lohr   Jérôme Kerdreux

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.

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.

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.

3. C. Lohr, J. Kerdreux, "Improvements of the xAAL Home Automation System", Future Internet 12, no. 6:104, https://doi.org/10.3390/fi12060104, 2020

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.

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

*Compares to "xAAL v0.7-draft rev 1b", this revision introduces the new specific target address (UUID)(0) for `is_alive` requests.

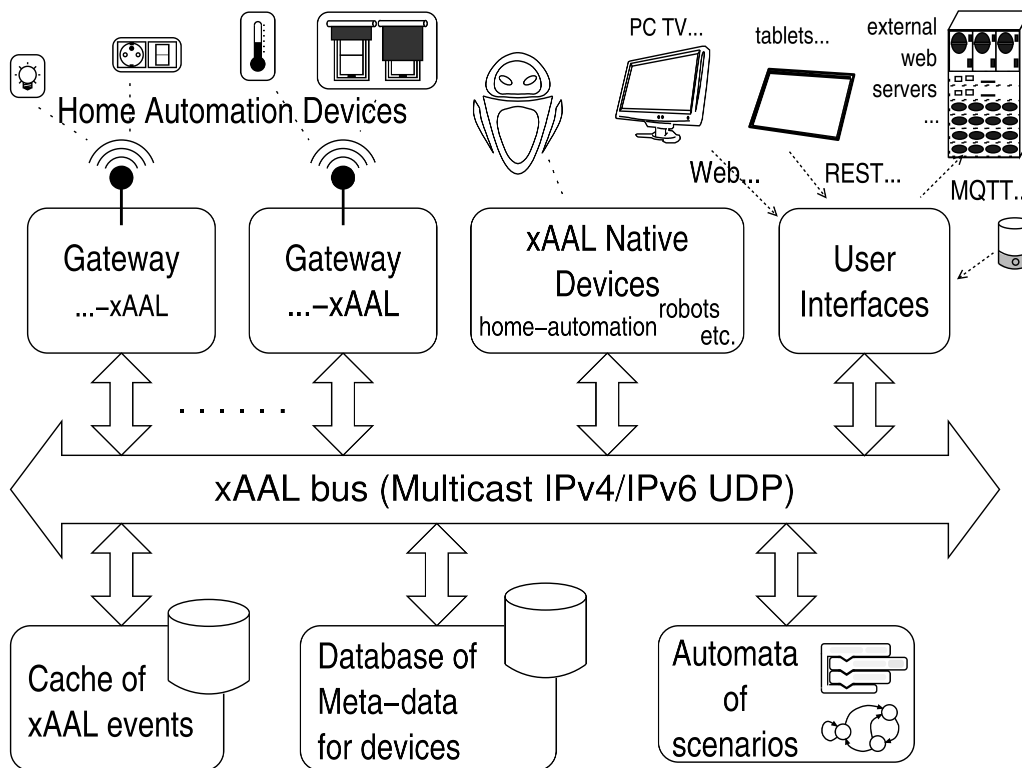

Figure 1: Functional Architecture of xAAL.

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.

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.

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. For instance: `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 proposed by a manufacturer (See 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 device's 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.[1] [2] If none is relevant, the unit should be one defined by 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 can be turned on and 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.

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

[2] 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 `out` 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 (i.e. the target field is left empty).

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 an `is_alive` request is generally the specific address `(UUID)(0)` reserved for this purpose. All devices should listen to this specific address, and expect a valid `is_alive` request in the message. This is the discovering mechanism proposed by xAAL.

  Alternatively, the target field of an `is_alive` may be a list of known devices one is waiting for.

  Note that the target field of requests in general is rarely empty. Devices should not send broadcast requests.

  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.

5 The xAAL Communication Protocol

xAAL is a distributed system. Participating entities need to communicate with each other.

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).
- [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.

```
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

```
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

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 for notifications and methods replies, and the one of the target(s) for methods requests.
- [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;
- 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).

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*)

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
}
```