URBI

UNIVERSAL ROBOTIC BODY INTERFACE

Jean-Christophe Baillie
Electronics & Computer Engineering Dpt.

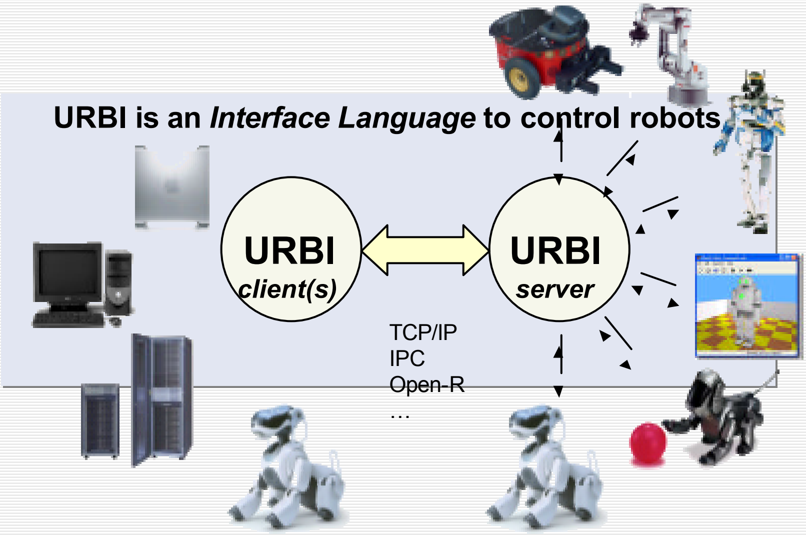
ENSTA

Ecole Nationale Supérieure des Techniques Avancées
Paris, France

Outline

- *Introduction*
- *Devices*
- *Examples*
- *Soft devices*
- *URBI usages*
- *Conclusion*

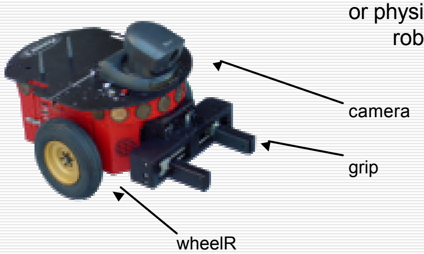
Introduction

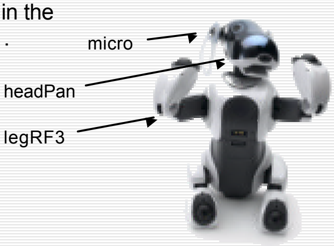


URBI is an *Interface Language* to control robots

Devices

Every sensor, motor, camera
or physical hardware in the
robot is a **device** .





A device is similar to a C++ object: it has **methods** and **properties**.

The **val** property is related to the device *value*

(telnet session, port 54000, with Aibo ERS7)

```
headPan.val = 15;
headPan.val;
[136901543:notag] 15.1030265089
accelX.val;
[136901543:notag] 0.002938829104
```

```
camera.val;
[145879854:notag] BIN 5347 jpeg 208 160
############# 5347 bytes ###########

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

Examples with Aibo

Soft device

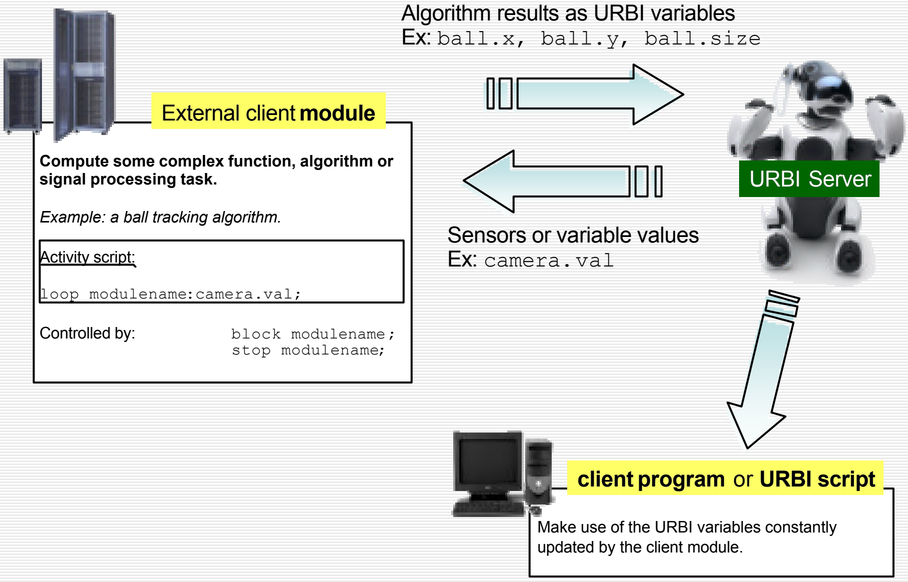
```
// The ball tracking program:

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

Group of devices (*virtual devices*)

```
// Stand up (leg1, leg2 and leg3 are virtual devices defined elsewhere)

getup: {
  { leg2.val = 90 time:2000 &
    leg3.val = 0 time:2000 } |
  leg1.val = 90 time:1000 |
  leg2.val = 10 time:1000 |
  { leg1.val = -10 time:2000 &
    leg3.val = 90 time:2000 }
};
```

for/while/if/…, functions, events, variables & more on www.urbiforge.com

Making reusable software modules



Algorithm results as URBI variables
Ex: `ball.x, ball.y, ball.size`

External client module

Compute some complex function, algorithm or signal processing task.

Example: a ball tracking algorithm.

Activity script:

```
loop modulename:camera.val;
```

Controlled by:

```
block modulename;
stop modulename;
```

URBI Server

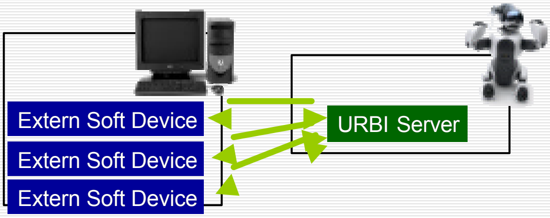
Sensors or variable values
Ex: `camera.val`

client program or **URBI script**

Make use of the URBI variables constantly updated by the client module.

URBI usages

Possible standard ?
*For **robotics** and **smart objects** in general*



- ✓ Boost **robotic software industry**
- ✓ Enable **home shared computing**
- ✓ **Educational** tool

Future extensions: **Server = Client**
(client functions included in the language)

Query methods:

```
devices; // return the list of devices
info headPan; // infos about a device
```

➔Each component can ask about what are the available components around and what they can do. Then, it can start to interact with it.

Conclusion

URBI Server for Aibo, Webots & liburbi are freely available for non commercial use at:

www.urbiforge.com

Next steps:

- Support **more robots** and **develop partnership/contracts with device manufacturers**:
  Currently: Aibo, HRP-2, Webots5 (robot simulator, Cyberbotics), Aldebaran Robotics
  Next: Sony, ActivMedia, Philips *iCat?*
- **Increase the number of languages** for the liburbi.
  Currently: C++, Java, Matlab, C++/OPENR.
  Next: Python, C, perl
- Continue to **add important features** in the kernel: client functions, debugging facilities, multi tagging, kernel 2 with multicore processors support and real-time scheduling.
- Develop **URBI plugins** to enhance the language and stimulate the community (GPL)
- Develop useful **associated software** & **tutorials** : URBI Lab, URBI center, URBI Dev

---

Demo

Short demo video:

URBI
UNIVERSAL ROBOTIC BODY INTERFACE

```
at (talk.finished == true)
      echo  « Thank you for » +
            « your attention »;
```

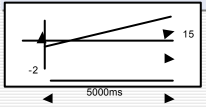
Complex Assignments

Simple assignment:

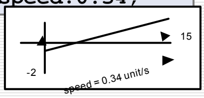
```
headPan.val = -2;
```

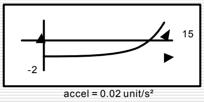
Numerical assignments can be specified via "**modifiers**"

```
headPan.val = 15 time:5000;
```

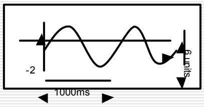


```
headPan.val = 15 speed:0.34;
```





This command never terminates



```
headPan.val = 15 accel:0.02;
```

```
headPan.val = -2 sin:1000 ampli:3;
```

Any function can be assigned as time parameterized trajectory wth the **function** modifier (0.9.7):

```
headPan.val'n = function(t):sqr(t)+sin(3*t+pi);
```

Time Operators

Commands can be executed in **serial** or **parallel** mode:

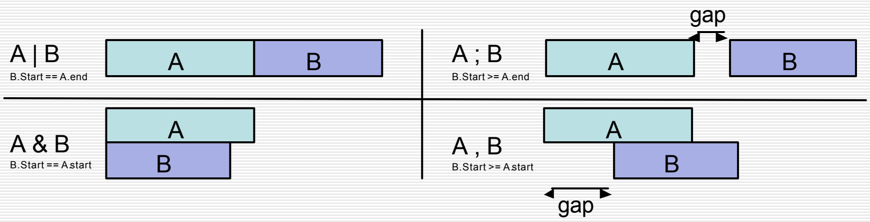
```
headPan.val = 15 & neck.val = 30;
```

Set headPan to 15 and neck to 30 *at the same time*

```
headPan.val = 15 | neck.val = 30;
```

Set headPan to 15 and *after*, set neck to 30.

Operators **,** and **;** are also available and have a semantics identical to **&** and **|** except that they have looser constraints:



A | B
B.Start == A.end

A ; B
B.Start >= A.end

A & B
B.Start == A.start

A , B
B.Start >= A.start

NB: Brackets can be used to group commands, like in C:

```
{ headPan.val = 15 | headTilt.val = 23 time:1000 } & neck.val = 10;
```

Event catching

Several event catching mechanisms are available:

```
at (test) {
   instructionsA;
};
```

```
at (test) {
   instructionsA;
}
onleave {
   instructionsB;
};
```

A will be executed <u>once</u> when *test* becomes true. Then, as soon as *test* becomes false, B is executed and the server waits for *test* to be true again. *test* can be set with an **hysteresis** on the number of successful test checks or time.

```
whenever (test) {
   instructionsA;
};
```

```
whenever (test) {
   instructionsA;
}
else {
   instructionsB;
};
```

When *test* becomes true, A is executed. When A is finished, it is executed again if *test* is still true, otherwise, B is executed.

```
waituntil (test);
```

usage:

```
waituntil (test) | instructions…
```

Terminates only when *test* becomes true.
=> *If given a number, the* ***wait*** *command pauses for this nb of ms.*

<u>In development</u>: `emit/catch` events with parameters

Advanced Tagging

stop / block / freeze

```
mytag: { commands...};

stop mytag;    // stops the commands

block mytag;  // kills any new command with tag "mytag«
unblock mytag;

freeze mytag; // freeze any running or new command
unfreeze mytag;
```

Flags: tag modifiers

```
+stop(x==1): { commands...}; // stops on condition

+freeze (headSensor.val > 0): { ... }; // freeze on cond.

+timeout (10000) : command; // stops after 10s

+error: command; // reports errors
+begin: command; // reports beginning of the command
+end:   command; // reports ending of the command
```

Variable Properties & Extensions

Variables can be assigned several properties, using the -> indirection:

`myvar->rangemin = x;`	Stores the **min** range for the variable `myvar`
`myvar->rangemax = 145;`	Stores the **max** range for the variable `myvar`
`device.val->speedmin = 24.4;`	**Speedmin** is the minimal speed value (units/s)
`device.v`	ive
`neck.val`	
`neck.val`	
`neck.val`	

Variables in URBI are not simple containers like in C or similar languages:

they are **dynamic structures** evolving over time.

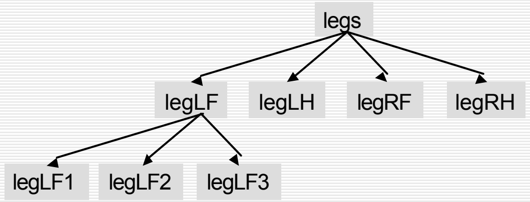
Variables eva

`myvar';` `device.x'';`	First/Second order derivative,from *assignments*
`device.val'd;` `device.val'dd;`	First/Second order derivative of `device.val`,from *sensors*
`neck.val'n;`	**Normalized** value (using rangemin/rangemax)
`neck.val'e;`	*Assigned – measured* error (approx. torque measure).

Device Grouping

Devices can be grouped into **virtual devices** to form a hierarchie:

```
group legLF {lefLF1, legLF2, legLF3};
group legs {legLF, legLH, legRF, legRH};
```



Commands go down the hierarchy, allowing useful factorizations:

```
legs.PGain = 0; // will affect the P Gain of all sub devices
```

The **@** prefix prevents the recursive descent:

```
@legs.PGain = 0;
```

Liburbi: quick review

Telnet is of course too limited => **liburbi** for **C++** programming.
Exists also for **JAVA, Matlab** and **OPENR** (soon: Python, C)

The URBI library (liburbi) give simple methods to **send commands** to and **receive messages** from the URBI server.

C++ example:

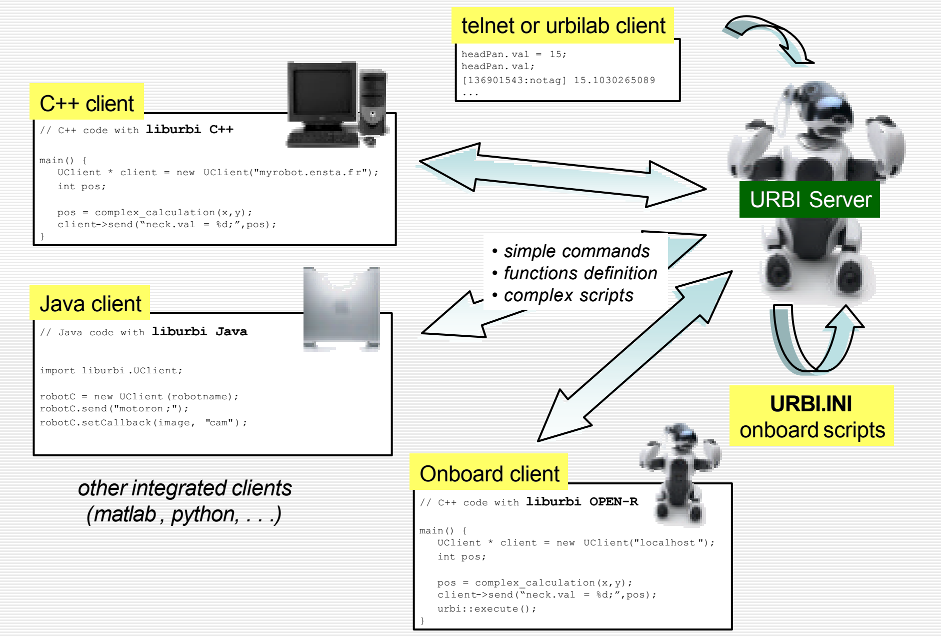
```
UClient * client = new UClient("myrobot.ensta.fr");

client->send(" headPan.val = %d", x);

client->syncGetDevice("neck",&neckVal); // exists also for binary

client->setCallback(&receiveMyImage,"imgtag");
client->send("loop imgtag:camera.val;");
```

synchronous: `client->syncGetDevice(...)`

asynchronous: `client->setCallback(...)` / `client->send("loop imgtag:camera.val;")`

```
UCallbackAction receiveMyImage(const Umessage &msg)
{
… /* handles the image contained in Umessage */
}
```

How to use URBI?



telnet or urbilab client

```
headPan.val = 15;
headPan.val;
[136901543:notag] 15.1030265089
...
```

C++ client

```
// C++ code with liburbi C++

main() {
   UClient * client = new UClient("myrobot.ensta.fr");
   int pos;

   pos = complex_calculation(x,y);
   client->send(“neck.val = %d;”,pos);
}
```

Java client

```
// Java code with liburbi Java


import liburbi.UClient;

robotC = new UClient(robotname);
robotC.send("motoron;");
robotC.setCallback(image, "cam");
```

other integrated clients (matlab, python, . . .)

- *simple commands*
- *functions definition*
- *complex scripts*

URBI Server

URBI.INI
onboard scripts

Onboard client

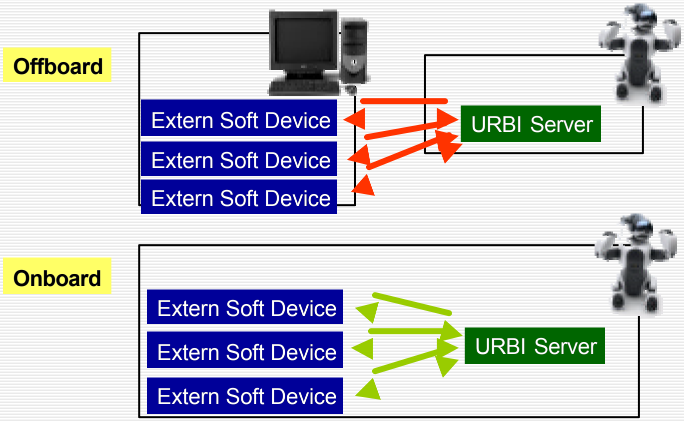
```
// C++ code with liburbi OPEN-R

main() {
   UClient * client = new UClient("localhost");
   int pos;

   pos = complex_calculation(x,y);
   client->send(“neck.val = %d;”,pos);
   urbi::execute();
}
```

Soft Devices

The creation of “soft” devices extends the devices available in the robot. This is done by creating a subclass of `USoftDevice`. Soft devices can monitor variables, redirect function calls (C++/URBI binding) and events relative to them.

Example:

`ball` => `ball.x, ball.y`
`voice` => `voice.say(“hello”), voice.hear(x)`



Offboard

Extern Soft Device
Extern Soft Device
Extern Soft Device

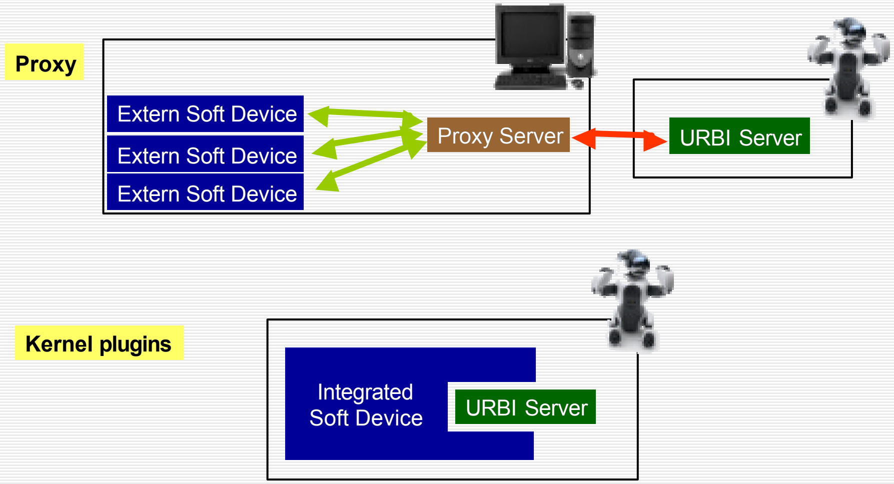
URBI Server

Onboard

Extern Soft Device
Extern Soft Device
Extern Soft Device

URBI Server

Soft Devices (2)



Proxy

Extern Soft Device

Extern Soft Device

Extern Soft Device

Proxy Server

URBI Server

Kernel plugins

Integrated Soft Device

URBI Server

With the **URBI Plugin Library**: <u>the same C++ code for all integration possibilities</u>

Performances

Performances

Test on a wifi 802.11B wireless connections, with a URBI server for ERS7, a linux C++ client and tests with both the client and the server on the robot, and OPEN-R based connection.

Latency Bandwidth:	**1.5ms** ± 0.5 ~**500Ko/s**	Latency Bandwidth:	**600 µs** ~**2Mo/s**
Jpeg compress (90):	10ms for 208x160 **3ms for 104x80**	**~30 motor commands per second per motor** 208x160 jpeg92 = **30fps**	

Good performances in general. For highly demanding low level action/perception loops, the best is to **run some URBI Clients on the robot** and leave the **high level slow control architecture offboard**.