URBI Tutorial for urbi 0.9

Basé sur URBI revision 142

Jean-Christophe Baillie
Antoine Robin (traduction de l'anglais)

URBI Tutorial for urbi 0.9: Basé sur URBI revision 142

par Jean-Christophe Baillie et Antoine Robin (traduction de l'anglais)

Publié

Table des matières

Liste des illustrations

Chapitre 1. Introduction

URBI (*Universal Robotic Body Interface*) est un langage de script conçu pour fonctionner selon un mode client/serveur dans le but de contrôler un robot, ou plus largement, tous les types d'appareils disposant de moteurs et de capteurs. Comme nous allons le voir dans ce tutoriel, URBI est plus qu'un simple outil de pilotage, il est un système universel de contrôle du robot apportant des fonctionnalités supplémentaires via des plugins et autorisant la création d'applications interactives complexes ouvertes.

Les principales qualités d'URBI sont les suivantes :

- Simplicité: Facile à comprendre et puissant à la fois, URBI convient aussi bien à l'enseignement qu'au développement d'applications professionnelles.
- Flexibilité: Indépendant du robot, de l'OS, de la plateforme, URBI est également interfaçable avec de nombreux langages (*C++*, *Java*, *Matlab*...).
- Modularité: L'architecture de composants orientés objet permet d'étendre les possibilités d'URBI grâce à une multitude de langages. Les composants peuvent être externes au serveur URBI (*remote*) ou intégrés (*plugin*).
- Traitements parallèles: Exécution parallèle de commandes, gestion des accès concurrentiels aux variables, programmation évenementielle, ...

Le point probablement le plus important de ce tutoriel est le suivant : URBI a été conçu dès le début avec un souci constant de simplicité. Il n'y a pas de philosophie ou d'architecture complexe à assimiler. Il est compréhensible en quelques minutes et peut être utilisé immédiatement. Que vous développiez une application se contentant de bouger quelques articulations du robot ou une application implémentant une intelligence artificielle complexe, URBI vous fournira les outils pour vous simplifier la vie.

URBI est disponible pour beaucoup de robots et leur nombre augmente. Actuellement, il existe une version d'URBI pour Aibo [http://fr.wikipedia.org/wiki/Aibo], pour l'humanoïde HRP-2 [http://fr.wikipedia.org/wiki/HRP-2], pour le simulateur universel Webots [http://www.cyberbotics.com/products/webots/], les robots Pioneer, l'iCat [http://www.research.philips.com/technologies/syst_softw/robotics/index.html] de Philips et d'autres humanoïdes vont suivre.

La compatibilité avec le simulateur Webots signifie qu'il est possible de passer du véritable robot à sa simulation via un simple changement d'adresse IP, et cela rend URBI particulièrement adapté à cet usage.

Dans ce tutoriel, nous avons essayé de faire une description progressive d'URBI qui s'étend des commandes simples pour les moteurs à de la programmation plus complexe ainsi qu'aux composants logiciels intégrés. Tout ceci a été rédigé dans le but d'être compréhensible par des personnes ayant peu ou pas d'expérience en matière de robotique et de programmation (exception faite des sections traitant du C++ qui exigent un minimum d'aisance avec ce langage). Cependant, de temps en temps, nous avons fourni des explications ou des compléments qui resteront probablement opaques au lecteur lambda.

Chapitre 2. Installer URBI

Nous ne pouvons montrer en détail dans ce tutoriel comment installer URBI pour chaque robot actuellement supporté, mais l'idée générale est d'avoir un serveur URBI chargé et démarré sur votre robot. La marche à suivre est normalement décrite dans le fichier `INSTALL` de l'archive que vous avez téléchargée. Idéalement, URBI est préinstallé sur votre robot.

Etant donné que nous allons donner beaucoup d'exemples appliqués à l'Aibo dans ce tutoriel, nous fournissons ici les instructions d'installation d'URBI sur Aibo. Nous expliquerons également comment installer URBILab qui est un client graphique multi-environnements à la fois simple et pratique pour remplacer *telnet*.

Préparer le memorystick pour Aibo

Tout d'abord, téléchargez le memorystick correspondant à votre robot. Deux possibilités s'offrent à vous pour l'instant :

- ERS2xx : http://www.urbiforge.com/ers200
- ERS7 : http://www.urbiforge.com/ers7

Instructions rapides :

Décompressez l'archive et copiez le contenu du dossier `MS-xxx` sur un memorystick vierge, puis mettez le fichier `WLANCONF.TXT` à jour en l'adaptant à la configuration de votre réseau.

Instructions détaillées :

1. Décompressez l'archive correspondant à votre Aibo. Vous devriez obtenir un dossier nommé `MS-ERS7` ou `MS-ERS200`. Entrez dans ce répertoire.

2. A partir du dossier `MS-ERS7` (ou `MS-ERS200`), allez dans `OPEN-R/SYSTEM/CONF`. Il devrait y avoir un fichier `WLANCONF.TXT` (sinon vous devez le créer), pour configurer le réseau convenablement. Il n'existe aucune documentation officielle sur la façon de modifier `WLANCONF.TXT`, mais voici un exemple dont vous pouvez vous inspirer :

```
HOSTNAME=aibo.mondomaine.com
ETHER_IP=192.168.1.111 # <— votre IP ici
#
# Réseau sans fil
#
ESSID=0a3902 # <— votre SSID ici
WEPENABLE=1 # <— WEP ou non
WEPKEY=0x4B2241785B # <— la clé: hexadécimale
#WEPKEY=ABCDE # <— en ASCII pour ERS2xx
APMODE=1

#
# Réseau IP
```

```
#
USE_DHCP=0
SSDP_ENABLE=1

# Cette partie est facultative
# Votre configuration réseau ici —>
ETHER_NETMASK=255.255.255.0
IP_GATEWAY=192.168.0.3
DNS_SERVER_1=192.168.1.1
```

Vous pouvez utiliser URBI sur un Aibo sans le réseau si vous ne disposez pas d'équipement Wi-Fi en appelant vos programmes depuis le fichier `URBI.INI`.

3.

Copiez le contenu du dossier `MS-ERS7` ou `MS-ERS200` à la racine d'un memorystick rose spécialement conçu pour la programmation (un "PMS"). [1] Prenez garde à ne pas utiliser le memorystick contenant *Aibo Mind* ou un memorystick bleu : à l'heure actuelle, il est indispensable d'acheter un memorystick spécial programmation à *Sony*, ce n'est malheureusement pas inclus dans les accessoires de base. Une fois fait, insérez le stick et démarrez le robot. Votre robot URBI est prêt.

Vous pouvez ouvrir une connexion *telnet* [2] sur le port *54000* du robot pour vous assurer que tout est OK :

```
telnet aibo.gostai.com 54000
```

Vous devriez voir s'afficher le message d'accueil d'URBI qui se présente comme suit:

```
[00020380:start] *** ************************************************************
[00020380:start] *** URBI Language specif 1.0 - Copyright (C) 2006  Gostai SAS
[00020380:start] *** URBI Kernel version 1.0 rev. 100
[00020380:start] ***
[00020380:start] ***   URBI Engine 1.0 for Aibo ERS2xx/ERS7 Robots
[00020380:start] ***   (C) 2004-2006 Gostai SAS
[00020380:start] ***
[00020380:start] *** URBI comes with ABSOLUTELY NO WARRANTY;
[00020380:start] *** This software is free, and you are welcome to use
[00020380:start] *** it under certain conditions; see LICENSE for details.
[00020380:start] ***
[00020380:start] *** See http://www.urbiforge.com for news and updates.
[00020380:start] *** ************************************************************
[00020380:ident] *** ID: U595075704
```

Un des avantages de l'architecture client/serveur d'URBI est que vous pouvez immédiatement envoyer des commandes à votre robot via un simple client *telnet*. Bien entendu, il est envisageable d'interfacer URBI avec un programme écrit en *C++* ou en *Java*, comme nous le verrons prochainement en parlant de *liburbi* (chapitre La liburbi en C++). Nous allons pour l'instant nous contenter d'une interface *telnet*.

Toutefois, *telnet* est quelque peu spartiate (et ne marche pas toujours bien sous *Windows* [3]) et nous avons développé un logiciel multi-plateforme disposant d'une interface graphique que nous avons appelé *URBI Remote* et que nous vous encourageons à utiliser. *URBILab* est gratuit et mis à disposition sous license GNU-GPL. Vous pouvez le télécharger ici (disponible fin 2006) :

[3] La situation s'améliore si vous utilisez *Cygwin*, sans quoi les retours à la ligne sont mal interprétés par la commande *telnet* fournie dans *Windows*.

http://www.urbiforge.com/urbiremote

D'autres applications graphiques tierces-parties comme *Aibo-Telecommande* peuvent être téléchargées dès maintenant sur urbiforge.com [http://www.urbiforge.com/].

Chapitre 3. Premiers pas

Dans ce qui suit, nous allons étudier des exemples prévus pour l'Aibo, mais vous pouvez les transposer pour votre propre robot. Chaque partie du robot (capteurs, moteurs, caméra, ...) est un objet et possède un nom. Par exemple, en ce qui concerne l'Aibo, il existe deux moteurs dans sa tête qu'URBI nomme **headPan** et **headTilt**. L'objet associé à la caméra est appelé **camera**. Par la suite, nous emploierons le terme **device** pour désigner un objet qui gère une partie matérielle du robot. Ainsi nous parlerons du *camera device*, des *motor devices*, etc.

Vous trouverez la liste des devices pour votre robot en consultant sa documentation associée sur (http://www.urbiforge.com/doc), ou simplement en tapant la commande **group objects;**.

Affecter et lire une valeur associée à un moteur

Nous allons utiliser les moteurs dans ce qui suit, donc, avant toute chose, nous les mettons en route :

```
motors on;
```

motors off est bien sûr également disponible, comme normalement n'importe quel device ou objet avec **nom_du_device on/off**. Commençons par déplacer le moteur HeadPan à 30 degrés:

```
headPan = 30;
```

Maintenant, demandons quelle est la valeur du device HeadPan:

```
headPan;
[139464:notag] 30.102466
```

Le serveur répond avec un message (écrit ici en italique pour le distinguer des commandes) précédé d'une marque temporelle (*timestamp*) et d'une étiquette (*tag*) entre crochets. Ici la commande n'a été associée à aucune étiquette, par conséquent le terme **notag** est utilisé par défaut. Il est très simple d'associer une étiquette à une commande en URBI. Il suffit de précéder la commande d'un mot suivi du symbole deux-points (**:**):

```
monetiquette:headPan;
[139464:monetiquette] 30.102466
```

Le message possède maintenant l'étiquette **monetiquette**. Cela va s'avérer indispensable lorsqu'il s'agira de déterminer qui envoie quoi dans un contexte de commandes s'exécutant en parallèle, ou pour arrêter une commande en cours d'exécution en tâche de fond.

Vous pouvez essayer d'agir ainsi sur différents moteurs de l'Aibo tels **legRF1** ou **tailTilt**, ou bien vous amuser avec les lumières (**leds**) telles **ledF1** ou **ledBMC**, ou bien encore lire les valeurs de capteurs comme le détecteur de distance **distanceNear** ou l'accéléromètre avec **accelX**, **accelY** et **accelZ**. La syntaxe est toujours la même: **device = value;**.

En réalité **device = value** est compris par URBI comme **device.val = value;** qui affecte le paramètre **val** du device à **value**. Mais, afin de simplifier encore les choses pour les débutants, tous les devices d'Aibo ont un alias construit de la façon suivante:

```
alias headPan headPan.val
```

Du coup, vous n'avez pas à vous préoccupper d'ajouter le paramètre **val** pour les travaux basiques sur les moteurs. Il est toutefois possible de retirer ces alias (définis dans le fichier `URBI.INI`) avec la commande **unalias**.

Régler la vitesse, la durée ou la gestion sinusoïdale des mouvements

Les exemples précédents affectent la valeur d'un device aussi rapidement que le matériel le peut. Bien entendu, vous souhaitez certainement faire quelquechose de plus élaboré comme atteindre une certaine valeur dans un temps donné (en millisecondes):

```
headPan = 30 time:3000;
```

qui va atteindre la valeur 30 (degrés) en 3000ms. Quand vous avez à exprimer une valeur temporelle avec URBI, vous pouvez toujours préciser explicitement l'unité que vous souhaitez employer:

```
headPan = 30 time:3s;
headPan = 30 time:3000ms;
headPan = 30 time:3m;
headPan = 30 time:3h26m15s;
```

Vous pouvez utiliser des jours (**d**), des heures (**h**), des minutes (**m**), des secondes (**s**) et des millisecondes (**ms**), avec des valeurs décimales. Par défaut, l'unité est la milliseconde si aucune unité n'est précisée ou lorsqu'une variable est utilisée.

D'autrepart, vous pouvez également régler la vitesse à la laquelle la valeur doit être atteinte, exprimée en *unité/s*:

```
headPan = 30 speed:1.4;
```

Ou l'accélération (exprimée en *unité/s²*):

```
headPan = 30 accel:0.4;
```

Une manière très utile d'affecter une variable avec un profil dynamique est d'employer l'oscillation sinusoïdale:

```
headPan = 30 sin:2s ampli:20,
```

Cela va faire osciller le device HeadPan autour de 30 degrés avec une amplitude de 20 degrés et une période de 2 secondes. Remarquez que la commande se conclue par une virgule et non un point-virgule. Nous expliquerons ceci en détail plus tard, mais, en deux mots, la raison est que l'oscillation ne termine jamais et la virgule signifie en quelque sorte "mais fais tourner ceci en tâche de fond" pour permettre aux commandes suivantes d'être exécutées. Autrement, avec un point-virgule, aucune autre commande ne pourrait ensuite être exécutée puisque l'oscillation ne termine jamais. Il s'agit d'une erreur courante de la part des débutants en URBI.

time, **speed** et **sin** sont appelés des *modificateurs*. Il en existe beaucoup d'autres comme **phase**, **getphase** ou **smooth**. Rendez-vous dans le document "URBI Language Specification" pour une liste exhaustive des modificateurs.

Un modificateur particulièrement puissant est une fonction qui affecte une fonction temporelle complexe en tant que variable de trajectoire. Tout ceci est décrit dans "URBI Language Specification" et ne sera disponible qu'à partir de la version 2.0 des serveurs.

A la découverte des variables

Avec URBI, vous pouvez utiliser des variables. Affecter une valeur à **x** suffit à créer une variable nommée **x** qui sera locale à votre connexion:

```
x = 4;
x;
[146711:notag] 4.000000
```

Structure générale des variables

Avec la version 1.0 du noyau URBI, les noms de variable sont toujours de la forme **préfixe.suffixe** et lorsqu'aucun préfixe n'est donné, un préfixe local à votre connexion est silencieusement ajouté afin que ce **x** n'interfère pas avec le **x** d'une autre connexion.

Par exemple, quand vous tapez **x**, URBI va en réalité utiliser **U596851624.x** dans sa mémoire, **U596851624** étant l'identificateur de votre connexion courante (celle dans laquelle vous avez tapé **x**). De la même manière, les appels de fonction possèdent un espace local de travail (*namespace*) afin que d'éventuels appels récursifs puissent cohabiter sans risque d'interférence. Tout ceci va être entièrement revu dans URBI 2.0, avec une gestion avancée des noms de variables et d'espaces de travail.

Les valeurs d'un device et l'alias .val

Comme nous l'avons vu précédemment, il existe une exception importante à la règle qui dit que les variables sans préfixe sont locales: quand vous tapez **headPan**, URBI ne le traite pas comme une variable locale mais applique un alias qui traduit l'expression en **headPan.val**, qui est une variable URBI standard contenant la valeur du device. Donc en réalité, headPan ne réfère pas à une variable locale mais à la variable globale **headPan.val**. Les alias sont habituellement définis dans le fichier `URBI.INI`.

Créer une variable globale

Il n'y a pas vraiment de concept de variables *locales* ou *globales* dans cette version d'URBI. Tout est de la forme **préfixe.suffixe**. Sans préfixe, la variable est locale à la connexion mais vous pouvez inventer votre propre préfixe pour rendre votre variable "globale":

```
monprefixe.x = "salut";
```

En fait, **monprefixe** peut être vu et défini comme un objet URBI. Nous le verrons dans le chapitre Les objets avec URBI qui détaille l'aspect orienté objet d'URBI.

Les expressions

Le type d'une variable (numérique, chaîne de caractères, liste ou même binaire, comme nous le verrons plus tard) est automatiquement déduit par URBI.

Vous pouvez évaluer des expressions complexes, composées de variables ou de fonctions standard comme **sin**, **cos** ou **random** (voir le document "URBI Specification" pour consulter la liste complète):

```
x=pi/2;
calc:sqrt(1+sin(x));
[148991:calc] 1.414213
```

Une propriété intéressante est que les modificateurs présents au sein d'affectations complexes sont constamment ré-évaluées pour que, dans le cas où ils contiennent des variables, la valeur du modificateur puisse évoluer dans le temps en même temps que la variable évolue de son côté. Voici un exemple qui affecte à **x** une oscillation sinusoïdale à l'intérieur d'un champ sinusoïdal compris entre 15 et 25:

```
mon_amplitude = 20 sin:10s ampli:5,
x = 0 sin:2s ampli:mon_amplitude,
```

D'importantes interactions entre variables et devices peuvent ainsi être établies.

Les listes

Avec URBI, vous pouvez stocker des éléments dans une liste, simplement en les encadrant par des crochets:

```
maliste = [1,2,35.12,"salut"];
maliste;
[139464:notag] [1.000000,2.000000,35.120000,"salut"]
```

Ajouter de nouveaux éléments ou des listes entre elles se réalise simplement:

```
maliste = [1,2] + "salut";
maliste;
[146711:notag] [1.000000,2.000000,"salut"]
x = 1;
maliste + [45,x];
[148991:notag] [1.000000,2.000000,"salut",45.000000,1.000000]
```

Pour accéder successivement à chaque élément d'une liste, utilisez la commande **foreach**:

```
liste = [1,2];
foreach n in liste { echo n };
[151228:notag] *** 1.000000
[151228:notag] *** 2.000000
```

Pour des raisons purement techniques, le code exécuté dans une commande **foreach** doit être encadré par des accolades, même s'il n'est composé que d'une commande.

Vous pouvez accéder à n'importe quel élément en fournissant sa position dans la liste, comme avec les tableaux de la majorité des langages de programmation:

```
maliste = [1,2,"salut"];
maliste[2];
```

```
[146711:notag] "salut"
```

Pour accéder aux éléments d'une liste contenue elle-même dans une autre liste, utilisez des index multiples comme **maliste[3][4]**.

Enfin, vous pouvez demander le premier élément de la liste (ce que l'on nomme la *tête* ou **head**) et le reste (ce que l'on nomme la *queue* ou **tail**):

```
maliste = [1,2,"salut"];
head(maliste);
[146711:notag] 1.000000
tail(maliste);
[146711:notag] [2.000000,"salut"]
```

Exécuter des commandes en parallèle

Une commande URBI peut durer un certain temps. C'est une nouveauté parmi la plupart des autres langages. Nous avons vu précédemment que l'on peut affecter des valeurs avec une certaine durée, une certaine vitesse, voire même avec une évolution sinusoïdale sans fin. Il existe de nombreuses façons de faire fonctionner ces commandes en parallèle. Nous avons déjà vu comment le faire en utilisant une virgule pour séparer les commandes au lieu d'un point-virgule.

Il y a une autre manière pour indiquer que les commandes doivent s'exécuter en parallèle: utiliser **&** :

```
x=4 time:1s & y=2 speed:0.1;
```

La différence avec le séparateur virgule est qu'ici on force les deux commandes à démarrer exactement au même moment. En particulier, cela signifie que la première commande ne peut démarrer tant que la seconde n'est pas complètement disponible. Ainsi, taper **x=4 time:1s &** dans la console ne produira rien car URBI attendra alors de savoir ce qui suit, après le & (c'est la raison pour laquelle le séparateur virgule existe, car ce dernier est moins contraignant et permet de lancer les commandes intéractivement).

Dans la même idée, les commandes peuvent être lancées en série, exactement l'une après l'autre, en utilisant un séparateur | (appelé *pipe*, *tuyau* en anglais):

```
x=4 time:1s | y=2 speed:0.1;
```

Il n'y aura aucun temps mort entre les deux commandes donc, là encore, URBI attend que la seconde commande soit disponible: la seconde commande doit démarrer exactement après la première, par conséquent elle doit être lue en avance.

Utiliser le point-virgule ou la virgule est plus permissif car ces deux séparateurs démarrent immédiatement la commande se trouvant juste avant eux. Mais si votre projet réclame des contraintes temporelles fortes, les séparateurs **&** et | sont là pour vous aider.

Il est possible de regrouper plusieurs commandes en les encadrant avec des accolades et ainsi élaborer des structures temporelles complexes:

```
{ { x=4 time:1s | y=2 speed:0.1 } & z=0 sin:200ms ampli:4 } | t=2,
```

Conseil: En général, pensez à terminer les commandes entrées dans une console (*URBILab* ou *telnet*) par une virgule, pour éviter de bloquer la connexion après avoir saisi une commande sans fin.

Affectations conflictuelles

Comme il est possible d'exécuter des commandes en parallèle, il se peut que des conflits apparaissent. Par exemple, que se passerait-il si le code suivant était exécuté ?

```
x=1 & x=5;
```

x=5 est une affectation conflictuelle car elle accède à la variable **x** en même temps que la première affectation. Pour cela, URBI possède plusieurs modes de mélange (*blend*) pour prendre en charge les éventuels conflits et vous pouvez indiquer le mode souhaité grâce à la propriété **blend** de la variable. Par exemple:

```
x->blend = add;
```

Cela va demander à URBI d'additionner les valeurs numériques de toutes les affectations conflictuelles sur la variable **x**. Ainsi, le résultat de la commande précédente sera l'affectation de la valeur *6* à la variable **x**. Il existe également un mode **mix** qui réalise la moyenne des affectations conflictuelles (le résultat serait alors *3*) ainsi qu'un mode **queue** qui écrase les affectations conflictuelles (le résultat serait alors *5*). D'autres modes sont disponibles et décrits dans le document "URBI Language Specification".

Avec URBI, les variables possèdent des propriétés qui peuvent être accédées avec l'opérateur **->**. Une propriété n'est pas l'attribut d'un objet. Les propriétés font partie intégrante de la sémantique du langage et par conséquent ne peuvent être redéfinies. Il existe de nombreuses propriétés disponibles comme **rangemin**, **rangemax**, **speedmax**, **delta**. Elle sont décrites dans le document "URBI Language specification."

Les modes de mélange s'appliquent également aux devices sonores, tels le haut-parleur de l'Aibo et passer alors de **mix** à **queue** passera d'une superposition sonore à une succession sonore (les sons, quoiqu'il arrive, sont joués les uns après les autres).

Les modes **add** et **mix** s'avèrent très utiles pour superposer des affectations sinusoïdales pour élaborer des mouvement périodiques complexes, en utilisant des transformations de Fourier du signal et en ne retenant que le coefficient le plus significatif.

Variables et propriétés utiles des devices

Pour l'Aibo comme la plupart des robots, vous trouverez les variables de moteurs suivantes utiles (remplacez **device** par le nom du device):

- **device.load** : règle le couple d'une articulation, entre *0* (complètement lâche) et *1* (rigide).
- **device.PGain** : règle le gain *P* d'une articulation du *PID* associé.
- **device.IGain** : règle le gain *I* d'une articulation du *PID* associé.
- **device.DGain** : règle le gain *D* d'une articulation du *PID* associé.

Et voici des propriétés utiles, qui ne sont pas des variables au sens strict (les propriétés font partie de la sémantique du langage), mais vous pouvez les lire et les règler:

- **device->rangemin** : valeur minimale du device
- **device->rangemax** : valeur maximale du device

- **device->delta** : précision du device, utilisé pour les tests flous
- **device->unit** : unité du device (pour indication seulement dans URBI 1.0)
- **device->blend** : le mode de mélange du device (**normal**, **mix**, **add**, **queue**, **discard** ou **cancel**)
- **device->info** : des renseignements sur le device.

Ces propriétés sont en réalité celles de **device.val**, sachant que l'on considère que les alias sont opérationnels.

Quelques commandes utiles

Voici une brève liste de commandes qui pourraient s'avérer utiles dans vos programmes:

- **reset** : réalise un redémarrage logiciel. Utile pour supprimer un paquet de scripts et envoyer une toute nouvelle version.
- **stopall** : arrête toutes les commandes de toutes les connexions. Un peu brutal mais bien utile parfois.
- **reboot** : redémarre le robot.
- **shutdown** : éteint le robot.
- **uservars** : affiche la liste des variables utilisateurs.
- **strict** : enclenche le contrôle de définition de variable (voir le document "URBI Language Specification").
- **unstrict** : annule l'effet de **strict**.

Chapitre 4. Fonctions plus avancées

Désormais, vous êtes capable de lire et de régler des capteurs et des moteurs dans votre robot, exécuter des scripts ou actions complexes et superposer des séquences de mouvements. Cela pourrait suffire pour la plupart des utilisateurs, mais URBI va plus loin en vous proposant l'ensemble des structures algorithmiques disponibles dans les langages de programmation modernes ainsi que de nouvelles adaptées à la robotique.

Le branchement conditionnel et les boucles

Le branchement conditionnel et les boucles disponibles avec le C et le C++ sont présents dans URBI : **if ... else**, **for** et **while**. Les exemples suivants illustrent ces constructions (la commande **echo** que vous trouverez parmi les exemples affiche tout simplement l'expression en tant que message-système).

if

if réalise un test et exécute la commande associée, à la condition que le test ait réussi:

```
if (backSensorM > 0) {
       appuye = 1;
       echo "Capteur du dos appuyé";
};
```

La dernière commande entre accolades ne nécessite pas d'être conclue par un point-virgule comme dans l'exemple précédent. En effet, les points-virgules sont des séparateurs de commande et non des terminateurs de commande. Vous pouvez conclure par un point-virgule comme en *C*, mais cela est inutile (cela ajoute une commande vide).

```
if (distance < 10)
  echo "Obstacle détecté"
else
  echo "Aucun obstacle";
[167322:notag] *** Aucun obstacle
```

Remarquez qu'il n'y a pas de point-virgule avant **else** mais qu'il y en a un (ou tout autre séparateur de commande) après l'accolade finale.

distance et **backSensorM** sont deux devices: l'un pour le capteur de distance infra-rouge situé dans la tête de l'Aibo et l'autre pour le capteur du dos du milieu.

while

La construction d'une boucle **while** est similaire à celle du *C*:

```
i=0;
while (i<=2) {
  i:echo i;
  i++;
};
[151228:i] *** 0
[151228:i] *** 1
```

```
[151228:i] *** 2
```

for, foreach

La construction d'une boucle **for** est similaire à celle du *C*:

```
for (i=0;i<=2;i++)
  i:echo i;
[151228:i] *** 0
[151228:i] *** 1
[151228:i] *** 2
```

Contrairement au *C*, URBI possède des constructions temporelles de **for**: **for&**, **for|** et **while|**. Ces constructions démarreront chaque itération en parallèle (avec &) ou en série (avec |) avec une garantie de contrainte temporelle. Plus de détails dans le document "URBI Language Specification".

Comme nous l'avons mentionné auparavant, il existe également **foreach** et **foreach&** pour énumérer les listes:

```
foreach i in [0,1,2] {
  i:echo i;
};
[151228:i] *** 0
[151228:i] *** 1
[151228:i] *** 2
```

foreach est une exception: même quand il n'y a qu'une seule commande, comme dans l'exemple précédent, vous devez l'encadrer par des accolades.

loop, loopn

Pour des raisons pratiques, URBI a ajouté deux constructions supplémentaires, **loop** et **loopn**, pour créer des boucles infinies pour le premier et pour boucler un nombre fixé de fois pour le second. La syntaxe est la suivante:

```
loop { ... }
```

et

```
loopn (n) { ... }
```

Les mécanismes de capture d'événements

at

at fonctionne un peu comme **if**, à la différence qu'il tourne en permanence en tâche de fond:

```
at (distance < 50)
  echo "Obstacle détecté";
```

La commande **echo** dans l'exemple précédent s'exécutera dès que le test deviendra vrai, et ce, une seule et unique fois. Pour être plus précis, **at** déclenche la commande lorsque le test bascule de vrai à faux. Ceci

est très utile pour démarrer une action lorsque une condition est remplie pour réagir à cette condition. Si vous exécutez le code précédent, le message

```
Obstacle détecté
```

s'affichera une fois lorsque vous mettrez votre main devant l'Aibo.

onleave quant à lui se rapproche de **else** et est suivi d'une action qui sera exécutée dès que le test basculera de vrai à faux:

```
at (distance < 50)
  echo "Obstacle détecté"
onleave
  echo "Obstacle disparu";
```

whenever

whenever fonctionne un peu comme **while**, à la différence qu'il ne termine jamais et tourne en tâche de fond:

```
whenever (distance < 50)
  echo "Il y a un obstacle";
```

La commande **echo** sera exécutée tant que le test restera vrai. Et s'il redevient vrai par la suite, la boucle redémarrera pour, à nouveau, durer tant que le test restera vrai. Comparé à l'exemple précédent, la différence est que le message

```
Il y a un obstacle
```

sera affiché de nombreuses fois, aussi longtemps que vous laisserez votre main devant la tête du robot.

Il est possible d'ajouter une construction **else** pour indiquer une action à réaliser pendant que le test est faux:

```
whenever (distance < 50)
  echo "Il y a un obstacle"
else
  echo "Il y n'a pas d'obstacle";
```

whenever et **at** sont deux constructions fondamentales que vous aurez à utiliser lorsque vous programmerez des réactions et des captures d'événements sur votre robot.

wait, waituntil

La commande **wait (*n*)** attendra pendant *n* millisecondes avant de s'arrêter. Pratique pour provoquer une pause lors d'une séquence de commandes, typiquement des commandes motrices:

```
headPan = 0 | wait(1s) | headPan = 90;
```

La commande **waituntil(*test*)** attend que le test devienne vrai. Utile pour synchroniser des programmes parallèles.

timeout, stopif, freezeif

La commande **timeout (*n*) cmd** exécute la commande **cmd** et l'arrête ensuite si après n millisecondes elle n'a toujours pas fini.

```
timeout(10s) loop legRF2 = legLF2;
```

La commande **stopif (*test*) cmd** exécute la commande **cmd** et l'arrête ensuite si le test devient vrai avant qu'elle n'ait fini.

```
stopif(distance<50) robot.walk();
```

La commande **freezeif (*test*) cmd** exécute la commande **cmd** et la gèle ensuite si le test devient vrai avant qu'elle n'ait fini. La commande est dégelée si le test devient faux à nouveau.

```
freezeif(!ball.visible) trackball();
```

Cela peut s'avérer très utile pour indiquer que certaines portions de code ne tournent que lorsque certaines conditions sont réunies.

Les tests coulés

Les tests utilisés dans les commandes de capture d'événements comme **at**, **whenever**, **waituntil**, **stopif** ou **freezeif** peuvent être complétés de contraintes temporelles, devenant des *tests coulés* (ou *soft tests*):

```
at (headSensor >0 ~ 2s)
  echo "Quelquechose s'est posé sur ma tête ...";
```

Cela signifie que le test doit rester vrai pendant deux secondes pour qu'il devienne vrai aux yeux de la commande **at**. Vous pouvez spécifier la durée en *s* ou *ms* en employant le suffixe approprié et le tout doit être séparé du test par un tilde ~.

Les tests coulés sont utilisables dans toutes les commandes de capture d'événements et ils sont très utiles en robotique en tant que filtres pour capteurs.

Emettre un événement

La programmation événementielle est pratique et elle est une méthode à privilégier pour programmer un robot. L'idée générale de la programmation événementielle est que certaines commandes émettent des événements et d'autres les capturent et réagissent en conséquence.

Evénements simples

Pour émettre un événement, il y a la commande **emit** en URBI et vous pouvez utiliser **at** ou **whenever** pour le capturer:

```
at (boom) echo "boom!";
  emit boom;
[139464:notag] *** boom!
```

L'événement **boom** est local à la connexion. Si vous souhaitez que l'événement soit visible par une autre connexion, ajoutez-y un préfixe, comme **myprefix.boom**.

Les événements avec paramètres

Vous pouvez ajouter des paramètres à un événement, comme ceci:

```
emit monevenement(1, "salut");
```

Les paramètres peuvent être récupérés lors de la capture:

```
at (monevenement(x,y))
  echo "capture deux: " + x + " " + y;

at (monevenement(1,x))
  echo "capture un: " + x;
```

Le second **at** est particulièrement intéressant car il réalise un filtrage sur les paramètres de l'événements en n'acceptant uniquement que les événements dont le premier paramètre est égal à 1:

```
emit monevenement(1, "salut");
[146711:notag] *** capture deux: 1.000000 salut
[146711:notag] *** capture un: salut
emit monevenement(2, 15);
[148991:notag] *** capture deux: 2.000000 15.000000
```

Durée d'un événement

Un événement possède normallement une durée nulle, il est juste un pic (fonction de Dirac du temps). Cependant, vous pouvez demander à un événement de durer un certain laps de temps, indiqué enre parenthèses:

```
emit(10s) boom;
emit(15h12m) monevenement(1, "salut");
```

Cela fera une différence entre **at** et **whenever** par exemple: **whenever** bouclera pendant toute la durée de l'événement.

Evénements pulsés: la commande every

Vous pouvez imposer à une commande de s'exécuter à intervalle régulier en utilisant la commande **every**. L'exemple suivant affiche

```
salut
```

toutes les dix minutes:

```
every (10m) echo "salut";
```

Une utilisation classique de ceci est d'émettre régulièrement un évenement à intervalle régulier:

```
every (100ms) emit pulsation;
```

Pour arrêter l'émission, utilisez simplement **stop** sur la commande **every** avec l'étiquette appropriée:

```
monmetronome:every (100ms) emit pulsation;
stop monmetronome;
```

Etiquettes, drapeaux et contrôle des commandes

Le système d'étiquettage décrit au début de ce tutoriel est en réalité plus qu'une simple identification de messages. Par exemple, vous pouvez arrêter n'importe quelle commande avec la commande **stop**, depuis n'importe quelle connexion:

```
maboucle:loop legRF2 = legLF2,
...
stop maboucle;
```

Vous pouvez aussi geler une commande avec la commande **freeze** et la dégeler (elle reprendra là où elle en était) avec la commande **unfreeze**. Il existe aussi la combinaison de commandes **block**/**unblock** qui permet d'empêcher de nouvelles exécutions de la commande possédant l'étiquette donnée. Une étiquette peut préfixer un ensemble de commandes entre accolades, comme { ... }, et ainsi être associée à une large portion de code et pas forcément une seule commande.

A la suite de l'étiquette, il est possible d'ajouter un ou plusieurs drapeaux (*flags*). Un drapeau est un mot-clé préfixé par le signe +. Les drapeaux les plus utiles sont **+begin** et **+end** qui envoient un message-système lorsque la commande démarre et lorsqu'elle s'arrête, ou **+bg** qui place la commande en tâche de fond. Voici quelques exemples illustrant tout ceci:

```
monetiquette+begin:
loop legRF2 = legLF2,
[139464:monetiquette] *** begin
```

```
+begin+end:
wait(1s);
[521200:monetiquette] *** begin
[522200:monetiquette] *** end
```

D'autres drapeaux sont présentés dans le document "URBI Language Specification".

Depuis URBI 1.0, vous pouvez utiliser des étiquettes hiérarchiques comme **monetiquette.sousetiquette**. L'avantage de ceci est que vous pouvez arrêter une famille entière basée uniquement sur l'étiquette-mère: l'étiquette précédente peut aussi bien être arrêtée avec un **stop monetiquette.sousetiquette** qu'avec un **stop monetiquette**, et vous pouvez ainsi regrouper facilement plusieurs commandes. Une prochaine version incluera également le multi-étiquettage pour accroître encore les possibilités.

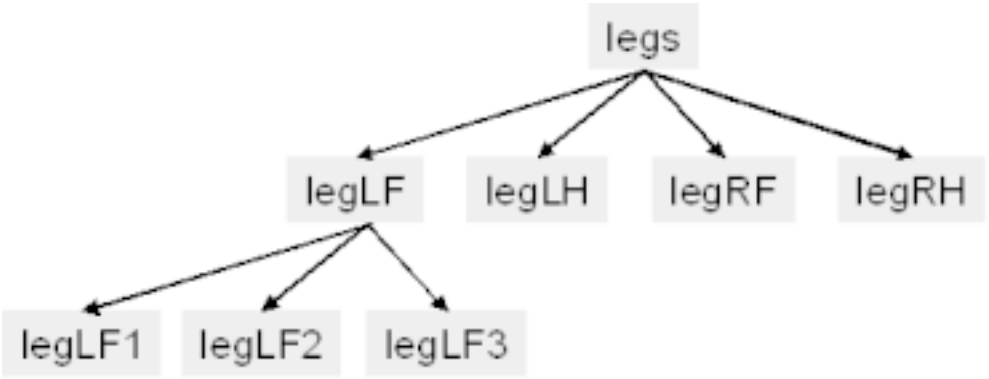
Le regroupement d'objets

Une possibilité importante offerte par URBI est de pouvoir regrouper les objets en hiérarchies. Cela se fait grâce à la commande **group**: **group nomdugroupe { objet1, objet2, ...}**, par exemple:

```
group patteavantgauche {legLF1, legLF2, legLF3};
```

```
group pattes {patteavantgauche, pattearrieregauche, patteavantdroite, pattearriere
```

Figure 4.1. Une hiérarchie typique de device moteurs



Cette fonction de regroupement est associée à la notion de *broadcasting*, qui est utilisée pour différentes choses. L'une d'entre-elles est l'affectation multi-objets : toute affectation est exécutée pour le groupe et, récursivement, passée aux sous-groupes-enfants. En d'autre termes, en utilisant l'exemple précédent, la commande **legLF.val = 0** règlera la valeur de legLF1.val, legLF2.val et legLF3.val à zéro (notez que les alias fonctionnent également si vous le souhaitez).

```
group ab {a,b};
ab.n = 4;
a:a.n, b:b.n,
[167322:a] 4.000000
[167322:b] 4.000000
```

Pour tout robot supporté, il y aura une hiérarchie de regroupement d'objets disponible au départ. Les commandes établissant cette hiérarchie se trouve dans le fichier `URBI.INI` ou le fichier `std.u`.

Par exemple, pour Aibo, il y a un groupe de l'ensemble des moteurs et un groupe de l'ensemble des lumières. Vous pouvez ainsi régler toutes les lumières à une valeur aléatoire avec la commande suivante:

```
leds = random(2);
```

Pour s'amuser, vous pouvez tenter: **fun: loop leds = random(2)**, et admirer le résultat.

Il reste plusieurs choses à dire au sujet des groupes et du broadcasting, qui est une fonction très puissante d'URBI. Nous reviendrons sur le sujet dans le chapitre Les Objets avec URBI.

Définir une fonction

Pour définir une fonction, vous devez utiliser le mot-clé **function** suivi du nom de votre fonction en notation **préfixe.suffixe** (ou simplement **suffixe** pour une fonction locale à la connexion), et les paramètres entre parenthèses (ou des parenthèses vides s'il n'y a aucun paramètre). Vous pouvez utiliser **return** pour retourner une valeur ou quitter la fonction, comme en *C*:

```
function ajouter(x,y) {
  z = x+y;
  return z;
};
```

```
function ecrire(x) {
  echo x;
  if (x<0)
    return
  else
    echo sqrt(x);
};
```

Il faut obligatoirement un point-virgule ou un autre séparateur de commande à la fin de la définition de la fonction, puisque que définir une fonction est une commande comme tout autre commande URBI.

Les paramètres sont toujours locaux à l'appel de la fonction. Les variables non globales (i.e. sans préfixe) dans le corps de la fonction sont également locales à l'appel de la fonction. Etudions l'exemple suivant:

```
a=4;
b=5;
function afficher(b) {
  afficher_b:b; // b est local
  a=b;          // cree une variable locale a
  afficher_a:a;
};
display(10);a:a;b:b;
[139464:display_b] 10.000000
[139464:display_a] 10.000000
[139464:a] 4.000000
[139464:b] 5.000000
```

Une bonne habitude est d'isoler vos fonctions dans un fichier séparé comme `mesfonctions.u`, et de les charger avec la commande: **load("mesfonctions.u")**. Cela peut être fait dans le fichier `URBI.INI` par exemple, ou quand vous avez réellement besoin.

Pour retirer la définition d'une function, utiliser simplement:

```
delete mafonction;
```

Messages d'erreur et messages-système

Lorsqu'une commande URBI échoue, cela envoie un message d'erreur, préfixé par trois points d'exclamation:

```
impossible:1/0;
[167322:impossible] !!! Division by zero
[167322:impossible] !!! EXPR evaluation failed
```

Remarquez que l'étiquette de la commande est utilisée pour le message d'erreur, ce qui s'avère très pratique pour savoir ce qui a coincé dans un programme complexe.

Les messages d'erreur sont différents des messages-système, préfixés eux par trois étoiles. Un exemple simple est la commande **echo** avec drapeau **+begin** et un drapeau **+end**:

```
monetiquette+begin+end:echo "salut tout le monde!";
[146711:monetiquette] *** begin
```

```
[146711:monetiquette] *** salut tout le monde!
[146711:monetiquette] *** end
```

Chapitre 5. Les objets avec URBI

La programmation orientée objet est intégrée à URBI, avec de nombreuses innovations comme les attributs virtuels et la diffusion (*broadcasting*). Ce chapitre aborde l'aspect le plus important des objets en URBI. Il peut être ignoré par les programmeurs débutants même si cela n'est en vérité guère compliqué et vraiment instructif.

Définir une classe

Tout comme en *C++*, on définit une classe en URBI avec le mot-clé **class**:

```
class maclasse;
```

Vous pouvez naturellement définir ce qui se trouvera dans la classe, c'est à dire des variables, des fonctions et des événements:

```
class maclasse {
  var x;
  var y;
  function f(a,b);
  event signalemoi(s);
};
```

Il est important de remarquer que, contrairement aux classes *C++*, **maclasse** dans l'exemple précédent est également une instance [1] et vous pouvez donc tout à fait affecter des valeurs à **maclasse.x** et l'utiliser.

Une fonction importante que vous souhaitez certainement définir est **init**, qui est le constructeur de la classe (ceci est une autre différence avec le *C++*, le constructeur n'est pas nommé avec le nom de la classe). Cette fonction ne devrait rien retourner ou retourner zéro pour indiquer la réussite de la création de l'instance et tout autre valeur pour indiquer son échec.

Pour définir le corps d'une méthode de classe, faites-le en dehors de la définition de classe, de la manière suivante:

```
class maclasse {
  var x;
  function init(a);
};

function maclasse.init(a) {
  x = a;
};
```

Vous pouvez définir une sous-classe (ou une instance, souvenez-vous qu'il n'y a aucune différence), avec une commande **new**, comme en *C++*:

```
masousclasse = new maclasse(42);
```

[1] Cela s'appelle un langage orienté prototype, comme le *javascript*.

Cela va créer une sous-classe et appeler **masousclasse.init(42);**.

masousclasse hérite de **maclasse**, par conséquent tous les attributs et les méthodes de **maclasse** sont aussi disponibles dans **masousclasse**. Nous allons voir par la suite la question de la définition par défaut et celle de la redéfinition [2].

masousclasse peut hériter de plusieurs classes en appelant **new** sur ces différentes classes:

```
masousclasse = new maclasse (42);
masousclasse = new monautreclasse ();
```

C'est une façon assez spéciale de traiter l'héritage multiple, comparé au *C++*.

En appelant **new** sans parenthèse, avec juste le nom de la classe, on exécute le constructeur **init** sans paramètre:

```
masousclasse = new maclasse;
maclasse();
```

Si **init** n'est pas défini, ou si **init** retourne une valeur synonyme d'erreur (non vide et non nulle), un message d'erreur sera produit et la création avortée.

Les classes peuvent être complétées pendant l'exécution simplement en créant des fonctions ou des attributs se référant à elles. Exemple:

```
class maclasse {
  var x;
};
var maclasse.nouvelattribut;
maclasse.nouvelattribut = "salut";
...
function maclasse.f(a) {
  nouvelattribut = a;
};
```

Méthodes virtuelles et attributs

En URBI, toutes les méthodes (fonctions de classe) et tous les attributs (variables de classe) sont virtuels, ce qui signifie que si votre classe le redéfinit, il devient sa propre définition, autrement la définition (ou la valeur) de la classe-mère sera utilisée.

Considérons l'exemple suivant:

```
class maclasse {
  var x;
  function f();
};
function maclasse.f() {
  echo "Je suis dans maclasse";
```

[2] Actuellement, il n'existe pas de notion d'accès privé, public ou protégé. Elles seront intégrées dans URBI 2.0.

```
};
```

D'où:

```
sous = new maclasse;
sous.f();
[139464:notag] *** Je suis dans maclasse
```

La définition de **f** est récupérée depuis **maclasse**. Nous pouvons maintenant la redéfinir:

```
function sous.f() {
  echo "Je suis dans sous!";
};
```

Observons la différence:

```
sous.f();
[139464:notag] *** Je suis dans sous!
```

De la même façon, les attributs obtiennent leur valeur de la classe-mère, à moins qu'ils soient redéfinis dans la classe-fille. L'exemple suivant illustre ce cas avec la classe précédente **maclasse** et les prototypes de **sous**:

```
maclasse.x = 1;
sous.x;
[146711:notag] 1.000000
sous.x = 4;
sous.x;
[146711:notag] 4.000000
maclasse.x;
[146711:notag] 1.000000
```

Les groupes

Nous avons vu dans les chapitres précédents comment les groupes peuvent être utilisées pour affecter simultanément une valeur à plusieurs variables d'objets. En fait, le mécanisme est bien plus général et est associé au concept de diffusion qui sera défini précisément dans la prochaine section.

Tout d'abord, quelques mots sur les groupes. Nous avons déjà appris que nous pouvions définir des groupes avec la commande **group**. De la même façon, vous pouvez ajouter un membre à un groupe avec la commande **addgroup** et en retirer un avec **delgroup**, ce qui vous permet la gestion dynamique de vos groupes:

```
group a {a1,a2};
addgroup a {c,d};
delgroup a {a1,d};
```

Vous pouvez examiner le contenu d'un groupe en invoquant la commande **group** avec seulement le nom du groupe:

```
group a {u,v,b};
group a
[146711:notag] ["u","v","b"]
```

Le regroupement de sous-groupes est possible. Dans ce cas, l'évaluation du contenu du groupe retourne la liste des membres terminaux seulement:

```
group a {u,v,b};
group b {x,y};
group a
[146711:notag] ["u","v","x","y"]
```

L'utilisation typique de cela est l'énumération de devices, comme des moteurs, qui ont été regroupés dans le même groupe:

```
foreach m in group motors {
  $(m) = ...
}
```

Le constructeur **$** retourne la variable dont le nom est la chaine de caractères donnée en paramètre. Dans l'exemple précédent, nous supposons qu'il y a un alias sur **.val**, autrement il faudrait écrire: **$(m+".val")**

Maintenant, nous allons voir comment utiliser en pratique les groupes avec la notion de diffusion.

La diffusion

Quand vous exécutez une commande à l'échelle d'un groupe, qui peut être un appel de fonction ou une affectation, la commande sera propagée en parallèle à chaque membre du groupe et à leurs sous-groupes. Cela s'appelle la diffusion (ou *broadcast*). Cela peut s'appliquer aussi aux classes, puisque vous pouvez définir un groupe contenant chaque instanciation de classe-fille. Une approche habituelle est de nommer le groupe associé à une classe avec le pluriel du nom de la classe, en ajoutant un simple "s".

Tout d'abord, observons comment une affectation est diffusée:

```
class a;
a1 = new a;
group as {a,a1};
a1.x = 42;
as.x = 4;
a.x;
[139464:notag] 4.000000
a1.x;
[146711:notag] 4.000000
```

La diffusion fonctionne de manière similaire avec les fonctions:

```
class a {
  var x;
  function f();
};
```

```
function a.f() {
  echo x;
};
a1 = new a;
group as {a,a1};
a.x = 1;
a1.x = 2;
as.f();
[139464:notag] *** 1.000000
[139464:notag] *** 2.000000
```

La fonction précédente **f** est en réalité exécutée de la façon suivante:

```
a.f() & a1.f();
```

Diffuser revient donc à dupliquer les commandes en parallèle.

Les sous-groupes sont naturellement parcourus dans le processus.

Les fonctions diffusées peuvent s'avérer très utiles pour exécuter des tâches en parallèle dans un groupe d'objets, sans avoir à utiliser **for&** ou une construction similaire. La diffusion et l'héritage se complètent mutuellement, ainsi quand la diffusion est achevée, la définition de la fonction peut être recherchée en remontant dans la hiérarchie de classe, comme dans l'exemple suivant:

```
class a {
  var x;
  function init(v);
  function f();
};
function a.init(v) { x=v; };
function a.f() {
  echo x;
};
a1 = new a(1);
a2 = new a(2);
a3 = new a(3);
function a1.f() { echo "Je suis différent !"};
group unetdeux {a1,a2};
unetdeux.f();
[139464:notag] *** 2.000000
[139464:notag] *** Je suis différent !
```

La diffusion est clairement un concept nouveau dans les mains des programmeurs. Vous pouvez l'utiliser ou non, mais nous croyons que cela permettra d'obtenir des programmes plus concis, en regroupant les actions logiques en une ligne, au lieu d'utiliser des boucles **for** ou un concept itératif similaire. Cela renforce également l'idée que certaines actions doivent être exécutées en parallèle sur un groupe d'objets, ce qui peut rend votre code plus sensé.

Chapitre 6. L'exemple de la détection de balle

La meilleure façon d'apprendre un nouveau langage est d'étudier de petits exemples pour voir ce qui peut être fait en pratique. Dans ce tutoriel, nous allons nous focaliser sur le détecteur de balle pour Aibo qui se révèle intéressant car son comportement n'est constitué que de deux états et parce qu'il implique une boucle perception-action qui est typique dans le domaine de la robotique. Nous allons voir comment URBI peut aider à contrôler l'exécution du comportement de façon simple grâce aux étiquettes.

Détection de la balle

Détecter une balle implique un traitement de l'image et ceci ne peut être écrit directement en URBI pour cause de performance. La meilleure façon de produire de tels composants algorithmiques (comme le traitement de l'image ou du son) est d'écrire un *composant UObject* en *C++*, *Java* ou *Matlab* et de le connecter à URBI. Nous n'allons pas nous attarder pour l'instant sur l'écriture d'un tel composant mais plutôt en utiliser un: l'objet **ball**.

L'objet **ball** est directement intégré dans l'*Aibo URBI Engine* et vous pouvez l'utiliser directement, comme n'importe quel device. Il n'y a pas de variable **ball.val** mais des variables **ball.x** et **ball.y** qui sont égales aux coordonnées de la balle dans l'image, comprises entre *-1/2* et *1/2*. Lorsqu'une balle est visible, **ball.visible** vaut *1*, ou *0* dans le cas contraire. Il existe également une variable **ball.ratio** qui donne la proportion de pixels de la balle dans l'image, exprimée en pourcentage. Ces simples variables sont déjà intéressantes pour de nombreuses applications, tout comme nous allons le voir par la suite.

Le programme principal

Le programme de détection de balle est donné en exemple dans le kit de développement de *Sony* (*SDK OPEN-R*) et réalise les actions suivantes: lorsqu'une balle se trouve en face du robot, ce dernier va la suivre en bougeant la tête. Autrement, il va la chercher en bougeant la tête en cercles.

Orienter la tête vers la balle peut être écrit très simplement en URBI avec les deux lignes suivantes:

```
headPan  = headPan  + camera.xfov * ball.x &
headTilt = headTilt + camera.yfov * ball.y;
```

Cela aura pour effet de bouger en même temps (c'est ce que veut dire le séparateur &) les deux moteurs de tête **pan** et **tilt**, d'une quantité proportionnelle aux positions x et y de la balle dans l'image de sa caméra. Les coefficients **camera.xfov** et **camera.yfov** viennent du device **camera** que nous découvrirons dans le chapitre suivant. Ils représentent l'angle x et l'angle y du champ de vision de la caméra de l'Aibo qui sont utilisés pour convertir l'intervalle normalisé *[-1/2;1/2]* de **ball.x** et **ball.y** en degrés.

Pour suivre la balle et non plus s'orienter une fois dans sa direction, nous allons utiliser la commande **whenever**:

```
whenever (ball.visible) {
  headPan  = headPan  + camera.xfov * ball.x &
  headTilt = headTilt + camera.yfov * ball.y;
```

```
};
```

Ce programme fait seulement trois lignes et réalise le comportement de suivi de balle que l'on souhaitait. Cependant, avec un Aibo, cela risque d'être trop réactif et conduira à de faibles oscillations de la tête autour de l'orientation de la balle. Pour éviter cela, une technique robotique simple est d'utiliser un coefficient d'atténuation, **ball.a**, pour limiter la réactivité du système. Par exemple:

```
ball.a = 0.8;
whenever (ball.visible) {
  headPan  = headPan + ball.a * camera.xfov * ball.x &
  headTilt = headTilt+ ball.a * camera.yfov * ball.y;
};
```

La prochaine étape est de basculer de ce comportement vers le comportement de recherche quand la balle n'est pas visible. Le comportement de recherche peut être exprimé avec un simple mouvement sinusoïdal sur à la fois **headPan** et **headTilt**. Nous utilisons dans l'exemple suivant l'extension de variable **'n** [1] qui indique que l'on travaille avec une valeur normalisée de la variable, comprise entre *0* et *1*, calculée à partir des propriétés **rangemin** (la limite inférieure) et **rangemax** (la limite supérieure). Cela s'avère très pratique d'éviter de contrôler la valeur actuelle d'un device et de le manipuler d'une façon générique:

```
periode = 10s;
headPan'n  = 0.5 sin:periode ampli:0.5 &
headTilt'n = 0.5 cos:periode ampli:0.5,
```

Le modificateur **cos** est identique à **sin** mais avec un décalage de phase de *pi/2*. Remarquez comme la veleur centrale *0.5* avec l'amplitude *0.5* permet de couvrir toute l'étendue du device: [*0..1*].

La commande précédente réalise le mouvement circulaire requis mais lorsque ce comportement est lancé, la position initiale sera atteinte brusquement depuis l'emplacement où se trouvant la tête avant le lancement de la commande. Pour éviter cela, nous pouvons précéder notre commande avec une transition douce d'une seconde vers la position initiale du cercle qui est **headPan'n = 0.5** et **headTilt'n = 1**:

```
headPan'n  = 0.5 smooth:1s &
headTilt'n = 1   smooth:1s;
```

Le modificateur **smooth** est similaire à **time**, à la différence qu'il ajoute la douceur du tracé d'un "S", au lieu de faire un mouvement linéaire.

Désormais, nous pouvons tout connecter en un seul comportement, en utilisant la capteur d'évenement **at** comme ciment. Afin d'éviter de basculer du balayage circulaire au suivi de la balle trop souvent, nous ajoutons également un test coulé, et nous utilisons la fonction **loadwav** pour précharger deux fichiers sonores que l'on affecte au device **speaker** (le haut-parleur, nous décrirons ce device plus tard) pour jouer un son lorsque la balle est trouvée ou perdue:

```
// Parameters initialization
ball.a = 0.9;
period = 10s;
trouvee = loadwav("found.wav");
perdue  = loadwav("lost.wav");
```

[1] D'autres extensions sont disponibles dans URBI. Les extensions sont de puissants outils pour moduler l'évaluation d'une variable. Consultez le document "URBI Language Specification" pour de plus amples détails.

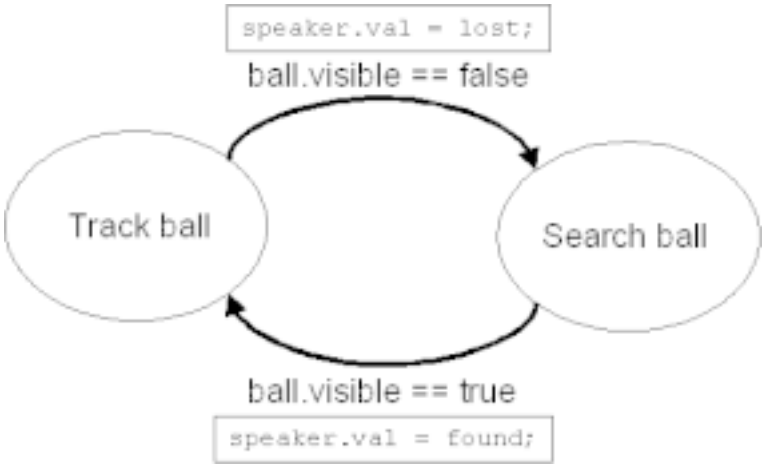
```
// Comportement principal
whenever (ball.visible ~ 100ms) {
  headPan  = headPan  + ball.a * camera.xfov * ball.x &
  headTilt = headTilt + ball.a * camera.yfov * ball.y;
};

at (!ball.visible ~ 100ms)
recherche: {
          { headPan'n  = 0.5 smooth:1s &
            headTilt'n = 1   smooth:1s } |
          { headPan'n  = 0.5 sin:period ampli:0.5 &
            headTilt'n = 0.5 cos:period ampli:0.5 }
           };

at (ball.visible) stop recherche;

// Comprtement sonore
at (ball.visible ~ 100ms) speaker = trouvee
onleave speaker = perdue;
```

Vous pouvez aussi utiliser la construction **onleave** pour regrouper les deux commandes **at (ball.visible)**, mais vous devez alors employer la commande **at&**, pour placer la commande **recherche** en tâche de fond (car c'est une commande sans fin et donc **at** ne rendrait jamais la main).

Programmer par un graphe de comportement

Le programme précédent fonctionne, est facile à comprendre et à modifier. Pourtant, il est courant en robotique de concevoir les programmes en termes de comportements, exprimés sous forme de machines à états finis, qui sont des graphes d'*états* reliés entre-eux par des *transitions*. La figure 6.1 illustre le graphe de comportement du programme de détection de balle, qui est exemple très simple de comportement à deux états.

Figure 6.1. Le graphe de comportement de la détection de balle



Les ellipses représentent les états (dans lequels le robot boucle des actions ou une surveillance) et les flèches les transitions, étiquetées par des conditions. Les rectangles associés aux transitions indiquent certaines actions à exécuter lorsque la transition s'effectue.

La meilleure façon de programmer ce genre de graphe de comportement en URBI est d'utiliser une conjonction des fonctions avec les commandes **at** et **stop** pour relier le tout. Tout d'abord, définissons les deux fonctions associées aux deux états du programme de détection de balle:

```
// Etat de suivi
function suis() {
  whenever (ball.visible) {
    headPan  = headPan + ball.a * camera.xfov * ball.x &
    headTilt = headTilt+ ball.a * camera.yfov * ball.y;
  }
};

// Etat de recherche
function cherche() {
  period = 10s;
  {
    headPan'n  = 0.5 smooth:1s &
    headTilt'n = 1   smooth:1s
  } |
  {
    headPan'n  = 0.5 sin:period ampli:0.5 &
    headTilt'n = 0.5 cos:period ampli:0.5
  }
};
```

Maintenant, nous pouvons simplement relier les états entre-eux en établissant les transitions avec deux commandes **at** et en terminant l'état précédent avec des commandes **stop**:

```
// Transitions
at (ball.visible ~ 100ms) {
  stop recherche;
  speaker = trouvee;
  suivi: suis();
};

at (!ball.visible ~ 100ms) {
  stop suivi;
  speaker = perdue;
  recherche: cherche();
};
```

L'avantage de ré-écrire le programme de détection de balle en termes de machine à états finis peut ne pas vous apparaître évident pour l'instant car le programme est très simple. Mais, avec des comportements plus riches de dizaines d'états, chacun avec plusieurs transitions, il s'agit de la façon la plus sure de programmer. Cela rend le code modulaire, clair et facile à modifier.

Les machines à états finis sont une excellente façon de décrire les comportements des robots. Elles ne sont pas parfaites mais c'est pour l'instant la technique la plus employée en robotique. URBI est également capable de décrire des architectures subsumées, hiérarchisées ou réactives et bien d'autres paradigmes.

Contrôler l'exécution du comportement

Les possibilités de geler, arrêter et bloquer les commandes avec URBI forment un outil très puissant pour contrôler l'exécution du comportement. Par exemple, si les transitions, exprimées avec une commande **at** sont préfixées par une étiquette, comme ceci:

```
transition_de_suivi:
     at (ball.visible ~ 100ms) {
        stop recherche;
        speaker = trouvee;
        suivi: suis();
     };

transition_de_recherche:
     at (!ball.visible ~ 100ms) {
        stop suivi;
        speaker = perdue;
        recherche: cherche();
     };
```

Il devient très facile de suspendre temporairement ou de ré-activer une transition de la manière suivante:

```
freeze transition_de_suivi;
...
unfreeze transition_de_suivi;
```

D'autrepart, il est possible de bloquer l'exécution d'un état, sans pour autant empêcher les transitions vers lui (attendant silencieusement une autre transition qui fera passer le robot à un autre état):

```
block recherche;
...
unblock recherche;
```

En utilisant **freeze**, **block** et **stop**, il est simple de modifier les comportements et ré-affecter les priorités en direct, ce qui est très pratique en robotique. Les possibilités sont innombrables car ces réglages peuvent être contrôlés par des évenements ou par d'autres programmes tournant en parallèle, ou même par un programme de contrôle à distance, ou encore une session telnet.

Chapitre 7. Images et sons

Jusqu'ici, nous n'avons cotoyé que des variables numériques, comme **headPan.val**. Cela n'est bien sûr pas suffisant pour transmettre images et sons. Certains devices, comme **camera**, **micro** ou **speaker** pour l'Aibo, sont des devices binaires. Dans ce cas, la variable **device.val** n'est pas une valeur numérique mais une valeur binaire.

Lire une valeur binaire

Vous avez certainement déjà tenté d'évaluer l'une de ces variables binaires:

```
camera;
[139464:notag] BIN 5347 jpeg 208 160
.................5347 bytes.................

micro;
[139464:notag] BIN 2048 wav 2 16000 16 1
.................2048 bytes.................
```

URBI préfixe chaque donnée binaire d'un en-tête binaire commençant par le mot-clé **BIN**, suivi de la taille de la donnée (en octets ou *bytes*) et un mot-clé indiquant le type de la donnée. Certains paramètres optionnels, comme la taille de l'image, la fréquence d'échantillonnage, le status mono ou stéréo d'un son peuvent suivre. Ensuite, après un retour à la ligne, la donnée binaire en elle-même est retournée (affichée ci-dessus comme une série de points **....**), ce qui peut embrouiller un client telnet mais pas un client logiciel URBI [1] .

Ce que l'on appelle un *client logiciel* est un client ou un composant écrit en *C++* ou *Java*, comme décrit dans le chapitre La liburbi en C++. C'est la manière habituelle de gérer les données binaires quand on souhaite traiter un signal avec URBI.

Affecter une valeur binaire

Vous vous en doutez sûrement, affecter une valeur binaire à un device **speaker** (le haut-parleur) par exemple n'est guère plus complexe que de le lire. Pour jouer un son sur un Aibo, vous pouvez envoyer au serveur une commande comme celle-ci:

```
speaker = bin 54112 wav 2 16000 16;
..............54112 bytes..............
```

L'en-tête doit se terminer par un point-virgule, et rien d'autre. Le contenu binaire commence immédiatement après le point-virgule donc nul besoin d'un retour à la ligne supplémentaire.

Bien sûr, une telle affectation binaire ne peut être faite depuis telnet ou URBIRemote, puisque vous souhaitez probablement que le programme envoie le contenu binaire, et que vous n'ayez pas à le saisir vous-même dans le terminal ! (bien qu'il soit possible de faire jouer un son depuis un client telnet).

Ce petit exemple montre une affectation binaire et une lecture binaire en URBI depuis un client telnet. Mais gardez à l'esprit que ceci n'est qu'une démonstration:

[1] URBI Remote comprend les en-têtes URBI, affiche les images et lit les sons, en tenant compte du type.

```
mybin = bin 3;ABC
mybin;
[146711:notag] BIN 3
ABC
```

Vous pouvez ajouter d'autres paramètres après la taille de la donnée binaire, ils seront stockés avec le contenu binaire à l'intérieur de l'en-tête:

```
mybin = bin 3 salut tout le monde 33;ABC
mybin;
[146711:notag] BIN 3 salut tout le monde 33
ABC
```

Ne confondez pas donnée binaire et donnée textuelle (chaîne de caractères). L'exemple précédent est différent de:

```
mystring = "ABC";
mystring;
[148991:notag] "ABC"
```

Attributs associés

Habituellement, un objet device binaire dispose de plusieurs attributs. Un exemple classique est le device **camera** d'un Aibo qui fournit les attributs suivants:

- **camera.shutter** : la vitesse d'obturation de la caméra: *1*=lente (par défaut), *2*=moyenne, *3*=rapide
- **camera.gain** : le gain de la caméra: *1*=lent, *2*=moyen, *3*=rapide (par défaut)
- **camera.wb** : la balance des blancs: *1*=intérieur (par défaut), *2*=extérieur, *3*=fluorescent
- **camera.format** : le format de l'image: *0*=YCbCr *1*=jpeg (par défaut)
- **camera.jpegfactor** : le facteur de compression JPEG (de *0* à *100*). *80*, par défaut.
- **camera.resolution** : la résolution de l'image: *0*:*208*x*160* (par défaut) *1*:*104*x*80 2*:*52*x*40*
- **camera.reconstruct** : reconstruction de l'image haute résolution (lent): *0*:non (par défaut) *1*:oui
- **camera.width** : largeur de l'image
- **camera.height** : hauteur de l'image
- **camera.xfov** : Angle de vue horizontal, en degrés
- **camera.yfov** : Angle de vue vertical, en degrés

Pour le device **speaker**, en charge de la production sonore de l'Aibo, vous disposez de:

- **speaker.playing** : égal à *1* si un son est en cours de lecture, *0* dans le cas contraire
- **speaker.remain** : nombre de millisecondes de son à jouer restantes, *0* quand le tampon est vide.

Avec l'objet **speaker**, il existe également une méthode qui peut être utilisée pour jouer un son directement à partir d'un fichier présent sur la memorystick:

```
speaker.play("monson.wav");
```

Autrement, pour éviter d'avoir un accès disque vous ralentissant dans le cas d'accès fréquent, vous pouvez opter pour le stockage en mémoire. Pour cela, utilisez la fonction **loadwav**:

```
monbinaire = loadwav("monson.wav");
speaker = monbinaire;
```

Exemples d'opérations binaires

Avec URBI, vous pouvez ajouter des binaires, pour concaténer de sons par exemple. Par exemple, considérons le programme suivant:

```
son = bin 0;
timeout(10s) loop son = son + micro;
speaker = son;
```

Ce code enregistrera dix secondes de son provenant du device **micro** et les stockera dans la variable **son**, et les jouera suite à l'affectation au device **speaker**. Ceci montre à quel point il peut être simple de manipuler les tampons binaires avec URBI pour de simples tâches comme la concaténation.

Chapitre 8. La *liburbi* en *C++*

Qu'est-ce que la *liburbi*?

Utiliser URBI depuis un client telnet est trop limité. Vous avez besoin d'envoyer des commandes et de recevoir des messages en utilisant le langage de programmation de votre choix ou, de manière plus générale, vous avez besoin d'interfacer URBI avec d'autres langages.

C'est la raison pour laquelle URBI est un *Interface Language*: il est bien plus qu'un simple protocole car c'est un langage de script complet agissant comme un protocole. Dans la plupart des applications où intervient l'imagerie informatique ou le traitement sonore, URBI est utilisé conjointement avec *C++* ou tout autre langage véloce afin d'effectuer la partie algorithmique. URBI est là pour orchestrer vos comportements, vos boucles action/perception et tout autre élement de haut niveau, avec comme source de décisions la sortie du code rapide de *C++*, *Java* ou *Matlab*.

Qu'est-ce que *liburbi*? Vous pourriez certes programmer une couche TCP/IP pour *C++* ou pour votre langage favori mais c'est relativement trivial et serait fait une fois pour toutes. Voilà pourquoi nous avons créé *liburbi*. Voici les fonctionnalités que l'on est en droit d'attendre:

- Ouvrir une connexion vers le robot depuis son langage favori (comme *C++*),
- Envoyer une commande au robot depuis ce langage,
- Demander la valeur d'une variable et la recevoir,
- Recevoir les messages provenant du robot et réagir de manière appropriée.

Le dernier point est très important et, même si cette approche est très différente de votre façon de programmer, il est essentiel de vous adapter à cette logique-ci de programmation (appelée *programmation asynchrone*) car elle est la plus adaptée à la robotique. Les robots sont fondamentalement des systèmes asynchrones. On attend des messages de la part du robot pour y réagir ensuite. (ceci est également appelé *programmation évenementielle*). C'est ce qu'un robot fait la plupart du temps: réagir à des évenements [1].

Ce chapitre est une brève introduction à *liburbi*. Vous devriez lire la documentation officielle de *liburbi* sur http://www.urbiforge.com si vous souhaitez une description plus étoffée. Si vous programmez en *C++*, nous vous conseillons d'utiliser l'architecture *UObject* décrite plus tard dans ce tutoriel, *liburbi* n'étant qu'un complément à la nouvelle et prometteuse technologie UObject.

Les composants et *liburbi*

Etendre URBI avec du code écrit en *C++*, *Java* ou *Matlab* qui sera accessible à vos scripts URBI peut être réalisé de deux façons. La première est d'exploiter la *liburbi* de votre langage préférée (*C++*/*Java*/*Matlab*) pour construire un client URBI. C'est ce que nous allons décrire dans ce chapitre.

La seconde, plus puissante, est de créer un *composant UObject* qui est associé à un objet *C++*. Ceci est expliqué au chapitre Créer des composants: l'architecture UObject. L'objet sera accessible comme tout autre objet URBI, partageant ses méthodes et ses attributs. Il s'agit de la manière la plus portable et la plus flexible d'incorporer des fonctionnalités inédites à URBI. Mais commençons par la traditionnelle *liburbi*. L'un des aspects séduisants de la *liburbi* est qu'elle est disponible pour de nombreux langages, bien

[1] Traditionnellement en Intelligence Artificielle, la manière dont le robot réagit peut être modifiée par des activités cognitives de plus haut niveau (architecture hiérarchique) ou par des propriétés (architecture subsumée) ou par une combinaison complexe de processus délibératifs et réactifs (architecture hybride)

que le jumelage-objet ne soit pas toujours possible et qu'il soit pour l'instant limité au *C++*. Aborder la programmation avec la *liburbi* permet déjà de mener à bien certains projets de programmation et, chose intéressante pour les débutants, de s'initier à l'aspect asynchrone de la robotique.

Il existe actuellement une version *C++*, une version *Matlab*, une version *Java* et une version *Python* de la *liburbi* si vous désirez contrôler votre robot dans l'un des langages. Il existe également une version *OPEN-R*, permettant de recompiler un programme pour le faire tourner uniquement sur l'Aibo. Cependant, nous vous déconseillons cette dernière approche au profit des *UObject*. La seule raison d'être de cette version *OPEN-R* est qu'elle offre une gestion implicite du multi-threading non bloquant, indisponible dans le système d'exploitation de base de l'Aibo, *Aperios*.

Nous ne décrirons pas ici la totalité des implémentations de *liburbi* mais seulement celle dédiée au *C ++*, permettant cependant de s'introduire aux concepts généraux. Les autres versions sont similaires et possèdent chacune leur documentation. Nous supposons dans ce qui suit que vous disposez d'un minimum de connaissances sur le *C++*. Dans le cas contraire, consultez un rapide tutoriel *C++* pour vous familiariser aux concepts de base de ce langage populaire.

Premiers pas

Pour commencer, il vous faut pouvoir compiler un programme basé sur la *liburbi*. Il y a plusieurs façons de s'y prendre selon que vous soyez sous *Linux* ou *Windows*, avec un compilateur *Borland* ou *Microsoft*, etc. D'une manière générale, il suffit d'inclure `liburbi.h` et de lier votre code avec la *liburbi* (**-lurbi** par exemple pour le compilateur *gcc*). Consultez la documentation appropriée pour de plus amples détails.

Du côté du code, la première chose à faire est de créer un client qui se connectera à votre robot, disons à l'adresse *monrobot.mondomaine.com*. Pour cela, vous disposez d'une classe **UClient**:

```
UClient* client = new UClient("monrobot.mondomaine.com");
```

Si vous connaissez l'adresse IP, vous pouvez la mettre à la place.

Vous pouvez également appeler explicitement la classe UClient, en utilisant une fonction de l'espace de nommage **urbi**:

```
UClient* client = urbi::connect("monrobot.mondomaine.com");
```

Bien sûr, vous pouvez créer ainsi autant de clients que vous le désirez.

Envoyer une commande

L'objet **UClient** possède une méthode **send** qui fonctionne comme **printf**:

```
client->send("motor on;");
for (float val=0; val<=1; val+=0.05)
  client->send("neck'n = %f;wait (%d);", val, 50);
```

Vous pouvez également utiliser votre objet-client comme un flux si vous préférez cette manière typiquement *C++*:

```
client << "headPan = " << 12 << ";";
```

Il existe également une manière très pratique d'envoyer des blocs entiers de code URBI depuis votre programme *C++*, grâce à la macro **URBI((...))**:

```
URBI((
      headPan = 12,
      echo "salut" | speaker.play("test.wav") & leds = 1
    ));
```

Le texte brut entre les doubles parenthèses sera envoyé directement au premier client créé par votre programme, par défaut. Cela peut se régler en appelant **urbi::connect(...)**. La première possibilité que nous avons abordée, celle employant la méthode **send**, est plus appropriée dans la plupart des cas et la macro URBI ne devrait servir qu'à envoyer des scripts d'initialisation en début de programme ou pour prototyper.

Gardez à l'esprit que vous pouvez toujours faire repartir votre robot sur des bases neuves (reboot virtuel) en lui envoyant la commande **reset**. Cela vous évitera les définitions multiples à chaque nouvelle exécution de votre client. Voilà pourquoi de nombreux programmes principaux débutent par **client->send("reset;");**.

Envoyer une donnée binaire

Pour envoyer une donnée binaire, vous utiliserez la méthode **sendBin**, au lieu de **send**:

```
client->sendBin(DonnéeSonore, TailleDuSon,
                        "speaker = BIN %d raw 2 16000 16 1;",
                        TailleDuSon);
```

Les deux premiers paramètres sont le son lui-même et sa taille. Viens ensuite l'en-tête URBI et enfin les paramètres otpionnels, employant une syntaxe à la **printf**.

Pour envoyer un son, il existe une méthode spécialisée appelée **sendSound**, plus pratique et plus efficace:

```
client->sendSound(son, "finduson");
```

Le premier paramètre est une structure USound, décrivant le son à jouer. Le second est une étiquette optionnelle qui sera utilisée par le serveur pour conclure la lecture du son par un l'émission d'un message-système.

Vous pouvez utiliser la fonction **convert** pour convertir un son vers différents formats.

Avec **sendSound**, il n'y a pas de limite quant à la taille du tampon sonore, puisqu'il sera découpé en petits morceaux par la bibliothèque. Les données étant copiées par *liburbi*, le paramètre USound et ses données associées peuvent ensuite être libérées dès que le fonction se termine.

Recevoir des messages

Les étiquettes URBI vont s'avérer très utiles pour la réception des messages du serveur: chaque commande se voit associée à une étiquette (**notag** par défaut), et cette étiquette est transmise dans tout message originaire de cette commande. La classe **UClient** gère la réception de ces messages dans un thread indépendant créé par le constructeur, les analysent et remplit une structure **UMessage**. Les fonctions-callback, nom que l'on donne aux fonctions d'un client qui doivent réagir automatiquement à tout emploi d'une étiquette donnée de la part du serveur, peuvent être référencées avec la méthode **setCallback**: à

chaque fois q'un message avec cette étiquette est envoyée par le serveur, la fonction-callback sera appelée avec la structure UMessage comme argument.

```
typedef UCallbackAction (*UCallback) (const UMessage &msg);

UCallbackID setCallback (UCallback cb, const char *tag)
```

Le premier paramètre **cb** est un pointeur sur la fonction à appeler. La fonction-callback doit retourner la valeur **URBI_CONTINUE** ou la valeur **URBI_REMOVE**. Dans ce dernier cas, la fonction sera déréférencée.

La meilleure façon d'apprendre à utiliser les callback est d'étuider les exemples de la documentation:

http://www.urbiforge.com/eng/liburbi.html

Les types de données

Le type de donnée utilisé par la *liburbi* est décrite ci-dessous:

UMessage

```
class UMessage {
 public:

  UAbstractClient    &client;   // connexion à l'origine du message
  int                timestamp;     // timestamp côté serveur
  char               *tag;          // étiquette associée

  UMessageType       type;          // type du message
  UBinaryMessageType binaryType;    // type du message binaire

  union {
    double        doubleValue;
    char          *stringValue;
    char          *systemValue;
    char          *message;        // rempli si le type est unknown (MESSAGE_UNKN
    USound        sound;           // rempli si la donnée binaire est sonore
    UImage        image;           // rempli si la donnée binaire est une image
    UBinary       binary;          // rempli si la donnée binaire est de type ind
  };
}
```

Le champ **type** peut être **MESSAGE_DOUBLE**, **MESSAGE_STRING**, **MESSAGE_SYSTEM**, **MESSAGE_BINARY** ou **MESSAGE_UNKNOWN**. En fonction de ce champ, la valeur correspondante dans l'union sera remplie. Si le message est binaire, **binaryType** donnera des informations supplémentaires sur le type de la donnée (**BINARYMESSAGE_SOUND**, **BINARYMESSAGE_IMAGE** ou **BINARYMESSAGE_UNKNOWN**), et la structure appropriée (**sound** ou **image**) sera remplie.

USound

```
class USound {
 public:
 char                  *data;            // pointeur sur la donnée sonore
 int                   size;             // taille totale, en octets
 int                   channels;         // nombre de canaux audio
 int                   rate;             // fréquence d'échantillonnage, en Hertz
 int                   sampleSize;       // qualité d'échantillonnage, en bits
 USoundFormat          soundFormat;      // format du son
                                            // (SOUND_RAW, SOUND_WAV, SOUND_MP3..
 USoundSampleFormat    sampleFormat;     // format des échantillons
};
```

UImage

```
class UImage {
 public:
  char                  *data;            // pointeur sur la donnée graphique
  int                   size;             // poids de l'image, en octets
  int                   width, height;    // taille de l'image
  UImageFormat          imageFormat;      // format de l'image
                                              // (IMAGE_RGB, IMAGE_YCbCr, IMAGE_J
};
```

Opérations synchrones

La classe dérivée **USyncClient** implémente des méthodes permettant d'obtenir les résultats des commandes URBI de manière synchrone. Vous devez savoir que ces fonctions sont moins efficaces et qu'elles ne fonctionnent pas encore avec la version OPEN-R de la *liburbi*. Par ailleurs, travailler avec un robot en mode synchrone est à éviter.

Lecture synchrone de la valeur d'un device

Pour obtenir la valeur d'un objet-device (possédant un attribut **val**), vous pouvez utiliser la méthode **syncGetDevice**. Le premier paramètre est le nom du device (ici nous prendrons comme exemple **neck**), le second est un réel (type **double**) qui est affecté de la valeur recue:

```
double ValeurNeck;
syncClient->syncGetDevice("neck", ValeurNeck);
```

Obtenir une image de façon synchrone

Vous pouvez faire appel à la méthode **syncGetImage** pour obtenir une image de façon synchrone. La méthode enverra la commande appropriée et attendra le résultat, bloquant ainsi votre thread tant que l'image ne sera pas arrivée.

```
client->send("camera.resolution = 0; camera.gain = 2;");
int width, height;
client->syncGetImage("camera", myBuffer, myBufferSize,
                             URBI_RGB, URBI_TRANSMIT_JPEG, width, height);
```

Le premier paramètre est le nom du device de la caméra. Le second est le tampon qui recevra la donnée de l'image. Le troisième doit être une variable de type entier égale à la taille du tampon. La fonction placera la taille de la donnée à l'intérieur. Si le tampon est trop petit, la donnée sera tronquée.

Le quatrième paramètre est le format dans lequel vous souhaitez recevoir l'image. Les valeurs possibles sont **URBI_RGB** pour une image non compressée d'une profondeur de 24 bits, **URBI_PPM** pour un fichier PPM, **URBI_YCbCr** pour une image YCbCr non compressée, et **URBI_JPEG** pour un fichier compressé JPEG.

Le cinquième paramètre peut être soit **URBI_TRANSMIT_JPEG** soit **URBI_TRANSMIT_YCbCr** et spécifie comment l'image sera transmise entre le robot et le client. Transmettre une image JPEG augmente la fréquence de rafraîchissement et améliore nettement les performances.

Enfin la largeur et la hauteur de l'image sont données par les paramètres **width** et **height**.

Obtenir du son de façon synchrone

La méthode **syncGetSound** permet d'obtenir un échantillon sonore du serveur, quelle que soit sa taille.

```
client->syncGetSound("micro", duree, son);
```

Le premier paramètre est le nom du device à qui demander du son, le second est la durée souhaitée, en millisecondes. **son** est une structure **USound** qui sera remplie avec le son enregistré.

Fonctions de conversion

Nous fournissons également quelques fonctions de conversion entre différents formats d'image et de son. L'utilisation des fonctions de conversion d'image est assez directe:

```
int convertRGBtoYCrCb (const byte* source, int longsource, byte* dest);
int convertYCrCbtoRGB (const byte* source, int longsource, byte* dest);
int convertJPEGtoYCrCb(const byte* source, int longsource, byte* dest, int &taill
int convertJPEGtoRGB  (const byte* source, int longsource, byte* dest, int &taill
```

Le paramètre **taille** doit être réglé à la taille du tampon de destination. Au retour de l'appel de la fonction, il contiendra la taille de la donnée en sortie.

Pour convertir entre différents formats sonores, vous pouvez utiliser la fonction **convert**. Elle prend deux structures **USound** en paramètres. Les deux formats audio actuellement supportés sont **SOUND_RAW** et **SOUND_WAV**, mais le support de formats compressés tels l'Ogg Vorbis ou le MP3 sont prévus. Si l'un des champs est fixé à zéro, la valeur de la source sera utilisée.

L'exemple d'*urbiimage*

URBIimage est un petit programme écrit en *C++* avec la *liburbi-C++* pour acquérir et afficher des images d'un serveur URBI. *URBIimage* fait deux choses: il place un callback sur l'étiquette **uimg** pour recevoir les images, et puis les affiche. Examinons le code. Commençons par l'interface de callback:

```
Monitor *mon;
```

```
/* Notre fonction-callback */
UCallbackAction showImage(const UMessage &msg)
{
  ...
}
```

Et maintenant, le programme principal :

```
int main(int argc, char *argv[])
{
  mon = NULL;
  client = new UClient(argv[2]);
  if (client->error() != 0)
    exit(0);

  client->setCallback(showImage, "uimg");

  // Some image initialization
  client->send("camera.resolution = 0;");
  client->send("camera.jpegfactor = 80;");

  // Start the loop
  client->send("loop uimg: camera,");
  urbi::execute();
}
```

Le code gérant l'image est stocké dans "showImage":

```
UCallbackAction showImage(const UMessage &msg)
{
  if (msg.binaryType != BINARYMESSAGE_IMAGE)
    return URBI_CONTINUE;

  unsigned char buffer[500000];
  int sz = 500000;
  static int tme = 0;

  if (!mon)
    mon = new Monitor(msg.image.width, msg.image.height);

  convertJPEGtoRGB((const byte *) msg.image.data,
                   msg.image.size, (byte *) buffer, sz);

  mon->setImage((bits8 *) buffer, sz);
  return URBI_CONTINUE;
}
```

Il commence par tester le type du message, et retourne sans rien faire dans le cas où l'on ne se trouve pas face à un message de type **BINARYMESSAGE_IMAGE** (par exemple, si le callback est réveillé sur un message d'erreur).

Ensuite, la fonction de conversion **convertJPEGtoRGB** est utilisée pour transformer le tampon d'image en quelquechose d'exploitable par l'objet **Monitor**, qui reçoit ensuite l'image.

Enfin, **URBI_CONTINUE** est retourné pour prendre en charge d'éventuels futurs callbacks.

Ce petit programme illustre très bien l'organisation d'un programme basé sur la *liburbi*: on place des callbacks, on envoie des scripts URBI et on reçoit les callbacks dans des fonctions écrites pour l'occasion. Vous pouvez jeter un oeil au code source GPL d'*URBILab*, ce dernier étant construit avec la *liburbi-C++*. Il montre un usage plus poussé de cette méthodologie.

Chapitre 9. Créer des composants: l'architecture *UObject*

L'architecture *UObject* est la façon la plus avancée d'étendre les possibilités d'URBI et d'intégrer de puissants composants au langage. Elle est pour l'instant limitée au *C++* mais devrait se généraliser aux autres langages à l'avenir. L'idée est de prendre une classe *C++* et, après quelques petites modifications, d'être capable de l'incorporer dans URBI de manière à ce que ses méthodes et ses attributs soient accessibles comme s'il s'agissait d'une classe URBI pure. Un peu de terminologie: l'architecture *UObject* permet d'ajouter un *composant* au langage, et ce composant sera vu comme un *objet*.

Il y a actuellement deux manières d'intégrer votre classe *C++* dans URBI:

- Le mode *plugin*: vous pouvez incorporer l'objet directement dans URBI (le lier à l'*URBI Engine*) et il fera alors partie du code binaire de l'*URBI Engine*.

- Le mode *remote*: vous pouvez sinon l'exécuter en tant processus distant qui se connectera à votre *URBI Engine* et ajoutera de façon transparente l'objet au langage, tout comme dans le mode plugin, mais de loin.

Dans les deux cas, nous vous fournissons les outils nécessaires à ce lien. La bonne nouvelle est que le code source *C++* de votre objet est exactement le même dans les deux cas, et la façon dont vous l'utilisez est également transparente. Vous pouvez donc choisir d'incorporer n'importe quel composant, à volonté (à l'avenir, il sera même possible de gérer la liaison à chaud).

Nous allons maintenant voir comment transformer votre classe *C++* en une classe *UObject*, et comment lier les méthodes et attributs de votre classe avec URBI.

UObject

Les bases

Créons un objet **colormap**, composé du fichier `colormap.cpp` et du fichier `colormap.h`. Le fichier `colormap.h` devrait débuter ainsi:

```
#include <uobject.h>
using namespace urbi;

class colormap : public UObject
{
  public:
  colormap(string);
  ...
};
```

Votre constructeur doit être renommé **init**. Le constructeur par défaut **lenomdemaclasse(string)** qui est adapté aux *UObject*s doit être utilisé à la place. Par exemple, définissons le constructeur **init** qui prend un point RGB (rouge/vert/bleu) comme définition colorimétrique:

```
public:
  colormap(string);
  int init (int r, int g, int b);
  ...
```

Pour l'instant, c'est tout se dont vous avez besoin du côté définition de classes. Passons au code de `colormap.cpp`:

```
#include "colormap.h"

UStart(colormap);

colormap::colormap(string s) : UObject(s)
{
  UBindFunction(colormap, init);
}

int colormap::init(int r, int g, int b)
{
}

...
```

Deux nouvelles choses ici: vous devez invoquer la ligne "magique" **UStart(monobjet)** afin que le system en prenne connaissance. Ensuite, vous devez vous assurer que le constructeur par défaut appelle bien le constructeur *UObject*, passe la chaîne de caractères (**string**), lie la fonction **init** pour la rendre visible et l'exporter vers URBI. Cette étape est nécessaire si vous voulez que le constructeur **init** soit appelé par URBI à la création d'un objet. La méthode **init** doit retourner *0* en cas de succès, et tout autre nombre en cas d'échec (vous pouvez aussi retourner **void** qui est considéré comme un succès).

Il n'y a rien de plus à savoir. A ce stade, vous disposez déjà d'un objet exportable appelé **colormap** avec une méthode **init**. Maintenant, vous pouvez le compiler et obtenir le code binaire, prêt à être lié.

Supposons que vous ayez lié le code à l'URBI Engine, pour en faire un composant en mode plugin (nous verrons comment un peu plus tard). Maintenant, comment exploiter ce nouvel objet **colormap** ? Hé bien, il n'y a rien à faire de plus: tout est là. Rappelez-vous qu'en URBI il n'existe aucune différence entre une classe et une instance (langage prototypé), donc définir **colormap** est suffisant pour obtenir un objet **colormap** fonctionnel. Vous pouvez l'évaluer et observer le résultat:

```
colormap;
[139464:notag] OBJ [load:1.000000]
```

NB: Par défaut, il existe un attribut **load** exporté, ignorons-le pour le moment.

Définissons une sous-classe de **colormap**. Cette action appelera le constructeur **init** côté *C++* et génerera une nouvelle instance de la classe *C++* **colormap** mais, bien entendu, tout est fait automatiquement et vous n'avez pas à vous en préoccuper:

```
ball = new colormap(123,45,12);
ball;
[139464:notag] OBJ [load:1.000000]
```

Vous pouvez constater que la syntaxe de création de l'objet est identique à celle de *C++*. Dans la mesure du possible, nous nous sommes efforcés de conserver la syntaxe bien connue du *C/C++* dans URBI, car nous considérons qu'il n'y a pas de temps à perdre à apprendre des choses que nous connaissons déjà (du moment qu'il n'y ait pas de confusion en terme de sémantique).

Ajouter des attributs

Notre objet **colormap** n'est pour l'instant pas vraiment enthousiasmant. Pour le rendre plus utile, ajoutons-lui des attributs et rendons-les disponibles dans URBI. Pour ajouter une variable **x**, nous allons tout simplement ajouter **UVar x;** à l'intérieur de la définition de la classe:

```
#include <uobject.h>
using namespace urbi;

class colormap : public UObject
{
public:
  colormap(string);

  UVar x; // définition de la variable exportée
  ...
};
```

et ajouter le code de liaison dans la méthode **init**:

```
int colormap::init(int r, int g, int b)
{
  UBindVar(colormap, x);
  ...
}
```

En réalité, vous pouvez mettre votre code de liaison (UBindVar) où vous voulez, en particulier il peut être dans le constructeur d'objet *C++* ou dans la méthode **init**. Si vous le placez dans le constructeur *C++*, la variable sera disponible pour l'instance de base (celle qui est là dès le début et qui ne nécessite pas d'être crée par quelconque **new**), ou si vous le placez dans la méthode **init**, seuls les autres objets créés avec **new** auront cette variable. Cela peut être utile si vous ne comptez pas utiliser l'instance de base dans le cas où vous souhaitez plutôt la dériver pour la spécifier. Dans ce cas, placez toutes vos liaisons uniquement dans la méthode **init** et l'instance de base ne sera alors qu'une instance fantôme. Sachez que **UObject::derived** est un booléen qui vous dit si la classe a été dérivée avec **new** ou s'il s'agit de la classe de base.

Vous pourrez vérifier: maintenant **colormap.x** et **ball.x** seront là.

Pour affecter une valeur à **x** depuis l'intérieur de votre classe *C++*, utilisez-là simplement comme une variable normale, *UObject* fera le reste:

```
x = 42;
  ou
x= "hello";
```

L'opérateur = de *C++* a été redéfini pour **UVar**, il n'y a donc pas d'inquiétude à avoir et vous pouvez donc affecter des valeurs à **x** comme vous le feriez dans URBI.

Désormais, comment lire la variable ? Là encore nous nous sommes efforcés de rendre les choses simples: vous pouvez employer un "casting" à la C pour obtenir la valeur dans le type C++ approprié. Attention cependant car il n'y a, pour le moment, aucune exception de levée en cas d'erreur:

```
x = 42;
printf("Valeur de x: %d\n",(int)x);
```

x est ce que l'on appelle un "hook" sur la variable URBI **colormap.x**. Vous pouvez aussi définir des hook sur n'importe quelle variable en définissant votre propre instance **UVar** où que vous vouliez (la liaison sera automatique, nul besoin de **UBindVar**, le constructeur **UVar** se charge de tout). Voici quelques exemples:

```
UVar("camera.val");
UVar("camera","val");
UVar* myvar = new UVar("headPan","val");
```

La raison pour laquelle vous devez appeler **UBindVar** pour une **UVar** définie dans le corps de votre classe est que cette **UVar** n'est pas allouée dynamiquement. Une telle UVar ne connaît pas encore son nom et la macro UBindVar lui dit simplement qui elle est. Cette étape n'est pas nécessaire avec le constructeur UVar(string) qui prend comme nom son argument.

Bien entendu, votre objet *C++* peut contenir plusieurs attributs qui ne seront pas exportés vers URBI et resteront "privés" au niveau de la classe *C++*. Pour rendre donc un attribut disponible à URBI, vous devez le définir comme une UVar ou réaliser un **UBindVar** sur une **UVar** de la définition de votre objet.

Il est parfois intéressant de surveiller les accès ou les modifications apportées à une variable. Pour cela, on affecte une fonction-callback à la variable, en indiquant si la fonction doit être appelée à l'accès ou à la modification de la dite variable:

```
UNotifyChange(x,&colormap::moncallback);
UNotifyAccess(UVar("doo.daa",&colormap::monautrecallback);
UNotifyChange("uneautre.variable",&colomap::unautrecallback);
```

Notifier à la modification (**UNotifyChange**) signifie que le callback est appelé à chaque fois que la variable est modifié du côté d'URBI (pour les variables liées à des capteurs, à chaque fois que la valeur du capteur est mise à jour). Notifier à l'accès signifie que le callback sera appelé à chaque fois que quelqu'un évaluera la variable côté URBI, vous permettant de mettre à jour la valeur avant l'évaluation. Dans ce cas, il est conseillé de coder un mécanisme de cache dans votre callback si la variable est destinée à être évaluée fréquemment.

Ces lignes **Notify** seront généralement placées dans la fonction **init** ou dans le constructeur de votre objet, le choix entre l'un et l'autre étant dicté par les mêmes raisons que pour **UBindVar**. Remarquez que vous devez passer un pointeur à l'une des méthodes de votre objet. Il existe deux façons de prototyper pour ce genre de callback:

```
UReturn moncallback();
UReturn moncallback(UVar&);
```

La première est la plus simple: la fonction est appelée lorsque la condition est remplie. La seconde fait la même chose mais passe en plus l'UVar en paramètre. Ainsi vous pouvez utiliser le même callback avec plusieurs variables et obtenir celle qui est en rapport avec l'appel courant.

Lier une fonction ou un évenement

Tout comme avec les attributs, vous pouvez relier une fonction à l'objet côté URBI. Il n'y a rien de spécial à faire, seulement respecter la construction suivante:

```
int colormap::init(int r, int g, int b)
{
  UBindFunction(colormap, faisquelquechose);
  ...
}

std::string colormap::faisquelquechose(int, float)
{
  ...
}
```

Cela rendra la méthode **faisquelquechose** visible de l'extérieur. Ne vous préoccuppez pas des paramètres, ils seront reconnus et exportés pour vous. Pour l'instant, il n'est cependant pas possible de surcharger une fonction avec ce mécanisme (et notamment, vous ne pouvez surcharger le constructeur **init**).

De la même façon, vous pouvez relier un évenement à l'une des méthodes de votre objet. Ainsi cette méthode sera appelée à chaque fois que l'évenement correspondant sera émis côté URBI, et vous obtiendrez les paramètres par la même occasion. Pour cela, faîtes:

```
UBindEvent(colormap, reagisacela);
```

Vous pouvez également demander à être informé de la fin d'un évenement (comme vous le savez, un évenement peut durer un certain temps en URBI). Par exemple, si vous désirez être notifié en appelant la méthode **cestlafin** de votre objet, faîtes:

```
UBindEvent(colormap, reagisacela);
UBindEventEnd(colormap, reagisacela, cestlafin);
```

cestlafin doit être prototypé comme ci:

```
void colormap::endthis();
```

Minuteurs

Vous pouvez facilement placer des minuteurs (**timers**) qui appeleront votre fonction à intervalle régulier. La syntaxe est la suivante:

```
USetTimer(temps_en_ms, &monobjet::moncallback);
```

moncallback étant une méthode de votre objet avec le prototype suivant:

```
UReturn monobjet::moncallback();
```

Vous ne pouvez pas utiliser une fonction venant d'un autre objet.

Types binaires avancés

L'affectation des entiers, des réels et des chaînes de caractères ainsi que leur lecture par "cast" d'**UVar** sont simples. Pour une donnée binaire comme une image ou un son, vous aurez besoin des types **UImage** et **USound**. Voici en citation leur définition dans le fichier `uobject.h`:

```
///Class encapsulating an image.
class UImage {
  public:
    char                *data;          ///< pointer to image data
    int                 size;           ///< image size in byte
    int                 width, height;  ///< size of the image
    UImageFormat        imageFormat;
};

///Class encapsulating sound informations.
class USound {
  public:
    char                *data;          ///< pointer to sound data
    int                 size;           ///< total size in byte
    int                 channels;       ///< number of audio channels
    int                 rate;           ///< rate in Hertz
    int                 sampleSize;     ///< sample size in bit
    USoundFormat        soundFormat;    ///< format of the sound data
    USoundSampleFormat  sampleFormat;   ///< sample format
}
```

Vous les reconnaissez: ce sont les types utilisés dans la liburbi. Si votre UVar est une image, comme **camera.raw**, vous pouvez simplement la "caster" en une **UImage** et les différents attributs seront renseignés, en particulier le contenu binaire sera dans **data** et la taille dans **size**. Idem pour un son.

Méfiez-vous de **camera.val**: il se peut que ce soit un binaire compressé avec la méthode JPEG. Dans ce cas, vous devrez le convertir avec l'une de ces fonctions, décrites à la section Fonctions de conversion:

```
int convertRGBtoYCrCb (const byte* source, int sourcelen, byte* dest);
int convertYCrCbtoRGB (const byte* source, int sourcelen, byte* dest);
int convertJPEGtoYCrCb(const byte* source, int sourcelen, byte* dest, int &size);
int convertJPEGtoRGB  (const byte* source, int sourcelen, byte* dest, int &size);
```

Si vous souhaitez affecter un son, disons à speaker.val, remplissez une variable USound et affectez-la à l'UVar appropriée, l'opérateur = a été redéfini pour gérer cela. Cependant, nous ne prenons en charge pour l'instant que le format WAV.

L'attribut load

Nous avons déjà mentionné l'attribut **load**, défini comme une UVar liée par défaut dans *UObject*. Cet attribut peut être utilisé pour tester, dans votre code *C++*, si l'objet est activé ou non du côté d'URBI. Dans URBI, un appel à **monobjet on;** mettra **load** à *1* et un appel à **monobjet off;** le mettra à *0*. Vous pouvez donc facilement tester dans différentes fonctions si vous avez à réaliser les calculs ou non, en vous basant sur la valeur de **load**.

Cela s'avère très utile si vous voulez être en mesure d'activer/désactiver un calcul lourd pour le processeur qui pourrait tourner pour rien en tâche de fond. Par exemple, vous pouvez éteindre la détection de balle de l'Aibo avec:

```
ball off;
```

Notez qu'il s'agit d'une construction diffusable: si vous faites on/off sur un groupe, cela se répercutera récursivement sur chaque membre du groupe. C'est exactement ce qui se passe avec la commande **motors on;**.

NB: Vous disposez également de **monobjet switch;** pour basculer entre **on** et **off**.

L'attribut remote

Dans la définition d'*UObject*, l'attribut **remote** est là pour savoir si votre objet tourne en tant que composant distant (**remote**) ou en tant qu'intégré (**plugin**). Cela peut s'avérer utile si vous souhaitez vous comportez différemment dans les deux cas, classiquement lors du tranfert d'un important volume de données, que l'on peut effectuer avec ou sans compression. **colormap** met à profit cet attribut.

L'exemple de colormap

Voici un exemple dans la pratique d'un objet colormap tel qu'il est utilisé dans Aibo pour calculer la position approximative d'un blob de couleur, définie par une sous-plage de l'ensemble de la plage de couleurs YCrCb. Vous allez pouvoir constater la liaison du callback avec la source, habituellement la caméra. Le callback est réglé sur l'attribut **.val** ou **.raw** de l'objet source, en fonction du status de l'objet, distant ou non. En mode distant, nous souhaitons utiliser la compression JPEG et travailler avec l'image en résultant, alors qu'en mode intégré, on peut utiliser de la mémoire partagée sur le tampon **raw** pour obtenir une meileurre image dépourvue d'artefact et ainsi éviter de compresser/décompresser pour rien.

Vous verrez également comme les affectations aux attributs **x** et **y** ainsi que les autres attributs décrivant la forme du blob sont simples:

First the colormap.h file (extracts only):

```
#include <uobject.h>
using namespace urbi;

class colormap : public UObject
{
  public:

    colormap(string);
    int init(string,int,int,int,int,int,int,ufloat);

    UVar   x;
    UVar   y;
    UVar   visible;
    UVar   ratio;
    UVar   threshold;
    UVar   orientation;
    UVar   elongation;
    UVar   ymin, ymax, cbmin, cbmax, crmin, crmax;
```

```
    UReturn newImage(UVar&);
};
```

Ici nous utilisons **ufloat** au lieu de **float** car il est adapté aussi bien au 32bits, qu'au 64bits, voire même aux cartes-mères dépourvues d'unité de calcul à virgule flottante. Il est donc préférable pour les applications embarquées.

Maintenant, voici le code principal:

```
#include "colormap.h"

UStart(colormap);

//! colormap constructor.
colormap::colormap(string s) :
  UObject(s)
{
  UBindFunction(colormap,init);
}

//! colormap init function
int
colormap::init(string source,
               int _Ymin,
               int _Ymax,
               int _Cbmin,
               int _Cbmax,
               int _Crmin,
               int _Crmax,
               ufloat _threshold)
{
  UBindVar(colormap,x);
  UBindVar(colormap,y);
  UBindVar(colormap,visible);
  UBindVar(colormap,ratio);
  UBindVar(colormap,threshold);
  UBindVar(colormap,orientation);
  UBindVar(colormap,elongation);
  UBindVar(colormap,ymin);
  UBindVar(colormap,ymax);
  UBindVar(colormap,cbmin);
  UBindVar(colormap,cbmax);
  UBindVar(colormap,crmin);
  UBindVar(colormap,crmax);

  if (remote)
    UNotifyChange(source+".val",&colormap::newImage);
  else
    UNotifyChange(source+".raw",&colormap::newImage);
```

```
  // initialization
  ymin        = _Ymin;
  ymax        = _Ymax;
  cbmin       = _Cbmin;
  cbmax       = _Cbmax;
  crmin       = _Crmin;
  crmax       = _Crmax;
  threshold   = _threshold;
  x           = -1;
  y           = -1;
  visible     = 0;
  orientation = 0;
  elongation  = 0;
  ratio       = 0;

  return 0;
}

//! colormap image update
UReturn
colormap::newImage(UVar& img)
{
  if ((ufloat)load < 0.5) return(1);

  UImage img1 = (UImage)img;  //ptr copy

  if (remote)
    convertYCrCb(img1); // this function is available in UObject 1.0 only

  int w = img1.width;
  int h = img1.height;

  //lets cache things
  int ymax = this->ymax; int ymin = this->ymin;
  int crmin = this->crmin; int crmax = this->crmax;
  int cbmin = this->cbmin; int cbmax = this->cbmax;

  long long x=0,y=0,xx=0,yy=0,xy=0;
  int size = 0;
  for (int i=0;i<w;i++)
    for (int j=0;j<h;j++) {

      unsigned char lum = img1.data[(i+j*w)*3];
      unsigned char cb  = img1.data[(i+j*w)*3+1];
      unsigned char cr  = img1.data[(i+j*w)*3+2];




      if ( (lum  >= ymin) &&
           (lum  <= ymax) &&
           (cb >= cbmin) &&
           (cb <= cbmax) &&
```

```
          (cr >= crmin) &&
          (cr <= crmax) ) {
        size++;
        x += i;
        y += j;
        xx += i*i;
        yy += j*j;
        xy += i*j;
      }
    }

  this->ratio = ((ufloat)size)/((ufloat)(w*h));
  if (size > (int)((ufloat)threshold * (ufloat)(w*h))) {

    this->visible = 1;
    this->x = 0.5 - ((double)x /
        ((double)size * (double)w));
    this->y = 0.5 - ((double)y /
        ((double)size * (double)h));

    //orientation: first eighenvector of covariance matrice
    double m00 = (double)xx - (double)(x*x)/(double)(size);
    double m11 = (double)yy - (double)(y*y)/(double)(size);
    double m01 = (double)xy - (double)(x*y)/(double)(size);

    //bigest eighenvalue
    double l = (m00+m11)/2.0 + 0.5*sqrt((m00+m11)*
               (m00+m11)-4*(m00*m11-m01*m01));

    //first eighenvector orientation
    double angle = atan2(l-m00, m01);
    this->orientation = angle* 180.0 /M_PI;

    //variance on new axis => elongation
    double angle2 = angle + M_PI/2.0;
    double X = x*cos(angle)+y*sin(angle);
    double Y = x*cos(angle2)+y*sin(angle2);
    double XX = xx*cos(angle)*cos(angle)+yy*sin(angle)*sin(angle)+
                2.0*xy*cos(angle)*sin(angle);
    double YY = xx*cos(angle2)*cos(angle2)+yy*sin(angle2)*sin(angle2)+
                2.0*xy*cos(angle2)*sin(angle2);
    double vX = XX - X*X/(double)size;



    double vY = YY - Y*Y/(double)size;

    this->elongation = sqrt(vX/vY);

  }
  else {
    this->x=-1;
    this->y=-1;
    this->visible = 0;
```

```
  }

  return(1);
}
```

L'objet **colormap** est alors incorporé dans l'URBI Engine et il est employé pour créer un détecteur de balle depuis le fichier `URBI.INI`:

```
ball = new colormap("camera",0,255,120,190,150,230,0.0015);
```

En pratique, comment profiter d'un *UObject*?

Nous mettons à disposition une série de scripts qui ont pour but de vous aider à construire un composant distant ou un composant intégré le plus facilement possible. Sous *Linux*, tout cela est fait par un fichier `Makefile` que vous devez placer dans le répertoire où se trouve le code source de votre *UObject*. Sous *Windows*, bibliothèques et projets seront fournis pour les compilateurs les plus courants, mais pour l'instant *Mingw* et l'environnement *Cygwin* doivent être utilisés. Dans ce qui suit, nous supposons que vous utilisez un environnement de la famille Unix.

Comment installer le kit de développement pour construire/lier des composants pour votre robot ?

Pour construire ou utiliser des composants pour un serveur URBI donné, vous devez installer le kit de développement (*SDK*) correspondant à ce serveur. Téléchargez-le sur le site Web d'URBI (ou depuis le site Web du constructeur de votre robot), décompressez l'archive, et, pour l'installer, taper en tant qu'administrateur (*root*):

```
make install
```

Comment créer et utiliser un composant distant ou intégré ?

Les composants distants et intégrés sont construits de la même façon. Copiez le fichier `Makefile.module` que vous trouverez dans `/usr/local/share/urbicore` dans le répertoire de votre composant et renommez-le en `Makefile`. Son comportement par défaut est de compiler toutes les sources du répertoire (éditer-le si vous avez besoin d'ajouter des actions supplémentaires ou de modifier les options de compilation).

Pour construire votre composant et le lier à un serveur particulier, tapez:

```
make TARGET=<cible> link
```

En remplaçant <cible> par la cible correspondant à votre serveur (en l'occurrence: **aibo**). Cette commande produira un nouveau serveur URBI avec votre composant intégré à lui. Remplacez l'ancien serveur avec celui-ci (le fichier `URBI.BIN` pour Aibo) et voilà !

Pour construire votre composant et en faire un composant distant, tapez:

```
make TARGET=remote link
```

Cela produira un exécutable qui prendra comme unique argument le nom d'hôte du robot ou son adresse IP.

Méfiez-vous: certaines cibles possèdent des dépendances supplémentaires, comme pour **aibo** par exemple qui nécessite le *OPEN-R SDK*. En d'autres termes, il vous faut installer le kit de développement de *Sony* avant de pouvoir construire des composants intégrés pour Aibo.

Comment incorporer plusieurs composants dans votre serveur URBI ?

Pour incorporer plusieurs composants dans votre serveur URBI, construisez-les comme expliqué ci-dessus si vous avez leurs sources, en omettant le **link** à la fin de la commande:

```
make TARGET=<target>
```

Cela va créer un ensemble de bibliothèques prêtes à lier. Tapez alors dans l'un des répertoires-sources:

```
make TARGET=<target> link LIBS=<libs> LIB_DIR=<libdir>
```

où **libs** est le nom de toutes les bibliothèques que vous souhaitez lier, et **libdir** le chemin de ces bibliothèques. Par exemple, si vous possédez un Aibo, que vous réalisez un composant **monopoly** dans le répertoire courant, et que vous désirez l'intégrer avec le composant **detecteargent** que vous avez téléchargé dans `/home/moi/recus` en tant que `libdetectmoney.a`, tapez:

```
make TARGET=aibo link LIBS=detecteargent LIB_DIR=/home/moi/recus
```

Comment distribuer l'un de vos composants et le proposer à d'autres ?

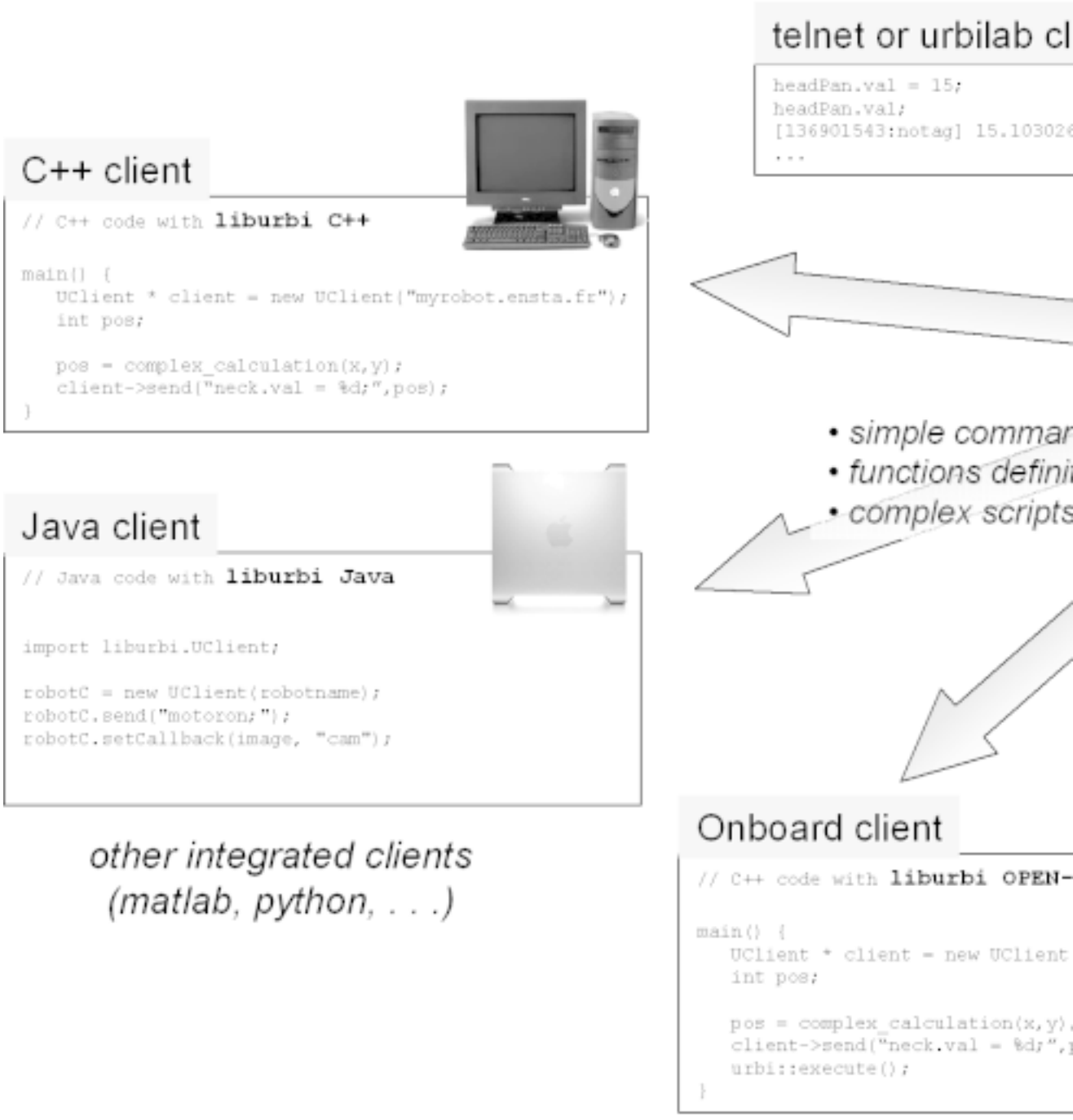
Vous pouvez distribuer soit les sources de votre composant, soit la bibliothèque (`.a`) générée par le processus de construction comme décrit précédemment (**make TARGET=<target>**). On pourra alors lier votre composant à un serveur URBI comme expliqué précédemment. Notez bien que la bibliothèque est dépendante de l'architecture: un composant compilé pour Aibo ne peut être lié en tant que module distant, les sources doivent être recompilées. Nous vous recommandons fortement de publier à la fois votre code source pour d'éventuels recompilations si votre licence le permet ET une ou plusieurs versions binaires pour les personnes désireuses de seulement lier votre composant et l'utiliser comme composant distant.

Le site Web http://www.urbiforge.com est une plateforme pour échanger les composants et les scripts URBI. Vous pouvez y envoyer votre travail afin que la communauté en profite.

Chapitre 10. Mettre tout cela ensemble

Le diagramme suivant vous montre une architecture typique de clients et de logiciels pour une application URBI. Vous avez des clients en *C++*, en *Java* et en *Matlab* tournant sur différentes machines (sous *Linux*, *Windows* et *MacOS X*), des *UObjects* distants, des clients sur place, des *UObjects* intégrés et des scripts tapés depuis *telnet/URBILab*. Il y a également un ensemble de scripts de contrôle lancés depuis le fichier `URBI.INI`. Cet exemple montre combien URBI peut être souple en permettant à tous ces systèmes de fonctionner en parallèle sur votre robot.

Figure 10.1. L'architecture générale d'URBI, en mettant tout ensemble



Exemples d'utilisations classiques

Vous disposez d'un serveur URBI sur votre robot et ...

1.

Contrôle à distance d'un robot

Votre robot est équipé d'une connexion WiFi, tel Aibo, et vous exécutez un programme complexe d'intelligence artificielle sur un puissant ordinateur personnel pour le contrôler. Ce programme est en réalité un client URBI écrit en *C++* qui utilise la *liburbi-C++* ou la bibliothèque *UObject* pour envoyer des commandes URBI au robot quand cela est nécessaire, et pour recevoir des messages URBI du serveur de manière asynchrone en vue d'y réagir de façon appropriée. Vous pouvez remplacer *C++* par *Java*, *Matlab* ou tout autre langage. Vous avez également quelques composants *UObject* exécutant d'intéressants algorithmes de vision.

2.

Un robot entièrement autonome avec un client URBI (des *UObjects*) à bord

Cette fois-ci, vous exécutez le client URBI ou les *UObjects* sur le robot et non à distance. Là encore, tout ceci est écrit en *C++* avec la *liburbi-C++* ou avec l'architecture *UObject*. Au lieu d'une connexion TCP/IP WiFi entre le client et le serveur, vous avez une communication directe entre les processus sur l'hôte local ou un accès direct à la mémoire partagée avec le mode intégré d'*UObject*.

3.

Un robot autonome contrôlé entièrement par des scripts URBI

Dans ce cas, cela signifie que vous avez trouvé toutes les fonctionnalités dont vous avez besoin dans URBI (pas de programmation externe en *C++* ou *Java* de nécessaire). Vous rédigez directement vos boucles perception-action avec des scripts URBI tournant sur le serveur, utilisant des composants pré-existants ou téléchargés en mode intégré. Vous n'avez besoin que d'un telnet ou d'URBI Remote pour envoyer vos scripts URBI au serveur, et voilà. Vous pouvez également stocker directement le script dans le fichier `URBI.INI` et votre robot le lancera alors au démarrage (nul besoin d'un client ou d'un ordinateur).

4.

Un peu des trois

Vous avez un robot contrôlé par plusieurs clients URBI en même temps, certains sur le robot, d'autres sur l'ordinateur, certains en *C++*, d'autres en *Java*, d'autres encore en *Matlab*. Par dessus tout cela, vous avez également plusieurs scripts URBI faisant tourner sur le serveur des boucles perception-action, utilisant de puissants UObjects écrits par vos soins mais aussi d'autres téléchargés sur Internet. Tout ceci est lancé depuis `URBI.INI` mais aussi dynamiquement chargé par certains clients, au besoin. C'est la situation la plus intéressante, exploitant au maximum la souplesse d'URBI.

Annexe A. Copyright

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

   1. "Collective Work" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License. 2. "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License. 3. "Licensor" means the individual or entity that offers the Work under the terms of this License. 4. "Original Author" means the individual or entity who created the Work. 5. "Work" means the copyrightable work of authorship offered under the terms of this License. 6. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive,

perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

1. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works; 2. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Derivative Works. All rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights set forth in Sections 4(d) and 4(e).

4. Restrictions.The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

1. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. 2. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works. 3. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied;

and to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit. 4.

For the avoidance of doubt, where the Work is a musical composition: 1. Performance Royalties Under Blanket Licenses. Licensor reserves the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work if that performance is primarily intended for or directed toward commercial advantage or private monetary compensation. 2. Mechanical Rights and Statutory Royalties. Licensor reserves the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions), if Your distribution of such cover version is primarily intended for or directed toward commercial advantage or private monetary compensation. 5. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor reserves the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions), if Your public digital performance is primarily intended for or directed toward commercial advantage or private monetary compensation.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR

EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

1. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License. 2. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

1. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License. 2. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable. 3. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent. 4. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.