

# Service UPnP pour dispositifs autonomes

par Vincent Hourdin

Fonction : Ingénieur en informatique de l'École Polytechnique Universitaire de Nice – Sophia Antipolis

par Stéphane Lavirotte

Fonction : Docteur en Informatique de l'Université de Nice Sophia Antipolis, Maître de Conférences à l'IUFM Célestin Freinet – Académie de Nice

par Jean-Yves Tigli

Fonction : Docteur en Informatique de l'Université de Nice Sophia Antipolis Maître de Conférences à l'École Polytechnique Universitaire de Nice – Sophia Antipolis

<b>1</b>	<b>LES PROTOCOLES DE DECOUVERTE DE DISPOSITIFS.....</b>	<b>3</b>
1.1	SLP : <i>SERVICE LOCATION PROTOCOL</i> .....	3
1.2	BONJOUR (RENDEZVOUS) .....	4
1.3	BLUETOOTH.....	4
1.4	UPNP .....	4
1.4.1	<i>Adressage</i> .....	5
1.4.2	<i>Recherche et découverte</i> .....	5
1.4.3	<i>Présentation</i> .....	7
1.5	CONCLUSION.....	7
<b>2</b>	<b>UPNP : UNE ARCHITECTURE ET UNE INTERFACE NORMALISEE.....</b>	<b>7</b>
2.1	EXEMPLES DE DISPOSITIFS UPNP .....	8
2.1.1	<i>Serveurs multimédia</i> .....	8
2.1.2	<i>Routeur et configuration NAT</i> .....	8
2.2	SERVEUR UPNP DIT « DISPOSITIF UPNP » .....	8
2.2.1	<i>Description, Devices et Services</i> .....	9
2.2.2	<i>Contrôle</i> .....	9
2.2.3	<i>Notification d'évènements</i> .....	9
2.3	CLIENT UPNP DIT « POINT DE CONTROLE ».....	10
<b>3</b>	<b>LA PILE UPNP EN DETAILS .....</b>	<b>10</b>
3.1	ADRESSAGE : DHCP ET AUTO-IP.....	10
3.2	DECOUVERTE : SSDP .....	11
3.2.1	<i>Device disponible : ssdp:alive</i> .....	11
3.2.2	<i>Device indisponible : ssdp:byebye</i> .....	12
3.2.3	<i>Recherche de devices : ssdp:discover, toujours sur HTTPMU</i> .....	12
3.2.4	<i>Réponse au ssdp:discover</i> .....	12
3.3	FICHIERS DE DESCRIPTIONS .....	13
3.4	CONTROLE : SOAP .....	16
3.4.1	<i>Invocation</i> .....	16
3.4.2	<i>Réponse</i> .....	17
3.5	ÉVÈNEMENTS : GENA .....	18
3.5.1	<i>Opérations sur les abonnements</i> .....	18
3.5.2	<i>Évènements</i> .....	20
3.6	PRESENTATION : HTTP .....	20
<b>4</b>	<b>SDK, OUTILS, ET EXEMPLES D'UTILISATION.....</b>	<b>21</b>
4.1	OUTILS.....	21
4.2	SDK INTEL POUR WINDOWS.....	22
4.3	LIBUPNP PORTABLE .....	24
4.3.1	<i>Descriptions</i> .....	24

4.3.2	<i>Serveur</i> .....	27
4.3.3	<i>Point de contrôle</i> .....	38
<b>5</b>	<b>PERSPECTIVES ET CONCLUSION .....</b>	<b>42</b>
5.1	VERS DES SERVICES WEB POUR DISPOSITIFS.....	42
5.2	CONCLUSION.....	44

« *Plug and Play* » (*PnP*) ou littéralement « *on branche et ça marche* », caractérise la facilité d'installation d'un nouvel équipement dans un système informatique. Techniquement, le système d'exploitation reconnaît le périphérique que l'on vient d'ajouter à l'ordinateur, retrouve le pilote nécessaire pour le faire fonctionner ou demande de charger ce pilote et lance le travail après avoir réadapté ses paramètres pour tenir compte du nouveau dispositif. L'installation du matériel est ainsi grandement simplifiée par la configuration automatique des paramètres du pilote tels que l'interruption utilisée, la plage des ports d'entrées/sorties utilisés, etc.

« *Universal Plug and Play* » (*UPnP*) reprend les concepts de *PnP* pour les étendre à tout le réseau, facilitant la découverte et le contrôle de dispositifs, tels qu'une imprimante réseau, un routeur ADSL ou tout autre équipement périphérique maintenant connecté au réseau local. Cette technologie n'est pas la seule à proposer une telle approche. Dans la première partie de ce document nous comparerons les grandes caractéristiques d'*UPnP* avec des technologies plus ou moins proches telles que Bonjour, *SLP* et *Bluetooth* notamment en ce qui concerne les protocoles de recherche et de découverte utilisés. Nous verrons ainsi qu'une des caractéristiques forte et spécifique à *UPnP* repose sur l'utilisation de protocoles très proches de ceux déjà éprouvés dans le domaine des Services Web. Dans une seconde partie nous présenterons l'architecture générale d'*UPnP*, décomposée en dispositifs *UPnP* (serveurs) et points de contrôle (clients) sur le réseau et les interfaces associées normalisées. Nous détaillerons dans la troisième partie la pile protocolaire *UPnP* avant de nous attarder sur la mise en oeuvre logicielle d'*UPnP* dans la quatrième partie. Enfin dans le cadre de la conclusion nous soulignerons les perspectives ouvertes par cette nouvelle technologie qui, au-delà de sa vocation première et de par sa proximité avec les Services Web, pose clairement les bases d'une extension vers des Services Web pour Dispositifs.

## 1 Les protocoles de découverte de Dispositifs

Dans l'optique de faciliter la configuration de matériel ou plus simplement de découvrir des services sur un réseau, *UPnP* n'est pas la seule solution. D'autres protocoles de découverte existent, chacun ayant des fonctionnalités propres. La vue d'ensemble qui suit nous permet de replacer *UPnP* dans son contexte, et d'en extraire ses spécificités.

### 1.1 SLP : Service Location Protocol

*SLP* est un protocole de découverte de services développé par le groupe de travail IETF SrvLoc depuis 1999. Son but est de définir un standard indépendant des fabricants pour les réseaux TCP/IP. Il peut fonctionner aussi bien dans un petit réseau grâce à l'utilisation de la diffusion pour la découverte, que dans un réseau d'entreprise en utilisant les annuaires de services. C'est un des rares protocoles de découverte de service qui peut utiliser ces deux modes.

De plus, *SLP* permet de rechercher des services avec des expressions booléennes conformes à la syntaxe LDAP, en utilisant les variables définies dans des schémas de descriptions de classes de dispositifs. Par exemple pour rechercher une imprimante, un prédictat peut être la vitesse minimale d'impression ou la capacité d'imprimer en couleurs. Dans tous les cas, le résultat de la recherche fournit des informations sur le matériel et ses capacités, et sa localisation géographique. Le serveur d'impression CUPS (*Common Unix Printing System*) peut utiliser *SLP* pour découvrir les imprimantes compatibles. Solaris 8, le système de SUN, utilise aussi *SLP*.

SLP a d'autres avantages : l'annuaire est découpé en scopes (groupes), qui peuvent avoir chacun une clé secrète pour tous les services qu'ils contiennent, ajoutant ainsi la notion d'authentification et de cryptage de données. Il est possible de renseigner un DHCP avec l'adresse de l'annuaire de service. En voulant obtenir un adressage, un client obtiendrait par la même occasion cette adresse, facilitant la découverte des dispositifs de son nouveau réseau.

## 1.2 Bonjour (Rendezvous)

Bonjour est une implémentation par Apple du standard ZeroConf. Cette architecture permet de découvrir automatiquement les ordinateurs, les équipements et les services dans les réseaux IP de petite taille, sans se baser sur des serveurs DHCP ou DNS existant. Pour cela, il utilise d'autres technologies : la configuration automatique d'adresse IP, MDNS (*Multicast DNS*) et DNS-SD (*DNS Service Description*) qui joue le rôle d'un annuaire distribué. MDNS permet d'assigner un nom d'hôte différent de ceux utilisés par les autres périphériques du réseau, DNS-SD assigne un nom à chaque service et permet aussi de découvrir les services du réseau, en utilisant la diffusion UDP (*multicast*).

Bonjour est utilisé dans les versions récentes de Mac OS X (à partir de 10.2) pour partager les imprimantes et des fichiers sur un réseau d'ordinateurs. Avant Bonjour, Mac OS X utilisait SLP, qui est toujours présent dans ce système.

## 1.3 Bluetooth

La pile Bluetooth dispose du protocole de recherche Bluetooth SDP. Il permet de rechercher une classe de service, des attributs de services, et de naviguer dans les services disponibles. Ce protocole est spécialement conçu pour les dispositifs Bluetooth peu complexes. Il ne s'occupe que du problème primaire de découverte de service et d'établissement de canal de communication. Il ne dispose pas de système d'annuaire, ni de système de notification ou d'abonnement, ni de système assurant qu'un service est toujours visible. C'est pourquoi on utilise d'autres protocoles de découverte de services au-dessus de Bluetooth SDP, tel que Salutation ou Jini.

Avec Bluetooth, les services découverts sont définis sous forme de profils. Ils définissent un type de dispositif, et donc le type de communications qu'il supporte. Le protocole Bluetooth établit un canal de communication RF entre le périphérique maître et le point d'accès, après avoir effectué une authentification par code PIN. Ce dernier peut être crypté lors d'une communication sécurisée.

## 1.4 UPnP

La technologie UPnP™ a été créée en 1999 par l'UPnP Consortium, démarré par Microsoft. Le Forum UPnP comporte plus de 700 membres industriels, et cette technologie est de plus en plus intégrée aux dispositifs (imprimantes, routeurs, commutateurs, lecteurs de DivX ou de DVD, ...). UPnP utilise les technologies réseaux suivantes : IP (éventuellement IPv6), TCP mais aussi UDP nécessaire à la recherche et découverte par diffusion pour la couche transport. Mais UPnP utilise aussi les technologies recommandées par le W3C : HTTP pour la couche session et XML pour la couche présentation.

UPnP permet à des clients (**points de contrôle**) de découvrir ou rechercher des dispositifs (**serveur UPnP sur les dispositifs**) sur un réseau local, par diffusion. La recherche peut contenir des filtres tels qu'un type de dispositif, un type de service ou encore un identifiant unique. UPnP ne se limite pas au problème de découverte de dispositifs, mais fournit aussi les protocoles pour communiquer avec eux.

#### 1.4.1 Adressage

La phase d'adressage permet à un nouveau dispositif d'obtenir une adresse et une configuration réseau. C'est la première étape du lancement d'un serveur UPnP, sans laquelle il ne pourrait communiquer. Cependant elle ne dépend pas des protocoles définis ou normalisés par UPnP, en l'occurrence DHCP et Auto-IP, mais doit être fait d'une manière décrite par la norme UPnP.

#### 1.4.2 Recherche et découverte

Pour connaître l'existence d'un serveur UPnP ou tout autre système de recherche, deux moyens sont possibles :

- Par recherche : les clients effectuent une méthode de recherche par diffusion (*multicast* ou *broadcast*). Les serveurs répondent alors seulement au client (*unicast*) qui a effectué la recherche, en indiquant les informations pour les contacter.

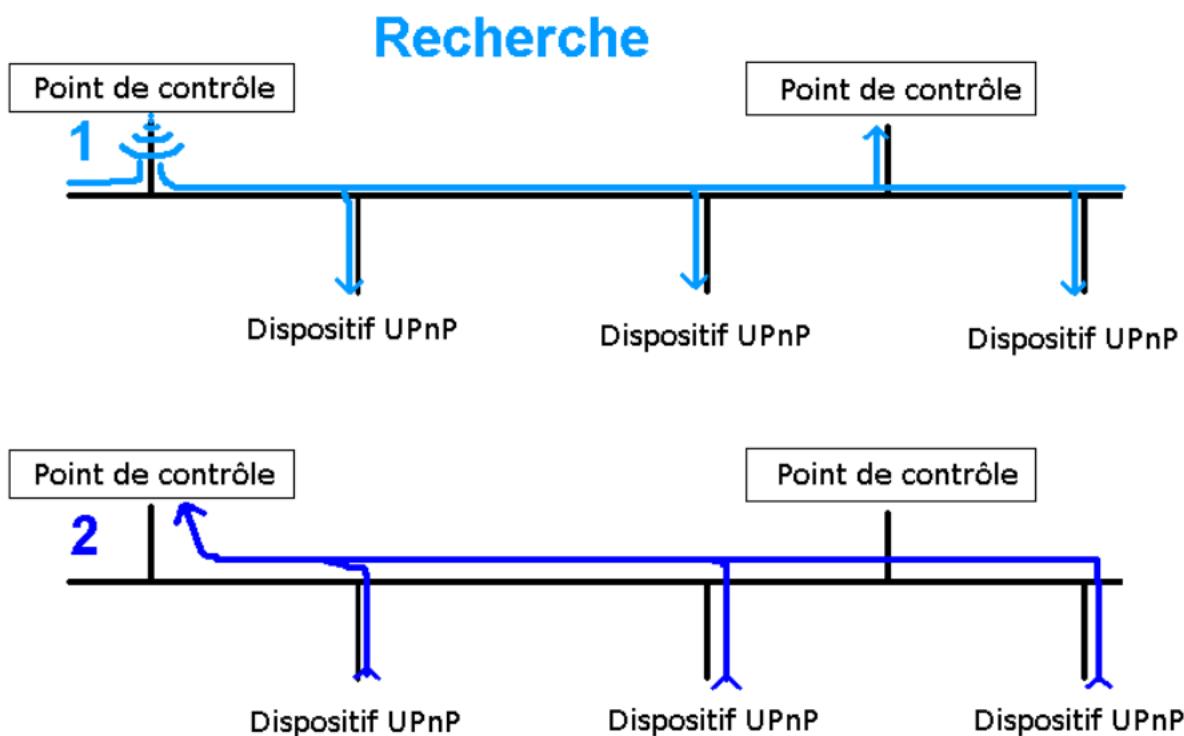


Figure 1 - Recherche de services sur un réseau local.

- Par découverte : les serveurs annoncent périodiquement leur présence par diffusion. Les clients découvrent alors les services et apprennent par cette annonce l'existence des dispositifs.

## Découverte

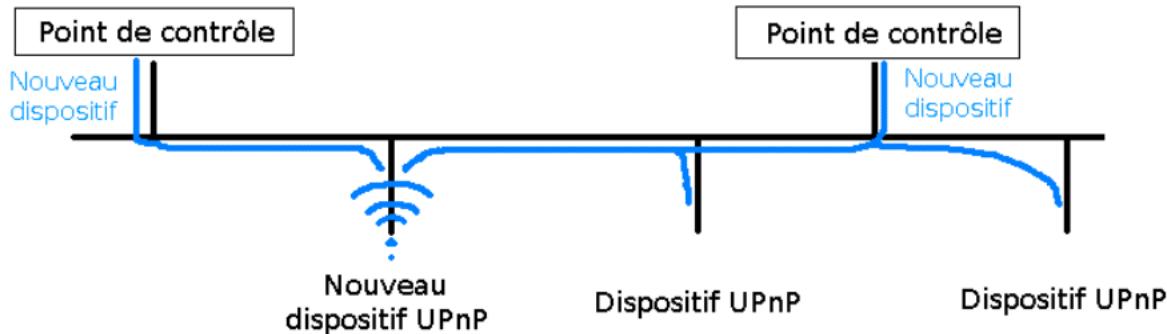


Figure 2 - découverte de services sur un réseau local.

Lorsqu'un serveur UPnP est lancé, il annonce sa présence sur le réseau local par un ou plusieurs paquets *multicast* (HTTPMU : HTTP Multicast sur UDP) définis par SSDP (*Simple Service Discovey Protocol*) et GENA (*General Event Notification Architecture*). La notification est de type *ssdp:alive*, et contient une URL pointant vers l'entité définie, qui peut être un *Root Device*, un *Device* ou un *Service*, et un UUID (*Universally Unique Identifier*). Ce paquet est envoyé sur l'IP de *multicast* 239.255.255.250 sur le port 1900. Cette adresse et ce port sont réservés à UPnP par l'IANA (*Internet Assigned Numbers Authority*).

Ce message de présence contient aussi un temps maximal de validité (qui devrait être supérieur ou égal à 1800 secondes pour limiter le trafic réseau), temps au bout duquel l'existence du service ne sera plus garantie pour les points de contrôle. L'annonce sera ré-émise avant la fin de ce temps maximal, permettant de plus aux nouveaux points de contrôle de connaître l'existence du serveur.

SSDP permet aussi de rechercher activement les services. Les messages de présence des serveurs permettent aux points de contrôle de les rechercher passivement, mais ces derniers peuvent aussi émettre un message *multicast* (toujours HTTPMU) contenant un filtre pour un type d'entité pour effectuer une recherche. Les serveurs UPnP recevant ces messages devront répondre en HTTPU (HTTP sur UDP) à l'initiateur de la demande, avec un message contenant les mêmes informations que les messages de présence.

Lorsqu'un serveur UPnP s'arrête, il doit envoyer, dans la limite du possible, un ou plusieurs messages *ssdp:byebye* par diffusion, qui indiquent aux points de contrôles que le serveur n'est plus disponible. En cas d'incident réseau ou de défaillance d'un dispositif, et que les points de contrôle ne reçoivent plus les messages SSDP, les serveurs seront automatiquement considérés comme perdus lorsque le temps de validité sera écoulé.

Les méthodes de diffusion sont valables sur un réseau Ethernet IP grâce à l'UDP, mais la couche de transport ne doit pas être une limite. Avec d'autres protocoles ou média tels que Bluetooth, WiFi, InfraRouge ou I<sup>2</sup>C, les *multicast* ou *broadcast* sont souvent gourmands en énergie et en bande passante [4]. Une solution à ce problème est de séparer la partie transport de la partie applicative, en dédiant une couche de transport optimale à chaque protocole.

Plutôt que de séparer les couches, une autre solution consiste à disposer de relais

ou ponts entre les protocoles. Ils permettent d'agrandir le réseau de services, en étendant le réseau classique à celui des protocoles moins bien supportés d'ordinaire, qui souvent ont un format propriétaire de communications. On peut aussi s'en servir comme relais de paquets *multicast* pour les réseaux IP, si on ne se sert pas de répertoire de services sur un réseau de taille moyenne.

#### 1.4.3 Présentation

Quand un point de contrôle découvre un serveur UPnP, il obtient une URL pointant vers un fichier de description du serveur. Dans ce fichier, dont le format et les caractéristiques seront expliqués dans la partie suivante, un champ correspond à l'URL de la page de présentation principale du dispositif. Cette page est au format HTML, ce qui signifie qu'elle pourra être vue par tout navigateur Internet classique. Pour offrir cette fonctionnalité, le serveur UPnP contient un serveur HTTP. C'est en fait un détournement des fonctionnalités premières d'UPnP, qui utilisent le protocole HTTP pour toutes les communications, comme nous l'avons vu avec la découverte par exemple.

### 1.5 Conclusion

Voici un tableau récapitulatif des fonctionnalités des protocoles de recherche et de découverte permettant une comparaison facilitée des protocoles présentés.

	Bonjour	SLP	Bluetooth	UPnP
Découverte	x	x	x	x
Recherche	x	x	x	x
Événements				x
Indépendance langage	x	x	x	x
Transport	IP	IP	Bluetooth	IP
Fonctionnalités non-standards	Configuration automatique du réseau.	Annuaire interrogeable avec expressions booléennes. Adresse de l'annuaire connue par DHCP. Scope de services : séparation, authentification et cryptage.	Profils : classes de dispositifs. Code PIN, cryptage.	Recherche par types de dispositifs, services ou identifiant unique. Pages HTML.

## 2 UPnP : une architecture et une interface normalisée

Le Forum UPnP a défini des spécifications pour certains dispositifs comme : les imprimantes, les routeurs, les commutateurs, les lampes, les systèmes audio/vidéo, la gestion de température et de ventilation d'une maison et bien d'autres dispositifs électroniques.

Ces spécifications portent sur les fichiers de description des serveurs.

Outre le fait de pouvoir communiquer avec un dispositif par sa page de présentation, UPnP définit des actions invocables à la façon des Services Web, et un système d'abonnement aux événements. C'est principalement ces services (et la liste des

actions et des variables) sur lesquels portent les normalisations des descriptions définies par le Forum UPnP. Les systèmes voulant exploiter des dispositifs appartenant à ces catégories connaissent donc les services avec lesquels ils pourront dialoguer.

## 2.1 Exemples de dispositifs UPnP

Contrairement à l'utilisation des pages de présentation des dispositifs, l'utilisation des fonctionnalités de services d'UPnP requiert un client (point de contrôle) spécifique à chaque application. L'avantage est qu'un point de contrôle pourra communiquer avec tous les matériels du même type, peu importe leur constructeur.

Pour reprendre l'exemple de l'imprimante cité plus haut, si un point de contrôle complet était déployé dans les systèmes d'exploitation, il n'y aurait plus besoin de pilotes pour imprimer, les services fournissant toutes les fonctionnalités d'impression les plus utilisées.

Bien sûr, les scénarios ne se limitent pas à l'utilisation des imprimantes. Voyons quelques autres exemples tout aussi révélateurs.

### 2.1.1 Serveurs multimédia

Si on possède un magnétoscope numérique relié au réseau conforme à la norme UPnP, on peut effectuer des opérations à distance avec son ordinateur équipé d'un point de contrôle, telles que la programmation d'un enregistrement par exemple. UPnP définit aussi des normes de contrôle de flux audio/vidéo, de sorte que l'on peut commander le magnétoscope pour qu'il envoie ses flux sur tout autre support compatible UPnP Audio/Vidéo.

On peut donc transformer ses divers équipements en véritable plate-forme décentralisée de rendu multimédia. On peut par exemple visionner sur un ordinateur équipé d'un logiciel de rendu compatible UPnP un film enregistré sur son magnétoscope numérique ou en lecture sur une platine DVD ou regarder sur sa télévision UPnP les films stockés sur le disque dur de l'ordinateur.

### 2.1.2 Routeur et configuration NAT

Les routeurs que nous installons dans nos maisons pour accéder à Internet doivent le plus souvent traduire les IP du réseau local vers l'IP Internet publique (on parle alors de NAT : *Network Address Translation*). Les ordinateurs du réseau ne sont plus librement connectés à Internet et ne peuvent pas recevoir des données sur un port et un protocole donné sans que la passerelle ne l'ait autorisé. C'est souvent une tache fastidieuse lorsque le nombre d'ordinateurs du réseau grandit, surtout si plusieurs veulent rediriger les mêmes ports.

UPnP définit la spécification *Internet Gateway Device* qui est implémentée dans les routeurs, dans le but de simplifier la gestion de redirection de ports vers le réseau interne. Les logiciels s'exécutant sur les ordinateurs du réseau local ont la possibilité de demander au routeur l'ouverture d'un port. Cette fonctionnalité est très utilisée de nos jours, pour les applications point à point, pour transférer des fichiers ou des flux vidéos ou bien encore pour jouer sur Internet.

## 2.2 Serveur UPnP dit « dispositif UPnP »

Après avoir rapidement étudié les fonctionnalités premières d'UPnP, nous allons maintenant comprendre celles mises en jeu dans les utilisations plus avancées : la description, le contrôle et les évènements.

### 2.2.1 Description, Devices et Services

L'URL donnée par les serveurs dans les phases de recherche et découverte pointe vers un fichier XML de description du serveur UPnP. Ce fichier contient des informations sur le constructeur de l'appareil ou du serveur UPnP, telles que son nom, l'adresse de son site Web, le nom et la version du dispositif, un numéro de série... Les informations pertinentes pour UPnP sont les champs `deviceType` et `UDN` correspondant respectivement à un espace de nommage de *Devices* et un numéro d'identification unique à chaque *device*. Ils sont envoyés dans les messages d'annonce des serveurs, de sorte qu'à sa réception, on connaît le type du serveur sans télécharger le fichier de description.

Un serveur est composé de *Devices* et *Services*. L'architecture d'un serveur est un arbre : il y a toujours un *Root Device*, qui peut contenir des *Devices* imbriqués et des *Services*. Les *Devices* contiennent les *Services*, et fournissent des informations supplémentaires. Les *Services* sont les services au sens architecture orientée service, contenant une interface (liste d'actions et variables d'état) avec laquelle communiquer. Les actions sont les fonctions invocables des *Services*. Les variables d'état sont des variables pouvant émettre des événements lorsque leur valeur change (*evented variables*) ou représentant des arguments de fonctions. L'architecture d'un serveur est définie dans le fichier de description représentant l'arbre sous forme d'un fichier XML.

Chacun des *Services* contient cinq informations :

- `serviceType` : définition de l'espace de nommage de services ;
- `serviceId` : nom unique du service, en général celui vu par les utilisateurs ;
- `SCPDURL` : URL pointant vers la description du service ;
- `controlURL` : URL avec laquelle communiquer pour les invocations d'actions ;
- `eventSubURL` : URL de de souscription aux évènements.

Le Forum UPnP a standardisé des spécifications pour les descriptions des serveurs UPnP. Elles devront être implémentées pour qu'un dispositif puisse être défini comme compatible UPnP. Ces standards se trouvent sur le site Web du consortium (<http://upnp.org/>). On y retrouve les spécifications pour un serveur ou un générateur de rendu de média, une caméra numérique de surveillance, une chaufferie, une lampe, une passerelle Internet, un point d'accès aux réseaux sans fil, une imprimante simple ou une autre plus complexe, un scanner, une télécommande...

### 2.2.2 Contrôle

La récupération des fichiers de description des *Services* (les liens `SCPDURL`) fournit les informations sur les actions et les variables que chaque *Service* contient. En utilisant l'URL de contrôle (`controlURL`) avec le protocole SOAP (*Simple Object Access Protocol*), bien connu dans le monde des Services Web, on peut invoquer les actions et récupérer les valeurs des variables. L'accès direct aux valeurs des variables est cependant déconseillé par le consortium, au profit de l'utilisation d'actions qui retournent ces informations.

SOAP est un protocole d'appel de fonction distante (RPC) basé sur XML, transporté le plus souvent par HTTP.

### 2.2.3 Notification d'évènements

UPnP fournit un système de notifications d'évènements : dans la liste des variables

d'un Service, celles dont l'attribut `sendEvents` a pour valeur `yes` peuvent faire l'objet d'un envoi de message avertissant d'un changement de valeur. Pour recevoir ces informations, les points de contrôle doivent s'abonner à un Service précis, en utilisant l'`eventSubURL` du fichier de description du serveur. L'abonnement à un Service correspond en fait à recevoir un message contenant les valeurs de toutes les variables évènementielles du Service à chaque changement de valeur d'une variable. On ne peut donc pas s'abonner à une seule variable. La conception des Services sera influencée par ce fait : il vaut mieux séparer en plusieurs Services les variables qui n'ont aucun lien entre elles. Ainsi, le traitement des données sera plus simple, la charge réseau améliorée, et donc le temps de réponse sera meilleur. On peut ajouter que pour les dispositifs mobiles fonctionnant sur batterie, réduire le temps de traitement des données augmente le temps d'autonomie.

### 2.3 Client UPnP dit « Point de Contrôle »

Un point de contrôle dans un réseau UPnP est un contrôleur capable de découvrir et contrôler les autres dispositifs. Après la découverte, un point de contrôle peut :

- Récupérer la description du *Device* et obtenir la liste des *Services* associés.
- Récupérer les descriptions de *Services* pour les *Services* intéressants.
- Invoquer des actions pour contrôler le *Service*.
- Souscrire aux évènements d'un *Service*. À chaque changement d'état du *Service*, le serveur d'évènements enverra un évènement au point de contrôle.

Un point de contrôle est susceptible d'interagir avec tous les dispositifs ayant la même interface (description) ; ainsi, ce client peut être développé qu'une seule fois pour un grand nombre de serveurs, même de fabricants différents dans le cas d'une description UPnP conforme aux descriptions normalisées.

## 3 La pile UPnP en détails

Nous allons maintenant étudier plus en détail le format des données transitant sur le réseau, pour chacune des fonctionnalités présentées dans la partie précédente.

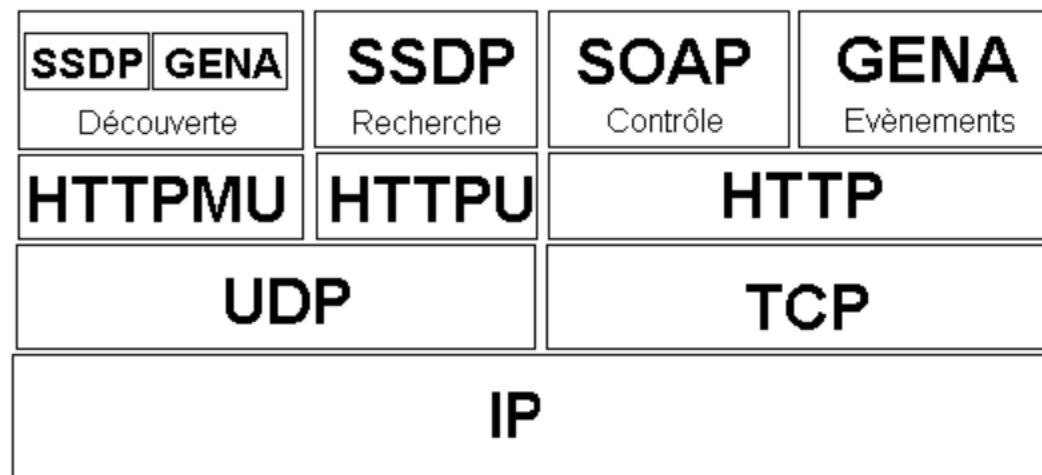


Figure 3 - pile des protocoles d'UPnP.

### 3.1 Adressage : DHCP et Auto-IP

L'adressage est réalisé en plusieurs étapes : le dispositif envoie un paquet `DHCPDISCOVER` sur le réseau, pour tester l'existence d'un serveur DHCP qui

pourrait lui affecter une adresse et lui fournir la configuration du réseau. S'il obtient une réponse, l'adressage est effectué en fonction de ces données. Par contre, si au bout d'un temps arbitraire, le dispositif n'a pas de réponse, il doit choisir une adresse IP automatiquement. Pour cela, on utilise Auto-IP, qui est défini par l'IETF. La méthode est simple : on choisit aléatoirement une IP sur la plage 169.254/16 et on teste si elle est déjà utilisée. Si c'est le cas, on en choisit une autre, et on recommence l'opération un nombre arbitraire de fois. Pour tester si une adresse IP est déjà utilisée, on envoie une requête ARP de demande de résolution de l'IP en question, et s'il y a une réponse ou une autre demande pour la même IP, l'adresse est définie comme utilisée.

Dans le cas où on utiliserait Auto-IP pour choisir son adresse IP, il faudrait périodiquement envoyer un paquet DHCPDISCOVER dans l'espoir de finalement trouver un serveur DHCP et se reconfigurer avec. Le dispositif peut choisir de garder un moment l'adresse IP auto-configurée pour garder les connexions actives.

L'adressage n'est pas réellement géré par la pile UPnP, mais plutôt par le système d'exploitation qui embarque le serveur UPnP. Nous n'en parlerons donc plus dans nos futurs exemples.

## 3.2 Découverte : SSDP

### 3.2.1 Device disponible : ssdp:alive

Lors de son lancement, puis périodiquement avant la fin de la validité de l'annonce, un serveur UPnP s'annonce au réseau, en émettant plusieurs paquets par diffusion. Un message sera envoyé pour annoncer le *Root Device*, un pour l'UUID, et un pour chaque *Device* et *Service* que le serveur comprend. En général, ces paquets sont envoyés en double, à cause de la non fiabilité de réception des paquets UDP.

Voici un paquet typique au format HTTP correspondant à l'annonce d'un *Root Device*, lors du lancement d'un serveur UPnP. Il est envoyé en UDP Multicast, formant finalement un paquet HTTPMU.

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
LOCATION: http://192.168.0.1:48009/
SERVER: Windows NT/5.0, UPnP/1.0, Intel CLR SDK/1.0
NTS: ssdp:alive
USN: uuid:39621552-6fad-4cea-bc33-3bf65cc10494::upnp:rootdevice
CACHE-CONTROL: max-age=1800
NT: upnp:rootdevice
Content-Length: 0
```

La ligne NOTIFY \* HTTP/1.1 est définie par GENA pour l'envoi de notifications et évènements, l'étoile signifiant que le message est général et non spécifique.

La ligne HOST dans tous les paquets SSDP aura pour valeur l'adresse de diffusion assignée par l'IANA (Internet Assigned Numbers Authority), 239.255.255.250:1900.

LOCATION est l'URL de base du serveur, l'emplacement du fichier XML de description SCPD.

Server fournit des informations sur l'OS sur lequel est lancé le serveur UPnP, et le dispositif ou la pile UPnP.

NTS doit être ssdp:alive.

CACHE-CONTROL définit le temps de validité de l'objet décrit, en secondes.

Les champs USN et NT sont liés et peuvent prendre plusieurs valeurs différentes ;

NT définit le type de notification (défini par GENA):

```
upnp:rootdevice
uuid:device-uuid
urn:schemas-upnp-org:device:devicetype:version
urn:schemas-upnp-org:service:servicetype:version
```

USN est défini par SSDP et prend pour valeur `uuid:device-uuid` concaténé avec `::` puis la valeur de NT, sauf si NT vaut `uuid:device-uuid`.

### 3.2.2 Device indisponible : `ssdp:byebye`

Lorsqu'un serveur se déconnecte, il doit envoyer un message pour avertir les points de contrôle qu'il ne sera plus disponible. Comme pour le `ssdp:alive`, il s'agit ici aussi de plusieurs messages.

Voici un exemple de paquet envoyé pour l'indisponibilité d'un service, HTTPMU aussi :

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
NTS: ssdp:byebye
USN: uuid:39621552-6fad-4cea-bc33-3bf65cc10494::urn:schemas-upnp-
org:service:light:1
NT: urn:schemas-upnp-org:service:light:1
Content-Length: 0
```

Les champs présents dans ce paquet sont semblables à ceux du `ssdp:alive`, sauf le NTS qui prend pour valeur `ssdp:byebye`.

### 3.2.3 Recherche de devices : `ssdp:discover`, toujours sur HTTPMU

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: ssdp:discover
MX: 3
ST: urn:upnp-schemas-org:device:InternetGatewayDevice:1
```

Les champs requis sont Host, Man indiquant le type de paquet SSDP, prenant pour valeur `ssdp:discover` obligatoirement, MX étant le nombre de secondes maximal durant lequel le point de contrôle attend une réponse des serveurs, et ST définit la cible de la recherche. Ce champ peut prendre plusieurs valeurs :

- `ssdp:all` pour rechercher tous les *Devices*
- `upnp:rootdevice` pour rechercher uniquement les *Root Devices*
- `uuid:device-uuid` pour rechercher un *device* d'un `uuid` particulier
- `urn:upnp-schemas-org:device:device-type:version` pour rechercher les devices du type *device-type* et d'une version précise
- `urn:upnp-schemas-org:service:service-type:version` pour rechercher les Services du type *service-type* et d'une version précise mais il ne peut y avoir qu'un champ ST par paquet.

La ligne M-SEARCH \* HTTP/1.1 est définie par SSDP pour signifier qu'on effectue une recherche générale, et que tous les serveurs UPnP doivent écouter le message.

### 3.2.4 Réponse au `ssdp:discover`

Voici un exemple de réponse de *Root Device* à une recherche. Ce message est envoyé seulement à celui qui a fait la requête, en HTTPU (HTTP sur UDP).

```

HTTP/1.1 200 OK
ST: upnp:rootdevice
EXT:
SERVER: Windows NT/5.0, UPnP/1.0, Intel CLR SDK/1.0
USN: uuid:c6f29f0f-adc5-460c-bc1c-c84c142f461a::upnp:rootdevice
CACHE-CONTROL: max-age=1800
LOCATION: http://192.168.0.1:48009/
Content-Length: 0

```

On voit un message typique de HTTP, avec le code 200 et le OK au début de la réponse. Les champs ST et USN sont liés de la même manière que NT et USN dans les paquets `ssdp:alive`.

### 3.3 Fichiers de descriptions

Les fichiers de descriptions des serveurs UPnP sont au format XML. Un point de contrôle les télécharge à l'adresse indiquée dans le champ LOCATION des messages SSDP, par une requête HTTP GET classique (sur TCP). Ils contiennent les informations du constructeur, l'architecture des *Devices* imbriqués et des *Services* qu'ils fournissent. Dans l'exemple ci-dessous, les données qu'il faut modifier sont en italique. Cependant, certaines sont optionnelles (`URLBase`, `manufacturerURL`, `modelURL`, `UPC`, `iconList`, `deviceList`) et d'autres recommandées, mais pas obligatoires (`modelDescription`, `modelNumber`, `serialNumber`, `presentationURL`).

```

<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <URLBase>URL de base pour toutes les URL relatives</URLBase>
  <device>
    <deviceType>urn:schemas-upnp-org:device:deviceType:v</deviceType>
    <friendlyName>titre court compréhensible</friendlyName>
    <manufacturer>nom du fabriquant</manufacturer>
    <manufacturerURL>URL du site du fabriquant</manufacturerURL>
    <modelDescription>titre long</modelDescription>
    <modelName>nom du modèle</modelName>
    <modelNumber>numéro du modèle</modelNumber>
    <serialNumber>numéro de série du fabriquant</serialNumber>
    <modelURL>URL du site du modèle</modelURL>
    <UDN>UUID</UDN>
    <UPC>Universal Product Code</UPC>
    <iconList>
      <icon>
        <mimetype>image/format</mimetype>
        <width>largeur en pixels</width>
        <height>hauteur en pixels</height>
        <depth>nombre de bits de couleur</depth>
        <url>URL de l'icone</url>
      </icon>
      Les autres déclarations d'icônes, s'il y en a, vont ici.
    </iconList>
    <serviceList>
      <service>
        <serviceType>
          urn:schemas-upnp-org:service:serviceType:v

```

```

</serviceType>
<serviceId>
    urn:upnp-org:serviceId:serviceID
</serviceId>
<SCPDURL>URL vers le fichier de description du service</SCPDURL>
<controlURL>URL pour le contrôle</controlURL>
<eventSubURL>URL pour les évènements</eventSubURL>
</service>
Les déclarations pour les autres services, s'il y en a, vont ici.
</serviceList>
<deviceList>
Les descriptions des devices imbriqués, s'il y en a, vont ici.
</deviceList>
<presentationURL>URL pour la page de présentation</presentationURL>
</device>
</root>

```

Jusqu'à URLBase, le fichier doit être strictement identique. L'élément URLBase permet de définir explicitement le chemin qui sera utilisé comme préfixe pour toutes les URL du fichier. Par exemple si sa valeur est http://192.168.0.1/, et qu'un Service contient comme SCPDURL service1.xml, l'URL du SCPD du Service sera http://192.168.0.1/service1.xml. Si l'élément URLBase n'est pas renseigné, il sera automatiquement déduit de l'adresse de base de ce fichier, obtenue lors de la découverte ou de la recherche du serveur.

Le préfixe urn:schemas-upnp-org de l'élément deviceType peut être changé pour des dispositifs non-standards, remplacé par urn: et un nom de domaine ICANN. Dans tous les cas, la taille du deviceType doit être inférieure ou égale à 64 caractères.

Le friendlyName est le nom qui sera utilisé par les points de contrôle pour afficher les serveurs aux utilisateurs. Les éléments suivant jusqu'à serialNumber, ainsi que UPC sont à remplir par le fabricant, et ne sont pas d'une importance capitale.

L'UDN (*Unique Device Name*) permet d'identifier un device UPnP de façon unique, même parmi ses devices imbriqués. Il est identique à celui utilisé dans les messages de découverte. Il doit toujours être le même pour un device, et donc doit survivre aux redémarrages.

On peut spécifier une liste d'icônes, qui pourront être affichées à l'utilisateur, comme le friendlyName. Tous les éléments de icon doivent être renseignés.

Les listes serviceList et deviceList définissent l'architecture réelle des Services. serviceList peut ne contenir qu'un seul Service, et deviceList, la liste des Devices imbriqués, peut être vide et alors omise. Les éléments serviceType et serviceid sont soumis aux mêmes règles que deviceType, à l'exception que serviceId doit être unique parmi les Services d'un même Device. Les trois URL des Services peuvent être relatives à l'URL de base (explicite ou non). SCPDURL pointe vers le document qui va être décrit ci-après, controlURL est l'URL que contacte un point de contrôle pour invoquer des actions, et eventSubURL est l'URL de souscription aux évènements d'un Service. Si un Service ne dispose pas de variable qui envoie des évènements, l'élément eventSubURL doit quand même être présent et vide.

L'adresse de la page principale de présentation est spécifiée par l'élément presentationURL.

Les fichiers de description des Services contiennent les informations permettant de dialoguer avec les ressources des Services. Ils définissent les actions et les variables d'état que le Service fournit.

```

<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
<specVersion>
    <major>1</major>

```

```

<minor>0</minor>
</specVersion>
<actionList>
  <action>
    <name>nom de l'action</name>
    <argumentList>
      <argument>
        <name>nom du paramètre formel</name>
        <direction>in/out : paramètre d'entrée ou de sortie</direction>
        <retval />
        <relatedStateVariable>variable d'état liée</relatedStateVariable>
      </argument>
      Les déclarations de paramètres supplémentaires, s'il y en a, vont
      ici.
    </argumentList>
  </action>
  Les déclarations pour les autres actions, s'il y en a, vont ici.
</actionList>
<serviceStateTable>
  <stateVariable sendEvents="yes">
    <name>nom de la variable</name>
    <dataType>type de la variable</dataType>
    <defaultValue>valeur par défaut</defaultValue>
    <allowedValueList>
      <allowedValue>valeur énumérée</allowedValue>
      Les autres valeurs autorisées, s'il y en a, vont ici.
    </allowedValueList>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>nom de la variable</name>
    <dataType>type de la variable</dataType>
    <defaultValue>valeur par défaut</defaultValue>
    <allowedValueRange>
      <minimum>valeur minimale</minimum>
      <maximum>valeur maximale</maximum>
      <step>pas</step>
    </allowedValueRange>
  </stateVariable>
  Les déclarations des autres variables d'état, s'il y en a, vont ici.
</serviceStateTable>
</scpd>
```

La liste des actions peut être vide, dans ce cas l'élément `actionList` ne sera pas présent. Chaque action est définie par un nom et une liste d'arguments, qui elle aussi peut être absente. Les arguments ont un nom, une direction qui peut être `in` pour un argument, `out` pour un résultat. De plus, en spécifiant l'élément `retval` pour au plus un argument, il sera utilisé comme valeur de retour pour l'action.

Chaque argument est lié à une variable d'état : elle renseigne qu'un évènement pourrait arriver sur cette variable lors de l'invocation de l'action, et elle définit le type de l'argument.

Une variable peut être définie de plusieurs façons : elle peut émettre des évènements ou pas, avoir une valeur par défaut ou pas, une liste de valeurs possibles ou une valeur minimale, maximale et un pas optionnel. Dans tous les cas, les champs obligatoires sont le nom et le type.

Voici la liste des types de variables spécifiés par le forum UPnP (champ `dataType`) :

<b>ui1</b>	Entier sur un octet, non signé. Même syntaxe que int. Doit valoir entre 0 et 255.
<b>ui2</b>	Entier sur deux octets, non signé. Même syntaxe que int. Doit valoir entre 0 et 65535.

<b>ui4</b>	Entier sur quatre octets, non signé. Même syntaxe que int. Doit valoir entre 0 et 4294967295.
<b>i1</b>	Entier sur un octet. Même syntaxe que int. Doit valoir entre -128 et 127.
<b>i2</b>	Entier sur deux octets. Même syntaxe que int. Doit valoir entre -32768 et 32767.
<b>i4</b>	Entier sur quatre octets. Même syntaxe que int. Doit valoir entre -2147483648 et 2147483647.
<b>int</b>	Nombre entier. Peut commencer par un signe ou des zéros. Pas de groupage des chiffres avec des virgules ou autres signes.
<b>r4</b>	Flottant sur quatre octets. Même syntaxe que float. Doit valoir entre 3.40282347E+38 et 1.17549435E-38.
<b>r8</b>	Flottant sur huit octets. Même syntaxe que float. Doit valoir entre -1.79769313486232E308 et -4.94065645841247E-324 pour les valeurs négatives, et entre 4.94065645841247E-324 et 1.79769313486232E308 pour les valeurs positives, i.e., IEEE 64-bit (8-Byte) double.
<b>number</b>	Identique à r8.
<b>fixed.14.4</b>	Identique à r8 mais avec pas plus de 14 chiffres à gauche de la virgule, et pas plus de 4 à droite.
<b>float</b>	Nombre à virgule flottante. La mantisse (gauche du décimal) et/ou l'exposant peuvent être signés ou débuter par des zéros. Le caractère décimal de la mantisse est le point. La mantisse est séparée de l'exposant par E. Pas de groupage des chiffres avec des virgules ou autres signes.
<b>char</b>	Chaîne de caractères Unicode. Un caractère de long.
<b>string</b>	Chaîne de caractères Unicode. Pas de limite sur la longueur.
<b>date</b>	Date dans un sous-ensemble du format ISO 8601 sans données de l'heure.
<b>dateTime</b>	Date au format ISO 8601 avec heure optionnelle mais sans fuseau horaire.
<b>dateTime.tz</b>	Date au format ISO 8601 avec heure optionnelle et fuseau horaire.
<b>time</b>	Date dans un sous-ensemble du format ISO 8601 sans date ni fuseau horaire.
<b>time.tz</b>	Date dans un sous-ensemble du format ISO 8601 sans date mais avec fuseau horaire optionnel.
<b>boolean</b>	0 ou 'no' pour faux, 1 ou 'yes' pour vrai.
<b>bin.base64</b>	Grand objet binaire codé en Base64. Prend 3 octets, les découpe en 4 parties, et associe chacun des 6 bits à un octet. Pas de limite de taille.
<b>bin.hex</b>	caractères hexadécimaux représentant des octets. Utilise deux caractères pour représenter un octet. Pas de limite de taille.
<b>uri</b>	Universal Resource Identifier.
<b>uuid</b>	Universally Unique ID. Caractères hexadécimaux représentant des octets. Les optionnels traits d'union sont ignorés.

### 3.4 Contrôle : SOAP

L'invocation des actions d'un Service nécessite de connaître l'URL de contrôle, le serviceType dans lequel l'action se trouve, le nom de la fonction, le nom des arguments et leurs valeurs éventuelles. SOAP définit l'utilisation de XML et HTTP pour l'appel de procédure distante (RPC). UPnP utilise SOAP, qui invoque les actions et fournit un résultat de retour. Voyons le format de ce genre de message :

#### 3.4.1 Invocation

**POST URL du contrôle HTTP/1.1**  
**HOST:** hôte de l'URL:port de l'URL de contrôle  
**CONTENT-LENGTH:** taille du corps en octets  
**CONTENT-TYPE:** text/xml; charset="utf-8"  
**SOAPACTION:** "urn:schemas-upnp-org:service:serviceType:v#actionName"  
  
<s:Envelope  
  xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">

```

    s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<s:Body>
    <u:nom de l'action xmlns:u="urn:schemas-upnp-
org:service:serviceType:v">
        <nom de l'argument>valeur de l'argument</nom de l'argument>
        les autres arguments et leurs valeurs, s'il y en a, vont ici
    </u:nom de l'action>
</s:Body>
</s:Envelope>

```

Les données qui ne sont pas en italique sont requises et non sujettes à modification. Celles en italique changeront suivant leur description en fonction de l'action à invoquer. Si l'action n'a pas d'argument, la liste d'arguments (dont les éléments ont pour noms les noms d'arguments) ne doit pas être présente.

SOAP utilise la méthode POST définie par HTTP pour effectuer son appel. Pour fournir une plus grande flexibilité d'administration aux pare-feu et proxies, SOAP spécifie que l'appel doit être effectué une première fois comme décrit ci-dessus.

### 3.4.2 Réponse

La réponse doit être retournée dans les 30 secondes suivant l'appel. Cette valeur est définie par la référence UPnP et n'est pas modifiable. Si une action doit prendre plus de temps, elle devra retourner rapidement et envoyer un évènement pour signaler sa fin. Cependant, si on peut borner le temps maximal de réponse, on ne peut pas assurer qu'une réponse sera reçue dans un délai plus court. Un grand nombre de causes peuvent faire que la réponse prendra plus de temps que prévu pour arriver, par exemple un établissement de connexion TCP qui échoue au premier essai, ou une trop forte charge d'utilisation du dispositif. On ne peut donc pas utiliser UPnP pour des applications temps réel, et SOAP n'en est qu'une limitation.

La réponse est de type HTTP, elle doit donc avoir pour valeur 200 de retour s'il n'y a pas d'erreur. On retrouve les données XML propres à SOAP dans le corps ; celles ci comprennent les valeurs des arguments de sortie et de retour. Les données à modifier sont en italique, et assez explicites.

```

HTTP/1.1 200 OK
CONTENT-LENGTH: taille du corps en octets
CONTENT-TYPE: text/xml; charset="utf-8"
DATE: quand la réponse a été générée
EXT:
SERVER: OS/version UPnP/1.0 product/version

```

```

<s:Envelope
    xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
    s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<s:Body>
    <u:nom de l'actionResponse xmlns:u="urn:schemas-upnp-
org:service:serviceType:v">
        <nom de l'argument>valeur de l'argument</nom de l'argument>
        les autres arguments et leurs valeurs, s'il y en a, vont ici
    </u:nom de l'actionResponse>
</s:Body>
</s:Envelope>

```

Si le Service rencontre une erreur dans l'invocation de l'action demandée, le format de la réponse change. On ne retrouve plus le '200 OK' de la première ligne, mais '500 Internal Server Error' à la place, et le corps du message change radicalement

apres <s:Body> :

```
<s:Body>
  <s:Fault>
    <faultcode>s:Client</faultcode>
    <faultstring>UPnPError</faultstring>
    <detail>
      <UPnPError xmlns="urn:schemas-upnp-org:control-1-0">
        <errorCode>code d'erreur</errorCode>
        <errorDescription>message d'erreur</errorDescription>
      </UPnPError>
    </detail>
  </s:Fault>
</s:Body>
```

Le code d'erreur peut prendre diverses valeurs :

- 401 : action invalide. Il n'y a pas d'action avec ce nom dans ce Service.
- 402 : arguments invalides. Un ou plusieurs arguments peuvent manquer, il peut y en avoir en trop, de nom inconnu ou d'un mauvais type.
- 403 : mauvaise synchronisation.
- 501 : échec de l'action. Peut être retourné si l'état courant du Service ne permet pas l'invocation de cette action.
- de 600 à 799 : ce sont des erreurs spécifiques définies par le forum UPnP.
- de 800 à 899 : ce sont des erreurs spécifiques définies par le vendeur.

### 3.5 Évènements : GENA

Un point de contrôle peut souscrire aux évènements d'un Service et ainsi être informé des mises à jour des valeurs des variables du Service. Pour souscrire, il doit envoyer un message de souscription à l'URL d'abonnement (`eventSubURL`) correspondant au Service, qui se trouve dans le fichier de description du serveur.

Une souscription n'est pas définitive : elle est acceptée avec un temps maximal arbitraire de validité. L'abonné devra la réactualiser avant la fin de cette période s'il veut la garder active. Il peut aussi, à tout moment, annuler la souscription, ce qui est recommandé lorsqu'une souscription devient inutile, pour ne pas effectuer de traitement inutile ni surcharger le réseau. Si un abonné se déconnecte sans se désabonner, le serveur tentera de lui envoyer les évènements éventuels jusqu'à la date de validité de la souscription.

Toutes les actions concernant les évènements sont au format HTTP étendu par GENA.

#### 3.5.1 Opérations sur les abonnements

Avant d'étudier les évènements proprement dit, nous allons, étudier les phases de souscription et de gestion des abonnements. Les messages GENA de cette section sont des messages HTTP sans corps.

##### 3.5.1.1 Souscription

Le point de contrôle voulant effectuer une souscription doit posséder un serveur HTTP écoutant sur un port arbitraire pour pouvoir recevoir les évènements. Ils sont envoyés en HTTP par le Service. L'URL pointant vers ce serveur HTTP du client doit être mentionnée dans la demande de souscription, dans le champ `CALLBACK`. Il peut y avoir plusieurs URL, le serveur les essayant dans l'ordre jusqu'à en trouver une qui fonctionne.

**SUBSCRIBE** *eventSubURL HTTP/1.1*  
**HOST:** *hôte et port du serveur*  
**CALLBACK:** *<URL de délivrance de messages>*  
**NT:** *upnp:event*  
**TIMEOUT:** *Second-durée demandée de souscription en secondes*

Si le serveur est capable de gérer une nouvelle souscription, il doit renvoyer la réponse suivante en moins de 30 secondes et dès que possible envoyer une notification avec l'état courant des variables du Service :

**HTTP/1.1 200 OK**  
**DATE:** *date de la génération de la réponse*  
**SERVER:** *OS/version UPnP/1.0 produit/version*  
**SID:** *uuid:subscription-UUID*  
**TIMEOUT:** *Second-durée en secondes de validité réelle*

Le serveur doit assigner un SID (*Subscription IDentifier*) différent à chaque abonnement qu'il gère simultanément. Il permettra à l'abonné d'annuler la souscription et d'identifier les messages qu'il recevra lors des notifications.

La durée de validité acceptée peut être différente de celle demandée, en général si elle était trop courte ou trop longue. Si le serveur n'accepte pas la souscription, il doit renvoyer un message d'erreur, en remplaçant le '200 OK' par :

- '400 Bad Request' si le SID est présent dans la demande ;
- '412 Precondition Failed' si l'URL de CALLBACK n'est pas valide ou si NT ne vaut pas upnp:event.
- '5xx': si le serveur n'est pas capable d'accepter la souscription, il doit répondre avec une erreur HTTP de la série 500.

Le serveur doit maintenir une liste d'abonnés, contenant les informations permettant de les contacter : un identifiant de souscription unique, l'URL où délivrer les messages, la clé d'évènement, et le temps de validité de la souscription. Quand une souscription expire, il peut supprimer l'abonné de sa liste. Les éventuels messages concernant son abonnement qu'il en recevra ne pourront être qu'une demande de souscription.

### 3.5.1.2 Renouvellement d'abonnement

Le renouvellement d'abonnement doit se faire avant l'expiration de l'abonnement. Dans le cas contraire, il faudra effectuer à nouveau une souscription. Le renouvellement utilise la méthode SUBSCRIBE aussi, mais les deux messages utilisent un ensemble disjoint d'entêtes, SID pour le renouvellement, NT et CALLBACK pour la souscription.

**SUBSCRIBE** *eventSubURL HTTP/1.1*  
**HOST:** *hôte et port du serveur*  
**SID:** *uuid:subscription-UUID*  
**TIMEOUT:** *Second-durée demandée de souscription*

Le serveur doit répondre à cette requête avec le même message que pour la souscription.

### 3.5.1.3 Annulation d'abonnement.

L'annulation d'un abonnement se fait en utilisant la méthode UNSUBSCRIBE :

```
UNSUBSCRIBE eventSubURL HTTP/1.1
HOST: hôte et port du serveur
SID: uuid:subscription-UUID
```

Après cette opération, la réponse doit être reçue dans les 30 secondes, et l'uuid n'est plus valide :

```
HTTP/1.1 200 OK
```

### 3.5.2 Évènements

Le serveur envoie les évènement en se connectant en HTTP/TCP sur l'URL spécifiée dans CALLBACK. Il envoie un message GENA, avec des entêtes HTTP, et un corps XML définissant les variables.

Chaque évènement est envoyé avec un numéro de séquence (la clé de l'évènement). Ainsi, le point de contrôle est certain de pouvoir les traiter dans l'ordre. Ce numéro est incrémenté à chaque évènement. S'il se rend compte qu'il n'a pas reçu un ou des évènements (à cause d'une panne quelconque), la norme lui impose de se désabonner et se réabonner, et ainsi obtenir un nouveau SID et numéro de séquence.

```
NOTIFY URL de délivrance de messages HTTP/1.1
HOST: hôte et port du client
CONTENT-TYPE: text/xml
CONTENT-LENGTH: taille du corps
NT: upnp:event
NTS: upnp:propchange
SID: uuid:subscription-UUID
SEQ: clé de l'évènement
```

```
<e:propertyset xmlns:e="urn:schemas-upnp-org:event-1-0">
  <e:property>
    <variableName>nouvelle valeur</variableName>
  </e:property>
  Les autres variables, s'il y en a, vont ici.
</e:propertyset>
```

Comme on le voit, ce format de communication n'est pas forcément des plus efficace. Chaque variable ajoute l'élément `<e:property>` et son fils. Pour limiter le trafic réseau et le temps de traitement il est donc recommandé de séparer les variables sans rapport entre elles dans plusieurs Services.

Pour accuser réception de l'évènement, l'abonné renvoie le message classique HTTP :  
`HTTP/1.1 200 OK`

## 3.6 Présentation : HTTP

Étant donné que toutes les actions sont effectuées par HTTP (sur TCP ou UDP), une pile UPnP doit logiquement intégrer un serveur HTTP. Fournir des pages de présentation HTML (un site web pour le dispositif) est une extension simple des fonctionnalités de l'UPnP. Un serveur peut ajouter un élément `presentationURL` dans sa description, qui correspond à l'URL de la page d'entrée de ce site. Les pages consultées pourront par exemple refléter l'état du serveur et ses variables ou offrir des possibilités de configurations aussi bien accessibles par les services.

La fonctionnalité de présentation d'UPnP n'est qu'un exemple d'utilisation du serveur HTTP qui traite les messages des protocoles d'UPnP, SOAP, GENA, SSDP. La page de présentation si elle existe doit être mentionnée dans le fichier XML de description de serveur, par l'élément `presentationURL`.

Pour récupérer une page de présentation, le point de contrôle effectue une requête HTTP GET, et le serveur renvoie la page. Les pages fournies doivent être au format HTML ; la forme de la page, ses couleurs, sa complexité, l'utilisation de langages de scripts (par exemple javascript), sont libres à l'implémentation du vendeur. Ils ne sont pas normalisés par le Forum UPnP.

A partir de toutes les spécifications que nous avons vu dans cette partie, certains constructeurs ou fabricants de logiciels ont écrit des SDK (*Software Development Kit*). Ils facilitent l'implémentation d'un serveur UPnP, il n'est plus nécessaire de développer toutes les couches du modèle.

## 4 SDK, outils, et exemples d'utilisation

Comme tout Web Service, l'implémentation des les protocoles UPnP est libre. Des SDK, plus communément appelés *piles UPnP*, sont disponibles dans de nombreux langages de programmation et pour une grande variété d'architectures, embarquées ou non. Une liste de SDKs est recensée sur le site du forum UPnP.

<http://www.upnp.org/resources/sdks.asp>

### 4.1 Outils

La solution la plus complète et la plus simple est la distribution *Intel Tools for UPnP Technologies* distribuée gratuitement par Intel. Cet ensemble d'outils pour Windows comprend le *Device Spy*, un point de contrôle générique qui facilitera énormément les tests des serveurs développés. Pour tester les piles UPnP, l'outil *Device Sniffer* permettra de voir les trames réseau que l'ordinateur reçoit, et étudier les données des paquets HTTP par exemple. Il y a aussi un outil de validation de serveur, le *Device Validator*, qui effectue des tests de toutes les fonctionnalités de la pile UPnP. Le *Device Relay* permet d'utiliser un serveur UPnP d'un réseau sur un autre.

La *Network Light* que l'on trouve dans ces outils est un serveur d'exemple, une lampe virtuelle graphique. Elle est souvent utilisée pour tester un point de contrôle, ou pour vérifier que les paquets *multicast* sont bien visibles sur le réseau (un pare-feu ou certains commutateurs de basse qualité ne les laissent pas passer).

On dispose également de diverses applications pour l'UPnP Audio/Vidéo (A/V), pour valider les serveurs UPnP A/V, partager des fichiers sur le réseau ou effectuer leur rendu.

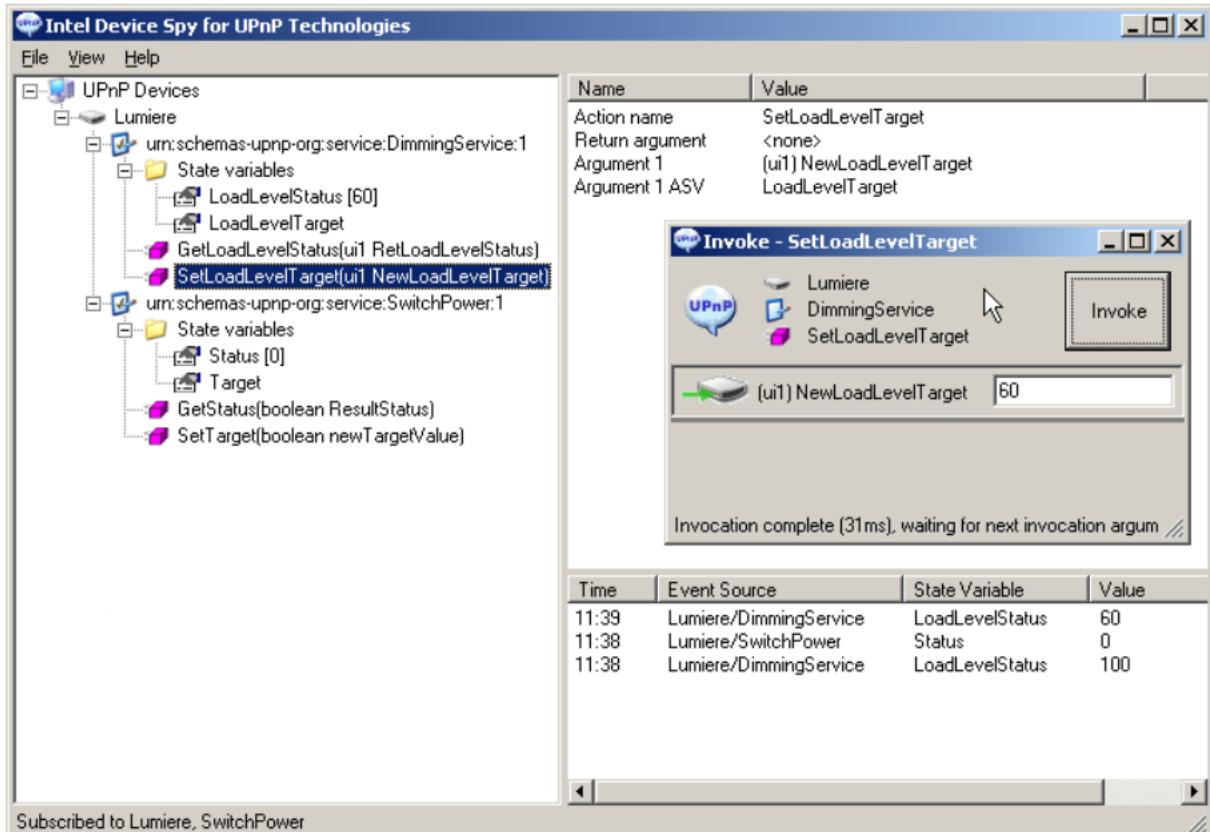


Figure 4 - Le *Device Spy* invoquant une action et recevant les évènements de la *Network Light* d'Intel.

Nous allons étudier deux outils importants, le *Service Author* et le *Device Builder*, qui sont utilisés pour créer des serveurs ou points de contrôle UPnP. Nous les illustrons avec le dispositif « Lumière » que nous utiliserons aussi pour l'exemple suivant. Le schéma de description de la lampe virtuelle est défini par le forum UPnP. Les noms d'actions et de variables que nous allons utiliser ne sont pas très explicites mais ce sont ceux du schéma officiel.

## 4.2 SDK Intel pour Windows

L'outil *Service Author* permet de générer les fichiers XML des Services avec une interface conviviale :

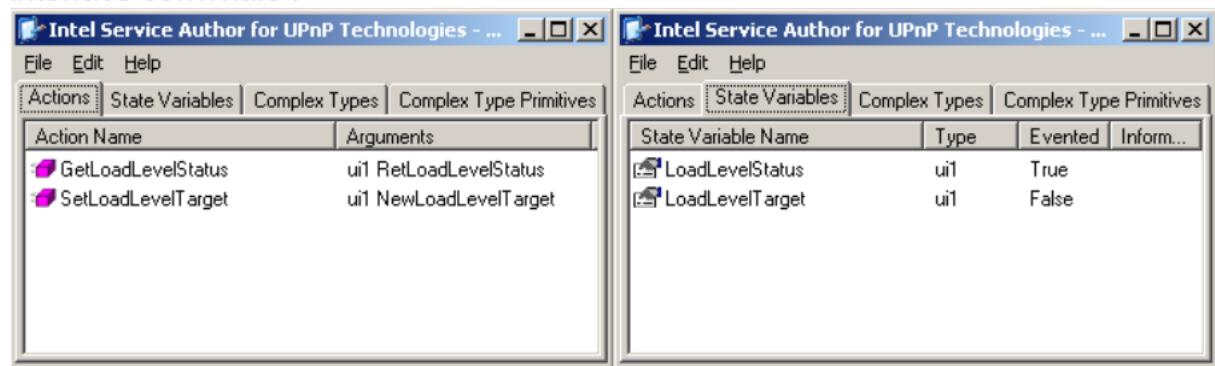


Figure 5 - La liste des actions et des variables d'état dans le *Service Author*.

Il faut prévoir les variables d'état nécessaires à la création des actions, puisque chaque argument est lié à une variable d'état. Une fois les variables et actions ajoutées, on génère le fichier XML (SCPD) correspondant au Service.

Nous allons maintenant utiliser l'outil *Device Builder*, qui construit le serveur, avec les Services fournis au format XML :

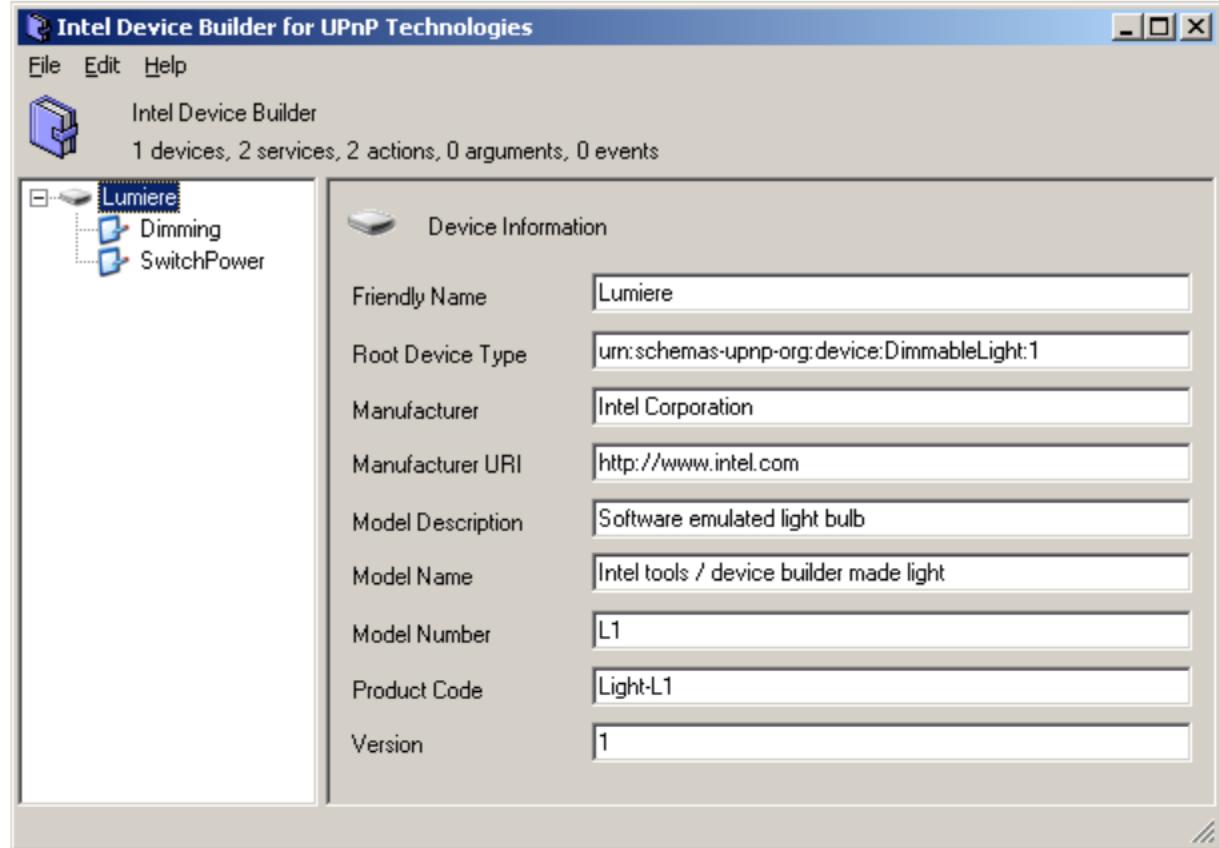


Figure 6 – Générateur automatique de code *Device Builder*

Cet outil peut générer le code du serveur et du point de contrôle dans plusieurs langages et pour plusieurs OS : C POSIX, C WinSock 1 et 2, C PocketPC, C# .NET, C++/ATL et Java 1.3. Avec le C POSIX, un makefile est généré, pour le C WinSock, le C++ et le C#, un projet Visual Studio est généré, et pour le Java, c'est un projet Borland JBuilder.

Dans le code généré en C, on retrouve toute la pile UPnP, et un squelette pour les actions à implémenter. On peut directement le compiler et le lancer pour le voir apparaître sur le réseau et vérifier que les descriptions sont bonnes.

Le code généré en C# utilise une bibliothèque gérée (code managé), UPnP.dll. La conception d'un serveur UPnP est extrêmement simple dans ce langage. Il n'est en effet même pas nécessaire d'écrire les fichiers XML, et l'utilisation des outils *Service Author* et *Device Builder* devient obsolète. Tout est déclaré par les classes UPnPDevice et UPnPServices, et la dll utilise la réflexion pour définir les types des actions à partir des fonctions C# .Net.

Ces outils simples et puissants, permettent de créer rapidement un serveur UPnP, mais plus pour un usage de prototypage plutôt que de production. Ils ont un gros inconvénient : la pile C contient des bugs qui empêchent, dans certains, cas l'envoi d'évènements et sa complexité est telle que résoudre ces problèmes demande un

temps non négligeable. La pile C# consomme une quantité de mémoire considérable et ne répond pas aux recherches, et on ne dispose pas de ses sources.

### 4.3 LibUPnP portable

Nous allons maintenant étudier une implémentation de serveur UPnP et d'un point de contrôle qui lui est associé, écrits en C pour LibUPnP. La LibUPnP est une implémentation libre d'un SDK portable pour les dispositifs UPnP. Intel avait développé cette bibliothèque en 2000 ; elle a depuis été mise à disposition sous licence BSD, pour lui assurer une meilleure croissance et un meilleur support. Écrite pour GNU/Linux à l'origine, elle a été portée pour divers systèmes d'exploitation, tels que FreeBSD, Solaris, et Windows. La communauté autour de ce projet open-source est active, ce qui garantit une bonne fiabilité du code et une bonne réactivité dans le cas peu probable où l'on trouverait une faille dans l'implémentation. C'est probablement la pile UPnP la plus stable et efficace.

Nous allons donc la mettre en œuvre avec un exemple, simple mais avec lequel nous allons pouvoir découvrir toutes les facettes de la programmation LibUPnP : une lampe virtuelle, semblable à l'*Intel Network Light*. Elle peut bien sûr être commandée pour être allumée ou éteinte, mais aussi pour faire varier l'intensité lumineuse, comme une lampe halogène à potentiomètre. Pour des raisons de simplicité et pour ne pas s'éloigner du sujet, elle ne disposera pas d'une interface graphique, mais textuelle.

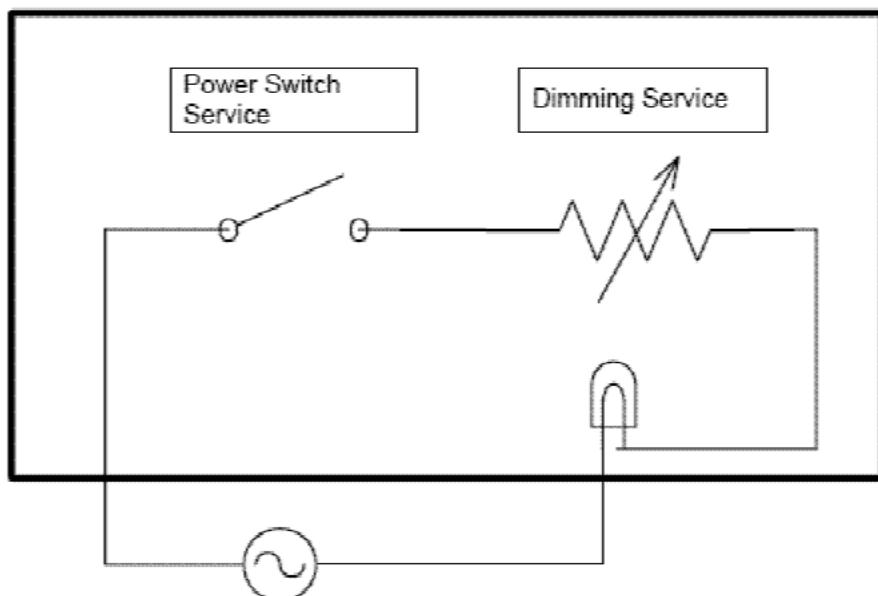


Figure 7 – Fonctionnement de la lampe virtuelle et utilisation de ses deux services.

#### 4.3.1 Descriptions

L'architecture du serveur est dictée par les fichiers XML de description du *Root Device*. Comme nous l'avons vu dans la section 3.3, cette description contient des informations variées sur le constructeur et le dispositif, ainsi qu'une liste de *Devices* imbriqués et de *Services*, et les URL qui permettent d'utiliser ces derniers.

Pour notre lampe virtuelle, nous utiliserons deux *Services*, un "*SwitchPower*" qui servira à allumer et éteindre la lumière, et "*Dimming*" qui gère la luminosité de la lampe.

## Device.xml

```
<root>
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <device>
    <deviceType>urn:schemas-upnp-org:device:DimmableLight:1</deviceType>
    <friendlyName>Lumiere</friendlyName>
    <manufacturer>Intel Corporation</manufacturer>
    <manufacturerURL>http://www.intel.com</manufacturerURL>
    <modelDescription>Software emulated light bulb</modelDescription>
    <modelName>Portable libupnp emulated light bulb</modelName>
    <modelNumber>PLU-L1</modelNumber>
    <modelURL>http://www.libupnp.org/</modelURL>
    <UDN>uuid:28083b9e-e3d7-4587-b70c-ade8bc058ba8</UDN>
    <iconList>
      <icon>
        <mimetype>image/png</mimetype>
        <width>32</width>
        <height>32</height>
        <depth>8</depth>
        <curl>icon.png</url>
      </icon>
    </iconList>
    <serviceList>
      <service>
        <serviceType>urn:schemas-upnp-org:service:Dimming:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:Dimming</serviceId>
        <SCPDURL>DimmingService.xml</SCPDURL>
        <controlURL>DimmingService/control</controlURL>
        <eventSubURL>DimmingService/event</eventSubURL>
      </service>
      <service>
        <serviceType>urn:schemas-upnp-
org:service:SwitchPower:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:SwitchPower</serviceId>
        <SCPDURL>SwitchPower.xml</SCPDURL>
        <controlURL>SwitchPower/control</controlURL>
        <eventSubURL>SwitchPower/event</eventSubURL>
      </service>
    </serviceList>
    <presentationURL>index.html</presentationURL>
  </device>
</root>
```

Voyons les points clés de ce fichier. Tout d'abord, il n'y a pas l'élément URLBase. On ne peut pas connaître à l'avance l'IP ou le hostname de la machine sur laquelle va s'exécuter le serveur, on ne peut donc pas la renseigner. Cependant une fois le serveur lancé, nous avons à disposition des outils pour ajouter ce nœud au fichier XML.

Le deviceType : il est normalisé et définit le nom d'un device. C'est l'information de base la plus utile pour déterminer le type de dispositif, les points de contrôle peuvent effectuer une recherche sur deviceType (ce que nous allons appliquer par la suite). L'UDN a été généré aléatoirement. Nous avons une icône représentant une lampe, ses informations sont décrites dans la description. Ainsi, un point de contrôle pourra afficher le dispositif avec une image, téléchargée par HTTP à l'URL indiquée.

La LibUPnP embarque un serveur HTTP. Fournir une page de présentation est donc

vraiment simple, comme nous le verrons. On décide que le fichier racine du site sera /index.html.

On voit nos deux Services dans la liste, et leurs informations d'identification, et de communications. Les URL de contrôle et de d'enregistrement sont des URL virtuelles, gérées par le SDK, on peut donc mettre celles que l'on veut, bien qu'une certaine hiérarchie soit préférable. La SCPDURL par contre devra pointer vers les fichiers réels, retranscrits ci-dessous :

### **DimmingService.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
    <specVersion>
        <major>1</major>
        <minor>0</minor>
    </specVersion>
    <actionList>
        <action>
            <name>GetLoadLevelStatus</name>
            <argumentList>
                <argument>
                    <name>RetLoadLevelStatus</name>
                    <direction>out</direction>
                    <relatedStateVariable>LoadLevelStatus</relatedStateVariable>
                </argument>
            </argumentList>
        </action>
        <action>
            <name>SetLoadLevelTarget</name>
            <argumentList>
                <argument>
                    <name>NewLoadLevelTarget</name>
                    <direction>in</direction>
                    <relatedStateVariable>LoadLevelTarget</relatedStateVariable>
                </argument>
            </argumentList>
        </action>
    </actionList>
    <serviceStateTable>
        <stateVariable sendEvents="yes">
            <name>LoadLevelStatus</name>
            <dataType>ui1</dataType>
            <allowedValueRange>
                <minimum>0</minimum>
                <maximum>100</maximum>
            </allowedValueRange>
        </stateVariable>
        <stateVariable sendEvents="no">
            <name>LoadLevelTarget</name>
            <dataType>ui1</dataType>
            <allowedValueRange>
                <minimum>0</minimum>
                <maximum>100</maximum>
            </allowedValueRange>
        </stateVariable>
    </serviceStateTable>
</scpd>
```

### **SwitchPower.xml**

```
<?xml version="1.0" encoding="utf-8"?>
```

```

<scpd xmlns="urn:schemas-upnp-org:service-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <actionList>
    <action>
      <name>GetStatus</name>
      <argumentList>
        <argument>
          <name>ResultStatus</name>
          <direction>out</direction>
          <relatedStateVariable>Status</relatedStateVariable>
        </argument>
      </argumentList>
    </action>
    <action>
      <name>SetTarget</name>
      <argumentList>
        <argument>
          <name>newTargetValue</name>
          <direction>in</direction>
          <relatedStateVariable>Target</relatedStateVariable>
        </argument>
      </argumentList>
    </action>
  </actionList>
  <serviceStateTable>
    <stateVariable sendEvents="yes">
      <name>Status</name>
      <dataType>boolean</dataType>
    </stateVariable>
    <stateVariable sendEvents="no">
      <name>Target</name>
      <dataType>boolean</dataType>
    </stateVariable>
  </serviceStateTable>
</scpd>
```

On voit que chaque *Service* contient deux actions et une variable qui envoie des évènements. Le *Service Dimming* règle l'intensité de la lumière, par l'action *SetLoadLevelTarget*, en émettant un évènement avec la variable *LoadLevelStatus* lorsqu'elle est modifiée. L'action *GetLoadLevelStatus* permet d'obtenir la valeur de cette variable. Le *Service SwitchPower* modifie l'état de la lampe : allumée ou éteinte. L'action *SetTarget* l'allume si son argument faut 1, et l'éteind s'il vaut 0, en émettant un évènement avec la variable *Status*. De même, *GetStatus* permet de connaître l'état de la lampe.

Maintenant que nous avons défini les fichiers de descriptions XML, nous allons pouvoir commencer la programmation du serveur et du point de contrôle.

#### 4.3.2 Serveur

La LibUPnP fournit les fonctions de base à l'exploitation d'un serveur UPnP. La programmation reste donc assez complexe, mais on a un meilleur contrôle de ce qui peut se passer dans la pile UPnP que dans l'implémentation C# des outils Intels par exemple. C'est de plus, probablement la clé de sa faible consommation en ressources.

Une bonne compréhension de la technologie UPnP est nécessaire : nous allons devoir développer toutes les fonctionnalités d'un serveur UPnP (avec l'aide des fonctions de la bibliothèque), à l'exception de la recherche, à laquelle la bibliothèque répond automatiquement.

#### 4.3.2.1 Première étape : initialisation, annonce.

Pour pouvoir utiliser la LibUPnP dans un programme, il faut bien entendu inclure ses fichiers d'en-tête :

```
#include <upnp.h>      /* fichier principal */
#include <upnptools.h> /* fonctions de génération de paquets de réponse */
```

La LibUPnP utilise un module interne de gestion de threads, un *thread pool*, qui lui permet d'effectuer plusieurs tâches en même temps (au moins virtuellement). Beaucoup de fonctions de la bibliothèque sont asynchrones. Elles retournent immédiatement mais prennent en argument une fonction de rappel (*callback*) qui sera appelée à la terminaison réelle de la tâche qu'elles effectuent. À cause des threads, il est fréquent qu'une fonction de rappel soit appelée plusieurs fois en même temps. Pour éviter ce problème, on définit une variable d'exclusion mutuelle (*mutex*) globale, qui protègera les données d'un accès simultané ; à l'entrée des fonctions de rappel, la mutex sera prise, et libérée à la sortie. Il faut faire attention à bien la libérer dans tous les chemins d'exécution possibles, surtout les cas d'erreurs qu'on a tendance par habitude à faire retourner immédiatement de la fonction. Voici sa déclaration, en variable globale :

```
pthread_mutex_t device_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Nous aurons aussi besoin, en variable globale, du handle, et des représentants dans leurs vrais types des variables d'état. Le handle est une sorte d'identifiant interne pour la bibliothèque, que nous devrons utiliser pour effectuer la plupart des actions UPnP. Il permet d'utiliser plusieurs serveurs dans le même programme, et ainsi de les dissocier lorsqu'on souhaite effectuer une action sur l'un deux.

```
UpnpDevice_Handle handle      = -1;
unsigned char    load_level   = 100;      /* intensité de la lampe */
unsigned char    target       = 0;        /* état de la lampe */
```

La LibUPnP ne peut utiliser qu'une adresse IP par serveur, on ne peut donc pas rendre un serveur visible sur toutes les interfaces réseau de la machine sur laquelle il est lancé (il est prévu que cette limitation soit supprimée dans une prochaine version). C'est pourquoi dans notre exemple, on permettra la spécification d'une adresse IP en argument de la ligne de commande.

La première fonction à appeler est UpnpInit(), qui prend en paramètre l'IP et le port du serveur. La valeur NULL pour l'IP lui fait prendre l'IP de la première interface et 0 pour le port en choisit un libre arbitraire supérieur ou égal à 49152.

```
if (UpnpInit(argv[1], 0) != UPNP_E_SUCCESS) {
    fprintf(stderr, "UpnpInit() failed.\n");
    UpnpFinish();
    exit(1);
}
```

Les fichiers de descriptions, et éventuellement les icônes et la page de présentation

sont transmis aux clients par HTTP. La LibUPnP embarque un serveur HTTP, auquel il faut préciser le chemin de ces fichiers pour qu'il puisse les servir. La fonction UpnpSetWebServerRootDir() remplit cette mission.

```
if (UpnpSetWebServerRootDir("www") != UPNP_E_SUCCESS) {
    fprintf(stderr, "Unable to set web server root directory\n");
    UpnpFinish();
    exit(1);
}
```

Il faut maintenant spécifier à la LibUPnP que l'on souhaite l'utiliser pour un serveur, et non pour un point de contrôle. Pour initialiser un serveur, il faut utiliser la fonction UpnpRegisterRootDevice(), qui prend en argument une URL qui permettra aux clients de télécharger la description XML et qui met en place la fonction de rappel pour tout ce qui peut arriver comme requête. Elle renvoie aussi le handle que nous avons déclaré, qui servira à identifier notre serveur dans diverses opérations. Il existe des fonctions pour récupérer l'IP et le port qui ont été choisis par la bibliothèque, que nous utilisons pour construire l'URL de description :

```
char desc_doc_url[200], buf[200];

sprintf(desc_doc_url, "http://%s:%d/device.xml",
        UpnpGetServerIpAddress(),
        UpnpGetServerPort());

if (UpnpRegisterRootDevice(desc_doc_url, callback, NULL, &handle) != UPNP_E_SUCCESS) {
    fprintf(stderr, "Unable to register the root device.\n");
    UpnpFinish();
    exit(1);
}
```

Lorsqu'un point de contrôle voudra invoquer une action ou souscrire à un Service de notre serveur, il identifiera ses actions avec les *deviceType*, *serviceType*, *UDN*, nom d'action, etc. Toutes ces informations se trouvent dans les fichiers XML de description. Cependant, pour ne pas alourdir le code avec un analyseur XML supplémentaire, nous avons préféré tenir à jour une table d'état du serveur. C'est une structure statique, remplie à la main à partir des informations du fichier XML, et en plus, les valeurs sous forme de chaîne de caractères des variables d'état qui envoient des évènements. Il faut alors définir les types représentant notre serveur :

```
typedef int (*upnp_action)(IXML_Document *request,
                           IXML_Document **out,
                           char **errorString);

struct service {
    char *service_id;
    char *service_type;
    char **var_names;
    char **var_values;
    int nb_vars;

    char **action_names;
    upnp_action *actions;
    int nb_actions;
};
```

```

struct device {
    char *UDN;
    struct service* services;
    int nb_services;
};

struct device root;

```

Pour plus de commodité, on définit des macros permettant de retrouver facilement les Services et les variables dans la table que l'on définit dans la fonction d'initialisation de la table d'état :

```

#define SERVICE_DIMMING 0
#define VARIABLE_LOADLEVELSTATUS 0

#define SERVICE_SWITCHPOWER 1
#define VARIABLE_STATUS 0

```

Dans notre `main()` on aura donc un appel à cette fonction :

```

void init_device_state_table() {
    root.UDN = "uuid:28083b9e-e3d7-4587-b70c-ade8bc058ba8";
    root.nb_services = 2;
    root.services = (struct service *)malloc(root.nb_services *
sizeof(struct service));

    /* données du premier service */
    root.services[SERVICE_DIMMING].service_id =
        "urn:upnp-org:serviceId:Dimming";
    root.services[SERVICE_DIMMING].service_type =
        "urn:schemas-upnp-org:service:Dimming:1";

    root.services[SERVICE_DIMMING].var_names = (char **)malloc(1 *
sizeof(char *));
        root.services[SERVICE_DIMMING].var_names[VARIABLE_LOADLEVELSTATUS] =
    "LoadLevelStatus";

    root.services[SERVICE_DIMMING].var_values = (char **)malloc(1 *
sizeof(char *));
        root.services[SERVICE_DIMMING].var_values[VARIABLE_LOADLEVELSTATUS] =
    (char *)malloc(4 * sizeof(char));
        strcpy(root.services[SERVICE_DIMMING].var_values[VARIABLE_LOADLEVELST
ATUS], "100"); /*ui1 LoadLevelStatus*/
    root.services[SERVICE_DIMMING].nb_vars = 1;

    root.services[SERVICE_DIMMING].action_names = (char **)malloc (2 *
sizeof(char *));
        root.services[SERVICE_DIMMING].actions=(upnp_action*)malloc(2*sizeof(
    upnp_action));
            root.services[SERVICE_DIMMING].action_names[0] =
    "GetLoadLevelStatus";
            root.services[SERVICE_DIMMING].actions[0] =
    Dimming_GetLoadLevelStatus;
            root.services[SERVICE_DIMMING].action_names[1] =
    "SetLoadLevelTarget";
            root.services[SERVICE_DIMMING].actions[1] =
    Dimming_SetLoadLevelTarget;
    root.services[SERVICE_DIMMING].nb_actions = 2;

    /* données du deuxième service */

```

```

        root.services[SERVICE_SWITCHPOWER].service_id = "urn:upnp-
org:serviceId:SwitchPower";
        root.services[SERVICE_SWITCHPOWER].service_type =
            "urn:schemas-upnp-org:service:SwitchPower:1";

        root.services[SERVICE_SWITCHPOWER].var_names = (char **)malloc(1 *
sizeof(char *));
        root.services[SERVICE_SWITCHPOWER].var_names[VARIABLE_STATUS] =
"Status";

        root.services[SERVICE_SWITCHPOWER].var_values = (char **)malloc(1 *
sizeof(char *));
        root.services[SERVICE_SWITCHPOWER].var_values[VARIABLE_STATUS] =
(char *)malloc(2 * sizeof(char));
        strcpy(root.services[SERVICE_SWITCHPOWER].var_values[VARIABLE_STATUS]
, "0"); /* bool Status */
        root.services[SERVICE_SWITCHPOWER].nb_vars = 1;

        root.services[SERVICE_SWITCHPOWER].action_names = (char **)malloc (2
* sizeof(char *));
        root.services[SERVICE_SWITCHPOWER].actions=(upnp_action*)malloc(2*siz
eof(upnp_action));
        root.services[SERVICE_SWITCHPOWER].action_names[0] = "GetStatus";
        root.services[SERVICE_SWITCHPOWER].actions[0] =
SwitchPower_GetStatus;
        root.services[SERVICE_SWITCHPOWER].action_names[1] = "SetTarget";
        root.services[SERVICE_SWITCHPOWER].actions[1] =
SwitchPower_SetTarget;
        root.services[SERVICE_SWITCHPOWER].nb_actions = 2;
}

```

Le champ `actions` de la structure `service` est un pointeur sur fonction, qui sera utilisée pour implémenter les actions des différents `Services`.

Nous sommes désormais prêts à recevoir les requêtes des points de contrôle, nous pouvons annoncer la présence du service sur le réseau. L'envoi des paquets *multicast* d'annonce(SSDP) se fait avec la fonction `UpnpSendAdvertisement()`, qui prend en paramètre le handle, et le temps de validité. Avant l'expiration des données publiées, la LibUPnP les renouvelle automatiquement.

```

if (UpnpSendAdvertisement(handle, 1800) != UPNP_E_SUCCESS) {
    fprintf(stderr, "Unable to advertise myself.\n");
    UpnpFinish();
    exit(1);
}

```

La fin du `main` consiste en une attente, les threads lancés par la LibUPnP s'occupant du traitement des données. Nous choisissons de lire l'entrée standard jusqu'à recevoir une fin de ligne, en affichant l'état de la lampe virtuelle à chaque entrée. Il est important, lorsque le serveur s'arrête, d'annoncer au réseau qu'il n'est plus disponible. Ceci doit être fait aussi bien lorsque le serveur est terminé proprement, avec la fin de ligne dans notre cas, que quand le serveur est tué, avec un Control-C par exemple. Nous mettons en place des gestionnaires de signaux qui effectuent les mêmes opérations que la fin du `main` ci-dessous, après le `while`:

```

signal(SIGINT, sighandle);
signal(SIGTERM, sighandle);

```

```

    while (fgets(buf, 200, stdin))
        printf("Status: %d. Level: %d\n", target, load_level);

    printf("exiting...\n");
    UpnpRemoveAllVirtualDirs();
    UpnpUnRegisterRootDevice(handle);
    UpnpFinish();
    pthread_mutex_destroy(&device_mutex);
    return 0;
}

```

#### 4.3.2.2 Deuxième étape : gestion des actions.

Nous sommes prévenus des requêtes des points de contrôle par l'appel de la fonction de rappel que nous avons fourni à `UpnpRegisterRootDevice()`. Il y a de nombreux types d'évènements possibles pouvant appeler cette fonction, nous nous limiterons aux deux principaux, les demandes de souscription aux évènements et les invocations d'actions :

```

int callback(Upnp_EventType event_type, void *event, void *cookie) {
    int retval = UPNP_E_INVALID_HANDLE;
    switch (event_type) {
        case UPNP_EVENT_SUBSCRIPTION_REQUEST:
            retval = handle_subscription_request(
                (struct Upnp_Subscription_Request *)event);
            break;
        case UPNP_CONTROL_ACTION_REQUEST:
            retval = handle_action_request(
                (struct Upnp_Action_Request *)event);
            break;
    }
    return retval;
}

```

La fonction `handle_action_request` doit vérifier plusieurs données avant de commencer l'invocation en elle-même : test de la validité de l'UDN (doit être identique à celui du XML), recherche du `serviceId` parmi les Services du serveur, recherche de l'action avec son nom parmi celles du Service. Lorsque l'action est identifiée, on peut appeler la fonction qui l'effectue, grâce au pointeur sur fonction de la table du serveur. En cas d'erreur, il faut fournir un code d'erreur HTTP, et une chaîne de caractères plus explicite.

```

int handle_action_request(struct Upnp_Action_Request *request) {
    int i, retval, service = -1, action = -1;
    char *error = NULL;

    request->ErrCode = 0;
    request->ActionResult = NULL;

    if (strcmp(request->DevUDN, root.UDN)) {
        fprintf(stderr, "Bad UDN for action request (%s != %s)\n",
                request->DevUDN, root.UDN);
        request->ErrCode = 401;
        request->ActionResult = NULL;
        strcpy(request->ErrStr, "Invalid UDN");
        return UPNP_E_INVALID_PARAM;
    }
}

```

```

/* trouver le service service */
for (i=0; i<root.nb_services; i++){
    if (!strcmp(request->ServiceID, root.services[i].service_id)){
        service = i;
        for (i=0; i<root.services[service].nb_actions; i++){
            if (!strcmp(root.services[service].action_names[i],
                        request->ActionName)){
                action = i;
                break;
            }
        }
        break;
    }
}

if (service == -1 || action == -1){
    request->ErrCode = 401;
    request->ActionResult = NULL;
    strcpy(request->ErrStr, "Invalid Action");
    return UPNP_E_INVALID_ACTION;
}

/* effectuer l'action */
retval = root.services[service].actions[action] (request-
>ActionRequest, &request->ActionResult, &error);

if(retval == UPNP_E_SUCCESS)
    request->ErrCode = UPNP_E_SUCCESS;
else {
    if (error)
        strcpy(request->ErrStr, error);
    switch (retval) {
        case UPNP_E_INVALID_PARAM:
            request->ErrCode = 402;
            break;
        case UPNP_E_INTERNAL_ERROR:
        default:
            request->ErrCode = 501;
            break;
    }
}
return request->ErrCode;
}

```

Voyons le code d'une action qui renvoie juste une information, en l'occurrence le pourcentage de luminosité de notre lampe, l'action `GetLoadLevelStatus`. Le code est simple, on doit juste construire la réponse, contenant cette valeur. Pour cela, la fonction `UpnpAddToActionResponse` (pour laquelle on doit inclure `upnptools.h`) construit la réponse SOAP (du XML) en prenant le nom de l'action, le `serviceType`, le nom de l'argument, et sa valeur :

```

int Dimming_GetLoadLevelStatus(IN IXML_Document * in, OUT IXML_Document
**out, OUT char **errorMessage){
    char *value;
    sprintf(value, "%d", load_level);
    *out = NULL;
    *errorMessage = NULL;

    pthread_mutex_lock(&device_mutex);
    value = root.services[SERVICE_DIMMING].

```

```

        var_values[VARIABLE_LOADLEVELSTATUS];
pthread_mutex_unlock(&device_mutex);

if (UpnpAddToActionResponse(out, "GetLoadLevelStatus",
                           root.services[SERVICE_DIMMING].service_type,
                           "RetLoadLevelStatus", value) != UPNP_E_SUCCESS) {
    *out = NULL;
    *errorString = "Internal Error";
    return UPNP_E_INTERNAL_ERROR;
}
return UPNP_E_SUCCESS;
}

```

Pour une action qui prend des données en argument pour modifier les variables internes, le code est légèrement plus complexe. Prenons par exemple l'action SetLoadLevelTarget, qui modifie l'intensité lumineuse de la lampe. La première étape est de récupérer la valeur de l'argument. Pour cela, on doit parcourir le fichier XML de la requête. Une fonction séparée `get_argument_value` a été écrite pour effectuer cette tache, puisqu'elle est utilisée pour chaque argument de fonction.

La deuxième étape est le test des valeurs des arguments. Elles doivent être conformes au type de la variable, aux valeurs minimales, maximale et au pas éventuels ou aux valeurs énumérées éventuelles. Dans notre exemple, la valeur doit être comprise entre 0 et 100.

La troisième étape est l'actualisation des valeurs internes avec les arguments. Si ces valeurs concernent des variables qui envoient des événements, il faut actualiser les valeurs des variables sous forme de chaîne de caractères de la table du serveur, et demander l'envoi des événements. Ces actions sont effectuées par une fonction séparée, `update_state_variable`.

Dernière étape, la construction de la réponse. Bien que cette action n'aie pas de paramètre ou de valeur de retour, il faut générer une réponse, pour que l'appelant sache que tout s'est bien passé. La réponse contiendra juste le nom de l'action et le serviceType.

```

int Dimming_SetLoadLevelTarget(IN IXML_Document * in, OUT IXML_Document
**out, OUT char **errorString){
    char *new_load_level;
    char value[4];
    int temp_level;
    *out = NULL;
    *errorString = NULL;

    if (!(new_load_level =
          get_argument_value(in, "NewLoadLevelTarget"))){
        *errorString = "invalid argument for SetLoadLevelTarget";
        return UPNP_E_INVALID_PARAM;
    }

    if (new_load_level[0] < '0' || new_load_level[0] > '9'){
        *errorString = "invalid argument for SetLoadLevelTarget";
        free(new_load_level);
        return UPNP_E_INVALID_ARGUMENT;
    }
    temp_level = atoi(new_load_level);
    free(new_load_level);
    if (temp_level < 0 || temp_level > 100){
        *errorString = "invalid argument for SetLoadLevelTarget";
        return UPNP_E_INVALID_ARGUMENT;
    }
}

```

```

    }

    if (temp_level != load_level){
        load_level = (unsigned char)temp_level;
        printf("Luminosité modifiée à %d pourcent\n", load_level);

        sprintf(value, "%d", load_level);
        update_state_variable(SERVICE_DIMMING,
            VARIABLE_LOADLEVELSTATUS, value);
    }
    *out = UpnpMakeActionResponse("SetLoadLevelTarget",
        root.services[SERVICE_DIMMING].service_type,
        0, NULL);

    return UPNP_E_SUCCESS;
}

```

Examinons les fonctions `get_argument_value()` et `update_state_variable()`. La LibUPnP comporte un module d'analyse XML de type DOM, le iXML. C'est celui que nous allons utiliser pour récupérer la valeur de l'argument. Heureusement, il existe une fonction de recherche d'éléments par leur nom. Nous l'utilisons en lui passant comme argument le nom de la variable à retrouver dans le document XML. La valeur du fils de cet élément est celle qui nous intéresse, la valeur de l'argument de l'action.

```

char *get_argument_value(IXML_Document * doc, const char *item){
    IXML_NodeList *nodeList = NULL;
    IXML_Node *textNode = NULL;
    IXML_Node *tmpNode = NULL;

    char *ret = NULL;

    nodeList = ixmldocument_getElementsByTagName(doc, (char *)item);
    if(nodeList) {
        if((tmpNode = ixmllist_item(nodeList, 0))) {
            textNode = ixmlNode_getFirstChild(tmpNode);
            ret = strdup(ixmlNode_getnodeValue(textNode));
        }
        ixmllist_free( nodeList );
    }
    return ret;
}

```

Nous avons décidé de protéger l'accès aux variables d'état sous forme de chaînes de caractères (utilisées pour les réponses d'actions et les évènements) par des exclusions mutuelles, pour ne pas avoir deux threads qui modifient en même temps les valeurs ou qu'un thread lise des données incorrectes lorsqu'un autre est en train de les écrire. La fonction `UpnpNotify` permet d'envoyer un évènement avec des modifications sur plusieurs variables d'un même Service.

```

int update_state_variable(int service, int variable, char *value){
    pthread_mutex_lock(&device_mutex);
    if (!strcmp(root.services[service].var_values[variable], value)){
        pthread_mutex_unlock(&device_mutex);
        return 1;
    }

    strcpy(root.services[service].var_values[variable], value);
}

```

```

        UpnpNotify(handle,
                    root.UDN,
                    root.services[service].service_id,
                    (const char **)
                        &root.services[service].var_names[variable],
                    (const char **)
                        &root.services[service].var_values[variable],
                    1);
    pthread_mutex_unlock(&device_mutex);
    return 0;
}

```

#### 4.3.2.3 Troisième étape : gestion des abonnements aux évènements.

Nous avons vu toutes les étapes de la gestion des actions de notre serveur UPnP. Ce n'est pas tout. Notre fonction de rappel (callback), nous l'avons vu, nous informe d'autres requêtes, particulièrement des demandes de souscriptions aux notifications de variables d'état d'un Service. La fonction suivante effectue cette tâche. Comme pour le gestionnaire d'actions, pour chaque requête, on vérifie que l'UDN est bien celui du serveur, et que le serviceId existe bien. Lorsqu'on accepte une souscription, on doit envoyer l'état actuel des variables concernées. La fonction UpnpAddToPropertySet regroupe les noms et les valeurs de ces variables, et UpnpAcceptSubscription se charge d'envoyer les données relatives à l'acceptation de la souscription, celles-ci contenant (entre autres) un UUID qui sera utilisé par le client pour déterminer la provenance des futurs évènements.

```

int handle_subscription_request(struct Upnp_Subscription_Request *request) {
    int i, j;
    IXML_Document *PropSet = NULL;

    if (strcmp(request->UDN, root.UDN)) {
        fprintf(stderr, "Bad UDN for subscription (%s != %s)\n",
                request->UDN, root.UDN);
        return UPNP_E_INVALID_PARAM;
    }

    pthread_mutex_lock(&device_mutex);
    for (i=0; i<root.nb_services; i++) {
        if (!strcmp(request->ServiceId, root.services[i].service_id)) {
            for (j=0; j< root.services[i].nb_vars; j++)
                UpnpAddToPropertySet(&PropSet,
                                    root.services[i].var_names[j],
                                    root.services[i].var_values[j]);

            UpnpAcceptSubscriptionExt(handle, request->UDN,
                                      request->ServiceId,
                                      PropSet, request->Sid);
            ixmldocument_free(PropSet);
            break;
        }
    }
    pthread_mutex_unlock(&device_mutex);

    if (PropSet == NULL){
        fprintf(stderr, "Bad serviceId for subscription\n");
        return UPNP_E_INVALID_PARAM;
    }
    printf("accepted subscription to %s\n", request->ServiceId);
}

```

```
    return UPNP_E_SUCCESS;
}
```

La réactualisation et l'annulation d'abonnements sont automatiquement gérées par la LibUPnP.

#### 4.3.2.4 Quatrième étape optionnelle : présentation.

La dernière fonctionnalité d'UPnP que nous n'avons pas étudié est la présentation. Comme nous l'avons vu dans les exemples, en pratique les pages de présentation sont parfois plus utilisées que les actions. Elles ont l'avantage de ne pas nécessiter le développement d'un client spécifique, puisque les navigateurs Web sont déjà très répandus. Nous allons donc poser les problèmes classiques de la génération des pages dynamiques, après avoir implémenté la fonctionnalité de présentation dans notre lampe.

Lors de l'initialisation, nous indiquons au SDK dans quel répertoire se trouvent les fichiers qui seront disponibles sur le serveur HTTP. On peut donc faire pointer la page de présentation sur un fichier existant dans ce répertoire.

La page de présentation d'un dispositif est en général utilisée pour fournir des informations sur l'état interne du serveur. Dans notre cas, on peut imaginer que l'on souhaite avoir une page Internet reflétant l'état de la lampe virtuelle, avec éventuellement une image de lampe éteinte ou allumée suivant l'état, et des informations ou des statistiques sur le serveur. Une telle page doit bien sûr être générée, puisqu'elle est dynamique.

Plusieurs méthodes sont possibles : pour les serveurs s'exécutant sur des dispositifs ayant une puissance de calcul assez bonne ou ne fonctionnant pas sur batterie, on peut générer la page HTML à chaque fois qu'une variable est modifiée ou qu'une action est appelée, si on veut des statistiques. Dans le cas où la page serait complexe et que la génération du code entier prendrait trop de temps, on peut envisager une solution de recollage de morceaux : on stocke les portions statiques de la page dans des fichiers, que l'on concatène avec les informations dynamiques. Effectuer de telles opérations à chaque action, alors que la page de présentation ne sera, en général, accédée que rarement, n'est vraiment pas optimal. Pour les appareils mobiles, effectuer ces opérations inutiles occupe le processeur et donc consomme de la batterie, deux inconvénients qui en font une mauvaise solution. De plus, le matériel embarqué utilise souvent un système de fichiers en lecture seule, écrire le fichier devient alors impossible.

Heureusement, il existe une solution dans la LibUPnP : les répertoires virtuels du serveur HTTP. Leur interface n'est pas des plus simples, mais ils permettent de générer une page HTML seulement lorsqu'elle est demandée, de donner la même page pour plusieurs noms de fichiers alors qu'ils n'existent pas sur le système de fichiers. On peut effectuer toutes les opérations que l'on veut sur un répertoire qui virtuel.

Si on souhaite utiliser cette fonctionnalité pour la page de présentation principale du serveur, il faut penser à modifier l'URL dans le fichier de description, qui pointe normalement vers un fichier réel.

Plus concrètement, on utilise la fonction `UpnpSetVirtualDirCallbacks()` pour mettre en place les fonctions qui seront appelées lorsqu'un client demande l'accès à un fichier d'un répertoire virtuel, et `UpnpAddVirtualDir()` pour informer le serveur HTTP qu'un certain répertoire sera virtuel. L'implémentation d'un répertoire virtuel revient un peu à développer un système de fichiers virtuel : il faut implémenter les fonctions de test d'existence et d'obtention d'informations (`get_info`), d'ouverture

(open), de lecture (read), d'écriture (write), de déplacement de tête de lecture (seek) et de fermeture d'un fichier (close). Les pointeurs sur ces fonctions sont enregistrés dans une structure, qui est passée comme argument à UpnpSetVirtualDirCallbacks(). Pour des pages simplement générées et accédées classiquement, les fonctions write et seek ne sont pas utilisées.

Quand un client effectuera un GET en HTTP sur un fichier se trouvant sous le répertoire configuré par UpnpAddVirtualDir(), le serveur de la LibUPnP appellera en premier notre fonction get\_info, qui devra donner des informations sur le fichier virtuel si on veut le servir, ou renvoyer -1 sinon. Parmi ces informations, on retrouve la taille du fichier, sa date de modification, si c'est un répertoire, s'il est lisible et son type MIME. Si le fichier "existe", la fonction open sera appelée. Elle devra retourner des données propres à l'implémentation, par exemple une structure contenant le flot de données du fichier, sa taille et l'emplacement de la tête de lecture. Ces données serviront de handle lors des appels de lecture, écriture, déplacement de tête de lecture, et fermeture du fichier. Par exemple le serveur appellera notre fonction read, avec une taille arbitraire de données à lire, nous copierons autant d'octets des données du fichier virtuel et nous avanceront la tête de lecture de cette même taille, si elle est plus faible que la taille du fichier bien entendu.

En résumé, cette méthode est plus complexe que de générer les pages web à chaque action, mais bien plus puissante, économique et flexible.

#### 4.3.3 Point de contrôle

Maintenant que nous disposons d'un serveur UPnP, nous souhaiterions pouvoir communiquer avec lui. Nous allons donc écrire un point de contrôle qui recherche et découvre ce serveur, s'abonne à ses Services, et permet d'invoquer ses actions.

Le principe de programmation du point de contrôle est assez semblable : à l'initialisation on fournit une fonction de rappel qui sera appelée principalement lorsqu'un message SSDP (découverte) ou un événement sont reçus. Il faut aussi faire attention aux threads qui appellent cette fonction simultanément.

La première fonction à appeler est ici aussi UpnpInit(), éventuellement avec une IP et un port, si la machine dispose de plusieurs interfaces réseau. On indique ensuite à la bibliothèque que l'on crée un client :

```
UpnpRegisterClient(callback, NULL, &handle)
```

De même que pour le serveur, on retrouve la fonction de rappel et le handle. Le deuxième argument, auquel on donne la valeur NULL, est appelé cookie. C'est un pointeur sur des données propres à l'utilisateur, que l'on souhaiterait retrouver dans la fonction de rappel, la LibUPnP l'appelant avec ce troisième argument. On trouve cette fonctionnalité dans le serveur également.

Le point de contrôle est maintenant opérationnel, on peut effectuer une recherche de serveurs. Dans notre cas, on souhaite se limiter au serveur correspondant à notre lampe virtuelle, on va donc effectuer une recherche sur type de device :

```
char *type = "urn:schemas-upnp-org:device:DimmableLight:1";
UpnpSearchAsync(handle, 5, type, NULL);
```

La recherche est de type asynchrone, la fonction retourne immédiatement, mais notre fonction de rappel sera appelée pour chaque réponse reçue :

```
int callback(Upnp_EventType event_type, void *event, void *cookie) {
```

```

    struct Upnp_Discovery *d_event;

    switch (event_type) {
        case UPNP_DISCOVERYADVERTISEMENT_ALIVE:
        case UPNP_DISCOVERYSEARCH_RESULT:
            d_event = (struct Upnp_Discovery *)event;
            add_device(d_event);
            break;
        case UPNP_DISCOVERYADVERTISEMENT_BYEBYE:
            d_event = (struct Upnp_Discovery *)event;
            pthread_mutex_lock(&DeviceListMutex);
            if ((dev = is_present(d_event->DeviceId)) >= 0){
                printf("Le device %s a quitté\n",
                       d_event->DeviceId);
                remove_device(dev);
            }
            pthread_mutex_unlock(&DeviceListMutex);
            break;
    }
}

```

Pour chacun de ces évènements (résultat de recherche ou découverte), les champs `DeviceId` et `Location` seront présents dans la structure `d_event`. Le `device` ID correspond à l'UDN de la description du `device`, il est donc supposé unique. Lorsqu'on effectue une recherche moins spécifique, par exemple de type `ssdp:all`, ou qu'un serveur s'annonce, les messages que l'on peut recevoir sont variés, et en général en double. Notre fonction de rappel est appelée à chaque réception de message, la fonction `add_device()` qui ajoute un `device` dans la base de données du point de contrôle doit donc vérifier qu'il n'existe pas déjà. Elle doit aussi verrouiller une variable d'exclusion mutuelle pour ne pas ajouter deux fois de suite le même `Device` ou `Service`. Lorsqu'on reçoit un message contenant le `deviceType` et qu'il est celui recherché pour notre lampe (`urn:schemas-upnp-org:device:DimmableLight:1`), on télécharge la description XML associée. On en extrait ensuite les diverses URL utilisées pour les `Services`, le `serviceType` et le `serviceId`. Si ces deux derniers ne correspondent pas à ceux attendus pour les deux `Services` de notre lampe, nous abandonnons ce faux serveur. Dans le cas contraire, nous nous abonnons aux changements de variables d'état des deux `Services`.

```

int add_device(struct Upnp_Discovery *d_event){
    int dev = -1;

    pthread_mutex_lock(&DeviceListMutex);

    if (!strcmp(d_event->DeviceType,
                "urn:schemas-upnp-org:device:DimmableLight:1") &&
        (dev = is_present(d_event->DeviceId)) == -1){
        struct device *new_dev = malloc(sizeof (struct device));
        new_dev->device_id      = strdup(d_event->DeviceId);
        new_dev->location        = strdup(d_event->Location);
        new_dev->device_type     = strdup(d_event->DeviceType);
        new_dev->description     = NULL;
        new_dev->services        = NULL;
        new_dev->nb_services     = 0;
        dev = nb_devices;
        roots[nb_devices] = new_dev;
        nb_devices++;
    }
}

/* téléchargement de la description XML */

```

```

        if (UpnpDownloadXmlDoc(new_dev->location,
                               &(new_dev->description)) != UPNP_E_SUCCESS) {
            new_dev->description = NULL;
            remove_device(dev);
            dev = -1;
            goto end;
        }
        /* récupération des informations de la description */
        if (get_services_infos(new_dev) != 2) {
            fprintf(stderr, "La DimmableLight n'a pas les deux
services attendus\n");
            remove_device(dev);
            dev = -1;
            goto end;
        }
        /* abonnement aux services */
        subscribe_to_service(&(roots[dev]->services[0]));
        subscribe_to_service(&(roots[dev]->services[1]));
    }
end:
    pthread_mutex_unlock(&DeviceListMutex);
    return dev;
}

```

La souscription à un *Service* se fait par la fonction `UpnpSubscribe()`, qui prend comme argument le handle, l'URL d'abonnement (l'élément `eventSubURL` du fichier de description), et un temps de validité. En résultat, on retrouve l'identifiant de l'abonnement (SID), le temps effectif de la souscription, et un code de retour pouvant prendre de nombreuses valeurs suivant les erreurs. L'exemple ci-dessous est assez simpliste, il ne fournit pas de message d'erreur lorsque l'abonnement échoue.

```

int subscribe_to_service(struct service *s){
    int retval, subscribe_time = 600;
    retval = UpnpSubscribe(handle, s->event_url,
                           &subscribe_time, s->event_sid);
    switch (retval){
        case UPNP_E_SUCCESS:
            printf("souscrit à %s pour %d secondes\n SID : %s\n",
                   s->service_type, subscribe_time, s->event_sid);
            return 0;
        default:
            fprintf(stderr, "erreur lors de l'abonnement\n");
            return 1;
    }
}

```

Nous avons découvert notre serveur, nous avons vérifié sa validité sur certains points, et nous nous sommes abonnés à ses *Services*. Le point de contrôle est un outil interactif de dialogue avec le serveur, il nous faut pouvoir invoquer des actions et afficher les évènements. Les évènements sont reçus par l'intermédiaire de la fonction de rappel ; on ne peut cependant pas les afficher directement, leur format étant aussi en XML. Nous allons donc ajouter un `case` au `switch` de la fonction `callback`, et utiliser une fonction simple de récupération de nom de variable et valeur du XML de l'évènement :

```

case UPNP_EVENT_RECEIVED:
    e_event = (struct Upnp_Event *)event;
    printf("event: %s %d\n", e_event->Sid, e_event->EventKey);

```

```

        ev = parse_single_event(e_event->ChangedVariables);
        printf("\t%s\n", ev);
        free(ev);
        break;

/* attention : les tests de nullité ont été supprimés pour des raisons de
simplicité et clarté */
char *parse_single_event(IXML_Document * doc){
    IXML_Node *var_node, *val_node;
    char *ret, *value;
    const char *var_name;
    var_node = ixmXmlNode_getFirstChild(ixmXmlNode_getFirstChild(
                                            ixmXmlNode_getFirstChild(&(doc->n))));
    var_name = ixmXmlNode_getLocalName(var_node);

    val_node = ixmXmlNode_getFirstChild(var_node);
    value = ixmXmlNode_getNodeValue(val_node);

    ret = malloc(strlen(var_name) + strlen(value) + 4);
    sprintf(ret, "%s = %s", var_name, value);
    return ret;
}

```

Dans la fonction ci-dessus, on part du principe qu'il n'y a qu'une seule variable qui envoie des événements par *Service*, et on ne cherche pas à en afficher plusieurs.

L'interface de notre point de contrôle est uniquement textuelle. Nous allons lire des lignes sur l'entrée standard et en extraire des commandes simples, pour invoquer des actions sur le serveur. Les résultats de ces actions et les événements en résultant seront affichés sur la sortie standard.

Notre `main()` pourra par exemple contenir le code suivant, qui commande l'allumage et l'extinction de la lampe lorsque l'on tape `on` ou `off` :

```

while (fgets(buf, 200, stdin)) {
    len = strlen(buf);
    buf[len-1] = '\0';
    if (len > 2 && (buf[0] == 'o' || buf[0] == 'O'))
        invoke_action(&(roots[0]->services[1]),
                      "SetTarget", "newTargetValue",
                      (buf[1] == 'n' || buf[1] == 'N') ? "1":"0");
}

```

Comme on le voit, il faut fournir le nom de l'action, de l'argument et sa valeur. Dans un point de contrôle plus générique, l'idéal serait de récupérer le fichier de description des *Services* pour connaître la liste des arguments et ne passer que leurs valeurs pour effectuer les appels.

L'invoque d'une action se fait en deux étapes :

- création du nœud XML de l'action, contenant son nom, le `serviceType`, et les paramètres éventuels,
- invocation, en incorporant ce nœud dans le document SOAP, et en envoyant la requête sur l'URL de contrôle du *Service*.

De nombreuses fonctions du SDK existent en version synchrone et asynchrone. Dans notre cas, le point de contrôle ne fait rien d'autre qu'attendre les commandes de l'utilisateur, les versions synchrones conviennent. Pour des utilisations plus

complexes, les appels asynchrones permettent au processeur de travailler en attendant une réponse, par exemple pour rafraîchir une interface graphique. De même que pour UpnpSubscribe(), UpnpSendAction() retourne de nombreux codes d'erreurs ayant une signification précise, et dans cet exemple nous n'en tenons pas compte pour une meilleure lisibilité. Dans notre lampe, les actions ne prenant pas d'argument en ont un en paramètre de retour, nous récupérons donc la réponse du XML (SOAP) avec la fonction `get_response()` lorsqu'il n'y a pas d'argument.

```
char *invoke_action(struct service *s, char *action, char *param, char
*val) {
    IXML_Document *action_node = NULL, *out_node;
    int retval;
    char *ret = NULL;
    if (UpnpAddToAction(&action_node, action, s->service_type, param,
val) != UPNP_E_SUCCESS){
        fprintf(stderr, "Echec dans l'ajout de parametre\n");
        return ret;
    }
    retval = UpnpSendAction(handle, s->control_url, s->service_type,
                           NULL, action_node, &out_node);
    switch(retval){
        case UPNP_E_SUCCESS:
            if (param == NULL)
                ret = get_response(out_node);
            ixmldocument_free(out_node);
            break;
        default:
            fprintf(stderr, "L'appel a échoué.\n");
    }
    return ret;
}
```

Nous pouvons maintenant compiler et lancer le serveur et le point de contrôle. Si on lance le serveur en premier, le point de contrôle effectuant une recherche à son lancement, il découvrira le serveur. Si le point de contrôle est lancé en premier, le serveur à son lancement annonce sa présence et il sera aussi découvert.

Le point de contrôle s'abonnera aux Services de la lampe virtuelle, et sera prêt à invoquer les actions demandées par l'utilisateur, et recevoir les évènement en résultant. On pourra aussi ouvrir la page de présentation du serveur dans un navigateur Web.

## 5 Perspectives et conclusion

### 5.1 Vers des Services Web pour dispositifs...

Tout au long de cet article nous avons vu les motivations d'UPnP, et les protocoles qui autorisent ses fonctionnalités. Quand on étudie plus en détail la pile UPnP, on se rend vite compte que les protocoles employés sont en partie ceux des Services Web, auxquels ont été ajoutés les fonctionnalités de recherche et découverte et de notification d'évènements.

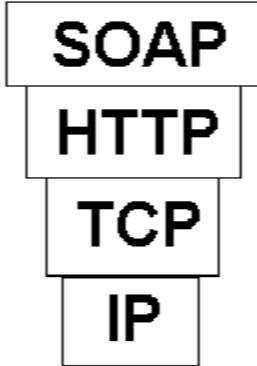


Figure 8 - pile des protocoles des Services Web

Ces nouvelles fonctionnalités respectent les recommandations du W3C concernant les Services Web, couvrant les couches du standard OSI :

- couche réseau : IP.
- couche transport : TCP (UPnP utilise aussi de l'UDP).
- couche session : HTTP. L'utilisation du HTTP permet de passer plus facilement les pare-feux.
- couche présentation : XML pour représenter les données. XML est utilisé aussi bien pour la description des services que pour les passages de messages :
  - descriptions : WSDL pour les Services Web classiques, une grammaire spécifique pour UPnP.
  - SOAP : *Simple Object Access Protocol*, pour invoquer les méthodes sur les Services Web.

Nous pouvons donc faire émerger, avec UPnP, un nouveau type de Services Web : les Services Web pour dispositifs [1]. Les Services Web classiques ne permettent pas de rechercher et découvrir de nouveaux services sur un réseau local par diffusion, ils utilisent plutôt des annuaires pour les recenser. Les services doivent s'y annoncer à leur lancement et terminaison, et les clients n'ont que l'annuaire comme interlocuteur pour obtenir une référence vers les services.

Les notifications par événements sont aussi une fonctionnalité ajoutée, les Services Web n'étant utilisés que pour invoquer des actions et obtenir leur réponse. Pour une utilisation sur dispositif, il est indispensable de disposer des événements, pour éviter de faire des attentes actives (*polling*) qui sont souvent coûteuses et non efficaces. On bénéficie donc toujours de la notion d'interruption matérielle au travers des Services Web pour dispositifs.

UPnP utilise IP pour ces communications, ce qui le limite de-facto à des environnements capables d'implémenter une pile IP. Cependant l'utilisation de ponts (*bridges*) entre les différents réseaux ou protocoles de transport résout ce problème [3]. De plus, l'implémentation de services dans les dispositifs avec des ressources limitées (batterie, microcontrôleur, peu de mémoire embarquée...) étant peu adéquate, un pont sur une machine plus puissante permet de faire le lien entre leur protocole natif et une interface de services ou Services Web pour dispositifs.

UPnP est aussi un standard émergent pour le contrôle à distance d'équipements domestiques (du multimédia au HVAC en passant par le monde du PC). UPnP

permet (en théorie) de faire le lien entre la micro-informatique personnelle (type Mac ou PC) et l'informatique enfouie dans le matériel.

## 5.2 Conclusion

Après avoir expliqué les objectifs premiers d'UPnP, nous avons détaillé des cas d'utilisation et les technologies utilisées depuis le protocole jusqu'à la conception même d'un premier dispositif et point de contrôle UPnP. UPnP peut-être vu comme une étape importante dans le prolongement de PnP où la connexion locale au PC des périphériques est remplacée par la connexion à distance des périphériques d'un réseau local. Après la distribution de disques, de fichiers, de CPU, voire de périphériques d'entrées/sorties tels que l'affichage au travers des serveurs graphiques, UPnP complète la panoplie du « partage réseau » en permettant la distribution sur un réseau local de tout périphérique connecté.

Eu égard aux choix technologiques qui ont été faits pour répondre au cahier des charges d'UPnP, cette approche ouvre d'autres perspectives moins attendues. En effet UPnP peut se concevoir comme une première extension des Services Web du W3C qui offriraient des fonctionnalités adaptées à la gestion de périphériques et plus largement de dispositifs (Services Web pour dispositifs). Vu sous cet angle UPnP présente alors certaines lacunes. Nous pouvons citer par exemple : des événements liés à une et une seule variable d'état du dispositif, des limitations dans le codage des types complexes, dans les mécanismes de recherche et découverte et l'absence totale de gestion de sécurité. Malgré cela le nombre de dispositifs UPnP est en pleine croissance, surtout dans le monde des équipements réseaux et maintenant multimédia, basés sur les récents standards HAVi/UPnP.

Cette démarche s'inscrit tout à fait dans l'évolution des sciences et technologies pour faciliter l'innovation dans le domaine des applications « Machine to Machine » ou M2M [2]. A la manière des Services Web pour les applications de traitements distribués d'informations, les Services Web pour dispositifs jouent un rôle prépondérant dans la classe des applications « Device to Device », plus interactives mettant en jeux des équipements communicants hétérogènes.

S'il était besoin de se convaincre de l'intérêt du sujet, il faut savoir que les successeurs d'UPnP sont déjà en préparation comme DPWS (*Device Profile for Services Web*) de Microsoft, qui sont cette fois directement annoncés comme Services Web étendus, contrairement à UPnP qui étend le concept de « Plug and Play ». UPnP est donc un précurseur dans ce domaine et doit donc encore avoir quelques beaux jours devant lui.

Pour en savoir plus

## Bibliographie

- [1] V. Hourdin, S. Lavirotte, J-Y. Tigli, "Étude et comparaison des systèmes de services pour dispositifs", Rapport de recherche, 2006.
- [2] FING, Machine To Machine (M2M) : enjeux et perspectives, mars 2006, Livre Blanc produit par la FING, Syntec Informatique et Orange, <http://www.fing.org>.
- [3] Qi He, Dan Muntz. "Multicast Gateway for Service Location in Heterogeneous Ad Hoc Communication", 2002. HP Technical Reports.
- [4] Marc Haase, Igor Sedov, Stephan Preuss, Clemens Cap, Dirk Timmermann,

Time and Energy Efficient Service Discovery in Bluetooth, in Proceedings of the 57th IEEE Semiannual Vehicular Technology Conference, Band I, S. 418-422, ISBN: 1090-3038, Jeju, Korea, 2003.

## Normes

## Réglementation

## Organismes

## Constructeur – Fournisseurs – Distributeurs

## Thèses

## Sites Internet

SLP : <http://srvloc.sourceforge.net/>

Bonjour : <http://www.apple.com/macosx/features/bonjour/>

Bluetooth : <http://www.bluetooth.org/>

Forum UPnP : <http://upnp.org/>

UPnP architecture : [http://www.upnp.org/download/UPnPDA10\\_20000613.htm](http://www.upnp.org/download/UPnPDA10_20000613.htm)

SOAP : <http://www.w3.org/TR/soap/>

XML: <http://www.w3.org/TR/2000/REC-xml-20001006>

HAVi : <http://havi.org/>