Introduction au protocole xAAL
Version 0.2

Jérôme Kerdreux Maria Teresa Segarra Christophe Lohr

5 juin 2014

Résumé

Ce document donne la description du protocole xAAL tel que défini par l'équipe IHSEV de Télécom Bretagne, et son état d'avancement en 2014.

1 Architecture générale

Le système xAAL s'articule autour d'un *bus multicast* IP (IPv4 ou IPv6). C'est une architecture fondamentalement distribuée : les fonctions décrites ci-après peuvent être présentes plusieurs fois, être implémentées et regroupées dans plusieurs composants matériels de différentes manières, et dialoguent toutes via ce bus xAAL.

Il y a plusieurs intérêts à une communication par bus :

— Ne pas gérer un ensemble de connexions point-à-point ce qui permet d'économiser les ressources des équipements domotiques.

— Un bus permet la *découverte* : lorsqu'un nouveau composant apparaît dans l'installation, il s'annonce ; toutes les autres entités peuvent alors le prendre en compte. De même, lorsque l'on ajoute un composant, celui-ci peut interroger le bus pour découvrir les éléments déjà présents. Cela facilite grandement la configuration et permet la dynamicité et l'évolutivité du système.

1.1 Architecture fonctionnelle

La figure 1 représente l'architecture fonctionnelle générale du système xAAL.

Equipements natifs. Des équipements domotiques (capteurs, actionneurs) peuvent dialoguer de manière *native* dans le protocole xAAL.

Gateways. Plus vraisemblablement, les équipements domotiques de chaque constructeur ne supportent que le protocole de communication du constructeur. Dès lors, des *gateways* se chargeront de traduire les messages entre les protocoles constructeur et le protocole xAAL.

À ce niveau-là on suppose qu'il y a autant de gateway que de protocoles constructeur à supporter dans le bâtiment.

Suivant les technologies et les protocoles constructeur, chaque gateway devra gérer les questions d'appariement entre la gateway et les équipements domotiques qu'elle gère, les problèmes d'adressage et de configuration des équipements domotiques, et gérer la persistance de ces informations de configuration.

Notons qu'il peut être intéressant de pouvoir interroger une gateway sur le mapping qu'elle fait entre les adresses xAAL et les adresses physiques constructeurs des équipements. Même

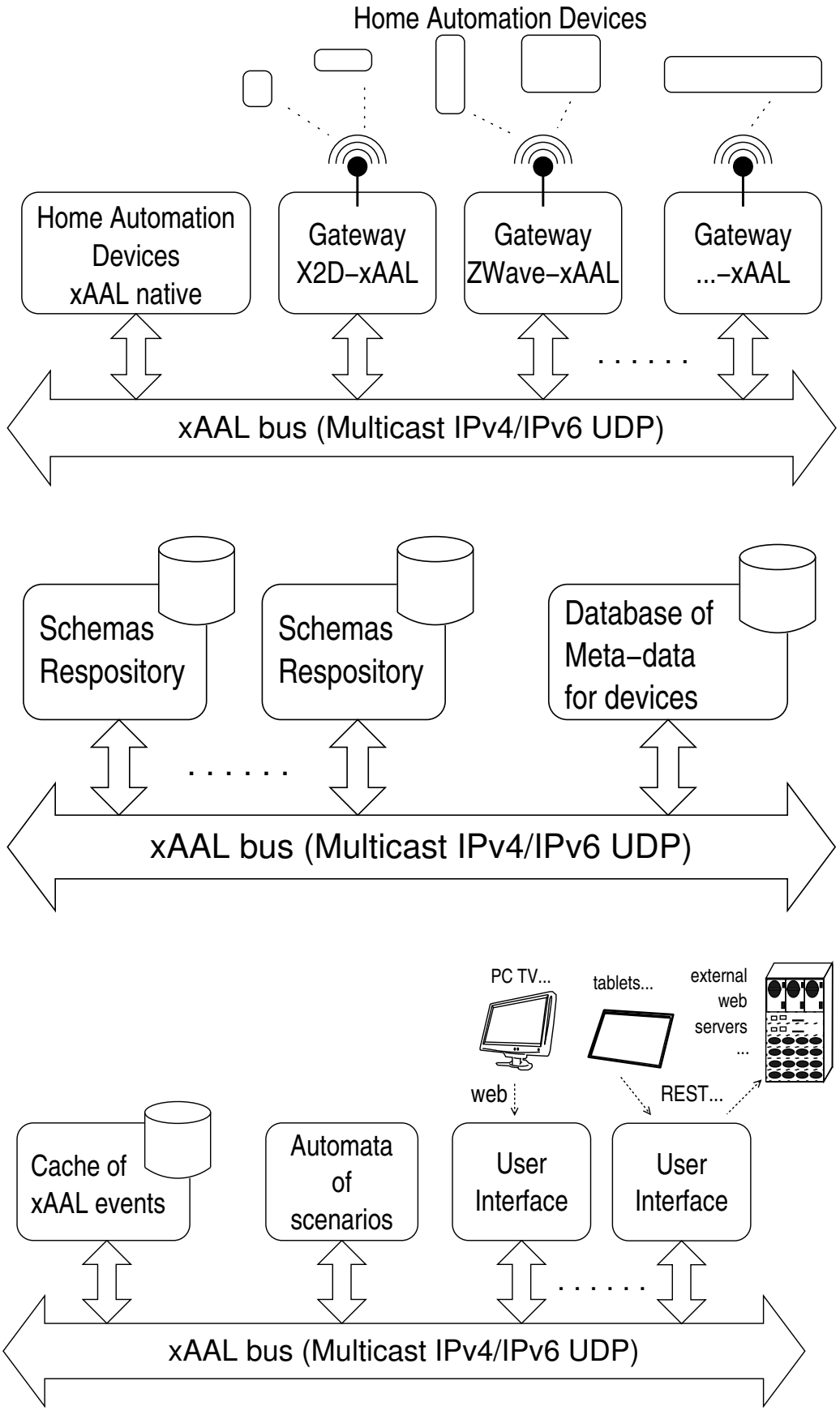

FIGURE 1 – Architecture fonctionnelle de xAAL.

si l'on interagit en xAAL avec les équipements via leur adresse xAAL, cette information de mapping que gère la gateway peut être intéressante pour la configuration de la base de données (décrite ci-dessous). En effet, lors de l'installation, l'installateur note généralement l'adresse physique constructeur de l'équipement avec sa localisation dans la maison. Pour faciliter la configuration de cette base de données (associer cette information de localisation avec l'adresse xAAL du device), il peut être intéressant de passer au début par l'identifiant ou l'adresse constructeur du device.

Registres de schémas (Schemas repository). D'une manière générale, les équipements domotiques doivent être décrits par un schéma (la nature de l'équipement, les interactions possibles). Ces différents schémas sont répertoriés dans un ou plusieurs *registres de schémas*.

Ces registres de schémas sont présents sur le bus xAAL. Les autres entités les interrogent, via ce bus, pour obtenir le schéma de description des autres équipements (que ces équipements soient actifs ou non sur le bus).

- On interroge le registre sur un identifiant de type, il nous renvoie son schéma, s'il le connaît. Plus précisément, il nous retroune une URL où nous pouvons télécharger le schéma.
- Associés à un schéma, en plus de la description du device lui-même, il peut y avoir des *métadonnées* : la langue dans laquelle sont rédigées les descriptions, les dates ou numéro de version, des url pour de la documentation, des url pour l'ihm (pointe vers des fragments de html, éventuellement hébergés sur internet), etc., une url pour les mise à jour, etc.
- Notons que le registre est également décrit lui-même par un schéma (à rédiger). Ce schéma doit donc être capable de décrire comment annoncer des schémas (on est donc un peu méta)

Dans une installation donnée, il peut y avoir zéro ou plusieurs registres de schémas sur le bus xAAL.

La définition du schéma d'un équipement doit être particulièrement soignée. Un certain nombre de ces schémas à vocation générale devront être rédigés par notre bureau international de standardisation xAAL ☺. Des schémas plus spécifiques pourront être rédigés par les constructeurs eux-mêmes.

Par voie de conséquence, il peut y avoir des conflits : pour un nom de schéma donné, plusieurs registres sur le bus pourraient renvoyer des schémas ayant des descriptions divergentes. Il faut alors prévoir un mécanisme de résolution de conflit : soit de manière statique où un administrateur vient invalider/effacer des schémas à problème dans un registre, soit de manière dynamique où les registres diffusent une information de priorité en plus du schéma lui-même (charge alors au récepteur de faire le tri). Ce point n'est pas encore tranché dans la spécification actuelle.

De plus, on peut envisager plusieurs façons de mettre à jour ces registres : soit via le bus xAAL lui-même, soit par Internet (le registre va télécharger des fichiers de schémas à des adresses qu'on lui aura indiquée), soit par un autre moyen (CD d'installation, etc.). Là aussi, ce point devra être précisé par la suite.

Base de données de métadonnées. À chaque équipement domotique présent dans une installation, il convient d'associer un certain nombre d'informations : tout au moins une information de localisation (p.ex. déclarer que l'équipement de type lampe et d'adresse X est situé dans la cuisine), et éventuellement un nom symbolique (p.ex. que l'équipement d'adresse X s'appelle "lampe-1").

Toutes ces informations sont regroupées dans une *base de données de métadonnées*. Concrètement, par soucis d'ouverture et d'extensibilité (au sens souplesse d'utilisation et non au sens

montée en charge), ces informations sont matérialisées par des tags.

Cette base de données contient en quelque sorte la configuration de l'installation domotique.

La base de données de métadonnées est présente sur le bus xAAL. Les autres entités l'interrogent via ce bus pour obtenir les tags associés à l'identifiant d'un device, ou inversement obtenir la liste des devices associés à un ou plusieurs tags.

- On le provisionne avec un ID de device, son type, une série de mot clef (tags) pour le localiser dans la maison
- On peut mettre à jour les informations sur un device
- On l'interroge pour connaître les tags associés à tel ou tel ID
- On l'interroge pour connaître les ID associés à tel tag ou tel type
- On l'interroge pour connaître les tags existants
- On peut supprimer un enregistrement
- Note : un device peut être déclaré dans cette base de données sans pour autant être présent sur le bus

Il conviendrait de disposer au moins d'une base de métadonnées sur le bus, mais il pourrait y en avoir plusieurs.

Là encore, la constitution de cette base de données doit être établie avec le plus grand soin. Il conviendra donc d'implémenter les mécanismes classiques d'organisation de tags : vérifier les incohérences, synonymes, ambiguïtés, proposer des facilités de renommage des tags, etc. Le piège classique étant de vouloir préciser une information de localisation à deux endroits, par exemple indiquer dans un tag *friendly name* :"lampe du salon", puis ajouter *location* :"salon" (et encore dans cet exemple l'information est redondante mais convergente...)

Note : On trouve cette notion de *friendly name* dans UPnP... Dans les faits ce mécanisme est une source de problèmes : soit il est détourné de son rôle pour y faire rentrer des informations de localisation comme dans l'exemple précédent, soit il est laissé à la valeur constructeur (combien de TV DLNA ou de smartphone androïd ont pour friendly name : *Media Server*...)

Cache. Il est courant d'avoir des capteurs monodirectionnels, que l'on ne puisse pas interroger, et qui envoient leurs informations de manière sporadique. (p.ex. un thermomètre qui n'envoie la température que si elle a changée.)

Il convient alors de disposer au moins d'un *cache* qui mémorise ce type d'informations et que les autres entités puissent interroger chaque fois que nécessaire.

Un tel cache doit mémoriser au moins les notifications émises par les devices. On pourra s'interroger sur la pertinence de mémoriser également les échanges de type commande/réponse.

Comme dans tout mécanisme de cache, il faut associer une estampille temporelle aux informations mises en cache. Lorsqu'une autre entité sur le bus xAAL demande une information dans le cache, il précise la durée maximum. Ce n'est pas le cache lui-même qui gère la pertinence des données en fonction de leur âge, mais le client qui fait sa requête.

Là encore on peut imaginer que des incohérences surviennent si deux caches sur le bus retournent des informations divergentes mais ayant la même estampille temporelle. On estime que ce phénomène est a priori rare, et par conséquent on n'envisage pas de définir un mécanisme particulier dans la spécification pour lever l'ambiguïté. On laisse le soin au client de trier les informations reçues en fonction de ses propres critères.

Automates de scénarios. Parmi les services avancés de la domotique, on trouve la notion de scénarios : par exemple exécuter toute une séquence d'actions à partir d'un clic de l'utilisateur, ou à heures programmées, ou surveiller des séquences d'événements puis y réagir, etc.

Pour ce faire, xAAL prévoit que cela soit pris en charge par un ou plusieurs entités de type *automates de scénarios*.

Ces automates de scénarios sont aussi le lieu de prédilection pour implémenter des devices virtuels : par exemple un scénario pourrait agréger et corréler tout un ensemble d'événements de devices réels pour synthétiser une information de type «présence» et générer une telle notification sur le bus, qui pourrait alors être ré-exploiter par d'autre entités. En procédant de cette manière, ce scénario devrait alors apparaître comme étant lui-même un device sur le bus, avec son adresse et son schéma.

Comme toujours, on peut avoir des incohérences ou des conflits entre scénarios (l'un allume la lampe, l'autre l'éteint), ou s'interbloquent, etc. Ce n'est pas la spécification xAAL en elle-même qui va l'empêcher. Les conflits sont censés être traités autant que possible off-line lors de l'édition de scénarios. La spécification n'impose pas le recours à un model-checker à base de Réseau de Pétri coloré temporisé, ni quoi que ce soit du genre... Cela reste un sujet de recherche, certes fort intéressant, mais hors cadre de la spécification xAAL. (Si quelqu'un le veut, il peut introduire un device de supervision, qui serait lui-même une sorte de moteur de scénarios un peu particuliers.)

Interfaces utilisateur. La, ou les, interfaces utilisateurs sont assurées par des entités spécifiques connectées au bus xAAL. Cela peut être un véritable équipement matériel avec un écran et des boutons ; ou bien un composant logiciel qui génère par exemple des interfaces Web utilisable dans un navigateur sur un PC, une télé connectée ou autre ; ou encore un logiciel qui fournit une API REST pour des applications mobiles (tablettes, smartphone), pour remonter des données vers un serveur MQTT, ou vers un serveur externe chez un prestataire.

Là encore, il semble naturel de pouvoir disposer de plusieurs interfaces utilisateurs (Smartphone, tablette, télécommande, etc.) au sein d'une installation domotique.

Synthèse Une telle architecture fonctionnelle permet de gérer les aspects dynamiques de l'infrastructure (modularité, évolutivité, adaptation, etc.). Ainsi, les fonctions avancées tels que les IHM ou les moteurs de scénarios, peuvent s'adapter automatiquement à l'infrastructure et à ses modifications (si un équipement entre/sort, si l'un tombe en panne, ou est remplacé, etc.).

Le fonctionnement typique d'une IHM pourrait être celui-ci :

1. interroger le bus pour repérer les équipements présents ;
2. interroger le registre de schémas pour connaître le type de ces équipements, le type de données des capteurs/actionneurs, les commandes possibles, des URLs pour récupérer des icônes ;
3. interroger la base de métadonnées des capteurs pour découvrir la configuration de l'installation (afficher le nom symbolique, la localisation et les autres tags associés à chaque équipement) ;
4. interroger le cache pour connaître les derniers états connus de ces équipements ;
5. construire dynamiquement des écrans d'affichage et des interfaces de commande pour ces équipements.

1.2 Architecture matérielle

L'intérêt d'une telle architecture fonctionnelle distribuée est d'autoriser une grande liberté concernant le placement et le regroupement des fonctions sur des composants matériels.

Par exemple :

— Un constructeur d'équipements domotiques pourrait commercialiser une *box* qui regrouperait : la gateway pour son propre protocole domotique, le répertoire de schémas pour les équipements de sa marque, le cache des équipements gérés par la gateway, quelques

automates de scénarios pour gérer les devices virtuels, et une petite base de métadonnées pour ses propres capteurs pour noter leur localisation.

— Un opérateur de services domotiques pourrait vendre une *box* (différente et complémentaire de la première) qui regrouperait : la base de données de localisation, les automates de scénarios, des interfaces web et REST pour dialoguer avec ses serveurs dans le cloud et offrir des services enrichis, etc.

— Un prestataire de services virtuels (c.à.d. ne vendant que du service et pas de matériel) pourrait fournir une application mobile pour smartphone ou tablette qui embarquerait : une interface utilisateur, un automate de scénarios, une interface vers ses serveurs externes pour des services avancés, etc.

— Un prestataire de services (p.ex. aide à la personne, télé-médecine, etc.) pourrait rajouter sa *box* (encore une autre) qui regrouperait : une gateway pour des équipements spécifiques (médicaux, télé-alarme, etc.), des automates de scénarios d'alerte, une interface vers ses serveurs externes, etc.

— Un constructeur d'objets connectés innovants pourrait vendre ces objets connectés qui embarqueraient : le composant domotique lui-même parlant de manière native en xAAL, le registre de schéma décrivant l'objet lui-même, un automate de scénarios, une interface vers ses serveurs externes, etc.

— Les services ou des scénarios un peu musclés (p.ex. à base d'actimétrie, ou de composition de services adaptatifs, ...) pourraient être composée d'un moteur de scénarios, couplé à du cache, couplé à une base de métadonnées.

Le fait de s'appuyer sur IP comme moyen d'interconnexion et ce protocole léger xAAL permet, on l'espère, une grande souplesse et une meilleure interopérabilité entre ces différents acteurs qui n'ont rien d'autre à révéler à la concurrence que des messages xAAL.

Le point véritablement dur est la définition de schémas et la rigueur avec laquelle on s'y plie dans les implémentations...

2 La couche de transport

On utilise du multicast IP (IPv4 et/ou IPv6), en UDP donc.

— L'adresse IP multicast est à définir.

— Le numéro de port est à définir également. Éventuellement se réserver une plage de ports.

— S'il y a plusieurs "réseaux domotiques" dans le domicile, ils utilisent des ports différents.

— Les messages auront une taille limitée par la taille max UDP (p.ex. 65507 octets en IPv4). On laisse la fragmentation IP opérer normalement. (Par opposition à des approches comme xPL qui borne à 1500, puis implémente des *continue*. La fragmentation IP gère déjà cela très bien.) Reste le problème que certains PICs ne gèrent pas forcément très bien la fragmentation IP (p.ex. la stack de base de Arduino). On recommande seulement d'essayer de se limiter à 1500 octets. Si des services ont besoin d'échanger des plus gros volumes de données entre devices, ils sont priés de dégager le bus domotique (qui n'est pas fait pour cela) au profit d'un autre canal de transmission (p.ex. une connexion TCP) qu'ils peuvent par contre annoncer sur le bus xAAL.

— La configuration IP des équipements est hors cadre du système xAAL. Les équipements peuvent utiliser de la configuration IP statique, du DHCP, les Router Advertisement IPv6, du ZeroConf, une adresse Link-Local (169.254.0.0/16, fe80 : :/64), etc.
Notons cependant que, du fait de l'utilisation d'un bus multicast, l'adresse source des émetteurs est assez peu importante (et pas utilisée dans notre cas). Certains équipements pourraient tout aussi bien utiliser une adresse IP bidon hardcodée...

— On attire l'attention des éventuels futurs développeurs sur les horreurs que l'on a pu constater dans les implémentations des bus xPL ou xAP (souvent en Java ou .Net) : le

recours à un "hub", un composant logiciel qui répète les messages du bus pour les applications situées sur une même machine. Cela est parfaitement inutile sur la quasi-totalité des systèmes d'exploitation actuels : les applications d'une même machine peuvent tout à fait avoir chacune leur propre socket sur le bus multicast, pour peu que l'on utilise les options de socket dédiées à cela (typiquement `SO_REUSEADDR` et `IP_MULTICAST_LOOP`)...

— Le choix du multicast IP impose quasiment le protocole UDP. UDP ne gérant pas la détection de perte de paquet et les retransmissions, on aurait pu trouver plus confortable de choisir TCP. Voici quelques arguments qui peuvent rassurer les gens frileux vis à vis d'UDP :

Dans le contexte d'un réseau au domicile, on peut faire l'hypothèse que la perte de paquets, quoique toujours possible, est a priori plutôt rare.

Par construction xAAL est assez peu exigeant vis à vis du traitement des cas d'erreur (cf. 7.3) ; de ce point de vue xAAL nous paraît plutôt *robuste*.

Utiliser TCP pour la gestion des pertes de paquets ne fait que déplacer le problème dans la pile protocolaire (et de manière lourde : c'est systématique, et l'automate TCP est consommateur de ressources). Dans xAAL, si l'on a besoin de gérer cela (ce qui, finalement, n'est pas nécessairement un besoin systématique), cela se fera au niveau de l'application de services (des entités implantées sur du matériel qui dispose généralement des ressources confortables).

3 La couche de présentation des données

Il s'agit de décider si les données manipulées par le bus xAAL sont plutôt de nature textuelle ou bien de nature binaire.

Ce point a fait l'objet d'un débat au sein de l'équipe, où deux écoles de pensées s'interrogent.

3.1 L'approche *protocole binaire*

La motivation principale est de rester à bas niveau : un protocole qui peut être codé dans un PIC (p.ex. en C de base), plutôt qu'un middleware sophistiqué issu du monde PC et des applications web.

Cela ne signifie pas pour autant que l'on vise des messages compacts/compressés, mais des messages simples à fabriquer et à parser (à coup de structure C à caster), auto-suffisant, dans un format contraint, avec une variabilité contrainte, une taille prédictible, etc.

Plus concrètement on déciderait de manipuler essentiellement des mots de 32 bits, en s'alignant sur 32 bits (donc, avec du bourrage s'il le faut).

L'inconvénient est que la phase de débuggage des messages est plus compliquée qu'une approche textuelle, car il faut alors un outil validé pour analyser les trames échangées. L'autre inconvénient et que l'on est trop éloigné des technos à la mode (autour du web), et qu'il sera a priori plus compliqué de faire y adhérer une communauté.

Finalement, ce n'est pas l'approche qui a été retenue par l'équipe pour l'implémentation de xAAL. Néanmoins, un draft de spécification a été défini et est disponible sur demande, c'est la *version 0.1-bin*

À l'avenir il pourrait être intéressant de se pencher sur le protocole Mihini/M3DA pour faire cela justement...

3.2 L'approche *protocole textuel*

L'avantage d'un protocole textuel est qu'il est plus facile à débugger, rien qu'en regardant les trames échangées.

L'inconvénient est que les messages sont plus complexes à construire, et encore plus compliqués à parser.

Le compromis adopté a été de choisir Json comme format d'échange de données. De nombreuses librairies sont disponibles dans de nombreux environnements de développement.

De plus, c'est assez proche du monde Web, et comme les dernières générations d'informaticiens ont été massivement et prioritairement formées aux technologies du web, on peut espérer une meilleure adhésion auprès de ce public, ainsi qu'auprès d'un certain public de décideurs...

C'est cette approche qui a été expérimentée dans les implémenations précédentes d'xAAL faites dans l'équipe : c'était la *version 0.1-json*. Les spécifications données dans la suite du document pour la *version 0.2* s'appuient sur cette approche *protocole textuel*, et l'étendent pour décrire la version actuelle...

Une variante intéressante serait peut-être d'utiliser certains aspects de CoAP [1] (ou un sous-ensemble)... À étudier pour l'avenir... même si la proximité avec le monde du web paraît malgré tout d'un intérêt technique limité...

4 Définition d'un device

Un *device* possède :

- un `devType` : ce nom référence le *schéma*, c'est-à-dire le type du device. Il est codé en dur dans le device.
  - Cet identifiant de schéma est une chaîne composée d'une paire de mots séparés par un point.
    - Le premier mot désigne une classe de type de device (p.ex. éclairage, chauffage, multimédia, etc.).
    - Le second mot désigne un type dans une classe donnée (p.ex. dans la classe éclairage, on a une lampe on-off, une lampe avec graduateur, une boule à facettes, etc.). Ce second mot pourra référencer les extensions de schémas constructeurs (cf. 5.1).
  - L'identifiant de type `"any.any"` est réservé, et désigne tous les types de toutes les classes.
  - Et par exemple le couple `"lamp.any"` désigne tous les types de la classe `"lamp"`.
  - On se réserve un nom particulier : `"experimental"` (c.-à-d. lorsque quelqu'un fait une implémentation mais n'a pas encore obtenu un nom par notre bureau international de standardisation d'xAAL). Et ceci pour la classe, ainsi que pour le type.
- une `address` : un identifiant de device unique sur le bus.
  - L'identifiant de device est un UUID (RFC 4122) : la représentation textuelle d'un nombre aléatoire de 128 bits.
    Par exemple `634c4471-bc3e-47f2-8cd3-eadfc08c6c6a`
  - L'adresse `00000000-0000-0000-0000-000000000000` est réservée. C'est l'adresse de broadcast et désigne tous les devices.
  - Comment est attribuée une adresse ?
    - Soit codée en dur, en usine.
    - Auto-générée (random) au moment de l'installation. Par contre il est préconisé que cette adresse soit un minimum "persistante" : c'est à dire qu'il faut (autant que possible) la sauvegarder lors des coupures d'alimentation par exemple.
    - Il y a en principe de très fortes probabilités de ne pas avoir de collision d'UUID. Cependant, il est techniquement possible de vérifier qu'un UUID n'est pas déjà utilisé sur le bus par une technique de type "arp gratuit", c'est à dire avec par des

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

requêtes `isAlive` pour vérifier que quelqu'un d'autre ne l'a pas déjà.
    - Mais surtout pas attribué par un superviseur de bus ou autre machin du genre !
- Un équipement peut être composé de plusieurs *devices*. C'est le cas typiquement d'une gateway (p.ex. un équipement qui va faire la passerelle entre notre bus domotique et du X10, ou bien du Zigbee, X2D, Somfy, etc.). Mais cela peut être également le cas d'une petite station météo qui mesure la température intérieure/extérieure/hygrométrie/etc.
    - Un tel équipement est alors qualifié de *device composite*,
    - et chacun de ses composants est un *device embedded*.
    - Le device composite s'annonce (éventuellement) lui-même sur le bus avec sa propre adresse et son propre nom de schéma. On peut interagir avec lui spécifiquement (e.g pour faire sa config, consulter son niveau de batterie, etc.). Il annonce également la liste de ses embedded (avec leurs adresse)
    - Chacun des embedded est également annoncé sur le bus comme un device ordinaire, bus-natif, avec sa propre adresse et son propre nom de schéma. Chacun est annoncé en précisant son parent, c.-à-d. le composite auquel il est raccroché. (Note : les devices véritablement bus-natif ont pour parent=0)
    - Un device composite peut également choisir d'être invisible, ne pas s'annoncer lui-même, ni signaler qu'il a des embedded, ni que ses embedded l'ont pour parent.

5 Définition d'un schéma (ou type d'un device)

Chaque device est typé, c.-à-d. décrit par un *schéma*. Le schéma est fortement inspiré de UPnP [2]. Nous n'avons pas encore défini de dialecte pour rédiger ces schémas. Vraisemblablement ce sera du XML (reste à écrire le DTD), ou du *schema JSON*[3] (à supposer que le draft proposé devienne un RFC).

- Le schéma donne un peu de sémantique à un device et décrit ses capacités : une liste de méthodes possibles sur ce device (mécanisme de type *requête-réponse*), une liste de notifications qu'il émet spontanément, et une liste de variables d'état de ce device (le device annonce spontanément sur le bus tout changement de valeur).
- Une liste de *méthodes*. Chaque méthode est décrite par :
    - un nom unique dans le schéma qui l'identifie.
    - une description textuelle (voir plus tard comment gérer la langue et les problèmes l'internationalisation)
    - une liste d'arguments, chaque argument étant défini par :
        - un nom
        - une direction (in/out)
        - l'unité (au sens Bureau International des Poids et Mesures...)
        - une description
- Une liste de *notifications*. Les notifications sont des messages que le device émet spontanément sur le bus. Ces notifications sont définies par :
    - un nom,
    - une description
    - une liste de variables (nom, unité)
- Une liste de *variables d'état*. Si leur valeur change, cela génère par défaut une notification sur le bus. (Notons qu'un device peut très bien être composé d'un certain nombre de variables internes supplémentaires qui ne sont pas monitorées et ne font pas l'objet de notifications...) Chaque variable d'état est définie par :

2. `http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf`, chapitre 2.
3. `http://json-schema.org/`

— un nom
— une description
— l'unitée
— une valeur par défaut (?)

5.1 Héritage de schémas.

Il y a une notion d'héritage entre schémas, un schéma peut *étendre* un schéma préexistant. Nous définissons les 3 premiers niveaux de cette généalogie :

1. Un schéma générique commun à tous les devices du bus, que tout le monde doit implémenter.
2. Un schéma spécifique à chaque classe, qui hérite donc du schéma général (*lamp*, *thermometer*, *switch*, *thermostat*, etc.)
3. Un schéma spécifique à chaque type, qui hérite donc d'un schéma de classe, et qui précise le niveau de fonctionnalités (p.ex. une lampe peut être de type on/off, certaines sont de type trigger, et d'autres avec graduateur...)
4. Par la suite, les constructeurs d'équipements domotiques vont naturellement définir leurs propres schémas et les décliner suivant leur gamme de produits. Par contre, il est interdit à ces schémas constructeurs d'être des schémas de niveau 1 (le schéma générique est seul et unique), ni des schémas de niveau 2 (schéma de classe). Les schémas des constructeurs sont donc nécessairement des extensions des schémas de niveau 2 ou plus. En termes de nommage des schémas (la paire pointée décrite au 4), le second mot est alors laissé à la discrétion du constructeur. (Les constructeurs frustrés de cet état de choses sont invités cordialement à discuter avec notre bureau de normalisation s'ils souhaitent l'introduction de nouveau schéma de niveau 2 ou 3.)

Note : pensez bien, en listant les APIs des devices décrits par ces schémas, que certains devices ne seront pas capable d'écouter le bus et ne feront qu'émettre dessus, uniquement des notifications de leur variables d'état.

5.2 Nommage des schémas

Il faudra accorder une grande rigueur dans le nommage des schémas afin de préserver un maximum de cohérence dans l'esprit de tous... Dans de trop nombreux systèmes (xAP, xPL pour ne pas les nommer), le nommage des devices et de leur typage est régulièrement détourné pour y faire rentrer des notions qui n'y ont pas leur place.

Ce qu'est un schéma. Deux choses : On souhaite que le nom du schéma renvoie avant tout aux fonctionnalités du device, aux moyens d'interagir avec lui. Ce nom doit également évoquer la nature du capteur.

Ce que n'est pas un schéma.
— Le nom n'est pas censé renvoyer à une technologie de constructeur (et encore moins à un modèle précis).
— Le nom du schéma ne doit pas renvoyer à un usage : un contacteur (qui transmet des notifications «ouvert/fermé») peut servir aussi bien à de la sécurité (sur les fenêtres ou la porte d'entrée) que de l'actimétrie (sur les tiroirs de cuisine) ; de même un thermomètre reste un thermomètre (qui envoie des notifications de température), qu'il soit dédié à une station météo ou à la gestion du chauffage. Il n'y a donc pas lieu d'avoir des

catégories de type *security* ou *hvac* (Heating, Ventilation and Air-Conditioning) ! Ce sont là des chapitres dans un catalogue de vente constructeur, pas des catégories de fonctions d'équipements domotiques. Tout au plus, le rôle associé à un capteur (sécurité, chauffage, etc.) est un tag à mémoriser dans la base de métadonnées de l'installation.
— On évitera soigneusement les catégories fourre-tout du type *soft* (ça s'est déjà vu !). Évidemment que l'on a un certain nombre de composants logiciels... De même, certains équipements seront purement logiciels (par exemple une application de web-radio dans un PC). Le fait que le composant soit en soft plutôt qu'en hard n'a aucun intérêt du point de vu xAAL.

6 Schéma générique

C'est le schéma de base. Tous les autres schémas doivent en hériter d'une manière ou d'une autre.

— Paramètres internes :
  — `devType` : le nom du schéma auquel obéit le device
  — `address` : une chaine, un UUID (l'adresse du device)
  — `Parent` : une chaine, un UUID (L'adresse du parent dans le cas d'un composite, ou 0 si non)
— Variable d'état :
  — aucune imposée dans le schéma générique
— Notification :
  — `alive` : émis lors de la mise en marche du device, puis régulièrement à un rythme laissé à la libre appréciation du device. Le message de notification ne contient pas de paramètres, pas de *body*.
  — `stateChange` : émis à chaque changement d'une des variables d'état. Le corps du message contient uniquement variables d'état qui ont changées.
  — `error` : émis lorsque l'équipement détecte une erreur.
    — `description` : la description textuelle de l'erreur
    — `code` : un code (numérique) d'erreur.

    Cela a vocation à être surchargé dans les définitions de schémas spécifiques.
— Méthodes :
  — `isAlive` : pas d'attributs, ni de valeur de retour. Cependant le (ou les) destinataire doit répondre autant que possible par une notification `alive` à tous (c.à.d. et non pas par un message `reply` à l'auteur de la requête).
  — `getDescription`
    — `vendorId` (out) : chaîne de caractères ou numéro attribué par le bureau ?
    — `productId` (out) : chaîne de caractères
    — `version` (out) : chaîne de caractères
    — `parent` (out) : adresse du parent s'il y en a un, ou 0.
  — `getState`
    — `<variables d'état>` : A priori cette méthode sera surchargée dans les définitions des schémas spécifiques de chaque device pour retourner tous les paramètres d'état spécifiques ajoutés par chaque type de device.
    — Attention : on ne renvoie pas les paramètres internes `devType`, `address` qui sont de toute manière dans l'entête, ni le `parent` qui est donné dans le `getDescription`.
  — `setBusConfig`
    — `busAddr` (in/out) : une chaîne de caractères donnant l'adresse IPv4 ou IPv6 du bus xAAL sur lequel basculer.

      Si l'adresse retournée n'est pas celle que l'on avait envoyée, c'est que le device n'a

```
{
 "header":
  {
   "version": "0.2",
   "source": "bd081741-ceb6-41eb-87e0-6628d4959657",
   "target": "83c84e87-a230-4245-8829-f788fa6365a0",
   "devType": "thermometer.queryable",
   "msgType": "request",
   "action": "getState",
   "cipher": "none",
   "signature": ""
  }
}
```

```
{
 "header":
  {
   "version": "0.2",
   "source": "83c84e87-a230-4245-8829-f788fa6365a0",
   "target": "bd081741-ceb6-41eb-87e0-6628d4959657",
   "msgType": "reply",
   "devType": "thermometer.queryable"
   "action": "getState",
   "cipher": "none",
   "signature": ""
  },
 "body":
  {
    "temperature": 9.3000000000000007
  }
}
```

FIGURE 2 – Exemple de messages xAAL

pas pu se configurer dessus (bref, il y a une erreur).

— `busPort` (in/out) : un entier non signé (enfin, limité à 65535), indiquant le port du bus xAAL sur lequel basculer.
De même, si la valeur retournée n'est pas celle que l'on avait envoyée, c'est qu'il y a eu une erreur.

— `TTL` (in/out) : un entier 8 bits non signé, indiquant le TTL utilisé pour l'émission de paquets multicast.
A priori on met 1 par défaut, mais après tout on pourrait aussi vouloir le changer...

7 Définition d'un message

Un message est composé d'une partie *header* et d'une partie *body*. Le header est obligatoire, alors que la présence du body est facultative.

La figure 2 donne un exemple d'échange de messages xAAL.

7.1 Header

L'entête d'un message comprend :

— `version` : la version du protocole. Le document présent décrit la version 0.2.

— `source target` : est adresses source et destination du message.
Notons que l'adresse destination peut être 0, auquel cas le message est de type broadcast. Les récepteurs (donc tout le monde) sont invités à filtrer la pertinence du message sur type (nom de schéma) du device. Les messages de notification sont souvent de type broadcast, mais pas nécessairement. En revanche, les messages de réponse à une requête sont renvoyées spécifiquement à l'adresse du demandeur.

— `devType` : le nom du schéma (couple *classe.type*) auquel se rapporte ce message. Donc typiquement si c'est un message de type requête, c'est le nom du schéma du device destinataire, et si c'est un message de type réponse ou notification, c'est le schéma de l'émetteur.
Notons que dans le cas d'une requête (broadcast ou non), le demandeur peut avoir recours au mot clef `any` pour la classe comme pour le type. Par contre, dans leur réponse (s'il y a lieu), les devices concernés précisent leur `devType` explicitement. (Concrètement on ne peut avoir de `any` dans les réponses ni dans les notifications.)

— `msgType` : le type de message parmi `request reply notify`. Donc, typiquement, un message `request` est suivi d'un `reply` s'il y a lieu de retourner des valeurs au demandeur.

Notons que l'on n'impose pas de numéro de séquence ou l'identifiant de requête-réponse... On peut donc avoir quelques incohérences : on en discute plus bas.

— `action` : le nom de l'action portée par le message parmi la liste des *méthodes* et des *notifications* décrites dans le schéma dont le nom est indiqué dans le message.

— `cipher` : le mécanisme de sécurité utilisé dans le message. Pour l'instant on a défini le mécanisme `none`...
  À l'avenir on peut envisager le mécanisme `SHA` (SHA1 SHA256 SHA512 ?) : une clef partagée sur tout le bus.
  On peut également envisager un mécanisme type `PGP` : un chiffrement asymétrique, la clef privée de chaque device étant hardcodée, et les clefs publiques écrite sur l'étiquette et partagées (et contresignées) au sein du bus...
  Voir plus bas pour une discussion sur le sujet.

— `signature` : la signature du message par le cipher choisi...

7.2 Body

Le corps du message contient les paramètres des requêtes et les valeurs de retour de réponses et notifications.

Le corps du message est optionnel : tout dépend de ce qui a été défini dans le schéma en question, en fonction de l'action et du type de message.

Donc, le cas échéant, le corps du message contient une liste de clefs-valeurs.

7.3 Cas d'erreur

L'esprit de la spécification xAAL est avant tout de s'intéresser à l'interopérabilité. On est donc guidé par la volonté de proposer, mais pas de contraindre... Par voie de conséquence, le protocole xAAL a très peu de garde-fous contre les cas d'erreur. C'est de la responsabilité des implémentations de composant de xAAL de faire au mieux.

La spécification ne dit rien quant à l'absence de réponse. D'une part, puisque l'on est en UDP, la requête comme la réponse peut se perdre. D'autre part, il y a des device qui ne savent qu'émettre et pas écouter sur le bus. La spécification n'impose pas de mécanisme de timeout : certains device n'ont pas nécessairement d'horloge sous la main, et peuvent gérer les réponses aux requêtes dans un buffer circulaire (et si une réponse arrive tellement tard que l'on a oublié la question, tant pis)

Autre argument concernant la non-réponse à une requête : dans le cas d'une requête non unicast (soit réellement broadcast, soit sur une classe de type), il paraît naturel que les équipements qui ne peuvent honorer la requête puissent choisir de se taire plutôt que de polluer le bus avec des messages d'erreur.

Dans certains cas on peut avoir envie de s'assurer que les messages ne sont pas perdus (p.ex. centrale alarme). Puisque la perte n'est pas gérée par la couche protocolaire xAAL elle-même, cela peut être pris en charge par l'application au-dessus. On peut envisager plusieurs stratégies :

— après une requête d'action vers un device, l'émetteur questionne le device pour vérifier s'il y a eu changement dans ses variables d'état ;

— l'api d'une action peut introduire un paramètre ad-hoc (p.ex. `requestId`) que le device doit répéter dans sa réponse ;

— etc.

Les cas d'erreur ou d'incohérence sont possiblement nombreux. La spécification n'impose pas un comportement précis. Donc, face à un message incohérent, des doublons, des réponses ou annonces contradictoires, *le comportement est non spécifié.*

On pourrait avoir envie de décréter malgré tout que, face à un message incohérent, on a l'obligation de le rejeter, de l'ignorer. Malheureusement, pour certains équipements, il n'est pas possible de défaire ce que l'on a commencé à faire (pas assez de mémoire, ou action partielle). Bref, la spécification n'impose rien. Charge aux implémentations de faire attention, et de faire au mieux.

Par exemple, dans certains cas c'est relativement facile de trancher :

— Un device reçoit une requête sur son adresse à lui, avec un type de device précis, mais qui n'est pas le sien. Il peut donc facilement identifier le problème et ignorer la requête.
— Un device reçoit une requête mais le nom de la méthode n'existe pas dans son schéma. Il peut donc facilement identifier le problème et ignorer la requête.
— Un device reçoit une requête mais certains paramètres sont du mauvais type, ou bien il manque des paramètres, ou il y en a en trop. Là, suivant les cas, il peut avoir commencé à réaliser partiellement un certain nombre de tâches correspondant à la méthode demandée et peut se retrouver en mauvaise posture. Il ne pourra peut-être pas défaire ce qu'il a commencé, et donc on ne peut pas lui imposer de rejeter le message. S'il le fait, c'est bien. Si non, tant pis. Tout au plus il positionne un code d'erreur qui va bien.
— Inversement, si dans une réponse certaines variables n'ont pas le bon type, s'il en manque ou s'il y en a en trop, le device peut avoir commencé à traiter partiellement la réponse... On lui demande de faire au mieux.
— Si les schémas sont rédigés soigneusement, on peut également prévoir un certain nombre de messages de notification d'erreur. Mais ceci est de la responsabilité du schéma, pas du bus xAAL.

8 Sécurité

Dans les travaux préliminaires, nous n'avons pas proposé de modèle de mécanisme de sécurité. Cependant, ce besoin paraît incontournable aujourd'hui. Cette section propose quelques éléments de réflexion.

On peut considérer que le lien sous-jacent apporte un certain niveau de sécurité. Considérons une télécommande infrarouge : la liaison RC5 (ou RC6) n'est pas chiffrée, et pour pirater la transmission entre la télécommande et la télé, il faut que le voisin puisse viser avec sa télécommande à travers la fenêtre. Ce sont les murs de la pièces qui assurent la sécurité de la transmission, pas le protocole lui-même.

Il en va de même pour un protocole IP : il faut déjà être sur le réseau pour pouvoir l'attaquer. Il faut être présent physiquement pour câbler en Ethernet, les liaisons WiFi ont des clefs wpa (généralement), reste qu'en CPL le chiffrement est existant mais rarement activé...

Et pour finir, pensez qu'à contrario la configuration du firewall du réseau de la maison peut être mauvaise, ou que des PC sur ce réseau peuvent être compromis par des virus...

8.1 Besoin de sécurité

Lorsque l'on parle de sécurité, il faut avant-tout définir contre quoi l'on veut se prémunir. Il ne s'agit pas ici de traiter les cas de défaillance, mais bel et bien des cas de malveillance.

Parmi les préceptes classiques de la sécurité (intégrité, confidentialité, disponibilité, non-répudiation, authentification), de quoi a-t-on besoin ?

Au minimum, il faut pouvoir assurer l'authentification de l'émetteur des messages, que les messages ne sont pas corrompus, et qu'il n'y a pas de rejeu.

Reste la question de la confidentialité : a priori ce n'est pas forcément bien gênant si le voisin peu écouter (tant qu'il ne peut pas injecter des faux messages). Cependant, dans certains cas

d'usage cela peut être gênant : rien que le fait de savoir que la centrale d'alarme est coupée ou que la porte d'entrée est déverrouillée peut-être problématique...

D'un autre côté, il faut garder à l'esprit que l'on a des équipements très légers et que les mécanismes de chiffrement sont gourmands en ressources.

De plus, pour les mettre en place il faut passer par une phase de configuration (les clefs), et que cela peut être problématique. (On pourra s'inspirer des mécanismes de configuration automatique de clef WiFi comme WPS.)

8.2 Propositions de sécurité

- `"cipher":"none"` : il faut se garder la possibilité de communiquer sans chiffrement, justement pour tous ces équipements qui sont trop légers pour embarquer du chiffrement et des mécanismes de configuration de clefs.
- `"signature":xxx` : rien qu'un mécanisme de signature par une fonction de hachage classique (MD5 SHA1 SHA-256 SHA-512 Blowfish ...) avec une clef partagée permet de garantir l'authentification et l'intégrité. Par contre cela implique de gérer la configuration des device pour leur donner cette clef.
  Une autre approche est d'utiliser des mécanismes à clefs asymétriques : la clef privée étant hardcodée dans le device, sa clef publique étant écrite sur l'étiquette, avec un mécanisme de diffusion, de partage et de contre-signature de ces clefs (à la PGP).
  Reste que pour éviter le rejeu il faut ajouter dans le message signé un petit quelque chose qui le rend unique... Pour cela, on peut ajouter une estampille temporelle (et éviter le rejeu de trame capturées il y a trop longtemps). Ou, dans le schéma du device on peut prévoir un mécanisme de challenge : d'abord demander au device un cookie, puis le lui retourner dans le message en signant tout. Un tel mécanisme est à prévoir dans le schéma du device, pas dans le protocole xAAL.
- La confidentialité : cela implique de chiffrer les informations elles-mêmes transportées dans le message. On peut envisager deux approches :
  - Cela peut être spécifié non pas dans le protocole xAAL mais dans le schéma du device qui en a besoin : on s'échange un paramètre public *data* mais dont la valeur est chiffrée.
  - Une autre approche consisterait à redéfinir dans le format de message un nouveau bloc body mais entièrement chiffré.

À l'heure actuelle, tout ceci est encore embryonnaire. Cependant l'idée générale est que la sécurité doit être possible mais optionnelle.

9 Bonnes pratiques

La spécification xAAL en elle-même permet un grand nombre de choses, des bonnes et des moins bonnes... Au fil des pages précédentes quelques suggestions de bonnes pratiques ont été évoquées. Nous les regroupons ici.

9.1 Device composite

Dans le monde de la domotique il est courant de trouver des équipements qui englobent plusieurs fonctions en un seul boîtier : par exemple thermomètre, contacteur de porte, fil pilote.

En xAAL, ceci s'organise autour de la notion de *device composite*. (C'est notre façon de faire de l'héritage multiple...) Ce device va alors exposer autant de devices xAAL que de fonctions élémentaires. Dans l'exemple ici : un device avec le schéma thermomètre, un device avec le schéma contacteur de porte, un device avec un schéma fil pilote.

De plus, le device va s'exposer lui-même avec le schéma *composite*, dont la seule fonction est de lister ses devices *embedded* (donner leur adresse et schéma). Le fait de s'exposer comme *composite* n'est pas obligatoire, c'est seulement recommandé : cela n'apporte pas de fonctionnalité domotique supplémentaire, mais cela permet d'exprimer l'architecture matérielle, ce qui peut aider au diagnostic et à la maintenance.

9.2 Gateway

Résumons le rôle d'une *gateway* :

- transformer des messages xAAL en des messages d'un protocole domotique constructeur ;
- gérer l'appairage entre elle-même et les devices constructeur ;
- gérer la configuration de ces devices (par exemple Z-Wave étant bidirectionnel, c'est la base Z-Wave qui pousse certains paramètres vers les devices : la gateway étant alors la base Z-Wave, elle gère ces paramètres Z-Wave mais ne les expose pas sur le bus xAAL) ;
- exposer les devices constructeur sur le bus xAAL avec leur adresse xAAL et leur schéma xAAL ;
- faire le mapping entre les adresse xAAL et les adresses constructeur ;
  Notons que ce mapping peut être intéressant pour aider au paramétrage de la base de métadonnées (par exemple, lorsque l'on rebranche le courant après avoir installé plusieurs équipements domotiques : tous vont apparaître dans le désordre sur le bus... il faut donc identifier qui est qui). On peut donc interroger la gateway sur ce mapping. Pour chaque device, ce mapping est typiquement une paire : adresse xAAL, adresse constructeur. D'un constructeur à l'autre, la notion d'adresse peut avoir une forme très variable (par exemple on a parfois la notion de groupe). Cependant, Json autorisant le typage dynamique, le format d'adresse pour chaque constructeur pourra être défini par la suite. (À minima c'est un dictionnaire qui contient la variable `protocolName`.)
- faire la traduction entre les fonctions des devices constructeur et les méthodes décrites dans les schémas xAAL ;
  Cette traduction est en principe assez directe, la gateway étant quasiment transparente de ce point de vue là.

Par construction une gateway apparaît comme un device composite. C'est un device composite un peu plus sophistiqué que le device composite simple décrit précédemment, mais c'est un device qui s'expose lui-même en tant que tel sur le bus xAAL.

Il y a le cas des gateway qui vont gérer des équipements constructeurs qui sont eux-mêmes de nature composite.

Prenons le cas du «Capteur d'ouverture 3 en 1 FIB-FGK-101».[4] Cet équipement Z-Wave fait : contacteur de porte, thermomètre, relais de données digitales (pour y connecter une extension). La gateway Z-Wave va donc exposer sur le bus xAAL :

- un device contacteur de porte (avec un schéma "contacteur de porte" et sa propre adresse xAAL) ;
  Ce device a pour *parent* le device composite décrit ci-après.
- un device thermomètre (avec le schéma "thermometer" et sa propre adresse xAAL) ; Ce device a pour *parent* le device composite décrit ci-après.
- un device «digital data» (avec un schéma "digital data" et sa propre adresse xAAL) ;
  Ce device a pour *parent* le device composite décrit ci-après.
- un device composite (avec le schéma "composite" et sa propre adresse xAAL) ;
  Ce device composite a pour *embedded* les trois devices précédents.

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

De plus, il a pour *parent* la gateway décrite ci-après.

— un device gateway (avec le schéma "gateway" et sa propre adresse). Ce device composite a pour *embedded* les quatre devices précédents.
Il n'a pas de *parent*.

9.3 Device virtuel

Dans l'architecture fonctionnelle xAAL, les automates de scénarios sont le lieu privilégié pour implémenter les services domotiques "intelligents". Par exemple :

— à la réception d'une commande, déclencher tout une série d'action (p.ex. "soirée tv") ;
— ou à l'inverse, en corrélant tout un ensemble d'événements, synthétiser une information (p.ex. "présence").

Ce faisant, l'automate apparaît alors lui-même sur le bus xAAL comme étant un device en tant que tel. C'est un *device virtuel*, mais un device avec ses propres méthodes, notifications, variables d'état : bref, il est défini par un schéma et les autres entités peuvent interagir avec lui via le bus xAAL.

9.4 Device surchargeant un autre

Lors d'une installation domotique on peut avoir recours à des équipements multifonctions. Considérons par exemple le «Micro-module Z-Wave Interrupteur on/off FGS-211». [5] Cet équipement a une taille étudiée pour pouvoir se glisser dans une prise murale et commander n'importe quel appareil électrique (lampe, ventilateur, volet roulant, etc.).

Ce type de micro-module, de plus en plus courant, offre des fonctionnalités complexes. Voici ce que dit le catalogue à propos de celui-ci :

— Fonctions on/off (relais) commandée par Z-Wave ou par les boutons connectés.
— Peut utiliser 1 ou 2 boutons au choix (le deuxième bouton peut commander un ou plusieurs autres modules associés par Z-Wave).
— Les boutons peuvent être bistables (on/off) ou monostables (impulsion par bouton-poussoir).
— Retour d'état vers Zwave.

Par conséquent, la gateway Z-Wave qui le prend en charge va exposer sur le bus xAAL les devices suivant :

— un device de type relais avec retour d'état (avec le schéma "relay", et sa propre adresse), correspondant à l'appareil branché dessus et ainsi commandé par ce micromodule (mais à ce niveau là on ne dit pas si c'est plus une lampe qu'un ventilateur ou un volet roulant) ;
— un device de type bouton (soit avec le schéma "bistable", soit le schéma "monostable" suivant la configuration choisie), et correspondant au deuxième bouton, l'éventuel premier bouton étant utilisé pour la commande manuelle du relais ;
— et accessoirement un device composite embarquant les deux précédents.

Par contre, il nous manque l'information de configuration comme quoi derrière ce module c'est bien une lampe que l'on contrôle et non pas un ventilateur ou un volet roulant.

Pour ce faire, on va avoir recours à une entité xAAL, un nouveau device qui va exposer le type "lampe" et faire la traduction entre les actions sur ce device et des actions sur le device "relais à retour d'état" décrit plus haut. Ce nouveau device est un peu virtuel, mais surtout il vient surcharger le device générique. On peut alors le qualifier de *device de surcharge*. (Et d'un point de vue xAAL, il aura comme parent le device générique.)

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

Notons que, puisque c'est la gateway qui expose ce nouveau device de surcharge, cette gateway n'est plus complètement transparente dans notre architecture : il faut la configurer pour déclarer comment sont spécialisés les devices génériques. Nous estimons que ce paramétrage là doit se faire au niveau de la gateway et non pas au niveau (par exemple) de la base de métadonnées. En effet, il s'agit du paramétrage du fonctionnement intrinsèque de device, et non pas d'un attribut dont on vient l'enrichir.

En toute rigueur, une gateway qui doit gérer un micromodule générique va donc exposer sur le bus xAAL deux devices : l'un générique, l'autre de surcharge. Cependant, pour des questions de simplicité de mise en oeuvre (et aussi parce que l'on sait bien comment sont les développeurs), il est tout à fait acceptable que la gateway fasse l'économie du device générique et n'expose sur le bus xAAL que le device de surcharge.

9.5 Device caché

Certains devices n'ont pas vocation à être présentés dans les interfaces utilisateurs. C'est le cas des gateways, composites, moteurs de scénarios, bases de métadonnées, caches, interfaces elles-mêmes, les devices génériques qui sont masqués par un device de surcharge, etc.

Il ne faut pas faire d'a priori sur le caractère caché ou non des devices, cela doit rester un tag explicite dans la base de métadonnées (éventuellement avec une valeur par défaut suivant la classe du schéma, mais qui puisse être modifié par la suite).

Les interfaces utilisateurs connaissent tous les devices (cachés ou non), mais choisissent de ne présenter que ceux qui ne sont pas cachés, sauf si l'utilisateur le demande explicitement (un peu comme dans son gestionnaire de fichiers sur son PC on peut choisir "afficher les fichiers cachés").

Cette information (préciser qu'un device est caché ou non) pourraît être une propriété purement locale à une IHM, et être gérée localement par l'IHM. Cependant on estime que c'est une caractéristique propre à l'installation domotique et qui doit donc être stockée dans la base de métadonnées et disponible à tous.

10 Remerciements

Nous tenons à remercier Corinne Le Moan et Florian Le Nestour pour leur participation active aux premiers travaux sur xAAL.

A Brouillons de schémas

À l'heure actuelle, la formalisation des schémas n'a pas été encore tranchée. Malgré tout, voici quelques idées générales sur ce que pourraient contenir ces schémas.

```
Notes:
- p:internal parameters
- v:state variables
- m:methodes
- n:notifications


generic
=======
p:devType
p:address
p:parent

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

m:isAlive()
m:getDescription(out vendorId, out productId, out version, out parent)
m:getState(out v:*)
m:getBusConfig(in/out busAddr, in/out busPort,  in/out TTL)



lamp.basic (extends generic)
==========
m:powerSwitch(in/out bool target)


lamp.queryable (extends lamp.basic)
==============
v:enlighten (bool)


lamp.dimmer (extends lamp.queryable)
===========
v:level (float, precent)
m:dimmer(in/out float level)


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

```
shutter.simple (extends generic)
==============
m:up()
m:down()
m:stop()


shutter.queryable (extends shutter.simple)
=================
v:position (open|closed|opening|closing)



door.simple (extends generic)
===========
v:position (open|closed)


door.locker (extends generic)
===========
v:position (locked|unLocked)
m:lock()
m:unLock()



media.simple (extends generic)
============
v:activity (play|pause|stop)
v:volume{left,right} (float percent / db ?)
m:play()
m:pause()
m:next()
m:previous()
m:stop()


media.spotify (extends media.simple)
=============
v:url
m:setUrl ??



composite.simple (extends generic)
================
v:embedded...


composite.gateway (extends composite.simple)
=================
do be done...
```