International Journal of Humanoid Robotics

URBI: A UNIVERSAL LANGUAGE FOR ROBOTIC CONTROL

JEAN-CHRISTOPHE BAILLIE
Laboratory of Electronic and Computer Engineering, ENSTA,
15 Bd Victor, 75015 Paris, FRANCE
jean-christophe.baillie@ensta.fr

The recent development of relatively cheap and powerful humanoid and animal-like robots has been accompanied by the development of a variety of softwares, libraries and other interfaces to control the robots. The diversity and complexity of these interfaces can only slow down the development of robotics and we believe that there is a need for a standard which could be universal, powerful and easy to implement. URBI, a Universal Robotic Body Interface, aims at providing the ground for such a standard. It is based on a client/server architecture. The server is running on the robot and accessed by the client, typically via TCP/IP. URBI includes a scripted language used from the client and capable of controlling the joints of the robot or access its sensors, camera, speakers or any accessible part of the machine. We describe here the principles behind URBI and how URBI differs from currently existing solutions, and present its specifications and capabilities using practical examples. We also introduce the URBI C++ Library which provides a simple and yet powerful way to program a client and to make a full use of the functionalities of URBI. We also give examples of URBI running on an Aibo ERS7 and analyse the performances of this implementation.

Keywords: Interface; SDK; Client/Server; Programming; Language; Standard.

1. Introduction

Humanoid and animal-like robots are more and more widespread and made available to researchers. Recently, there has almost not been a month in Japan without a company presenting its new humanoid robot. However, each company is developing its own programming language interface together with their robots, most of the time forcing researchers to relearn what they already know. Besides, some of these interfaces, like the Sony OPEN-R SDK[1], are notably difficult to master and, to our opinion, unnecessarily complex, which makes it troublesome to program even the simplest behavior[a].

We have developed URBI, a Universal Robotic Body Interface, in an attempt to provide the research community with a standard for robot control and interfacing, which would be both powerful and simple to use. URBI includes a scripted language

[a]With OPEN-R, moving a joint implies the creation of a shared memory region, which contains an array of commands describing sets of joints positions in future time frames and passing this shared memory structure to a remote object via an inter object message passing mechanism.

used from the client and capable of controlling the joints of the robot or access its sensors, cameras, speakers or any accessible part of the machine. The main characteristics of URBI, which make it different from other existing solutions are:

- URBI is an interpreted and scripted low level command language. Motors and sensors are directly read and set. Although complex high level commands can be written with URBI, the kernel of the system is low level by essence.
- Using a client/server architecture, URBI is designed to be independent from both the robot and the client system. This means that it is possible to use any platform and any language (C++, Java, etc) on the client side to program the robot. It relies on TCP/IP or Inter-Process Communication if the client and the server are both running on the robot.
- URBI is designed with a constant care for simplicity. There is no "philosophy" or "complex architecture" to be familiar with. It is understandable in a few minutes and can be used immediately with a simple telnet client, while it keeps advanced powerful features for more advanced users.

We will present shortly the two main existing solutions and why they differ from URBI. We will then present the client/server architecture used by URBI and the main characteristics of the language, with examples. We will also give explanations on the design decisions that we made. To facilitate programming with URBI, the C++ URBI Library is introduced and we give examples using the existing implementation with an Aibo ERS7 over a 802.11B wifi connection. We will show that, in this context, URBI is capable of handling simultaneously real-time complex motor control (walk or turn sequences), multiple simultaneous image retrievals at 30fps, stereo sound retrieval, sound streaming and smooth action/perception feedback loop (head tracking of the ball in the vision field).

2. Other approaches

Two important projects similar to URBI that we will shortly review here include Tekkotsu[2] (www.tekkotsu.org) and Player/Stage[3] (playerstage.sourceforge.net).

Tekkotsu is currently specialized for Aibo robots and requires to master a complex software architecture and API before being usable. It is built on top of OPENR, the Aibo official SDK, and provides high level functionalities, behaviors and algorithms. The price to pay for these high level features is an increase in complexity, which propagates to the use of low level features (it is necessary to understand a complex structure to do a simple task). URBI on the other hand is kept simple by leaving the high level programming to the user who might later develop or download high level URBI-based libraries if needed. Also, URBI is not specialized for Aibo robots.

Like URBI, Player/Stage is also based on a client/server architecture. It provides high level functionalities like the "GoTo(x,y,θ)" function and it is currently specialized for wheeled robots (θ is the orientation of the robot). Many of the features

related to position, odometry or speed in Player/Stage are difficult to translate in the case of a humanoid or a four-legged robot and we believe this approach, while being very interesting to investigate, is too specialized and high level to be a candidate for a low level standard. Player/Stage is also a complex system which requires time to be familiar with.

Other approaches like Soar[4], ACT-R[5], Orocos[6] (LAAS) and others, which are control architectures or cognitive architectures, go beyond the purpose of URBI which is only to provide the programmer with a simple interface to the robot's body and sensors.

3. Architecture and design principles

URBI is based on a client/server architecture. A URBI server is running on the robot and a client is sending commands to the server in order to interact with the robot. The channel between the client and the server can be a TCP/IP connection or direct Inter Process Communication if the client and the server are both running on the robot. In that later case, the latency is expected to decrease significantly.

The robot is described by its *devices*. Each element of the robot that can be controlled or each sensor is a device and has a device name. From a programmer's point of view, a device is an object. It has some mandatory methods and variables and a list of device-specific methods. Everything that can be done on the robot is done via the devices and the available methods associated to them.

The main advantage of using this architecture is the flexibility it allows. The client can be a simple telnet client or a complex program sending commands over TCP/IP. This client can run on linux, windows or any other system and it can be programmed in C++, java, LISP or any language capable of handling TCP sockets. For each new robot type, a new server has to be written. Once this server is running on the robot, it is straightforward to command the robot, whatever the robot is or how complex it is, as soon as one knows the list of devices and their associated methods. This list is made available in the documentation of the server and it is the only robot-specific piece of information required to know how to control a previously unknown robot.

The syntax used to access the devices is designed with simplicity in mind. More complex features of the URBI language are available and will be described here but understanding them remains easy and is an incremental, layered process: it is not necessary to understand the complex features to use the robot at a basic level[b]. This was an important design constraint for URBI.

[b]Many robot SDK require to master a complex software architecture before being able to do even a simple motor movement. There design is not "layered".

4. Performance issues

Performance is an important issue for robot control. In the last section, we will give results and benchmarks in the case of the Aibo robot, showing that URBI is delivering at the maximal speed allowed by the network. Moreover, if the server and the client are both running on the robot, the wireless lag can be cancelled.

Since URBI includes a scripted language, it is also possible to upload perception/action loops inside the URBI server and run them onboard, removing all the communication overhead. The "BallTrackingHead" example, described in 7.3 illustrates this technics.

These capabilities show that URBI is useable for demanding applications, requiring rapid perception/action loops and reactivity.

5. URBI language

The main part of URBI is the URBI language which defines how commands can be sent to the robot, what kind of scripting features are available and the syntax associated.

The working cycle of URBI is to send commands from the client to the server and to receive messages from the server to the client. Commands can be written directly in a telnet client on port 54000, where the messages will also be displayed or, as we will see later, using a program and a library.

5.1. *Getting and setting a device value*

As we said in the introduction, each element of the robot is called a device and has a device name. For example, in the case of Aibo, here is a short list of devices: legFL1, neck, camera, speaker, micro, headsensor, accelX, pawLF, ledF12... The first thing that can be done with a device is to read its value. This is done with the "val" method. For example, it can be done with a telnet client opened at port 54000 on the robot. We start every line with the ">" sign to show the command prompt but this is not visible in a normal telnet session, nor is it part of the command syntax. Other lines without ">" are messages from the server:

```
> neck.val;
[036901543:notag] 15.1030265089
```

The message returned is composed of a first part between brackets displaying a timestamp in milliseconds (from the start of the robot) and a command tag. In this case, the command tag is "notag", since no tag has been specified with the command. The tag can be specified before a ':', preceding the command. With the command tag, it is possible to retrieve the associated message later, possibly in a flow of other messages from the server:

```
> mytag: neck.val;
[041307845:mytag] 15.0040114317
```

This tagging feature is an essential part of URBI and the URBI C++ library where callback functions can be associated to any tag, enabling asynchronous message handling (see section 6).

Error messages are tagged with the "error" tag when they are not related to a specific command. This is useful when one wants to implement an error handling function, which will be triggered each time a message tagged with "error" is received. Warning messages also exists.

The second part of the message is the response of the server. In the case of our example, it gives the value of the Aibo *neck* device which is the position of the neck motor in degrees. The "val" command can be used with any device. The type of data returned depends on the device and can be checked with the "unit" instruction:

```
> unit neck;
[041447411:notag] "deg"
```

Available units are, among others, 'deg', 'bool', 'lum' (luminosity), 'cm', 'Pa' (pressure), 'm/s^2' (acceleration).

The minimal and maximal range are also available with any device, using the "maxrange" and "minrange" instructions. For example, in the case of the *headPan* device:

```
> therangemax: rangemax headPan; therangemin: rangemax headPan;
[057845441:therangemax] 91.0000000
[057845441:therangemin] -91.0000000
```

Note that the "valn" method also exists and is similar to "val" except that it handles a normalized value between 0 and 1, computed from the minrange and maxrange values of the device. This is useful when one does not know a priori the unit and range of a device or to write programs which are to a certain extend "robot-independent".

The "val" method can also be used to set a particular device value. If the device is a motor, it is going to move to the specified value. In the case of a LED, this will switch it to the corresponding illumination (between 0 and 1):

```
> motoron;
> headPan.val = 15;headTilt.val = 20;
```

The "motoron" command is necessary at the beginning to start the motors of the robot. The two next commands set the position of the head pan and tilt. Note that, by default, a setting command does not produce any return message. This can be modified with options associated to the tag at the beginning of the command, like the "+report" option which tells the server to be in "verbose" mode. The client

will be prompted when the command starts and when it finishes. With the "+error" flag, the server will notify the client if any error occurs (obstruction on the motor movement for example).

```
> mycommand +report: legLF2.val = 15;
[058477124:mycommand] *** start
[058477156:mycommand] *** stop
```

Note how meta information regarding the command are prefixed by ***. This is to make clear that the message is not a return value and has no type. Command specific error messages and warning messages are also prefixed by ***.

The tag "mycommand" is compulsory since if one uses only "+report", it would be difficult in general to know what the "start" and "stop" refers to when several commands are started at the same time.

5.2. *Modificators*

The value specified by a "val" command is reached as quickly as the hardware of the robot allows it. It is however possible to control the speed and other movement parameters using *modificators*.

The following example commands the robot to reach the value 80 deg for the motor device *headPan* in 4500 ms and the value 40 deg for *headTilt* with a speed of 12.5 degrees per seconds:

```
> headPan.val = 80 time:4500;
> headTilt.val = 40 speed:12.5;
```

The speed or time modificators are always positive numbers. It is possible to specify a speed without giving a targeted final value by setting the desired value to infinity (inf) or minus infinity (-inf). For example, in the case of a wheeled robot, one might need to control the right wheel speed with:

```
> wheelR.val = inf speed:120;
```

If the range of the device is not infinite, the command will stop when the value reaches maxrange or minrange.

Another interesting modificator is "accel" whose meaning is to control the acceleration, and "sin" which tells the robot to reach the value in a specified time and in a sinusoidal way. This is useful when a circular movement is required.

In future versions of URBI, it is planned to provide a way to define custom modificators, based on any motion profile and not only sinusoidal trajectories. This will be particularly useful to describe walk sequences.

Modificators can be combined, "time" being prioritary over "speed" which is prioritary over "accel":

```
> wheelR.val = 150 speed:120 time:2000;
```

This command means that the value 150 must be reached at speed 120. This speed must be reached in 2000ms (this sets an implicit first phase of acceleration of 60 deg/s^2). The priority between modificators tells which modificator is modifying the others in case of a combined use.

An important point about modificators is that it is not only available to set devices but for any kind of variable (variables of a device or global variables). Considering the following example:

```
> myvariable = 0;
> myvariable = 50 time:10000;
> myvariable;
[001410040:notag] 2.45471445
> myvariable;
[001412020:notag] 12.35471445
> myvariable;
[001442020:notag] 50.00000000
```

The first affectation sets the variable to zero and the second one tells the robot to reach the value 50 in 10 seconds. When the value of "myvariable" is checked over time, we see that it is evolving from 0 to 50 during this time interval. This is a unique and powerful feature of URBI compared to other existing languages and which makes it a fundamentally asynchronous and time-oriented language. It allows to create a dynamics for parameters, useful in many situations like for example in the design of a "walk" sequence for a legged robot.

5.3. *Binary data sending and receiving*

Some devices like cameras, speakers or microphones are handling binary data. In the case of Aibo, the camera device is called *camera* and its value (the current image) can be retrieved with a "val" command, just like any other device:

```
> camera.val;
[004757741:notag] BIN 5347 jpeg 208 160
########################################################
########### 5347 bytes of binary data ##############...
```

Binary data always starts with the word "BIN", immediately followed by the number of bytes that will be received. Extra information follows. In the case of an image, it is the image encoding type (jpeg or YCbCr for Aibo) and the image height and width. After the carriage return, the binary data starts. The system switches back to ascii mode after the last byte is sent.

It is possible to specify in which format the image should be sent (jpeg or YCrCb or, in future versions, mpeg) with the "format" method of the camera device. It is also possible to set the jpeg compression factor, the camera gain, white balance or shutter speed. Here is an example session:

```
> camera.format = 0;
> camera.jpegfactor = 80;
> camera.gain = 1;
> camera.wb = 2;
> camera.shutter = 0;
> camera.resolution = 0;
```

See the specific Aibo URBI server documentation for the precise meaning of the values. This example is interesting here because it shows how device-specific variables can be set using other methods than "val" or "valn". This is illustrating the ease of use of the "device.method" format chosen for URBI and how extensible it can be to support any kind of device with particular methods that might exist in future robots. This is an important point if URBI is to become a standard.

In the case of sound, for the Aibo *micro* device for example, one can request the incoming sound for 500ms with the following command:

```
> micro.val(500);
[004857788:notag] BIN 2048 wav 16000 16 2
########################################################
########### 2048 bytes of binary data ##################
#########################################...
```

If the parameter 500 is omitted, the sound is streamed without interruption, in small samples of 32ms (it can be stopped with a "micro.stop" command). The extra information after "BIN 2048" is the sound encoding (wav file in the case of Aibo), sample rate in Hz, encoding depth in bits and the number of channels.

To send sound, the syntax is similar:

```
> speaker.val = BIN 2048 wav;
########################################################
########### 2048 bytes of binary data ##################
#########################################...
```

Note that since the server is waiting for a command terminated by a semicolon, it is right after the semicolon that the binary data starts.

Of course, these binary transfers are not usable when the client is a telnet client and it is necessary to use a programming interface like the C++ URBI Library to make a full use of these features. As we will see, many functions are provided to easily and transparently send or receive binary data.

Note that it is possible to play a specific file on the robot with *speaker.play("myfile.wav")* sent from a telnet client.

5.4. *Serial and parallel commands*

One key feature of URBI is the ability to process commands in a serial or parallel way.

When separated by the "&" operator, two commands will be executed in *parallel*. Moreover, they will start at exactly the same time:

```
> headPan.val = 15 & headTilt.val = 30;
```

This will move the head pan and tilt together, with both motors starting at the same time.

In the same way, it is possible to *serialize* commands by separating them with a pipe. In that case, the second command will start just after the first one is finished, with no time gap.

```
> headPan.val = 15 | headTilt.val = 30;
```

This will move the head pan to 15 degrees and only when this value has been reached, and just after, it will start to move the headTilt motor.

Two commands separated by a semicolon have almost the same time semantics as the serial "|": the second will start after the end of the first, but the time gap between the end of the first and the beginning of the second is not specified. This is close to the standard semantics of C or C++. Most of the time, URBI commands will be separated by semicolons. The main difference in using semicolons is that commands are interpreted as they arrive, whereas with pipes the chain of commands cannot start until it is completely received (and terminated by a semicolon for example).

Finally, two commands can be separated by a colon. In that case, the time semantics is close to the parallel operator "&", except that the two commands don't necessarily start at the same time. The meaning of a colon terminated command is simply to start the command as soon as possible. In particular, as soon as the command is in the buffer, it will be executed, whereas with "&", the chain of commands must be integrally received before execution.

The following relationships represent those time dependencies:

```
a;b  :  b.start >= a.end
a,b  :  b.start >= a.start
a&b  :  b.start == a.start
a|b  :  b.start == a.end
```

The operator priority is the following: ; , & |

These time sequencing capabilities are another specificity of URBI and are very important features to design and chain complex motor commands or behaviors.

5.5. *Loops, conditions, event catching*

Several control structures are available, like the classical "for", "while" and "if". Some extra control structures like "loop", which is equivalent to "while (true)", or "loopn (n)" equivalent to "for(i=0;i<n;i++)" are also provided for convenience.

The syntax of "for", "while" and "if" is the same as in C. Here are some examples:

```
if (ledF11.val == 1) {
  if (ledF12.val == 0)
    ledF12.val=0.254 speed:0.1;
  ledF11.val = 1;
} else
  ledF11.val=1 time:1454;

for (i=0;i<100;i++) {
  headTilt.val = i/100 sin:124;
  ledF10.val = i/100;
};

while (legLF1.val < 12)
  legRF1.val = lefLF1.val;
```

As a specificity of URBI, event catching control structures like "whenever" and "at" are also available. The instruction "at (test) command" will execute the command only once at the moment when the test becomes true. It is possible to set a hysteresis threshold associated to the test so that the test has to be false n times before it can trigger the command again when it becomes true. This is done with the tilde separator in the test. The following example let the head move in circles, except when an object is detected in the 25cm short range for an Aibo:

```
period=2500;

at (distanceNear.val > 25 ~ 3)
   scanning: loop {
       { headTilt.val = 90 sin:period |
         headTilt.val = -90 sin:period }
       &
       { headPan.val  = 90 sin:period |
         headPan.val  = -90 sin:period }
   }
else
   stop scanning;
```

In this example, the hysteresis threshold is set to 3, which means that the test must be false 3 times before it can trigger the "loop" command again. The meaning of the "else" part is symmetrically identical. The "stop" command, followed by a tag name, means that any command with this tag will be stopped. This is used here to stop the head circular sweeping. Note how the serial and parallel operators are used to specify the circular head movement. The number after the tilde operator

can also be a time period. In that case the time unit must follow the number, like "350ms".

The instruction "whenever (test) command" will execute the command as long as the test is true. When the test becomes false, the command is not restarted once it is finished and the "whenever" instruction waits for the test to become true again.

Without entering into details, we can say that the mechanism used by the URBI kernel to process commands involves command substitution in the command stack. When executed, the command "if (test) command1 else command2" is transformed into "command1" if (test) is true, and "command2" otherwise. In the same way, when the command "while" is executed, it is immediately replaced by the following command:

```
while (test) instruction;
<=>
    if (test) {
        instruction;
        while (test) instruction;
    }
    else noop;
```

Note that "noop" is an instruction which take a cycle to execute and does nothing.

This substitution mechanism is computationally efficient and, as a side effect, gives a precise way to describe the semantics of the instructions. Here is the semantics of "at" and "whenever", as described in the kernel:

```
whenever (test) instruction;
<=>
    if (test) {
       instruction;
       whenever (test) instruction
    }
    else {
      noop;
      whenever (test) instruction
    }


at (test) instruction; <=>
    if (test) {
      instruction;
      at (!test)
        at (test) instruction;
    }
    else {
```

```
    noop;
    at (test) instruction;
  }
```

We are currently working on a more formal mathematical description of the URBI semantics, but the above description will be used as a reference.

Like "else" for the "if" instruction, there is a "else" part for whenever and a "onleave" part for "at". The semantics of "else" and "onleave" is symmetrically defined compared to the main body of the instruction.

5.6. *Device grouping*

An important feature of URBI is the capacity to group devices. This is done with the "group" command: group virtualdevice device1, device2, ...:

```
group legLF {legLF1, legLF2, legLF3};
group legs {legLF, legLH, legRF, legRH};
```

This grouping feature is useful when a method is called on a device (or a virtual device): it is executed for this device and then it is recursively passed to any child subdevice. In other terms, the command "legLF.PGain = 0" will set the P Gain value of lefLF1, legLF2 and legLF3 to 0. This child passing mechanism differs from the usual object oriented inheritance mechanism. The fundamental reason is that object oriented hierarchies are based on "is-a-kind-of" relationships whereas the grouping hierarchies are based on "is-a-part-of" relationships.

NB: It is possible to block the passing mechanism by prefixing the device name with a "@". In that case, only the specified device will execute the command.

For any robot, it is advised to design a hierarchy of devices, with *robot* on the top. This is usually done via a URBI.INI file stored on the robot (at the root of the filesystem). This file is by default executed by the server at start and it is the place where every robot-specific configurations should be defined. There is also a CLIENT.INI file that is executed at each new client connection.

5.7. *Multiplexing*

The URBI server running on the robot is a multi client server. This means that it is always possible that two contradictory commands are sent to the server, from two different clients. For example, what should be done if one client wants the *neck* device to be set to 20 degrees while the other one requests a value of -20?

Three strategies are available:

(i) [*discard*]: Ignore any conflicting command
(ii) [*queue*]: Queue the commands and execute them one after the other (default)
(iii) [*mix*]: Mix conflicting commands by averaging the instantaneous values

Each strategy can be selected via the parameterized "blend" instruction. Using the virtual device "robot", defined in a grouping command, one can set the whole set of devices to the third strategy by:

```
> blend[mix] robot.val;
> blend robot.val;
[087945428:notag] "mix"
```

In the case of a sound playing device, setting the blending strategy to "mix" enables the robot to play several sounds at the same time.

Like modificators, note that a blending strategy can be attributed to any variable, not only to "device.val".

5.8. *Method definition*

URBI implements a simple method definition functionality. This is useful to define custom methods for a device or a virtual device. The method can be seen as a function, defined with "def", and can return a value.

```
def robot.mymethod (x,y) {
  for (i=0;i<x;i++)
    legLF1.val = y + i &
  ledF11.val = 1;
}

def myadd (x,y) {
  return x + y + ledF1.val;
}
```

These examples shows that a function or a procedure can be specific to a device or a virtual device or can be global (myadd). The keyword "self" can be used to refer to the device when the definition is specific to a device. This is useful when the method is passed to children in a group hierarchy since it makes the method aware of the hierarchy level it is called from.

```
def robot.setval (x) {
  self.val = x;
}
```

Like in C, the method can be simply called by its name followed by the parameters, enclosed in brackets:

```
robot.setval(0);
robot.mymethod(45,12);
@robot.setval(1);
```

5.9. *Other commands*

We give here some other important commands available with URBI:

The "load" command, followed by a file name, is copying the file content in the execution pile. This is useful to store complex sets of commands on the robot and execute them in one shot. It can also be seen as a simple library call command if there are function definitions in the included file.

The "sleep(n)" or "device.sleep(n)" is pausing the robot or the device for n milliseconds. It can be useful in a serial set of commands.

"quit", "reboot", "shutdown", "motoroff" are self explanatory.

Some commands give also a limited control capability from one connection to another, like "killall connection123" which empties the command stack of the connection "connection123" (this connection name is accessible from URBI). These commands are useful to terminate an infinite loop in a connection from another connection and to ensure that URBI is robust in environments where it is not possible to reboot the robot.

6. URBI library

Even if some interesting behaviors and programs can be executed with a simple telnet client, it is of course necessary to use a more sophisticated programming interface to use the advanced features of URBI. We have developed a C++ Library (liburbi) for linux C++ programmers. Other languages and other platforms will be soon available.

This library is designed to encapsulate an URBI connection. It handles the TCP connection with the URBI server, and the dispatching of messages it sends. The library is thread-safe and reentrant.

The main component of the library consists of a C++ class, UClient, and a few helpful functions.

6.1. *Opening a connection*

To connect to an URBI server, simply create a new instance of UClient, passing the name (or the address) of the server as the first parameter, and optionally the port as the second parameter (standard URBI port is 54000):

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

This will automatically launch the associated receiving thread, which will listen for incoming messages from the URBI server.

6.2. *Sending commands*

The "send" function can be used to send a command. It accepts a syntax similar to the printf function:

```
for (float val=0; val<=1; val+=0.05)
    client->send("neck.val = %f;sleep(%d);", val, sleeptime);
```

To send binary data, like a sound to play, the "sendBin" and "sendSound" functions are available (see the documentation for more implementation details). The "sendSound" function implements useful chunk based decomposition of the sound buffer, to smoothly stream the audio data to the robot.

6.3. *Receiving messages*

Most of the messages received from the URBI server are the result of a previously sent command. The mechanism of URBI tags enables to relate a command to its reply message(s): with each command is associated a tag, and this tag is repeated in the reply message. Liburbi handles the reception of those messages and implements two different ways to access them: synchronous or asynchronous.

6.3.1. *Synchronous reception*

This is the simplest and quickest way. A set of functions defined in the USyncClient subclass of UClient allows to synchronously access all devices available from URBI. However it is not advised to use them if performance is an issue, since the synchronous mechanisms used are a source of inefficiency.

To get the value of a device, the function syncGetDevice or syncGetNormalizedDevice are provided. The first parameter is the name of the device, the second is a double that is filled with the received value. The difference between the two functions is that syncGetDevice retrieves the value with a "val" command, whereas syncGetNormalizedDevice uses "valn".

```
double neckVal;
client->syncGetDevice("neck",neckVal);
```

The function syncGetImage synchronously gets an image. The function will send the appropriate command, and wait for the result, blocking the thread until the image is received.

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

The first parameter is the name of the camera device. The second is the buffer which will be filled with the image data. The third must be an integer variable equal to the size of the buffer. The function will set this variable to the size of the data. If the buffer is too small, the data will be truncated.

The fourth parameter is the format in which the image data will be stored in the buffer. Possible values are URBI_RGB for a raw RGB 24 bit per pixel image, URBI_PPM for a PPM file, URBI_YCbCr for raw YCbCr data, and URBI_JPEG for a jpeg-compressed file.

The fifth parameter can be either URBI_TRANSMIT_JPEG or URBI_TRANSMIT_YCbCR.
It specifies how the image will be transmitted between the robot and the client. Transmitting JPEG images increases the frame rate and should be used for better performances. Setting a different format for transmission and storage induces an implicit conversion handled by the URBI library. For instance, one can transmit a jpeg image and use a PPM buffer.

Finally the "width" and "height" parameters are filled with the width and height of the image.

The function syncGetSound can be used to get a sound sample of any length from the server:

```
client->syncGetSound(duration, buffer, bufferLength);
```

The first parameter is the duration requested, in milliseconds. It will be a multiple of 32 milliseconds, rounded if necessary. The second is a buffer. It will be filled with RAW 16bit 16 KHz PCM data (no wav header). The third parameter is the size of the buffer, and is filled on return with the number of bytes written to it.

6.3.2. *Asynchronous reception*

Although the synchronous functions exposed above are certainly the simplest way to receive information from the URBI server, they are far from the best solution, especially if performance is a concern. Consider the following code snippet:

```
int width, height;
while (true) {
   client->syncGetImage("camera",
                        myBuffer, myBufferSize,
                        URBI_RGB, URBI_TRANSMIT_JPEG,
                        width, height);
   handleImage(myBuffer,width, height);
}
```

With this code, a lot of time is lost since while the program is handling the image the server could be sending an other one: using synchronous functions wastes all the benefits of asynchronous processing.

Liburbi implements an asynchronous mechanism via URBI tags. This is done by registering a callback function that will be called each time a message marked with a given tag is received from the server. There are two different prototypes for the callback functions:

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

typedef UCallbackAction (*UCustomCallback)
    (void * callbackData,
     const UMessage &msg);
```

UMessage is a class that contains all information related to the received message. In particular, the "timestamp" and "tag" fields retrieve the corresponding parsed information from the message. The rest of the information is stored in structures depending on the type of the message, which can be accessed in the "type" field of the UMessage class. The detail of those structures are available in the liburbi documentation and includes buffers, float values, strings,...

"callbackData" is a custom value that is specified when the callback is registered and which will be given back through the callback function each time the callback is activated (typical use of this is to pass a pointer to a structure related to the work done in the callback).

The function must return URBI_CONTINUE, or URBI_REMOVE if the callback is to be unregistered.

Registering a callback associated with a tag is done by calling the "setCallback" function. The parameters are: the function, the tag and the custom value if the callback is of the "custom" type. As said before, the function will then be called each time a message with this tag is received. setCallback returns an identifier that can be used to remove the callback (deleteCallback) or to duplicate it (duplicateCallback).

```
UCallbackID id = client->setCallback(receiveImage, "imgtag");
client->send("imgtag: camera.val;");
```

6.4. *Usage tips*

It is a good design principle to open one URBI connection (one UClient or USyncClient object) for each data channel type: image, sound in, sound out, motor commands. This avoids locking the bandwidth for image transfer while requesting several motor device values at the same time.

The callback functions should return as fast as possible, since all callbacks are called by the same thread. If there are time-consuming operations, it is a good idea to spawn another thread.

7. URBI on Aibo ERS7

The current version of URBI is 0.3 and is available on http://urbi.sourceforge.net. The server is currently only designed for an Aibo ERS7 and the C++ Library works on linux. We have implemented a subset of the URBI language in this 0.3 version and we have tested the performances with real applications. The programs

described below are all available in the URBI distribution on sourceforge and are programmed using liburbi, the C++ Urbi Library. URBI, liburbi and the other programs are released under the General Public License and made freely available for the research community.

7.1. *General performance measures*

We conducted a series of measures to evaluate the performance of the current URBI server implementation on the ERS7 Aibo robot. We have done the measures in two situations: when the client is running offboard on a remote PC and when the client is running on the robot.

The latency observed is 1.5ms to 2ms with the remote client and a bandwidth of approximately 500Ko/s. With the onboard client, the latency is below 1ms and the bandwidth goes up tp 2Mo/s. The jpeg compression used by the server to send smaller images also induces a latency: 10ms for 208x160 images and jpeg90 compression and 12ms for jpeg75 compression. With resolution 104x80 and jpeg90, the computation time is 3ms. Motor commands are executed almost immediately when issued.

These results shows good performance for general applications. For extremely demanding low latency perception/action loops, the best option is to fully write the loop with URBI and let it run on the server. In that case the latency will be down to micro seconds.

7.2. *Urbirecord, Urbiplay and Urbimirror*

We measured the efficiency of URBI on Aibo ERS7 in practical situations using several URBI applications. The first application is Urbirecord, a recorder that stores on a file all the joints positions of the robot over time. It can record any set of moves made by the robot. As an example, we have recorded a walk sequence performed independently by an OPEN-R object that we had retrieved from another laboratory, without any knowledge of the underlying algorithm.

Urbiplay is an application capable of replaying a record file created with Urbirecord. Using it, we have successfully reproduced the walk sequence previously stored.

Urbimirror is simply piping Urbirecord from one robot to Urbiplay on another robot. The result is that the destination robot is mirroring in real time any move from the source robot. We observe a lag of approximatively 250ms but no notable speed of accuracy problems.

We have measured that the server is capable of handling 30 "val" setting or getting commands per joints and per seconds, with a response time of approximately 3ms. The response time is measured with the Urbiping program (it includes upstream and downstream ping latency).

7.3. *Ball Tracking Head*

The ball tracking head program is a classical example of a perception/action loop with Aibo and is provided with the OPEN-R set of examples. We have implemented a version of this program using only URBI commands and the C++ URBI Library on a remote computer. The result is visualy as responsive as the fully OPEN-R based version but is slightly jerky. The reason is that the moving commands and the action loop are handled only by the client which induces a slight lag due to the network transfer with the server. The solution that we will use with the fully functional version of URBI (version 1.0) will be to have the command loop running on the robot, as an URBI program, with the parameters of the movement being constantly fed in the robot by the client. The client must set the variable "ballx" and "bally" to the position of the ball in the visual field (-1 if no ball is present) and the following URBI program should run on the robot:

```
coefx = 0.001503;
coefy = 0.004708;
period = 2500;

# Ball Tracking
whenever (ballx != -1) {
   headTilt.val = headTilt.val + coefx * ballx &
   headPan.val  = headPan.val  + coefy * bally
};

# Moving the head around, looking for the ball
at (ballx == -1 ~ 3)
   scanning: loop {
     { headTilt.val = 90  sin:period |
       headTilt.val = -90 sin:period } &
     { headPan.val  = 90  sin:period |
       headPan.val  = -90 sin:period}
   }
onleave
   stop scanning;
```

The only thing the client does is to retrieve the images (30fps) and to send the commands "ballx = ..." and "bally = ..." to the server, ballx and bally containing the ball position. The expected reactivity lag is 6ms (using 104x80 images + jpeg90), according to previous measures we have done.

We plan to run the client side of this test program inside the robot itself using the OPENR declination of the liburbi. We expect the response time to drop even more in that case.

7.4. *Urbiimage*

The program urbiimage is simply retrieving the camera image from the robot and displays it in real time on the screen of the client (using the XShm extension for efficiency). When the image (resolution 208x160) is transferred in jpeg format with a compression factor up to 92, the frame rate is equal to the frame rate of the camera (30fps). With a jpeg compression factor of 80, one can open up to 7 different connections with a frame rate above 15fps.

The measured bandwidth of our wireless connection (802.11B) is approximatively 500KB/s and one image in jpeg 80 is approximatively 5KB, which give a maximum frame rate around 100fps. This is sufficient for most programming needs and it is at least as fast as the frame rate available on the robot, using OPEN-R directly.

To test the robustness of the current version of URBI, we have started several clients at the same time, making sure that the bandwidth was not saturated: three image clients, one client retrieving the stereo sound from the robot and playing it on the PC speakers, one client streaming music on the robot, and we have started the URBI based ball tracking program on top of this. We have not observed any slow down or performance problem on the robot.

8. Conclusion

We have presented the URBI language and the C++ URBI Library. The library (liburbi) is fully functional but the current version of URBI (0.3) implements only a subset of the URBI language. The final version (1.0) will be available soon as a beta for Aibo ERS2xx and ERS7. We are also working on a pioneer version of the URBI server for wheeled robots. The next step will be to develop a humanoid version of the server, as soon as a public API for a robot will be made available.

Practical applications and use of the current version of URBI have shown the validity of our approach both in terms of performance and ease of use. Most programs that we have presented here are less than 50 lines long and are easy to maintain, whereas this would certainly require a more important effort to develop with SDKs like OPEN-R. URBI is simple to learn and simple to use, it is a low level language but includes scripting and procedural features that makes it extendable in a portable way.

An effort will be made to develop java, lisp and C# versions of the URBI Library, running on linux and windows. A version of liburbi-C++ for OPEN-R has been developped to run clients directly on the Aibo robot. This ensures that a program written with the C++ URBI Library and using no platform dependent system calls can work indifferently offboard or onboard.

Future work includes also the development of a security mechanism to allow "superclients" to lock the access of some devices or the whole robot, in order to have a system usable in untrusted environments.

URBI is still in an early stage of development but is already used on a daily

basis in our lab, and other labs working with Aibo have started to use it. We hope the robotics and humanoids research community will find it useful and that this work will help and contribute to the development of these domains.

References

1. Open-R SDK for Aibo robots, *http://www.openr.aibo.com*, Sony Corporation.
2. Tekkotsu Development Framework for AIBO Robots: *http://www.tekkotsu.org*, Carnegie Mellon University.
3. B. P. Gerkey, R. T. Vaughan, K. Sty, A. Howard, G. S. Sukhatme, and M. J. Mataric. Most valuable player: A robot device server for distributed control. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, Wailea, Hawaii, Oct. 2001.
4. Paul S. Rosenbloom, John E. Laird, and Allen Newell, editors. The Soar Papers: Research on Integrated Intelligence. *MIT Press*, 1993.
5. Anderson, J. R., John, B. E., Just, M. A., Carpenter, P. A., Kieras, D. E., and Meyer, D. E. (1995). Production system models of complex cognition. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society* (pp. 9-12).
6. Open Robot Control Software: *http://www.orocos.org*, Orocos European Project (KULeuven/LAAS/KTH).

Jean-Christophe Baillie was a student of the French *École Polytechnique* and received his Ph.D. degree from the University of Paris VI, in 2002. From 2002 to 2003, he was a postdoctoral researcher at the Sony Computer Science Laboratory in Paris, and worked on vision algorithms for a new declination of the Talking Heads project. From 2003, he is Assistant Professor at the *École Nationale des Techniques Avancées* (ENSTA), in the Laboratory of Electronic and Computer Engineering. Jean-Christophe Baillie main research interests include Mind, Design and Architecture, Interaction, Learning. He is in charge of the Cognitive Robotics activity in the Laboratory of Electronic and Computer Engineering in ENSTA where he is currently working on developmental robotics.