Protocole domotique bas niveau
xAAL - Version 0.1 bin

Christophe Lohr

Mars 2012

1 Introduction

Cette page décrit une proposition de protocole domotique de philosophie *bas niveau*.

- Protocole domotique: un protocole réseau (en l'occurrence sur IP) par lequel des équipements (capteurs, actionneurs, interfaces utilisateurs, moteurs de scénarios, etc.) peuvent interagir entre eux.
- Domotique: cela se passe au sein du domicile, ou d'un établissement plus ou moins grand. Des éventuelles interactions ou services à travers Internet sortent du cadre de ce protocole ci et doivent être élaborés sur la base d'un autre protocole à définir éventuellement (ou un à réutiliser).
- Bas niveau: on vise un protocole qui peut être codé dans un PIC (e.g. 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, auto-suffisant, dans un format contraint, avec une variabilité contrainte, un taille prédictible, etc.
- Conséquences: on manipule essentiellement des mots de 32bits. On s'aligne sur 32bits (donc, avec du bourrage s'il le faut).

2 Positionnement et motivation

Dans le monde des protocoles domotiques sur IP il existe déjà un certain nombre de choses.

- UPnP: conceptuellement très intéressant. Différents profils (multimédia, routeur, domotiques, etc.). Des schémas XML de description. De l'introspection. De la découverte automatique. On peut également annoncer son IHM, etc. Bref, hyper générique... un peu trop justement. Trop souple (il faut parser les schémas annoncés par un device pour découvrir l'url sur laquelle on

peut interagir avec lui). Parfois c'est l'ihm qui englobe la partie commande (i.e. dans l'ihm annoncé par le device, il faut récupérer le plugin pour son InternetExplorer pour commander le device!). Même pour des fonctionnements simple, on a du XML à parser... Trop lourd (pensez: un termomètre !)

- xAP: un protocole plus bas niveau. Malheureusement c'est du broadcast (ça n'existe plus en IPv6). Les implémentations collent des "hub" partout car les gars ont visiblement appris le réseau sur du .Net et n'ont pas vu l'option de socket SO_REUSEADDR (alors que ça existe tout de même sous Windows). Le gros problème étant qu'il n'y a pas de découverte. Le mécanisme d'adressage est compliqué: il englobe l'identifiant du device, son type de schéma, et l'id de sa gateway: résultat, on utilise beaucoup le "*"... Le choix d'un format textuel est également criticable (l'éternel problème en informatique des chaînes de longueur non prédictible) (AMHA).
- xPL: un fork de xAP... mais en pire...

3 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 défini également. Éventuellement se réserver une plage de ports.
- S'il y a plusieurs "réseaux domotiques" dans le domicile, ils utilisent sur des ports différents.
- Les messages auront une taille limitée par la taille max UDP (e.g. 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 certain PIC ne gèrent pas forcément très bien la fragmentation IP (e.g. la stack de base de Arduino). On verra plus tard. En attendant, on recommande seulement d'essayer de se limiter à 1500 octets. Si des services ont besoin d'échagner 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.

4 Définition d'un device

Un *device* a:

- Un *schéma*, dont la référence est codée en dur dans le device.

  - Un schéma est déterminié par 2 mots de 32 bits (ça devrait laisser assez de marge!)
  - Le premier mot de 32 bits désigne une classe de type de device (e.g. éclairage, chauffage, multimédia, etc.). C'est le **ClassID**
  - Le second mot de 32 bits désigne un type dans une classe donnée (e.g. dans la classe éclairage, on a une lampe on-off, une lampe avec graduateur, une boule a facettes, etc.). C'est le **TypeID**
  - L'identifiant de type [0xFFFFFFFF,0xFFFFFFFF] est réservé, et désigne tous les types de toutes les classes (*any*)
  - L'identifiant de type [0x0, 0x0] est réservé, et désigne le protocole domotique lui-même (c'est assez conceptuel, je vous l'accorde)
  - Et par exemple le couple [0xA2,0xFFFFFFFF] désigne tous les types de la classe 0xA2.
  - On se réserve une plage particulière "expérimentale" (i.e. lorsque quelqu'un fait une implémentation mais n'a pas encore obtenu un numéro par notre bureau d'attribution des numéros...). Par exemple la plage 0xFFFFFFxx/24 (en retirant 0x00 et 0xFF), et ceci pour la classe ainsi que pour le type.

- **DeviceID**: Un identifiant de device (ou adresse), un numéro unique sur le bus.
  - L'identifiant de device est un entirer non signé de 64 bits (codé sur 2 mots de 32 bits).
  - L'identifiant de device 0xFFFFFFFFFFFFFFFF est réservé. C'est l'adresse de broadcast et désigne tous les devices.
  - L'identifiant de device 0x0 est réservé. C'est l'identifiant du protocole domotique lui-même.
  - Comment est attribuée une adresse ?
    * soit codée en dur, en usine (les 2 premiers octets sont un numéro de constructeur attribué par notre bureau d'attribution des numéros, le reste est à l'appréciation du constructeur (numéro de série ou autre), façon OUI en Ethernet)
    * auto-générée (random) si le OUI (i.e. les 2 premiers octets) est 0xFFFF ?
    * mais vérifiée par un genre de "arp gratuit" par des requêtes *Is alive?* 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 (e.g. 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 fait température intérieure/extérieure/hydrométrie/etc.

- Un tel équipement est alors qualifié de *device composite*,
- et chacun de ses composant est un *device embedded*.
- Le device composite s'annonce (éventuellement) lui-même sur le bus avec son propre DeviceID, ClassID, TypeID. 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 id)
- Chacun des embedded est également annoncé sur le bus comme un device ordinaire, bus-natif, avec son propre DeviceID, ClassID, TypeID. Chacun est annoncé en précisant son parent, i.e. 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, i.e. décrit par un *schéma*.

- Le schéma donne un peu de sémantique à un device et décrit ses capacités: une liste d'actions possibles sur ce device (dans un schéma requête réponse), et une liste de variables d'état de ce device (le device annonce spontanément sur le bus tout changement de valeur).
- Une action est décrite par:
  - **ActionID**: un numéro, un entier 32bits non signé. On commence à 0. Les numéros sont consécutifs.
  - **Parameters**: une table de paramètres en entrée, éventuellement vide, a fournir lors de la requête
  - **SuccessCode**: un entier 32bit signé, retourné dans la réponse, qui code le résultat de la requête (on prend la convention C ? i.e.: positif c'est bien, négatif c'est pas bien; ou la convention shell: 0 succes, !0 erreur?)
  - **Results**: une table de paramètres en sortie, éventuellement vide, retourné dans la réponse à la reqête.
- Une variable d'état est décrite par:
  - **VariableID**: un numéro, un entier 32bits non signé. On commence à 0. Les numéros sont consécutifs.
  - **VariableType**: un numéro, un entier 32bits non signé qui code le type (voir plus bas)

- Concrètement un schéma peut être décrit par une grosse structure C dans un .h (on a dit que l'on était bas niveau!)
- Il y a une notion d'héritage entre schémas. (Donc, héritage à la C, avec des structures qui se recouvrent et des cast). On a une généalogie à 3 niveaux d'ancètres:
  - Un schéma générique commun à tous les devices du bus, que tout le monde doit implémenter:
    * Paramètres:
      · **ClassID** et **TypeID**: entiers 32bits non signés (le type de device)
      · **DeviceID**: un entier 64bits non signé (l'adresse du device)
      · **ParentID**: un entier 64bits non signé (le DeviceID du parent, ou 0x0)
      · Power ? ça serait intéressant que tout le monde ait ça ?
    * Actions:
      · **GetDescription**
      · Pas d'entrée
      · Retourne:
      · un VendorID (chaîne de caractères ou numéro attribué par le bureau?)
      · un ProductID: chaîne de caractères
      · une Version: chaîne de caractères
      · à priori le SuccessCode est toujours OK
      · **SetIPv4Config**:
      · Entrée:
      · Mode: un entier 32bits indiquant: manuel, automoatique (et le genre d'automatique DHCP, avahi, BOOTP, etc.), allumé, éteind, etc. Bref, à définir plus tard... Note: a priori une seulle interface réseau !
      · IPv4Addr: un entier 32bits non signé, l'adresse IPv4
      · IPv4Mask: un entier 32bits non signé, masque réseau
      · IPv4Gateway: un entier 32bits non signé, la passerelle
      · Sortie:
      · SuccessCode a préciser
      · **GetIPv4Config**:
      · Pas d'entrée
      · Sortie:
      · IPv4Addr, IPv4Mask, IPv4Gateway: des entiers 32bits
      · La même chose en IPv6
      · **SetBusIPv4Config**

· Entrée:
· BusAddr: un entier 32bits non signé, l'adresse multicast IPv4 du bus
· BusPort: un entier 32bits non signé (enfin, limité à 65535), le port du bus domotique
· Sortie:
· SuccessCode a préciser
· **GetBusIPv4Config**
· Pas d'entrée
· Sortie
· BusAddr: un entier 32bits non signé
· BusPort: un entier 32bits non signé
· La même chose en IPv6

  - Un schéma spécifique à chaque classe, qui hérite donc du schéma général
  - Un schéma spécifique à chaque type, qui hérite donc d'un schéma de classe

- Note: penser 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.

6 Définition d'un message

Rappelons que l'on adopte une philosophie bas niveau pour les messages...

- On aligne sur des mots de 32 bits (pas forcément compact car il y a parfois du bourrage ou des types trop grands, mais c'est plus facile à parser (boucler sur des read de 4 octets, et faire des casts).
- On adopte l'endianness du réseau (i.e. big endian): c.f. man byteorder; man endian (cela devient partie intégrante de notre spécification...)
- Un message est composé d'une partie *header* et d'une partie *body*. Le header est obligatoire, alors que la présence du body dépend du type de message.
- **Header**: **MsgType**, **SrcID**, **DestID**, **DeviceType**
  - **MsgType**: un entier 32bits non signé qui code le type de message (requête, réponse, notif). De plus, l'octet de poids fort (i.e. le premier que l'on voit passer, vu que l'on est en big endian) code la version du protocole domotique.
  - **SrcID**: entier 64bits non signés, le **DeviceID** de l'émetteur

- **DstID**: entier 64bits non signés, le **DeviceID** du destinataire, ou l'adresse de broadcast 0xFF...FF
- **DeviceType**: une paire d'entiers 32bits non signés, i.e. un **ClassID** et un **TypeID** tels que définit plus haut.
  * si MsgType correspond à un message de type requête, ce DeviceType est celui du destinataire à qui on veut s'adresser
  * si MsgType correspond à un message de type réponse ou notification, ce DeviceType est celui de l'émetteur

7 Type de messages

7.1 Généralités

- Le type de message est codé par un entier 32bits non signé
- L'octet de poids fort indique la version du protocole (e.g. ici on parle du protocole 0x1)
- Le bit de poids faible indique si c'est un message de type requête (0x1) ou de type réponse/notification (0x0). Ce bit permet donc de déterminer la portée du DeviceType de l'entête (respectivement celui de la destination, respectivement celui de la source).
- Un message de type requête:
  - DeviceType = ClassID, TypeID (je répète: celui de la destination)
    * soit 0xFFFFFFFF,0xFFFFFFFF (*any*): on s'adresse à tous les types de devices sur le bus
    * soit un ClassID précis, et un TypeID=0xFFFFFFFF: on s'adresse à tous les types d'une classe donnée
    * soit un ClassID précis et un TypeID précis: on s'adresse à un type bien déterminé
  - SrcID: le DeviceID de l'émetteur du message (peut être l'adresse 0x0 dans certains cas, par exemple pour gérer le genre d'arp gratuit, lorsque l'on n'a pas encore choisi un ID pour soi-même)
  - DstID: le DeviceID d'un device particulier à qui l'on adresse une requête, ou bien l'adresse de boradcast 0xFF..FF
- Un message de type réponse ou notification
  - DeviceType: celui du device qui émet la requête (et pas de any d'aucune sorte)
  - SrcID: le DeviceID de l'émetteur du message
  - DstID: le DeviceID du device qui a fait la requête et à qui l'on répond, ou bien l'adresse de boradcast 0xFF..FF, notamment pour les messages de notification

7.2 Détail des messages

0x 01 00 00 01 01 : requête de type "*Who is alive?*"

- Pas de body
- Sollicite des annonces de type "*Alive*" pour découvrir qui est actif sur le bus
- De manière globale: DstID broadcast, et DeviceType any
- Spécifiquement à un ClassID précis
- Spécifiquement à un ClassID+TypeID précis
- Spécifiquement à un DeviceID précis

0x 01 00 00 01 00 : annonce (ou réponse) de type "*Alive*"

- Pas de body
- Le device s'annonce spontanément sur le bus lorsqu'il est branché
- Le device s'annonce régulièrement (selon une périodicité de son choix, voir jamais)
- Le device s'annonce suite à une requête "Who is alive?"
- Le device peut choisir de ne pas s'annoncer s'il est en mode économie d'énergie, ou s'il ne sait pas traiter des requêtes "*Who is alive?*"

0x 01 00 00 02 01 : requête de notification d'état

- Pas de body
- Sollicite un message de notification d'état
- De manière globale, spécifique, ou entre les deux (cf plus haut)

0x 01 00 00 02 00 : message de notification

- Body : L'intégralité des variables d'état de device tel que définit dans son type
- Le device émet ce message spontanément (e.g. lorsque son état a changé)
- En réponse à une requête de status du device
- Le device peut choisir de ne pas répondre à la requête correspondante (e.g. s'il ne sait pas écouter le bus)

0x 01 00 00 03 01 : requête de type action

- Toujours de manière spécifique (pas de broadcast ou de any)
- Body:
  - Un RequestID: un entier de 64bits non signé (aléatoire)
  - Un ActionID: le numéro de la requête tel que définit dans le schéma

        * Une table de paramètres: les paramètres requis pour l'action en question, tel que définit dans le schéma (Absent si l'action ne requiert pas de paramêtre)

0x 01 00 00 03 00 : réponse à une action

- Répond spécifiquement à l'émetteur de la requête (i.e. sur son DeviceID à lui).
- Body:
    * RequestID: recopie celui de la requête
    * ActionID: rappel le numéro d'action de la requête
    * SuccessCode: un entier 32bits signé. Toujours présent.
    * Une table de variables de retours tel que définit dans le schéma (Absent si l'action ne fournit pas de variables en retour)

- Note: 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 aurgument 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 parraît naturel que les équippements qui ne peuvent honorer la requête puissent choisir de se taire plutôt que de polluer le bus avec des messages d'erreur.

7.3 Cas d'erreur possibles

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. (Une prochaine spécification pourra définir des *best practices*...)

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

- Un device reçois 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 reçois une requête d'action mais le numéro de l'action n'existe pas dans son schéma. Il peut donc facilement identifier le problème et ignorer la requête.
- Un device reçois une requête d'action 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 l'action 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.

8 Devices particuliers

- Un (ou plusieurs) référentiel de type/schémas, i.e. une base de données qui stocke les schémas associés à quelques types (et pas nécessairement tous)
  - On l'interroge sur un identifiant de type, il nous renvoie son schéma, s'il le connaît
  - Associé a un schéma, il peut y avoir des *méta-données*: dates ou numéro de version, url pour de la documentation, url pour l'ihm (pointe vers des fragments de html, éventuellement hébergés sur internet), etc.
  - Ce device a également un schéma (à rédiger). Ce schéma doit donc être capable de décrire comment annoncer des schémas (on est donc un peu méta)
- Un (ou plusieurs) référentiel de localisation
  - 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 donnée sans pour autant être présent sur le bus

- Un ou plusieurs moteurs de scénarios
  - ...
- Il peut y avoir plusieurs instances de ces devices là sur le bus (par exemple, même s'il y a un PC sur le bus qui sait tout sur tout le monde, un device un peu musclé peut embarquer également un référentiel de schéma pour annoncer son propre schéma, ou un référentiel de localisation pour annoncer sa propre config). Ainsi, en réponse à une requête, il peut naturellement il y avoir plusieurs réponses fournies par plusieurs référentiels. Ces réponses peuvent malheureusement être discordantes. La spécification n'impose toujours pas de manière de traiter des incohérences... chacun fait au mieux.
- De même, on peut avoir des scénarios qui se bouffent le nez entre eux (l'un allume la lampe, l'autre l'éteind), ou s'interbloquent, etc. Les conflits sont sencés être traités autant que possible off-line lors de l'édition de scénarios (la spec impose un model-checker à base de Réseau de Pétri coloré temporisé (non je déconne, c'est de l'humour de toulousain)), ou pourquoi pas introduire un device de supervision.
- Est-ce qu'il vaut introduire spécifiquement une notion de device ihm ? Que mettre dans son schéma ?

9 Types des paramètres et variables

0x0 : entier sur 32bits non signé

0x1 : entier sur 32bit signé

0x2 : entier sur 64 bits non signé

0x3 : entier sur 64 bits signé

0x4 : booléen (sur 32bits!) 0=False, !O=True

0x5 : float simple précision (32bits)

0x6 : float double précision (64bits)

0x7 : chaîne de caractères

- 1 mot de 32 bits pour coder la longueur en octets
- x mots de 32 bits, et bourrage pour s'alligner sur 32bits

0xA : suite d'octets

- idem

0xB : un tableau

- 1 mot de 32 bits pour coder le nombre d'éléments
- 1 mot de 32 bits pour coder le type des éléments
- x mots de 32 pour chacune des valeurs des éléments

0xC : une table (ou une structure)

- 1 mot de 32 bits pour coder le nombre d'éléments
- x fois (pour chaque élément)
  - 1 mot de 32 bits pour coder le type de l'élément
  - x mots de 32 pour la valeur de l'élément

0xD : une liste de structures

- la suite se termine par le type [0xFFFFFFFF]

0xFFFFFFFF : réservé

- (Cela code un marqueur de fin de liste.)