FP7-ICT-2013-C618075 TWO!EARS Project

Deliverable 5.1: First Intermediate Report on Hardware/Software Integration and Robotics Test Bed

WP5 *

November 30, 2014

* The TWO!EARS project (http://www.twoears.eu) has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 618075.

Project acronym:	TWO!EARS
Project full title:	Reading the world with TWO!EARS

Work package:	WP5
Document number:	D5.1
Document title:	First Intermediate Report on Hardware/Software Integration and Robotics Test Bed
Version:	1

Delivery date:	30th November 2014
Actual publication date:	30th November 2014
Dissemination level:	Restricted
Nature:	Report

Editor:	Patrick Danès
Author(s):	Sylvain Argentieri, Benjamin Cohen-L'Hyver, Patrick Danès, Xavier Dollat, Thomas Forgue, Bruno Gas, Matthieu Herrb, Anthony Mallet, Jérôme Manhès, Ariel Podlubne
Reviewer(s):	Armin Kohlrausch

Contents

1 Executive summary

The computational framework of auditory perception and experience developed in TWO!EARS is realized as a *development* software system primarily based on *MATLAB*. The evaluation of the TWO!EARS model for different scenarios implies a *deployment* system, consisting in the interface of the development system with a robot. Work package WP5 aims at providing all the necessary ingredients to this deployment. To assess the active and exploratory features of the computational model and its ability to handle multimodality, robot platforms endowed with adequate mobility and multimodal sensor input must be designed. So, three different configurations of increasing complexity are planned: first, a binaural "head-on-a-stick" type system, that is, a head-and-torso simulator (HATS; used model: *KEMAR*) endowed with rotational movements; then, this same HATS equipped with stereovision; last, an implementation of this visio-auditive head on the *PR2* mobile robot so as to get translational degrees-of-freedom for long-range navigation. Besides, each test bed must be accompanied by a comprehensive real time software architecture. This architecture is typically composed of a low "functional" layer, where components—*e.g.*, perception, locomotion, navigation, etc.—must run concurrently under severe time and communication constraints, together with a high "cognitive" layer, where decisional processes take place at a higher level of abstraction. Extensive evaluations of each functional module are envisaged before its integration into the deployment system and evaluation in the various scenarios developed in WP1.

This deliverable documents the progress made during year 1 towards the definition of a physical simplified platform and its associated software. On the basis of the *GenoM3*[1] generator of modules for the *ROS*[2] robotics middleware, a robotics architecture is advocated which can host all the necessary functional components, written in C/C++ under *GNU/Linux*. A so-called *MATLAB bridge* was developed so as to connect a range of *GenoM3* functional modules with different *MATLAB* modules developed in WPs 2-4, thus linking the development and the deployment systems. A high-performance *audio stream server* was implemented for binaural acquisition and time-stamped audio streaming

1 *Gen*erator *o*f *M*odules *v3*, `https://git.openrobots.org/projects/genom3/wiki/Wiki` – This framework is one of the core software component distributed within the open-source collection developed at CNRS, as a result of two decades of research on real-time architectures for autonomous systems.

2 *R*obot *O*perating *S*ystem, `http://www.ros.org` – This open-source meta-operating system has been initiated by Willow Garage, and runs on the top of Linux.

from any board compatible with *ALSA* (Advanced Linux Sound Architecture). Last, a controllable azimuthal degree of freedom was added in the neck of a *KEMAR* HATS. It entails an original mechanical setup, an actuator, sensors, and a specific *GenoM3* module for homing, servocontrol and time-stamped streaming of proprioceptive and audio data. All these elements constitute a self-sufficient stable subset which enables the conduction of experiments spanning multiple work packages. The design of the embedded stereovision system is ongoing. The mounting of the head of the *KEMAR* HATS on a *PR2* mobile robot in the next project phases has been prepared by respective design work. In parallel, the visual rendering of robotics environments and scenes has been realized in the *MORSE*[3] simulator, complementing the physical robot implementations. Therein, *GenoM3/ROS* modules were developed to control a virtual HATS or PR2 and to stream/process images from virtual cameras.

3 *M*odular *O*pen*R*obots *S*imulation *E*ngine, `https://www.openrobots.org/wiki/morse/` - This Blender based versatile simulator enables realistic 3D simulation with one to tens of autonomous robots, and can be integrated with several robotics middlewares, including *ROS*.

2 Introduction

The main objective of WP5 is to integrate the whole set of modules from WPs 2–4 into a physical test bed enabling the global evaluation of the TWO!EARS computational framework against the two applications constituting WP6. This implies the development of three test beds: an anthropomorphic binaural head-and-torso simulator (HATS) endowed with an azimuth degree-of-freedom on its neck; this same system complemented with stereovision; the mounting of the binaural head of this HATS on a *PR2* robot so as to offer translation degrees-of-freedom and enable long-range motions. A comprehensive software modular architecture comes with this hardware. Its lower functional layer is composed of components which run concurrently under severe time constraints and communicate by control or data flow in real time. Via a specific bridge, it is connected with the cognitive layer realized in the development system. Therein, decisional processes take place, which handle symbolic data and are less subject to time-critical constraints. Extensive "atomic" evaluations of all developed parts must be performed—and their performance must be quantified when possible—so as to ensure their satisfactory behavior when case studies are addressed through the whole, integrated, deployment system.

2.1 Structure of the report and major achievements

WP5 is split into three tasks. However, for easier readability, the manuscript is not organized along these. Rather, it is organized along the main achievements over the first period, starting from software and going to hardware, with tests included all along the sections.

Chapter 3 describes the proposed **model-driven design of the robotics software architecture**, based on the *GenoM3* generator of modules and the underlying *ROS* middleware. This architecture enables the concurrent execution of *C/C++* "functional" modules under severe time and communication constraints, and an upper "cognitive/decisional" layer implemented in *MATLAB*.

Chapter 4 presents an **ergonomic and optimized** *MATLAB* **bridge**, which enables multiple *MATLAB* decisional processes to be client of any set of *GenoM3* functional modules.

Chapter 5 reports the achievement of a **high-performance** *GenoM3* **audio stream server** for acquisition and time-stamped streaming of binaural signals to any client in the functional or cognitive layer.

Chapter 6 concerns the **addition of a pan degree-of-freedom on the neck of the** *KEMAR* **head-and-torso-simulator**. This consists in: the mechanical setup; the mounting of actuator and sensors; the drivers for homing, control, and time-stamped proprioception; the encapsulation into a *GenoM3* module of all these functions, as well as time-stamped streaming of binaural signals.

Chapter 7 summarizes the work done towards *GenoM3/ROS* **modules for the control of virtual robots and video streaming from virtual cameras in the** *MORSE* **simulator**, so as to test the concepts of the cognitive layer from WP3-WP4 in synthetic environments in parallel to the development of the physical test bed.

Last, Chapter 8 summarizes the **ongoing work and short-term prospects**.

Appendix 9 concludes the report.

2.2 Structure of the report *vs* Tasks Decomposition

The first task of WP5, **Task 5.1 — Test bed: Robot platform and integrated audio/audiovisual sensors** was supposed to address four main issues during the period. These are:

Design of an anthropomorphic binaural head mounted on a pan-tilt unit This subtask has been completed with changes. Instead of mounting a head on a pan-tilt unit, the *KEMAR* HATS has been endowed a with an azimuthal degree-of-freedom. This is argued and explained in Chapter 6.

Binaural head with stereoscopic vision This is in progress, but introduces no delay in the project. This is overviewed in Chapter 8.

Data acquisition and processing, to compute high-quality low-level audio or visual cues This has been completed for the *KEMAR* HATS within the *GenoM3* architecture, as explained in Chapter 6. The application of the project also mentions the development of a "System-on-a-programmable-chip" based integrated audio/audiovisual sensor, embedding processor-based system on a FPGA combined with application-specific hardwired modules, for the binaural *PR2* robot. A decision on this point will be taken in year 2, with no induced delay on the project.

Less capable but more transportable HATS-based system The foreseen contribution from

RPI on this point has been shifted to years 2 and 3, since the planned matching grants did not arrive on time for the first year of TWO!EARS.

The second task of WP5, **Task 5.2 — Software architecture of the TWO!EARS framework** addresses the design of a modular software architecture underlying the implementation of the TWO!EARS computational framework, on the basis of a "functional" (low) and "decisional/cognitive" (high) layer, with adequates bridges in between. As mentioned before, a stable self-sufficient subset has been completed, which enables experiments transversal to several WPs. The relative developments constitute Chapters 3 to 6.

Last, as aforementioned, the work on **Task 5.3 — Modular tests and evaluations**, is not gathered into a separate section, but comes with the corresponding developments into the different chapters.

2.3 Licensed code

The TWO!EARS project follows the approach of reproducible research. So far, all the software components from this WP are under the BSD-3 Clause License, which can be viewed in Appendix 9.1. This licence allows to freely reuse the software material, provided that it appears in any redistribution form. The software introduced in this deliverable is available on the following TWO!EARS repositories:
genomix matlab bridge:
`https://dev.qu.tu-berlin.de/projects/twoears-matlab-genomix-bridge/repository`;
audio stream server:
`https://dev.qu.tu-berlin.de/projects/twoears-audio-stream-server/repository`;
KEMAR motorization:
`https://dev.qu.tu-berlin.de/projects/twoears-kemar-control-genom-module/repository`;
KEMAR motorization with *audio stream server*:
`https://dev.qu.tu-berlin.de/projects/twoears-kemar-genom-module/repository`;
control of virtual *KEMAR* in *MORSE*:
`https://dev.qu.tu-berlin.de/projects/twoears-morse-genom3-kemar/repository`;
control of virtual *PR2* in *MORSE*:
`https://dev.qu.tu-berlin.de/projects/twoears-morse-genom3-pr2/repository`.

2.4 An introduction to the robotics test beds

This section briefly introduces the platforms used in TWO!EARS (Figure 2.1).

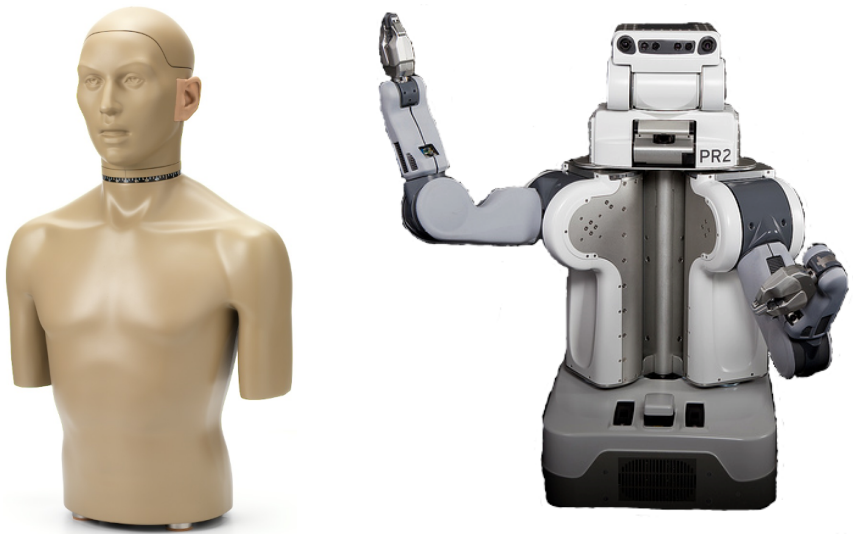

Figure 2.1: The KEMAR head-and-torso-simulator (HATS) and the *PR2* robot.

The *KEMAR* head-and-torso-simulator The *KEMAR* (*K*nowles *E*lectronics *M*anikin for *A*coustic *R*esearch) head-and-torso-simulator (HATS) is one of the most widespread acoustic simulators enabling human-like binaural acquisition. This anthropomorphic manikin is intented to reproduce the scattering and reflections undergone by acoustic waves on human upper bodies. It enables reproducible measurements, and is widely used to establish the performance of hearing aids and other electroacoustic devices, or to assess the quality of binaural recordings. It is based on worldwide average human male and female head and torso dimensions and meets the requirements of ANSI S3.36/ASA58-1985 and IEC 60959:1990.

The first version of this HATS dates back to 1972. The last versions (*e.g.*, the Type *45BB-2* used in the project) is built with a plastic composite that provides a more user friendly and ruggedized construction. It includes two ears, which can be selected from six different types, and come in a "Small" or "Large" size with two different shore hardnesses. The ears can be accurately positioned and easily dismantled for ear-canal exchange or calibration. The interior of the head can be easily accessed. The neck angle can be selected manually, locked at three predefined positions, and identified thanks to a visual marker.

In the framework of Two!Ears, it has been decided to endow the neck of the *KEMAR* HATS with a homemade controllable azimuth/pan degree-of-freedom. This way, experiments can be conducted which entail the actuation of this degree-of-freedom, such as exploratory movements, small motions for front-back disambiguation, etc. The visual

modality will be added in year 2, on the basis of a stereoscopic sensor.

The *PR2* mobile robot *PR2* is a mobile dual-arm manipulator from Willow Garage[1].

The *PR2* is a mobile robot built by Willow Garage to serve as a testbed for robotics research. It is endowed with two arms, with wrists and grippers; each of them has four, three and one degrees-of-freedom. The head can pan 350° and tilt 115°.

PR2 includes several types of sensors such as a Microsoft Kinect, a 5-Megapixel color camera as well as a Wide-Angle Color and a Narrow-Angle Monochrome camera on its head. It also embeds a Hokuyo UTM-30LX Laser Scanner on the base. On each forearm there is one Global Shutter Ethernet Camera and on each gripper there is a three-axis accelerometer and a fingertip pressure sensor array. The robot is endowed with two Quad-Core i7 Xeon processors, 24GB RAM, a removable 1.5TB hard drive and an internal 500GB hard drive. It is also equipped with an ethernet connection, wifi, a dedicated service access point and a bluetooth access point.

PR2 is undoubtedly the emblematic robot based on the *ROS* middleware. The open-sourcing of *ROS* code boosted the robotics research community to freely distribute modules. There is also a wide community exchanging information from the most basic things to the more advanced knowledge on the *PR2*, including troubleshooting. Many universities/research laboratories and known companies own a *PR2* for their research, such as *CNRS*, *UPMC*, Berkeley University, MIT, Samsung or Bosch to name a few. In total, there are 34 different institutions in more than 12 countries.

The head of the *KEMAR* HATS will be mounted on this mobile robot, so as to provide additional translational degrees of freedom for large-scale exploration.

1 `https://www.willowgarage.com/pages/pr2/overview`.

3 Overview of the robotics software architecture

An important objective of Two!Ears is the translation of the whole conceptual framework into a comprehensive robotics software architecture. This so-called "deployment" architecture is a necessary element to tackle the case studies of WP6. The present chapter first describes the needed step to bridge the gap between the Two!Ears conceptual framework and the deployment software at an abstract level, and then explains why and how the generator of functional modules *GenoM3* and the underlying middleware *ROS* can constitute a sound implementation framework.

3.1 From the Two!Ears conceptual framework to the deployment architecture

Though several approaches to robotics architectures exist, the benefits brought by a modular layered organization are widely acknowledged. From a robotics viewpoint, two basic layers can be exhibited in the Two!Ears deployment system. The *functional layer* is composed of modules which are subject to severe time constraints, for instance to achieve real time performance. These components must be able to communicate efficiently with each other, through control and data flows. They are in charge of sensorimotor functions, such as locomotion, proprioceptive or exteroceptive data acquisition and processing, obstacle avoidance, reactive navigation, localization, or even simultaneous localization and mapping (SLAM). As many components are in interaction with the environment, several local perception-action or perception-decision-action loops take place in this layer. Typical issues are components reusability, formal proofs of dependability and scalability.

Higher in the architecture, the *decisional/cognitive layer* hosts deliberation primitives. As argued in Ingrand and Ghallab (2015, in press), deliberation—which refers to purposeful, chosen or planned actions—is critical for robot autonomy against variable environments. Among the ingredients of deliberation in robotics, one can cite learning, goal reasoning, task planning, deliberate action, perception (which bottom-up as well as top-down) and monitoring. These abilities take place at a more abstract level, under lighter time constraints. An example of a general robotics architecture is shown on Figure 3.1.

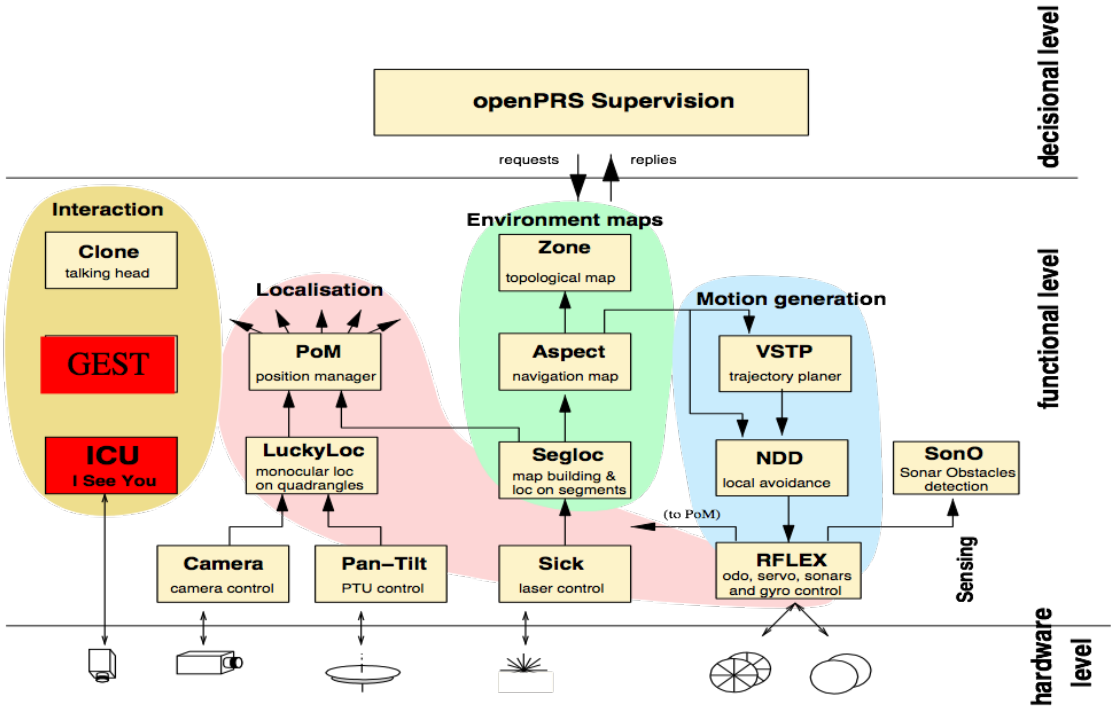

Figure 3.1: Example of a robotics architecture.

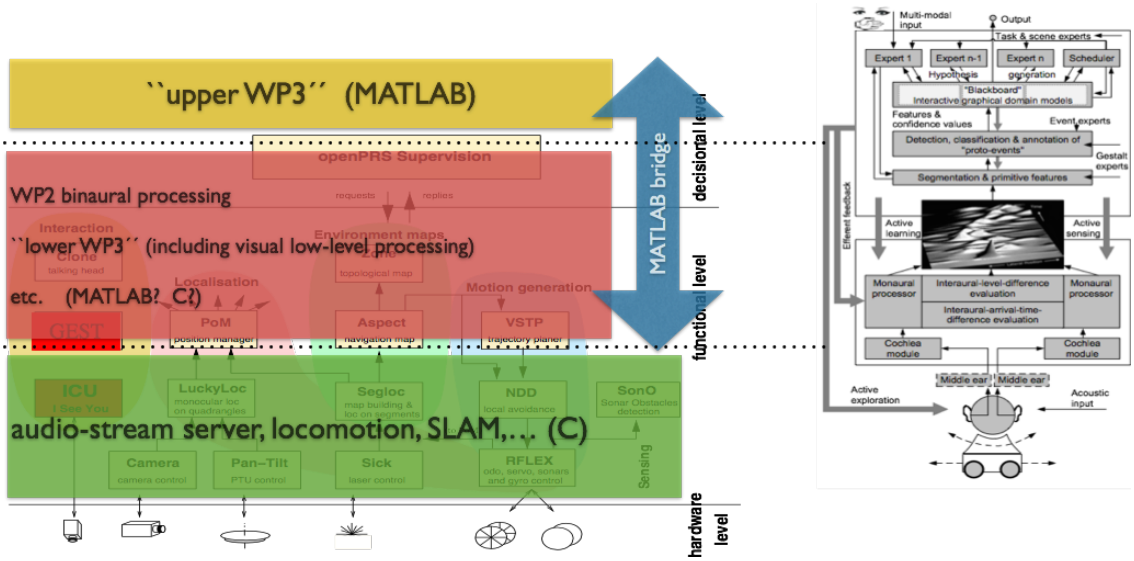

Figure 3.2: The Two!Ears conceptual framework (right) *vs* its implementation as a robotics architecture (left).

The TWO!EARS conceptual framework involves functions for robot locomotion, streaming of binaural signals, as well as SLAM. In view of the above, these functions come within the functional layer, and must be implemented into a high level language such as *C* or *C++*. Conversely, the cognitive part of WP3 straightly takes place within the deliberative layer. Quite uncommonly in comparison with robotics, the corresponding abilities are implemented in *MATLAB*. A *MATLAB bridge*, to be introduced later in this report, bridges these two extremal sets of primitives (Figure 3.2). Between them, several functions take place, such as the monaural and binaural processing stages from WP2 and the "lower" part of WP3 which includes low-level visual processing. In a first phase, these functions will constitute an intermediate layer. To run first integrated experiments, their currently available implementation will be used, namely *MATLAB* code for WP2 and *C* code for visual processing. Whether or not they will belong to the functional layer will be decided during year 2.

3.2 *ROS* and *GenoM3*

3.2.1 Basics

In the robotics community, *GNU/Linux* is by far the most common operating system, the *Ubuntu* distribution in particular. As aforementioned, components of the functional layer of a robotic software architecture have to deal with critical computing issues and timing constraints. Many middlewares are available off-the-shelf to run in between and take in charge their control and mutual communication/synchronization. Among them, *ROS*[1] from Willow Garage is undoubtedly the most widespread. *ROS* is open-source and runs on *GNU/Linux*. It benefits from a large community of developers, which gives access to a large choice of components in various domains. The *PR2* robot, which will constitute the most versatile test bed to assess the TWO!EARS conceptual architecture against case studies, runs *ROS*. For all these reasons, *ROS* has been selected. Sadly, some heavy constraints remain between the releases of *ROS*, the maturity of some basic related modules (*e.g.*, navigation, visual processing) and the underlying versions of *Ubuntu*, but they are not detailed here.

GenoM3 is one of the core softwares distributed within the *OpenRobots* open-source collection developed at CNRS. It comes as a result of two decades of research on real time architectures for autonomous systems[2] (Alami *et al.*, 1998)(Mallet *et al.*, 2010). It was decided to use *ROS* through *GenoM3*, because this significantly improves the encapsulation

1 *R*obot *O*perating *S*ystem, `http://www.ros.org`.

2 *Gen*erator *o*f *M*odules *v3*, `https://git.openrobots.org/projects/genom3/wiki/Wiki` (with included documentation).

of almost every kind of algorithm (periodic or aperiodic, synchronous or asynchronous, interruptible or not,...) into modules that can run several tasks in parallel and handle failures in a clean way. The *GenoM3* framework is middleware-independent and goes beyond by enforcing a clear internal organization of components, which helps preventing unsustainable design choices. The design of a module follows a model-driven approach in two steps:

1. First, the provided functionalities are described in a single specification file, the *.gen* file. This file defines the services provided by the module, the corresponding internal user-defined automata, as well as the module ports for data exchange with other components of the architecture.

2. On the basis of the *.gen* specification, *GenoM3* automatically generates the real time code related to the architecture of the module, and as well as a skeleton for the embedding of the code elements (or *codels*) of the algorithmic core. Once these codels are implemented (typically in *C* or *C++*) together with related external libraries, the module is ready to be built.

Thus, a *GenoM3* module can be generated for distinct middlewares without changing a line of code, as the *.gen* file and *codels* stay unmodified. Though *GenoM3* imposes a learning phase and strict guidelines to the user, its benefits are many: a clear specification of each component of the architecture in an associated file; a better organization of the code of each service thanks to the specification of its internal automaton; an easier reusability of the user algorithmic core; its decoupling of the underlying architecture; enhanced robustness thanks to the sharing of non-user code. Though not used in TWO!EARS, *GenoM3* can also be coupled with formal validation and verification tools[3].

3.2.2 Specification of a *GenoM3* module

The *.gen* file is written in a language called *dotgen*, close to *IDL*[4] (which can be used for syntax coloration in text editors). The main elements it can contain are listed below:

IDS stands for the Internal Data Structure of a module. It gathers the data shared by all its codels.

Services are the functionalities offered by the module. They are defined with one of the following keywords:

3 BIP/D-Finder for instance, `http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html`.

4 Interactive Data Language.

function A function is a simple, single-step service (*e.g.*, function returning the sum of two integers). It is coded into a single codel.

activity An activity is a more complex service defined as a finite state machine, each state of which is associated to a codel. There is at least a start codel which is the entering point of the state machine. If a stop codel is declared, then it is executed at the end of the activity, even if this happens prematurely (*e.g.*, if the activity is interrupted by another service, or if the module is killed while the activity is running). As an activity runs within a *task*, see below.

attribute An attribute is a service used to get or set parameters of the module, stored in the IDS.

Tasks Each module entails one or several tasks, each of which can be assimilated to a thread. Tasks can execute codels which are global to the module (*e.g.*, to be executed on a periodic basis), as well as codels which implement the associated activities. Tasks can have parameters such as period, priority, etc. Importantly, codels of a module are unbreakable from within *GenoM3*: if an interrupting event occurs, then the running codel is entirely executed by the task in charge of it before the activity stops.

Ports implement data communication from the module to the outside (*out port*), or the other way round (*in port*).

The *.gen* file can be completed with *.idl* files, commonly used to define data structures likely to be shared between *GenoM3* modules.

3.2.3 Blocking or non-blocking calls to services

A service can be called along two opposite ways:

Blocking A blocking service is waited to be done before the calling routine continues. This is the case of an activity which takes time to complete, and whose output is expected.

Nonblocking A nonblocking service keeps running while the calling routine requests other services, process other data, etc. Its output can be retrieved by its client once it is done, through its (scalar) ID.

3.2.4 Access of a running module

Prior to the execution of a *GenoM3* component, the corresponding middleware it was compiled for must be running. Once they are both launched, a client is needed in order to

access the services and ports provided by the module. There are different ways to get a client of *GenoM3* modules. These are described below.

The native and generic C client libraries

When a *GenoM3* component is built, C functions are generated in order to access its services and read its ports. These constitute the so-called *native C client library of the module*. On their basis, it is easy to program an executable client to the module.

GenoM3 also provides a *generic C client library*, which is made of more general functions so as to send requests to any module or to get information from any port of any module. Internally, data are encapsulated into *JSON* objects[5].

The *genomix* server and the *Tcl* client

The *genomix HTTP server* is a generic interface between one or several clients and any number of *GenoM3* components. Clients can access components by sending specific *HTTP GET* requests to *genomix*. The contents of these input requests from its clients is forwarded by *genomix* to its *GenoM3* server modules. Similarly, *genomix* forwards the replies from its *GenoM3* server modules to its clients. Data can also be relayed by *genomix* using *JSON* objects: from its clients to the *in ports* of its *GenoM3* server modules, as well as from the *out ports* of its *GenoM3* server modules to its clients[6].

A *Tcl*[7] client of *genomix* is part of the *GenoM3* distribution. It provides a *Tcl* command line interpreter as a client of *genomix*. In other words, *genomix* is a server to this *Tcl* interpreter and a client for the *GenoM3* components (Figure 3.3).

3.2.5 A toy example

Section 9.3.2 describes the specification and the implementation of a simple module that increments a counter on a regular time basis, and of another module that connects to the first one so as to display the value of the counter. This helps to understand the fundamentals of *GenoM3*, the *dotgen* language, and the *GenoM3* workflow, with features such as:

5 JavaScript Object Notation, `http://json.org/`

6 Data encapsulation into *JSON* objects is performed inside the clients and *GenoM3* server modules of *genomix*. The communication between *genomix* and its server modules uses internally (and transparently to the user) the aforementioned *generic C client library*.

7 Tool Command Language.

- declaring in and out ports and connecting them;
- declaring activities with codels defining a state-machine;
- declaring tasks with an initialization codel;
- using the module's internal memory to store data and define attributes;
- using validation codels to control the input of services.

3.3 Installation instructions for *ROS* and *GenoM3*

An easy procedure for the installation of the software supporting the TWO!EARS deployment architecture is provided in Appendix 9.2.

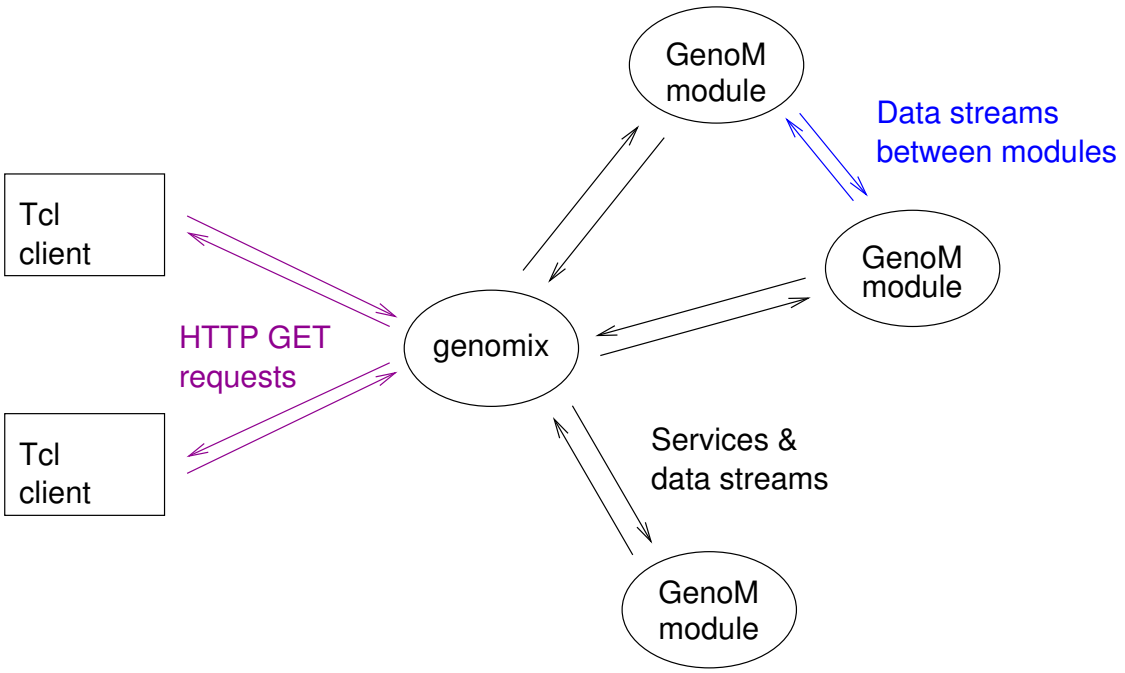

Figure 3.3: Example of an architecture using *genomix* with two *Tcl* clients.

4 The *MATLAB* bridge to *GenoM3* modules

4.1 Overview of a *MATLAB* client

Many aspects of the TWO!EARS architecture are implemented in *MATLAB*. This is the case for the decisional level, which takes autonomous decisions based on processed descriptors. On the other hand, robotic components such as the rotation control of the head and the acquisition of audio data from microphones are programmed in *GenoM3*, as they are part of the functional level, in charge of action and sensory input acquisition. The exact frontier between *MATLAB* and *GenoM3* will be fine tuned in the second year of the project.

Some functions can be implemented either in *GenoM3* or *MATLAB*. If they are coded in *GenoM3*, then *MATLAB* needs to get access to the descriptors they produce. If they are coded in *MATLAB*, then *MATLAB* must get the raw sensory data. In both cases, *MATLAB* must communicate with *GenoM3*. This is the reason why a significant effort has been spent on the development of a bridge between *MATLAB* and *GenoM3*.

The expected key features for this bridge can be summarized as follows.

- Any set of *GenoM3* modules must be reachable from *MATLAB*. This implies
  - the call of *GenoM3* modules services from *MATLAB*, either in a blocking or non-blocking way;
  - the access to ports of *GenoM3* modules from *MATLAB*, and the storage of the published data into *MATLAB* structures.
- Several *MATLAB* processes should possibly be clients of the same *GenoM3* module.
- *MATLAB* processes and *GenoM3* modules should be allowed to run on different computers, connected to a *TCP/IP* network.

4.2 Admissible solutions

Three solutions may be envisaged: the use of a native *ROS-MATLAB* bridge, by exploiting the fact that ultimately, in our specific context, *GenoM3* generates *ROS* code; the use of the *GenoM3 generic C client library* (Section 3.2.4); the access to *GenoM3* components from *MATLAB* through the *genomix* server (Section 3.2.4).

4.2.1 Use of a native *ROS-MATLAB* bridge

As the *ROS* middleware has been selected for the project, an option would be to rely on a native *ROS-MATLAB* bridge. The official *ROS I/O package*[1] exists for this purpose. However, its use would break the benefits brought by the aforementioned middleware independence of *GenoM3*. More importantly, this would require for the *MATLAB* user who wants to control a *GenoM3* module on the basis of its *.gen* file, to learn how this specification has been translated into *ROS* nodes with *ROS* topics, services and actions. Last, the *ROS I/O package* was tried at one point of the project to compare performances, but turned out to be somewhat instable and to provide poor flexibility.

4.2.2 Use of the *GenoM3 generic C client library*

The most efficient way to communicate with *GenoM3* modules is probably through the *GenoM3 generic C client library*, which is fast and very robust. On the basis of this library, a standalone C client can be developed and further encapsulated into *MATLAB MEX* files[2]. This option was considered for the toy example mentioned in Appendix 9.3.2, and its related standalone C client described in Appendix 9.4. This raised deep issues:

- *GenoM3* returns asynchronous events, for instance when services end. Handling them requires to get into the *MATLAB* event loop. This kind of real-time constraint is not officially featured in *MATLAB*. It would take a lot of effort (and probably a bit of "hacking") to come up with a solution, without any guarantee of success.
- The dynamic libraries related to any server module to be accessed have to be opened from within the client. This gets complicated if the server module and *MATLAB* run on separate computers.
- *MATLAB* uses its own set of *C* libraries, some of which are not compatible with those

1 `http://www.mathworks.com/hardware-support/robot-operating-system.html`
2 `http://www.mathworks.com/help/matlab/matlab_external/introducing-mex-files.html`

used by *ROS*. This reaches, in some sense, the limit of the middleware independence of *GenoM3*, in that the generated executable component must ultimately rely on libraries required by the middleware.

- Other unexpected behaviors were encountered at the *MEX* level, while underlying *C* standalone client was working flawlessly.

4.2.3 Use of the *genomix* server (chosen solution)

As outlined in Section 3.2.4, the generic *genomix* server enables the control of *GenoM3* modules by relaying *HTTP* requests from clients, the data transferred between *genomix* and its clients or servers being encapsulated into *JSON* objects. In the same way as the existing *Tcl* client enables the control of modules from a *Tcl* interpreter, a *MATLAB* client of *genomix* can be envisaged. This is indeed possible thanks to the *TCP/IP* support brought by the *MATLAB Instrument Control Toolbox*[3], which eases the sending of *HTTP GET* requests from a *MATLAB* client to *genomix*.

The following advantages and drawbacks appear.

+ *GenoM3* is used as it was meant to, and the middleware independance is kept.

+ The *Instrument Control Toolbox* offers tools for asynchronous communication, what solves the issue of handling *GenoM3* events (Section 4.3.2).

+ The dynamic libraries related to the *GenoM3* modules are handled by *genomix*, so *MATLAB* does not need to access them.

+ Only official *MATLAB* features are used, which reduces the risk on the project.

− *genomix* is an additional process between *MATLAB* and *GenoM3* modules. Although *genomix* is remarkably optimized, this brings a little overhead.

− All the data relayed by *genomix* between its clients and its servers are encoded into *JSON* objects. This is actually interesting for structuring these data. However, when it comes to raw binary data, such as audio streams published on the *out port* of a module, this raises two issues:

  1. The size of the data (in bytes) is increased when encapsulated into a *JSON* object. On the one hand, extra characters, such as brackets and quotes, are inserted. On the other hand, sending a number takes as many bytes as there are digits in its decimal encoding, as each digit is sent as an individual character.

3 `http://www.mathworks.com/products/instrument/supported/tcp-ip.html`

2. *JSON* objects need to be parsed so as to store them into *MATLAB* structures. This can take a significant time for big arrays.

In view of the above, this solution was chosen, and has led to the development of a so-called *genomix matlab bridge* (Figure 4.1). The detailed implementation follows.

4.3 Implementation of the *genomix matlab bridge*

4.3.1 *genomix* *HTTP GET* requests

When connecting to *genomix*, the client opens two sockets:

1. The first *socket* is used to start a session[4], with the request `get /session/start?s=`. This *socket* is then used to call services and read ports, and returns an *HTTP* content of type `application/json`. First *socket* will henceforth be referred to under the term *main socket*.

2. The second *socket* is used to receive asynchronous events sent by modules. Listening on this *socket* is made with the request `get /session/listen?s=...`, returning an *HTTP* content of type `text/event-stream`, which is the way to get events from an *HTML5* server. This is the only *GET* request sent on this *socket*, once at the beginning. This second *socket* will be referred to under the term *events socket*.

Table 4.1 lists the *GET* requests that the *genomix matlab bridge* uses to communicate with *genomix*.

4.3.2 *MATLAB* aspects of the solution

Using *TCP/IP* objects in *MATLAB*

The aforementioned *Instrument Control Toolbox* allows to create *TCP/IP* objects in *MATLAB*. The *genomix matlab bridge* creates two distinct *TCP/IP* objects, one per *socket*. On the *events socket*, the major issue was to know how to process the data as soon as it arrives, because it is sent asynchronously. In *MATLAB*, the `BytesAvailableFcn` field of a *TCP/IP* object allows to assign a callback function to be executed everytime new data arrives on the *socket*. The *main socket* does not have this feature as it works in

4 A session is a user defined context to which sent requests can be attached. cf. `https://git.openrobots.org/projects/genomix/gollum/protocol/sessions`

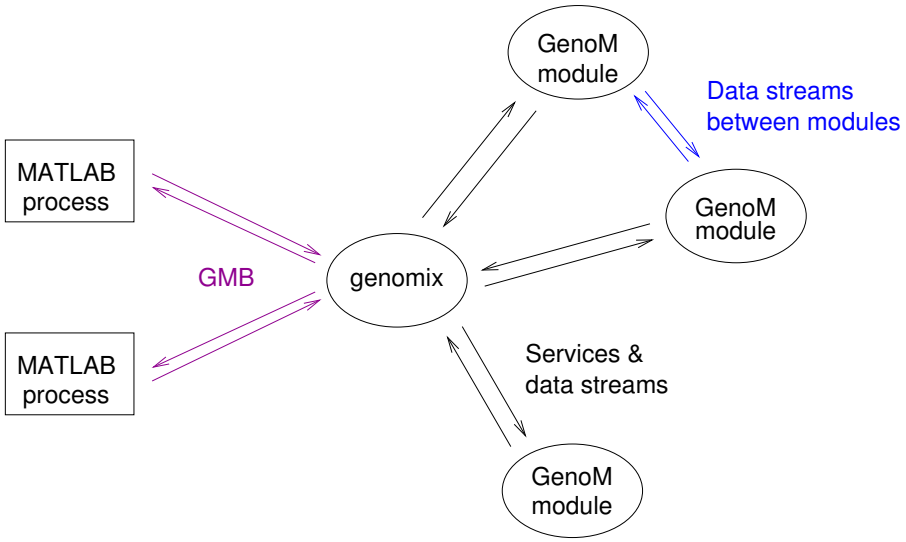

Figure 4.1: The *genomix matlab bridge* (*GMB*) in the software architecture.

Description	Request	Reply content
	main socket	
Start a session	`get /session/start?s=`	*sss*
Load a module	`get /load/`*mmm*`?s=`*sss*`&argv=`	*mmm*
Get module infos (cf. Section 4.3.3)	`get /module/`*mmm*`/info?s=`*sss*	*<JSON info>*
Call a service	`get /module/`*mmm*`/send/`*aaa*`?s=`*sss*`&input=`*iii*	*rrr*
Clean a request ID (cf. Section 9.3.1)	`get /module/`*mmm*`/clean/`*rrr*`?s=`*sss*	*<nothing>*
Read a port	`get /module/`*mmm*`/read/`*ppp*`?s=`*sss*	*<JSON data>*
	events socket	
Listen to events	`get /session/listen?s=`*sss*	*<events data>*

Variable	Meaning	Example
sss	session	`s1`
mmm	module	`demo`
aaa	service	`GotoPosition`
iii	service input in *JSON* format	`{"posRef":1}`
rrr	request ID	`0`
ppp	port	`Mobile`

Table 4.1: *GET* requests supported by *genomix*

a synchronous way: for every *GET* request sent on this *socket*, a reply from the server is waited for before continuing. The *main socket* has another property: as it is used to read ports, the size of received data can be very large, *e.g.*, when accessing a port which publishes raw audio data. Nevertheless, the `InputBufferSize` field of a *TCP/IP* object in *MATLAB* allows to set the maximum size of received messages on the *socket* to a value big enough for our needs.

Dealing with *JSON* objects in *MATLAB*

genomix encapsulates data into *JSON* objects. Therefore, a *JSON* parser in *MATLAB* is needed to convert the *JSON* objects into *MATLAB* structures. It was decided to use *jsonlab*[5], a free and open-source implementation of *JSON* encoding and decoding written in native *MATLAB*.

Although *jsonlab* was suitable for most of the manipulations on *JSON* objects required in the *genomix matlab bridge*, it showed a few efficiency limitations when trying to parse very large *JSON* objects from a port publishing raw audio data. This is why it was decided to write a custom parsing function just for the specific case of reading audio data from the corresponding *GenoM3* module (Chapter 5). The bridge would then allow the choice between *jsonlab* or this custom parsing function when reading ports. The custom function benefits from the fact that we have prior knowledge of how the data is organized in the *JSON* object and can write a parsing algorithm optimized for this particular case. It was eventually decided to encapsulate it into a *MEX* file, with good performance improvement (Section 5.3.4).

The `GenoM` global variable

The *genomix matlab bridge* needs to store some objects shared by its functions into memory, in particular the *TCP/IP* objects for the communication with *genomix* and the outputs of services that arrive on the *events socket*. These objects were chosen to be stored into a single global variable, called `GenoM`. The use of a global variable fits well this current case, as data received asynchronously on the *events socket* needs to be saved. Moreover, this turns out to be quite convenient for debugging purposes: by adding the global variable to the *MATLAB* workspace, it is easy to see the state of the bridge.

4.3.3 Releases

Two releases of the *genomix matlab bridge* were designed. The first one provided all the needed features:

- a function to connect to the *genomix* server and one to disconnect;
- a function to load any module and one to close any opened module;
- a function to call a module's service, in blocking or non-blocking mode;

5 Website: `http://iso2mesh.sourceforge.net/cgi-bin/index.cgi?jsonlab`

- a function to get the result of a non-blocking request;
- a function to read a module's port, with the possibility to provide an optional parsing function for efficiency.

In comparison to the *Tcl* client, the resulting design of the *MATLAB* client was missing a nice feature: automatic completion. When loading a module in the *Tcl* client, specific commands for the module are created. This is made possible thanks to the `get /module/mmm/info` request sent on the *main socket*: it returns a *JSON* object describing all the services and ports provided by the module, along with information about inputs and outputs of the services, and data types published on the ports. It was decided to develop a second release using this *GET* request so as to get the same kind of automatic completion in *MATLAB*.

Automatic completion in the *MATLAB* command window is possible for functions known by *MATLAB*. The idea was consequently to automatically generate *MATLAB* functions, based on the information returned by the *GET info* request.[6]

Following is an example using the `demo` module to show the improvement brought by the second release. The `demo` module, provided in the *GenoM3* distribution and intended to be a first example for new *GenoM3* developers, was employed for testing purposes when developing the *genomix matlab bridge*. It provides services for the control of a virtual robot moving forward or backward on a line, at a speed chosen by the user. In particular, a service named `GotoPosition` allows the user to move the robot to a position reference, given as input parameter of the service.

- In v1.0, a call to the `GotoPosition` service of the `demo` module with an input parameter `posRef` equal to 1, was made like this:

```
>> GMB_callModuleService('demo', 'GotoPosition', struct('posRef',1));
```

- In v2.0, with the corresponding generated function, it becomes:

```
>> GMB_demo_GotoPosition(1);
```

`GMB_demo_GotoPosition` is a native function automatically generated by the *genomix matlab bridge* when the `demo` module is loaded. As this function is then known by *MATLAB*, its automatic completion is enabled (Figure 4.2-left). An important remark can be made concerning the input parameter `posRef`: in v1.0, the user needed to provide the name of the input parameter, which required to have a look at the *.gen* file of the module to know its name. In v2.0, the name of the input parameter is retrieved thanks to the

6 Functions known by *MATLAB* are functions in the *MATLAB* path. It was then necessary to put the generated functions in a folder that is added to the *MATLAB* path by the bridge.

GET info request, and it is actually part of the prototype of the native function, so it also appears when typing the funcion in *MATLAB* (Figure 4.2-right).

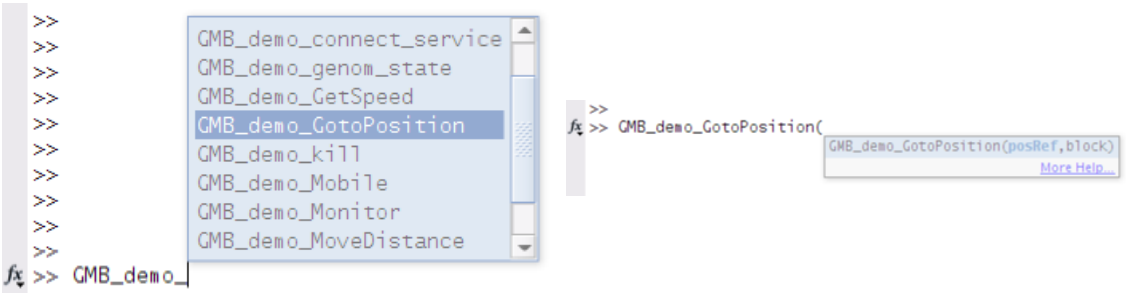

Figure 4.2: Left: Automatic completion in *MATLAB* with the `demo` module. Right: Suggestion of the input arguments in *MATLAB*

Figure 4.3 shows a parallel between the *Tcl* client and the *MATLAB genomix matlab bridge* v2.0 client we designed.

4.4 Timing concerns

One major concern of the *genomix matlab bridge*, during all its development process, has been related to timing. One of the major features originally intended with the bridge was to receive audio streams in real-time from a dedicated *GenoM3* module. Therefore, it had to be very efficient and minimize latency. Even though the final design for audio stream does not use the *genomix matlab bridge* (but another approach exposed in Section 5.4), communication latency was improved by the developers of *GenoM3* at *CNRS*. Two main updates of *GenoM3* provide the bridge with optimized timing performances. These are exposed below.

4.4.1 *JSON* serialization in *GenoM3*

When we started to work with *genomix*, our attempts to transfer large arrays of audio data revealed an issue: retrieving 1 second of audio data (2 channels at 44100Hz, with each sample encoded on 32 bits) was taking more than 1 minute. *JSON* serialization in *GenoM3* was improved to solve this issue. The serialization speed was reduced by a factor of 80[7].

7 For more information, see the commit message:
`https://git.openrobots.org/projects/genom3-ros/repository/revisions/a04e6142`

```
Tcl client                          | MATLAB client
**********                          | *************

# Connect to the genomix server and load the demo module.

eltclsh> genomix::connect           | >> GMB_genomixConnect('127.0.0.1');
genomix1                            |
eltclsh> genomix1 load demo         | >> GMB_loadModule('demo');
demo                                | Loading module 'demo'... Done.
                                    | Generating native functions... Done.


# Call the GotoPosition service with input 0.5, then call it in a non-blocking
# mode with input 1.

eltclsh> ::demo::GotoPosition 0.5   | >> GMB_demo_GotoPosition(0.5);
eltclsh> ::demo::GotoPosition 1 &   | >> GMB_demo_GotoPosition(1, 'nonblocking');
demo::0                             | GMB_demo_0


# Get the output of the non-blocking call. There are two possible results shown
# below. First result is if the service is still in progress when the function
# to get the output is called. Second result is if the service is completed.

eltclsh> ::demo::0                  | >> GMB_demo_0()
request demo::0 in progress (sent)  | ans = REQUEST_IN_PROGRESS
eltclsh> ::demo::0                  | >> GMB_demo_0()
                                    | ans = []


# Read the Mobile port of the demo module.

eltclsh> ::demo::Mobile             | >> GMB_demo_Mobile()
Mobile {position 1 speed 0}         | ans = position: 1
                                    |           speed: 0
```

Figure 4.3: Sketch of a session with the *Tcl* (left) and *genomix matlab bridge* (right) clients of the *GenoM3* `demo` module.

4.4.2 Reading topics in *ROS*

When using *GenoM3* to produce robotic components for the *ROS* middleware, the *ports* of modules in fact use *ROStopics*[8] to export or receive data: a given *out port* publishes its data on a *ROS* topic, and an *in port* connected to this *out port* suscribes to the same topic. Everytime a client of *genomix* requests for data published by an *out port*, *genomix* uses the *C* client to subscribe to the corresponding topic, and to retrieve the data. The subscribing process adds a little overhead. This point was improved as follows: instead of subscribing to the topic every time one reads the port, the *C* client only subcribes once when a first access to the port is attempted, and reuse the same subscription when one wants to access the port again. This improves the timing by about 100 ms on localhost.[9]

8 `http://wiki.ros.org/Topics`

9 For more information, see the commit message:
`https://git.openrobots.org/projects/genom3-ros/repository/revisions/2a2914f9`

5 A *GenoM3* module to stream binaural audio data

5.1 Overview of a binaural audio streaming module

The TWO!EARS project focuses most of its research on binaural audition. Sound is acquired through an interface that digitalizes the signals from the left and right microphones of the *KEMAR* head for further processing. Hence, the deployed software architecture of TWO!EARS must include a specific component, which captures the audio data from the sound interface and makes it available to other components of the *GenoM3/ROS* architecture. This *GenoM3* server module, in charge of streaming binaural data, will henceforth be termed *audio stream server*.

The used sound device is a *RME Babyface*. It runs in *Class Compliant* mode[1] for *GNU/Linux* compatibility. It can be accessed through *ALSA*[2] under *GNU/Linux*, and can be replaced by any other *ALSA*-compliant sound interface.

The required features for the *audio stream server* are the following:

- Services must be provided to start and stop the acquisition of binaural audio data from any *ALSA*-compliant sound interface.
- The *audio stream server* must stream binaural data through an *out port*. To limit the required bandwidth for data transfer, only a sliding window of the most recent captured data must be streamed. The module must provide the ability to select the length of this published window according to the frequency of data access by the clients so as to ensure no data loss inside them.
- Data time-stamping is needed so that the clients can keep track of what they receive, and can detect any data loss.

A client module must also be provided with the basic elements to receive the audio data

1 See *RME* documentation: `http://www.rme-audio.de/download/cc_mode_babyface_e.pdf`.
2 Advanced Linux Sound Architecture

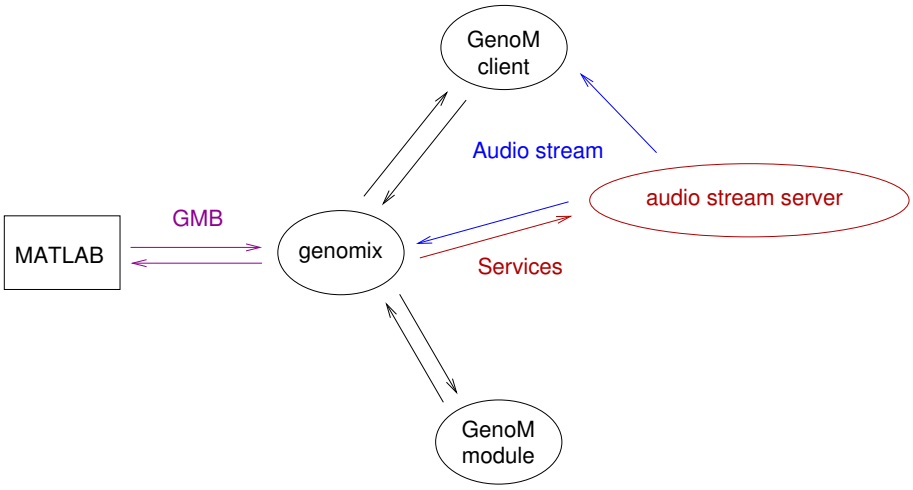

Figure 5.1: The *audio stream server* in the software architecture.

from the *audio stream server*. This client module must have an *in port* that can be connected to the *out port* of the *audio stream server*, and must propose services to retrieve the data and check their integrity. The *audio stream server* is also closely tied to the *genomix matlab bridge*, as one of the main goals is to get the audio data in *MATLAB*. Sections 5.3.4 and 5.4 will expose the accomplished work for optimizing the link between these two elements of the architecture.

5.2 Possible designs for the *out port* of the *audio stream server*

The *ALSA*-compliant sound interface internally delivers chunks of audio data. These must be published on the *out port* of the *audio stream server*, together with older data, typically up to a few past seconds. At least two designs of the *out port* of the *audio stream server* can be envisaged.

5.2.1 Linear design

An initial design of the *out port* can consist of a kind of *FIFO* stack: when a new chunk arrives from the *ALSA*-compliant sound interface, all the data currently on the port are shifted so as to make room for this new chunk. For instance, if the port contains `N` chunks, `1` being the oldest and `N` the newest, the following happens every time a new chunk is available (Figure 5.2):

- data in chunk `1` is deleted;

- data in chunk `2` is moved to chunk `1`;
- data in chunk `3` is moved to chunk `2`;
- ...
- data in chunk `N` is moved to chunk `N-1`;
- new chunk is copied in chunk `N`.

Importantly, if a client wants to read the port at a high frequency, only the last chunks of the port, constituting new data for the client, are relevant. But reading a port implies to read its whole content, so the client also gets previous chunks that were already internally saved from previous accesses. This constitutes a waste of time if the port is big and the client is only interested in the very last chunks.

5.2.2 Circular design

This problem can be circumvented through an alternative design. Instead of handling one large port, several small ports can be designed, each one publishing a single chunk of data. In other terms, instead of having `1` port of `N` chunks, `N` ports of `1` chunk can be proposed. The *audio stream server* then copies received chunks from the sound interface in a circular way (Figure 5.3):

- first arrived chunk is published on port `1`;
- next arrived chunk is published on port `2`;

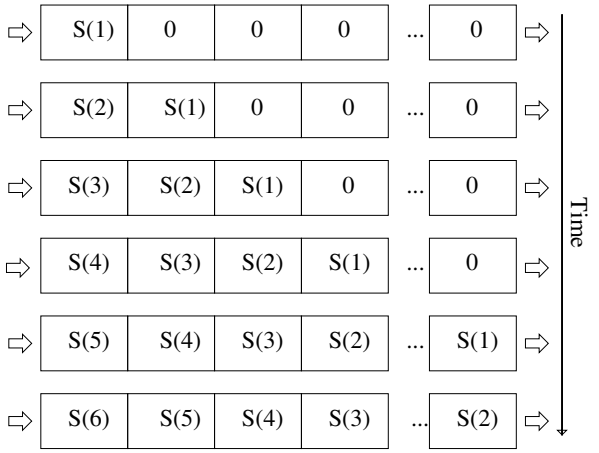

Figure 5.2: The linear design

- ...
- next arrived chunk is published on port `N`;
- next arrived chunk is published back on port `1`, overwriting the previous published data;
- etc.

An additional "index" port must display the absolute number of published chunks since the capture started, so that the client knows which ports need to be read. For instance, if the client already read `X` chunks and wants to get new data, then

- Accessing the index port, which displays the current index `Y` of the last published chunk, the client deduces that it must access chunks `X+1` to `Y`.
- As the chunks are published in a circular way on `N` ports, this is equivalent to accessing ports `(X+1 modulo N)` to `(Y modulo N)`.
- `Y-X > N` means that the client waited too long and that some needed chunks have been lost due to overwriting.

In this circular design, the client only reads the data it needs, which is an advantage compared to the linear design. This solution was implemented, but faced technical limitations: it turns out that for the *MATLAB* client, reading a port takes at least few hundreds milliseconds, no matter the size of the port. It is hence wiser to limit the number of reads, which leads back to the linear design. A solution to this issue is detailed in Section 5.3.3.

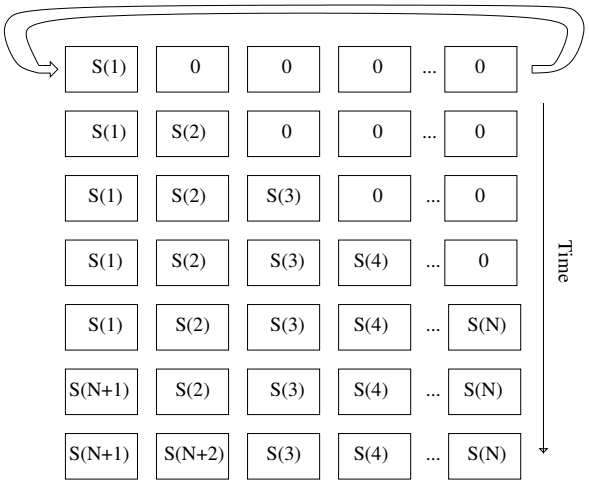

Figure 5.3: The circular design

5.3 Implementation aspects

5.3.1 Digital audio notions

The *audio stream server* deals with common digital audio notions. As it is important to have precise terms for defining these notions, some vocabulary is hereafter recalled to avoid any confusion:

capture This is the state of a sound device in charge of the acquisition of analog sound signals from microphones and of their analog-to-digital conversion. The other way is called *playback* and is outside the context of the *audio stream server*.

samples, frames and channels Two *channels* are involved in stereo audio: "left" is always considered as first, and "right" as second. A *sample* is a digital value encoding the signal of one channel at one point in time. A *frame* is a collection of samples, one from each channel, at one point in time.

ring buffer and transfer chunk Frames captured by the sound device are stored in a *ring buffer*, managed by *ALSA*. When an application, such as the *audio stream server*, wants to access these data, it regularly asks *ALSA* for little chunks of frames from the ring buffer. These are called *transfer chunks* and have a fixed size. Several applications can access the ring buffer concurrently. *ALSA* maintains for each application a pointer in the ring buffer to know which frames have not yet been read by the corresponding application. When one application asks for new data, the chunk corresponding to the next available frames for this application is sent.

overrun When the application does not read the *ALSA* ring buffer often enough, the buffer gets filled with unread data. The capture starts to overwrite on frames that have not been read yet, and are therefore lost by the application. This is called an *overrun*.

{non-}interleaved data There are two possible configurations for storing audio data in memory. Data are *interleaved* when frames are put one after each other, and they are *non-interleaved* when channels are separated.

```
Interleaved: data = {left[0], right[0], left[1], right[1], ...}
Non-interleaved: data.left = {left[0], left[1], ...} and
                 data.right = {right[0], right[1], ...}
```

5.3.2 *ALSA* and *GenoM3* notable features

The *audio stream server* uses specific *GenoM3* and *ALSA* features, some of which are commented below:

- Some *audio stream server* codels rely on the *ALSA API*[3] to prepare the sound interface to the capture and the retrieve of frames. The *RME Babyface* only supports one format to encode samples, namely signed integers encoded on 32 bits. Samples in the *ALSA* ring buffer are interleaved. The channels are separated prior to be published in non-interleaved format on the *audio stream server*'s *out port*.
- The *out port* of the *audio stream server* publishes, along with the audio data, an index of the last published chunk. This appears to be the most convenient way to keep track of time. For further needs in the project, it is also possible to get a high-resolution timestamp of the last published frame. This is made possible by a function of the *ALSA API* that returns the starting date of the capture. By combining it with the sample rate, the timestamp for any frame can easily be computed.

5.3.3 Synchronous and asynchronous releases

Two main versions of the *audio stream server* have been released: a first synchronous server fulfilling the required specifications, and a second asynchronous server with improved features.

The synchronous release

In this version of the module, the task dedicated to capture is synchronous. Its period depends on the size of the transfer chunks read from the *ALSA* ring buffer. For instance, if the aim is to read chunks of 2205 frames at a sample rate of 44100Hz (*i.e.*, exactly 50 ms of signal), then the task period has to be less or equal to 50 ms. If the task period is greater than 50 ms, then the ring buffer is accessed by the *audio stream server* at a lower rate than is updated by the audio interface, what eventually leads to an overrun.

A given number of frames for the transfer chunk does not necessarily correspond to a round number of milliseconds, as expected by *GenoM3*. In this case, a slightly shorter task period must be selected. As the task reads the ring buffer faster than it is filled up, there is no risk of overrun. However, there comes a time when the buffer does not contain enough data to build a complete transfer chunk. At this time, the *audio stream server* must simply wait a bit longer so that enough data is available in the ring buffer. As transitions between codels happen at multiples of the task period, the *audio stream server* has to wait for the next clock signal before reading the buffer again. Consequently, from time to time, retrieving the data takes two periods instead of one, so that *ALSA* has enough time to fill the ring buffer.

3 See the *ALSA C* library reference: `http://www.alsa-project.org/alsa-doc/alsa-lib/`.

The synchronous design has a big drawback: as the period task is hardcoded, the transfer chunk size and the sample rate must be hardcoded too. The transfer chunk time has been set to 50 ms with a sample rate of 44100 Hz. Three ports have been proposed following the linear design exposed in Section 5.2:

- one port of 500 ms (10 chunks of 50 ms);
- one port of 1000 ms (20 chunks of 50 ms);
- one port of 4000 ms (80 chunks of 50 ms).

To change the selected values (transfer chunk time, sample rate and port sizes), the module has to be recompiled.

The asynchronous release

The best design for a sound capture application is to implement an asynchronous loop. Instead of reading the ring buffer at a fixed time period, this approach proposes to wait until an event is raised by *ALSA*, indicating that there are enough frames in the ring buffer to constitute a new readable transfer chunk. The idea is to stay blocked in a function of the *ALSA API* (`snd_pcm_wait()`) without CPU consumption, until this internal event occurs. *GenoM3* offers a way to program this design: the task of the *audio stream server* has no period, and the codel containing the polling function is declared as "*asynchronous*" to let *GenoM3* know that it contains such a blocking function.

The task of the *audio stream server* being aperiodic, the sample rate and transfer chunk size are no longer hardcoded and can be set at runtime. Concerning the ports, they have to be rethought because the size of a chunk can now vary. *GenoM3* also offers a way to declare unbounded arrays of values, through the notion of "*sequences*". This brings another nice feature: there is only one port, which size can be set at runtime. For instance, a client can choose a size that best fits its needs. To sum up, here is the final prototype of the service for capturing sound, declared in the *.gen* file of the *audio stream server*:

```
activity StartCapture(
 in string device = ``hw:1,0'' : "Name of the sound device",
 in unsigned long transfer_rate = 44100 : "Sample rate in Hz",
 in unsigned long chunk_time = 50 : "Size of transfer chunks in milliseconds",
 in unsigned long Port_chunks = 20 : "Size of the Port in number of chunks")
```

For each parameter, one can read from left to right:

- its direction (`in` or `out`, `in` here because all parameters are inputs);

- its type (*e.g.*, `string`);
- its name (*e.g.*, `device`);
- its default value (*e.g.*, `''hw:1,0''`);
- its documentation (*e.g.*, `''Name of the sound device''`).

5.3.4 Timing results

The *audio stream server* was tested from within *MATLAB* through the *genomix matlab bridge*. The aim was to connect to it and retrieve the data published on its *out port*. Following the feature mentioned in Section 4.3.2, a custom parsing function was developed for the *audio stream server* so as to read the contents of its port efficiently from the *genomix matlab bridge*.

Table 5.1 reports the comparison of timing results with different parsing functions. Times have been measured for 3 different port sizes, and averaged across 10 operations. These results must be interpreted relatively to each other, as their absolute values depend on the power of the CPU running the test.

Port size	Parsing with *jsonlab*	Custom parsing in *MATLAB*	Custom parsing in *MEX* file
500 ms	39.7 ms	20.2 ms	2.1 ms
1000 ms	75.1 ms	38.7 ms	4.4 ms
4000 ms	293.5 ms	153.8 ms	19.3 ms

Table 5.1: Example of timing results for parsing data in *MATLAB*

5.4 Audio stream through a *dedicated socket*

5.4.1 Improving latency with a new approach

When clients of the *audio stream server*, such as *MATLAB*, retrieve the sensed binaural signals, the latency should be as short as possible. As previously exposed, the use of the *genomix matlab bridge* for this purpose shows notable drawbacks, namely the overhead brought by *genomix* and the fact that data are sent as *JSON* objects (which are bigger than binary data and need an extra parsing step, as explained in Section 4.2.3).

A new approach is proposed in order to minimize latency. It consists of a *dedicated socket* for direct *TCP/IP* communication between *MATLAB* and the *audio stream server*, bypassing *GenoM3*, *genomix* and the *genomix matlab bridge* (Figure 5.4). This solution allows data to be sent in a binary format, without *JSON* formating. The design of the *dedicated socket* is detailed below.

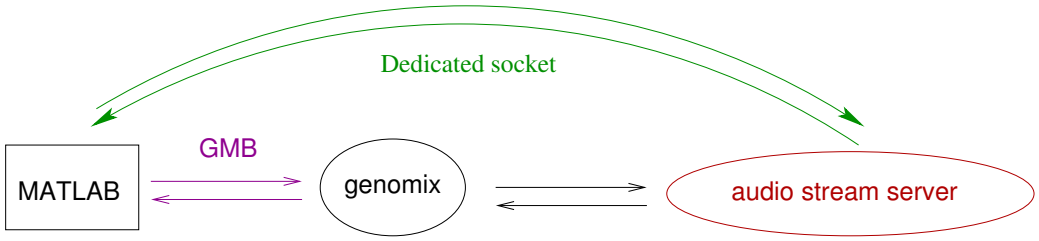

Figure 5.4: The dedicated socket in the software architecture

5.4.2 Communication aspects

Communication on the *audio stream server* side

On the server side, communication over a *dedicated socket* is made possible by embedding *C* code for socket communication directly into codels of the *audio stream server*. A new task is hence declared in the *GenoM3* module, with associated activities for allowing incoming connections or end the communication.

Implementation is done by using functions from the Internet Protocol Family library for inter-process communication (`sys/socket.h`). Several parameters have to be set before opening the connection. Main parameters include:

Domain indicating the used protocol for the communication. It is set to `AF_INET`, for *IPv4 Internet Protocol*.

Type specifying the communication semantics. It is set to `SOCK_STREAM`, which provides sequenced, reliable, two-way, connection-based byte streams.

Blocking type being set to `BLOCKING_SOCKET`.

Port number being the port used to listen to incoming connections. Port `8081` was chosen for this application.

Once the *socket* is opened, the module's task in charge of the communication starts listening

to incoming connections from clients in an infinite loop. The task asynchronously waits[4] for new events on the *socket*, using the standard `poll(2)` function on a set of file descriptors, one for each client. Three kinds of events can occur:

- A new client wants to establish the connection. It is added to the set of file descriptors.
- An existing client asks for audio data. Data are sent back to the client.
- An existing client closes the connection. It is removed from the set of file descriptors.

Data are sent in a binary format. As each sample is encoded on 32 bits, it takes exactly 4 bytes to send one sample.

Communication on *MATLAB* side

To ensure the communication over the *dedicated socket* on *MATLAB* side, the *Instrument Control Toolbox* providing *TCP/IP* support in *MATLAB* is again used[5].

Three main steps are designed in *MATLAB*:

1. A *TCP/IP* object is first created with a function called `dedicatedSocketOpen`, and the connection is established on port `8081`. Data will be received in an input buffer, the size of which is set thanks to the `InputBufferSize` field of the *TCP/IP* object.
2. Then, the `dedicatedSocketRead` function enables the client to request audio data. Once the request is sent, the client waits until the input buffer is filled with incoming data, which takes a few milliseconds. This is done by checking the value of the `BytesAvailable` field of the *TCP/IP* object, indicating the amount of bytes ready to be read from the buffer. This function should be regularly called by the client, in order to keep receiving newest data.
3. When the client wants to end the communication, it calls the `dedicatedSocketClose` function, deleting the used *TCP/IP* object.

As explained above, the server sends audio samples on the *dedicated socket* in a binary format, with 4 bytes per sample. In *MATLAB*, bytes are received in a long array and need to be structured: the first four bytes of the array will make the first sample, the next four bytes the second one, etc. This structuring step is only a matter of memory management, and is coded in a *MEX* function for timing efficiency. Eventually, the output of the `dedicatedSocketRead` function is a `Nx2` matrix, where the first column contains all

4 The codel implementing the loop is therefore declared as *asynchronous*.
5 As was the case for the *genomix matlab bridge* (Section 4.3.2).

left samples, the second column all right samples, and where N is the amount of frames that the server sent.

Left[1] Right[1]
Left[2] Right[2]
. .
. .
. .
Left[N] Right[N]

5.4.3 Complying with the *audio stream server* releases

As presented in Section 5.3.3, there were two main releases of the *audio stream server*. The *dedicated socket* also had two versions complying with the releases.

First release

In the first release, the *audio stream server* was publishing audio data on three ports of different sizes (500 ms, 1000 ms and 4000 ms). Thus, the *MATLAB* client could choose which one of those three sizes would be used for sending the audio stream over the *dedicated socket*. In his request, the client was specifying the chosen size.

Second release

In the current second release, there is only one port, the size of which may be chosen by the user. In this version, when the client requests new data, the first bytes sent back by the server indicate the size of data to be read, in order to let the client know how many bytes it must expect.

The current version of the *dedicated socket* even adds a new feature: when requesting new data, the client also indicates to the server the index of the last chunk it has read. This enables the server only to send back new data, and not the whole content of the port, hence keeping the spirit of what was intended with the circular design for the port (Section 5.2.2). This last point is another major improvement in the minimization of audio stream latency.

5.4.4 Timing results

A significant improvement has been reached by adopting the *dedicated socket* approach. In the context of the first release, timing bench tests were set up in order to measure the efficiency of the *dedicated socket* compared to the use of the *genomix matlab bridge* for audio stream. Timing results on localhost, averaged on a hundred tries, are presented in Figures 5.5 for the *genomix matlab bridge* and 5.6 for the *dedicated socket*.

Two main stages are involved in the communication:

1. Sending the request and reading the answer from the server.
    - In Figure 5.5 (with the *genomix matlab bridge*), the dark blue curve shows the time for both request and reading steps.
    - In Figure 5.6 (with the *dedicated socket*), the dark blue curve corresponds to the request and the green curve to the reading step.
2. Converting the answer into a *MATLAB* structure (which consists in parsing a *JSON* object for the *genomix matlab bridge* and binary data for the *dedicated socket*).
    - On Figure 5.5 (with the *genomix matlab bridge*), the red curve shows the time needed for parsing with *jsonlab* (Section 4.3.2) and the green curve corresponds to a custom parsing function (Section 4.3.2).
    - On Figure 5.6 (with the *dedicated socket*), the red curve shows the time for structuring data.

The total time for receiving data is shown with the light blue curve on both figures.

Table 5.2 shows timing results for the port with 4000 ms of audio data. It can be stated that using the *dedicated socket* reduces the latency by a factor close to 3.

Version	Request & read [ms]	Structure or parsing [ms]	Total [ms]
genomix	370	80	450
dedicated socket	157	3	160

Table 5.2: Times of each stage for reading `Port4000`

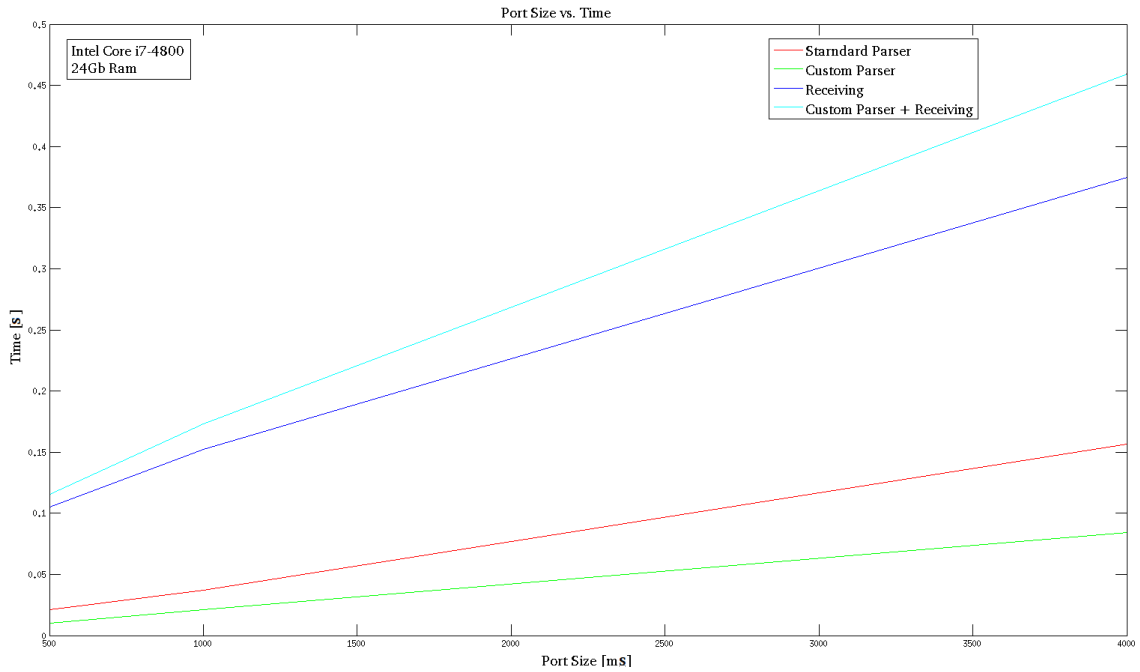

Figure 5.5: Time to get audio data from the *audio stream server* with the *genomix matlab bridge vs* port size

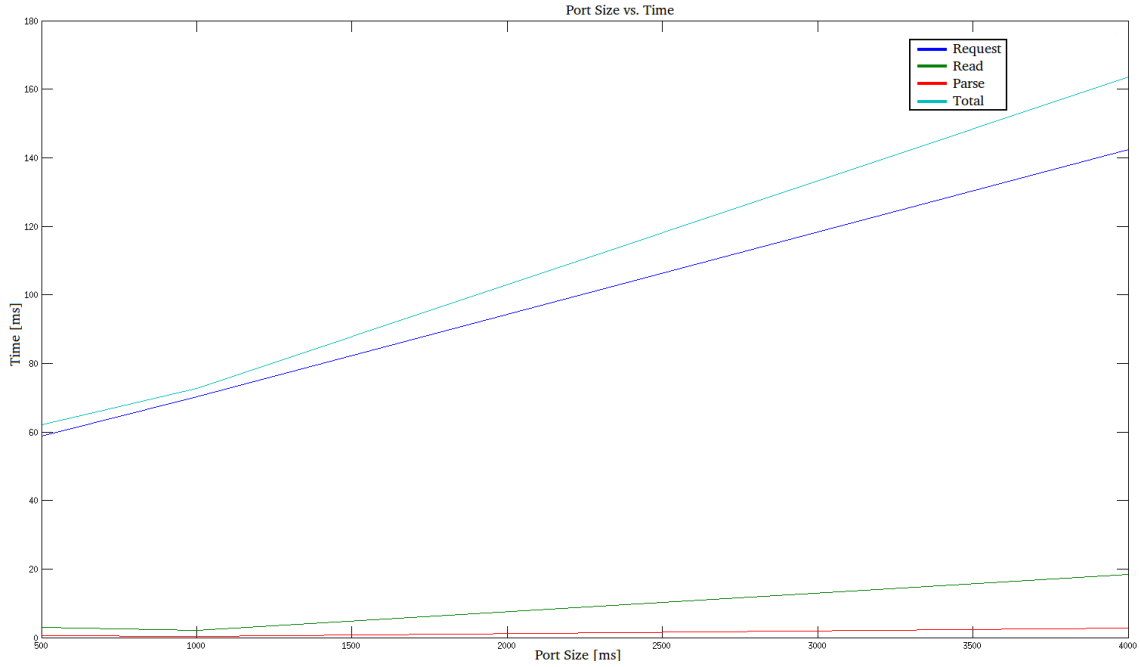

Figure 5.6: Time to get audio data from the *audio stream server* with the *dedicated socket vs* port size

6 The *KEMAR* HATS with controllable azimuthal degree-of-freedom and audio stream server

In the framework of Task 5.1 of Two!Ears, it was planned to mount an anthropomorphic binaural head on a pan-tilt unit, so as to get rotational degrees-of-freedom. Soon after the beginning of the project, the consortium decided to start from a complete head-and-torso-simulator (HATS) instead of restricting to a binaural head. The *KEMAR* HATS was then selected, in view of its widespread dissemination and because several instances were already available within the consortium. It was agreed to endow the neck of this HATS with an azimuthal degree-of-freedom. One major benefit of this motorized test bed is that it can still mimic acoustic waves scattering in humans even when the head is in motion. The tilt degree-of-freedom is no longer considered, because it does not contribute sufficiently to active motion, especially considering the effort which would be required to modify the *KEMAR* HATS accordingly. The targeted design then differs from the initially planned work, but is more complex, as mechanical parts have to be mounted inside the HATS: the servomotor to control the head azimuth, the limit sensors required to constrain the movement of the head, etc. During year 2, the head of the *KEMAR* HATS will be mounted on the *PR2* robot as committed in the project application, so as to get additional translational degrees-of-freedom for wider range of motion.

This chapter summarizes the work conducted in this respect, from the mechanical, control and software viewpoints. It is also shown why and how the *audio stream server* was merged with this hardware.

6.1 Characteristics of the genuine *KEMAR* head and of the used sensor supply module

The anthropomorphic *KEMAR* HATS mentioned in this report is a *Type 45BB-2* model, shown in Figure 6.1. It is fitted with "Large" European-like ears[1]. Its main features are summarized below.

Figure 6.1: *G.R.A.S. 45BB-2 KEMAR* HATS

1 `http://www.campbell-associates.co.uk/products/Gras/productdata/KEMAR-Manikin-Type-45BA.pdf`, see also `http://www.ee.bgu.ac.il/~acl/Equip/KEMAR.pdf` and `http://www.gras.dk/45bb-2.html`.

6.1.1 Microphones

Two *G.R.A.S Type 26CS* microphones[2] are placed inside the ears. This type of microphone is composed of a small ceramic thick-film substrate with a very high input impedance. Associated to each of them is the amplifier. This element has three functions. First, it receives and extracts the current coming from the connector to supply the microphone. Second, it injects and mixes the acquired audio signal into this same connector. Finally, a guard ring guaranties a shielding protection to minimize the influence of stray or parasitic capacitance and microphonic interference. Each amplifier has an integrated Microdot output connector, which may imply a Microdot-to-BNC cable to connect it to the audio acquisition system. The overall specifications are reported on Table 6.1.

6.1.2 The *IEPE Supply Module M28*

IEPE stands for "Integrated Electronics Piezo Electric". It is defined in the IEEE 1451.1 standard for the output of piezoelectrics transducers or microphones. The aim of this protocol is to provide a clean power supply for sensors placed far from the amplifier. Indeed, standard voltage supplies have the drawback to be sensitive to electrical noise when they travel along cables. Moreover, long cables attenuate the power so that it becomes impossible to set an accurate voltage. This problem is solved with current source supplies. Current is not influenced by electrical fields and feedback guarantees a selected reference current level.

2 `http://www.gras.dk/26cs.html`.

Figure 6.2: *G.R.A.S Type 26CS* Microphone

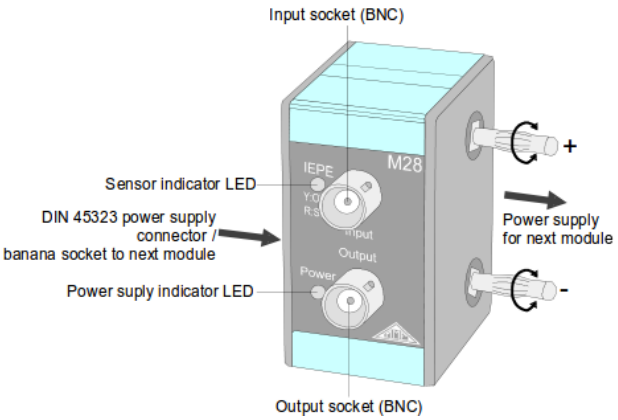

Figure 6.3: *IEPE Supply Module M28*

Specification	Value	Unit
Frequency Range	2.5 to 200 k	Hz
Slew Rate	20	V/μs
Input Impedance	20 // 0.4	GΩ // pF
Output Impedance	<50	Ω
Output Voltage Swing, min @ 24-28 V CPP voltage supply	8	Vp
Noise (A-Weighted) max	2.5	μV
Noise (A-Weighted) typical	1.5	μV
Noise (Linear 20Hz - 20kHz) max	6	μV
Noise (Linear 20Hz - 20kHz) typical	3.5	μV
Gain	-0.45	dB
Power Supply (Constant Current Power)	2 to 20 (typ. 4)	mA
DC bias voltage typical	12	V
Weight	3.0	g

Table 6.1: Specifications of *G.R.A.S Type 26CS* microphones

Each *26CS* microphone is connected on an *M28* module[3]. This device has multiple functions:

1. Generation of a 4 mA constant current for the microphone supply.
2. Injection of the current into the cable and combination with the sound signal.
3. Extraction of the sound signal coming from the microphone out of the bias current.
4. Amplification, with a defined gain, of the sound signal and band-pass filtering of this signal.

The circuit diagram is shown in Figure 6.4. The *IEPE* module injects a constant current I_{const} into the signal cable of the microphone. The output signal from the microphone may oscillate around the bias voltage. Therefore, the de-coupling capacitor C_C keeps DC components away from the output of the *M28* so that the instrument, in our case an *RME Babyface*, is free from DC components. The minimum output voltage is the saturation voltage of the integrated electronic (about 1 V). The maximum value is limited by the supply voltage of the constant current source (U_S=24 VDC with the *M28*).

Each IEPE Supply Module M28 contains the electronic circuit to supply one microphone. For multichannel applications, such as our binaural audition case, additional *M28* modules can be plugged into one another by means of screwed in banana plugs at the side wall. These plugs connect the power supply voltage to all modules.

3 `http://www.mmf.de/manual/m28mane.pdf`.

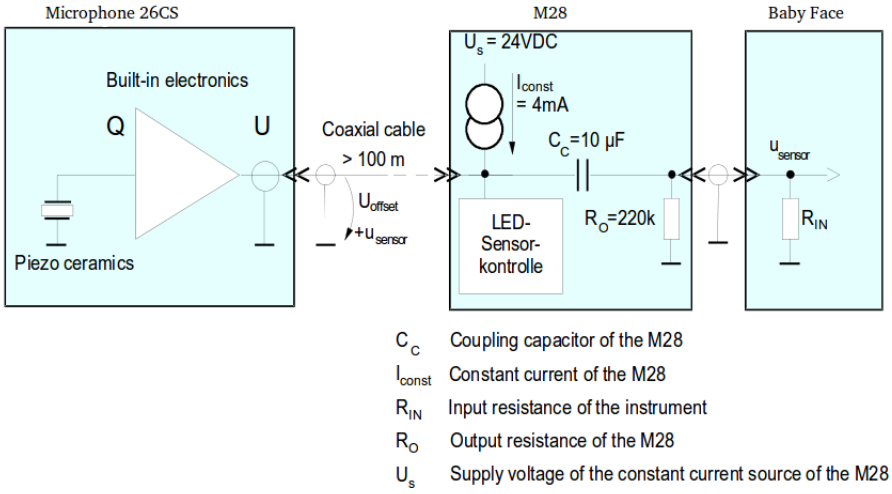

Figure 6.4: Internal electronic circuit of the M28

6.1.3 Mechanical Parts

By default, the head of the *KEMAR* HATS is not rigidly linked to the torso, and can be moved manually in azimuth, with the possibility to lock it at some specified angles (Figure 6.5). The assembly enabling this feature is shown in Figure 6.6. The black part goes inside the torso, and is endowed with an angle indicator. This angle indicator constitutes in some sense the neck, as it remains visible between the torso and the head. The grey part is rigidly attached to the head.

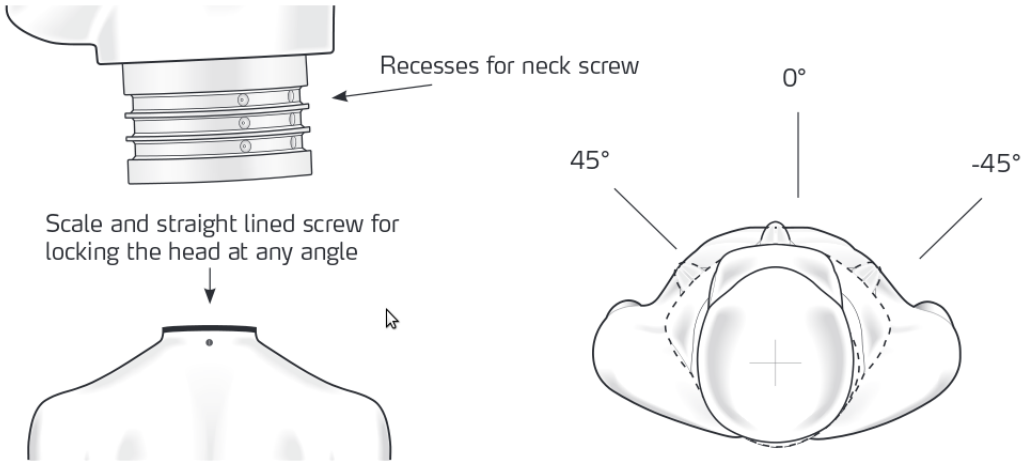

Figure 6.5: The *KEMAR* neck, with locks at 0°, 45°, −45°.

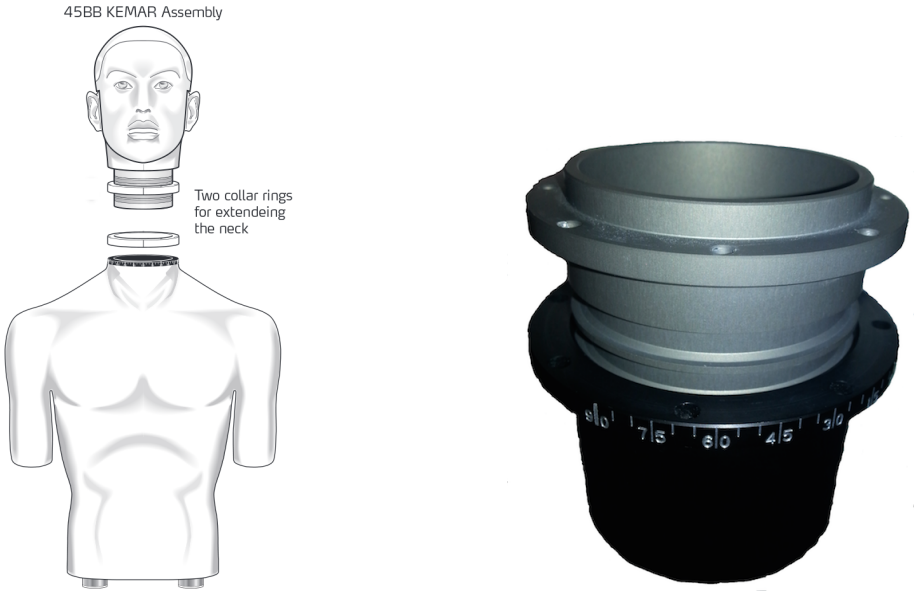

Figure 6.6: The assembly mechanism of the *KEMAR* HATS, with its angle indicator.

6.2 Devices for a controllable azimuthal degree-of-freedom on the *KEMAR* HATS

6.2.1 Mechanical design

As mentioned above, the aim is to design and manufacture a controllable azimuthal degree-of-freedom to be inserted in the *KEMAR* HATS. This section proposes a hardware design. The basic idea is to replace the two parts shown in Figure 6.6 by an aluminium mechanism designed on the basis of the CAD model of the *KEMAR* HATS (Figure 6.7). The part shown in Figure 6.8-left is screwed on the genuine mounting holes of the *KEMAR* torso, in exactly the same way as the original black angle indicator (Figure 6.6). Similarly, the complementary part displayed in Figure 6.8-right is fixed to *KEMAR*'s head by using the existing holes. So, the integrity of the *KEMAR* HATS is ensured[4]. Importantly, both parts are endowed with holes so that the cables connected to the two microphones can transmit the binaural data to the acquisition device through the lower part of the torso.

4 G.R.A.S. even accepted to guarantee the *KEMAR* HATS after the introduction of this motorization system.

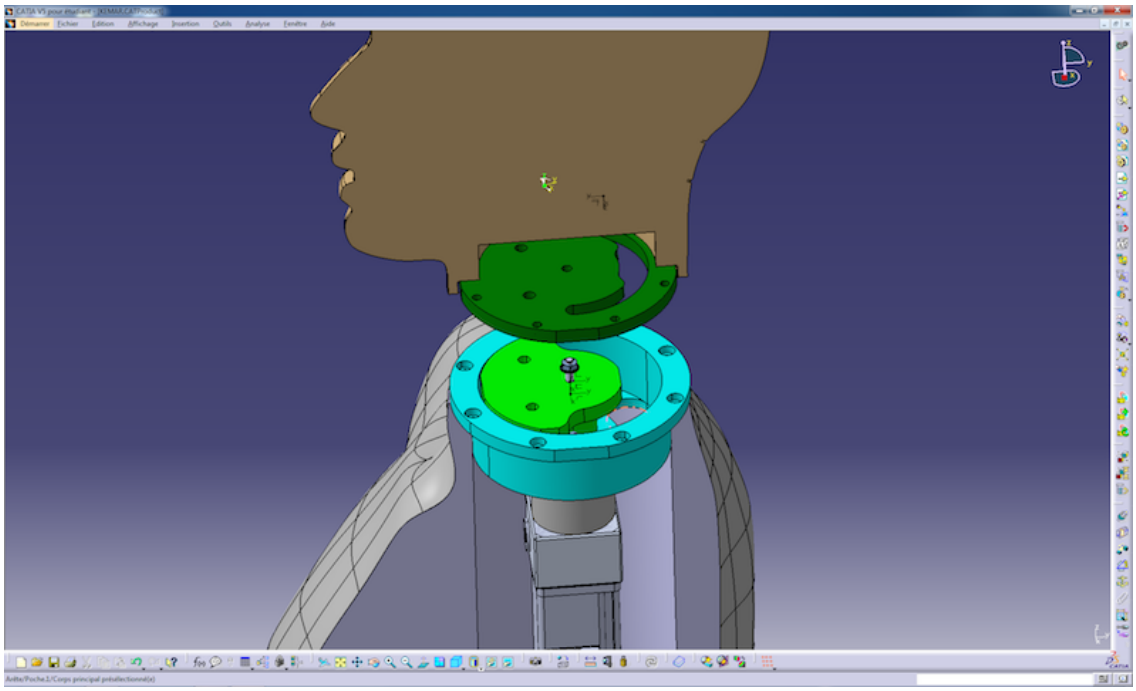

Figure 6.7: CAD design of the *KEMAR* motorization system

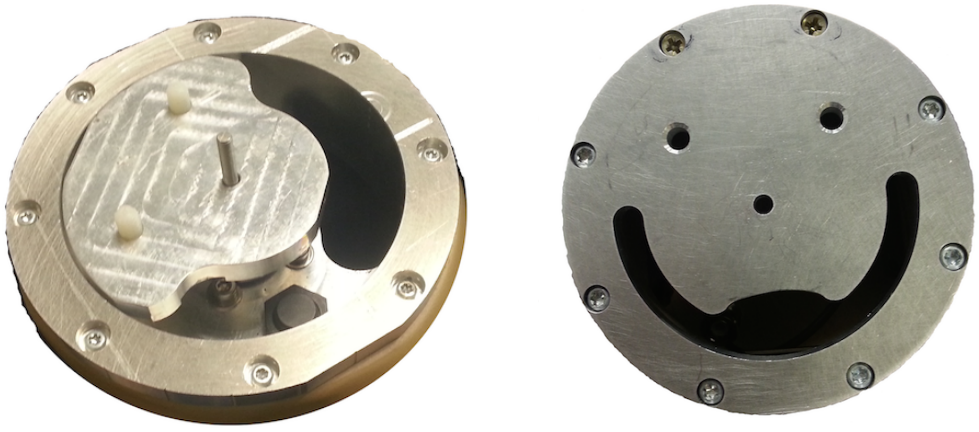

Figure 6.8: Aluminium parts to be fixed inside the torso (left) and inside the head (right).

The final device is displayed in Figure 6.9, where all pieces are assembled as they must fit inside the torso. This figure also shows the servomotor, described in the next section.

6.2.2 Actuator and sensors

To get an accurate and efficient control of the head, a set including a servomotor, its gearhead, an encoder, and an electronic controller were selected. Two criteria were studied in the motor selection.

Motor technology Various solutions exist with *DC* or *AC* supplies. The last one is mainly reserved for big and/or heavy loads. In adition, they are more complex to drive. *DC motors* are easier to operate due to their compactness and simplicity.

Figure 6.9: *KEMAR*'s full motorization system.

In this domain two versions exists. The first one called *"DC brushed motor"* is a well-known solution. However, more recently, a new DC motor technology has grown up called *"Brushless DC motor"*. These two technologies differ in their commutation methods (phase commutation) necessary to create rotation. The old *DC brushed motor* is based on a mechanical commutation. It is simple to control but it also has several drawbacks. Indeed, due to the mechanical commutation, *DC brushed motors* are noisy and request a regular maintenance due to the friction of the brushes on the commutator. On the contrary, with an electronic commutation outside of the core, *Brushless DC motors* are less noisy and do not need such an important maintenance. However, their control is more difficult. This drawback has been, however, solved with modern integrated industrial controllers.

In our opinion, the *brushless DC motor* represents the best compromise for TWO!EARS as this technology generates the least audible noise.

Power and torque of the motor If we consider the head as a sphere of 2.2 kg with a diameter of 0.2 m, then its moment of inertia J_{sphere} is equal to:

$$J_{\text{sphere}} = \frac{2}{5} \times M \times R^2 = \frac{2}{5} \times 2.2\,\text{kg} \times (0.1\,\text{m})^2 = 0.0088\,\text{kg.m}^2. \tag{6.1}$$

Considering a maximum startup acceleration of $26\,\text{rad.s}^{-2}$ (see for instance (6.5) below), the maximum torque T_{start} at startup is determined by:

$$T_{\text{start}} = J_{\text{sphere}} \times \frac{d\Omega}{dt} = 0.0088\,kg.m^2 \times 26\,rad.s^{-2} = 0.2288\,\text{N.m}. \tag{6.2}$$

So the output torque after the gearhead must be at *least* of 0.2288 N.m.

The selected servomotor has a nominal power of 100 W, 0.32 N.m of nominal torque with up to 0.95 N.m of stall torque. A $1/9^{\text{th}}$ ratio planetary gearhead was inserted, reducing the output speed and increasing the delivered torque so that

$$T_{\text{start\_gearhead}} = T_{\text{motor}} \times \textit{Gearhead ratio} \times \eta = 0.95\,\text{N.m} \times 9 \times 0.8 = 6.84\,\text{N.m}. \tag{6.3}$$

Finally, completing the motor block set, an accurate encoder was added. Although the selected brushless motor already integrates sensors providing a position of the rotor within a range of 120°, an accurate 2048-step relative quadrature encoder was integrated at the output of the motor shaft for accurate position and velocity control,

leading to wished resolution.

$$\begin{aligned}\text{Resolution} &= \frac{2\pi}{Encoder \times 4(quadrature) \times GearheadRatio} \\ &= \frac{2\pi}{2048 \times 4 \times 9} = 8.52 \times 10^{-5}\,\text{rad/pulse}. \end{aligned} \tag{6.4}$$

As the maximum acceleration, determined by the motor's manufacturer, is 300000 pulse/sec^2, and taking into account the resolution given by equation 6.4, the maximum acceleration can be determined, and writes as:

$$\text{Acceleration}_{\text{MAX}} = 300000\,\text{pulse.s}^{-2} \times 8.52.10^{-5}\,\text{rad.pulse}^{-1} = 25.56\,\text{rad.s}^{-2}. \tag{6.5}$$

Completing the set, an *Harmonica Controller* from *ELMO* was selected. This "Compact and Smart Digital Servo Drive" comes from a series of intelligent compact digital servo drives for DC brush, brushless and linear motors. It supports up to 13.3 A continuous current and integrates all the processing and power switching elements necessary for proper commutation. The *Harmonica* is capable of delivering a peak power of 2200 W and 1100 W of continuous power. Based on *Elmo's SimplIQ motion control technology*, the *Harmonica* is capable of operating in position and current modes and contains a wide range of feedback and I/O options. The drive operates on 24 V DC power. Communication with the outside world is done through the *standard CAN bus protocol*.

Two limit sensors were inserted, so far based on the Hall effect, in combination with a magnet mounted on the moving part of the aluminium assembly displayed in Figure 6.8-left. They will soon be replaced by two photoelectric proximity sensors. Those will deliver much more accurate end-of-course positions due to their insensitivity to magnetic fields. As a side effect of the new design, the admissible range of the head azimuths will be increased from $[-80°; +80°]$ to $[-90°; +90°]$, with the same conventions as in Figure 6.5.

6.3 Low-Level Libraries

Three different custom low-level libraries for the motorization of the *KEMAR* head have been developed (Figure 6.10): the `Socketcan` provides an interface with the *GNU/Linux* socket CAN layer; on the top of it, the `Harmonica` provides an interface with the *Elmo Harmonica Motor Controller*; the `Kemar` library provides an interface with the *KEMAR* HATS itself.

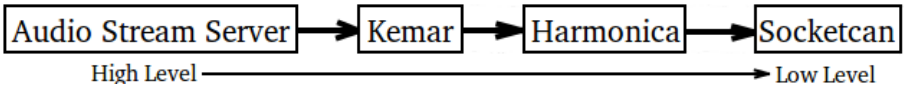

Figure 6.10: Low-level libraries Architecture

6.3.1 The `Socketcan` Library

This library is the very bottom layer of the communication with the *Harmonica* controller, which links the software with the hardware. Its main goals are:

- to initialize or end the communication with the CAN bus controller;
- to send or receive a message on the CAN bus.

6.3.2 The `Harmonica` Library

This library includes functions to initialize the motor controller, as well as to stop it. Other functions are included to set and get the position of the motor from the controller. Internally, these functions send the adequate word to the controller (who actually drives the motor) through the CAN bus using the above `Socketcan` library.

6.3.3 The `Kemar` Library

This library is intended specifically for the *KEMAR* HATS. It relies on the underlying `Harmonica` library. It includes functions for *Homing*, *Position control* or *Velocity control* of the head. These involve the aforementioned limit sensors mounted on the *KEMAR* aluminium assembly.

For a detailed information about each library, their functions and how they interact with each other, tables with function names and short descriptions are provided in Appendix 9.6.

6.4 *GenoM3* Integration

The above `Kemar` library was encapsulated into a *GenoM3* module for its integration in the TWO!EARS deployed robotics software architecture and the concurrent execution of other tasks.

6.4.1 Homing Procedure

The `Homing` activity aims at resetting the position of the head to a reference, so as to calibrate its position encoder. The process starts by moving the head to the left until it reaches the left limit sensor. At this point, the position encoder is reset to zero. Then, the head starts rotating to the right until it reaches the right limit sensor. The reference position is defined as the midpoint between these extremal positions. The homing is then completed, and the maximum admissible rotations along the left and right directions are deduced.

All this is performed in a state machine involving two *codels*. One is in charge of sending the command to the *Harmonica* controller and the second one, which is *asynchronous*, processes the replies from the *Harmonica* controller. Notice that the `Homing` function needs to be called at least once before using the other services described below.

6.4.2 Absolute Position Control

The `absolutePosition` activity drives the head to the position specified as the input parameter, considering that the zero position corresponds to the above homing position.

This activity can be divided into two codels. The first one is *synchronous*. It sends the requested position to the controller, through the position control implemented in the aforementioned `Kemar` library. The second one concurrently reads the current position of the head, and ends when the head reaches the requested position. It is declared as *asynchronous*, so that it does not block the overall behavior. For further details, see the *.gen* file in Appendix 9.5.2.

The behavior described above does not mean that the user sends a request for an absolute position and gets back the control of the software. This means that while the head moves, the user cannot perform any other control task on the servomotor. However, the module itself can perform parallel tasks if they are required. This would be the case if the *Motorization of the KEMAR* and the *audio stream server* were integrated into one single *GenoM3* module. While the head moves, the ALSA functions could keep recording audio (which is an *asynchronous* activity by itself) and the *GenoM3* module would retrieve new data when available.

6.4.3 Relative Position Control

The `relativePosition` activity is similar to the `absolutePosition` activity exposed above, but instead of providing an absolute targeted position as input parameter, this activity allows to move the head relatively to its current position. For instance, if the head is at +20° from the reference point established during the homing procedure and the `relativePosition` activity is called with an input parameter equal to +25°, the final absolute position will be +45°.

6.4.4 Velocity Control

This activity moves the head at a given constant speed defined in °/sec. The head keeps moving until this function is called again with the 0°/sec velocity parameter or until the maximum left or right limit established by the `Homing` is reached. This activity does not affect the default velocity. That is, if the `Velocity Control` activity is called to move the head at a given velocity, then the next time `Position Control` activity is called, the head moves again at the velocity that was set previously by `Set Speed` or by default in the `Homing` (section 6.4.1).

Unlike *Position Control* activities, this one only works *synchronously*. When the user calls this activity, the velocity is sent to the *Harmonica* controller and the user gets back the control of the software.

6.4.5 Get Current Position

This activity enables the user to retrieve the current position of the head at any time. This can be helpful for keeping track of the position during a move. The process of getting the position is the same as for any request sent to the controller: a message is sent and the answer is waited for in an asynchronous codel. The process is quite fast: it only takes 1.5 ms for the motor controller to send the current position back to the *GenoM3* module.

6.4.6 Set Speed

The speed is set by default at 100°/sec after the Homing is done. With *Set Speed*, the user can set a new value for *Control in Position* at any time. To keep consistency with other functions, the speed is expressed in °/sec.

To conclude, if the *Homing* has not been called, then the *Position Control* or *Velocity*

Control functions do not perform the requested activity. This is so because the absolute zero has not been set, or the maximum left and right positions have not been calculated, so that the controller has no reference for the movement.

6.5 Merging the *audio stream server* and the Motorization of the *KEMAR* into a single *GenoM3* Module

The need to get motor and audio features simultaneously motivated the merging of the *audio stream server* and the *Motorization of the KEMAR* into the same *GenoM3* Module. This allows an easy, fast and effective synchronization of the audio data and the position of the head at the exact time of retrieving new chunks from *ALSA*. *Easy* means that all functions are in the same module, so that no external functions or modules are needed. *Fast* means that data are internally synchronized in the merged module before being sent to the *MATLAB* client over the *dedicated socket*.

When a new chunk of audio samples is about to be published on an *out port*, the current position of the head is read and both data are published on this port. So, when this port is accessed, *e.g.* from *MATLAB*, the user gets all the samples from the microphones (frames) as well as the head position corresponding to the final time of each chunk. For example, if the capture is started to record 80 chunks (Section 5.3.1) of 50 ms of audio data sampled at Fs=44100 Hz, then the published data are as follows:

CHUNKS	FRAMES	POSITIONS
Chunk[0]	Frame[0] Frame[1] ⋮ Frame[2205]	 Position[0]
Chunk[1]	Frame[2206] Frame[2207] ⋮ Frame[4410]	 Position[1]
Chunk[2] ⋮	Frame[4411] ⋮ Frame[(N-2)*2205+2205]	 ⋮ Position[N-2]
Chunk[N-1]	Frame[(N-1)*2205+1] Frame[(N-1)*2205+2] ⋮ Frame[(N-1)*2205+2205]	 Position[N-1]
Chunk[N]	Frame[N*2205+1] Frame[N*2205+2] ⋮ Frame[N*2205+2205]	 Position[N]

Each chunk of data has an *index number*. This number is incremented by one every time a new chunk of audio data is available. It is used as a reference value for synchronizing the position of the head with the binaural audio data. This index is also published on the *out port* along with the binaural audio data and the positions of the head.

A new feature has been added with this merge. When the *Control in Absolute Position* (Section 6.4.2) activity is called, the *index number* of the current chunk and the position of the head before it starts moving are published in a port named *Indexes*. When the head reaches the requested position, the current chunk and that position are also published in the port.

Another improvement has concerned the time it takes to retrieve the binaural audio through the *dedicated socket*. Due to the fact that some logic had to be added in order to include the position of the head in the buffer that is sent from the server, the logic involved in the client also had to be changed.

Data sent by the server to its clients now consist of the following elements:

- number of new chunks to be expected by the client;
- left and right samples (non-interleaved) from the new chunks;
- index of the last chunk;
- positions of the head synchronized with the audio.

In other words, in addition to the samples from the binaural audio data, extra values are inserted: the number of chunks or blocks to be sent over the *dedicated socket*, the last chunk's index and the positions of the head.

Instead of waiting for a fixed number of bytes to read (according to `Port500`, `Port1000` or `Port4000` for the first release of the *audio stream server*, described in section 5.4.3), the clients follow the following steps to retrieve the buffer sent by the server:

1. Wait until the value `BytesAvailable` field of the *TCP/IP* object (Section 5.4.2) is greater than 0 (this varies according to the connection and the available bandwidth).
2. Read all the available bytes in the buffer.
3. The first value is the number of chunks to be sent by the server (`N`). So, the client calculates the total number of bytes to read and stores the samples that have already come.
4. Wait until the remaining samples arrive and retrieve them.

This strategy improves the total time it takes to retrieve all the information through the

dedicated socket (shown in Figure 6.11) over *localhost*, described in section 5.4.4. While it was 160 ms, this time is reduced to only 86 ms with this new logic. Table 6.2 shows a more detailed result for different data sizes.

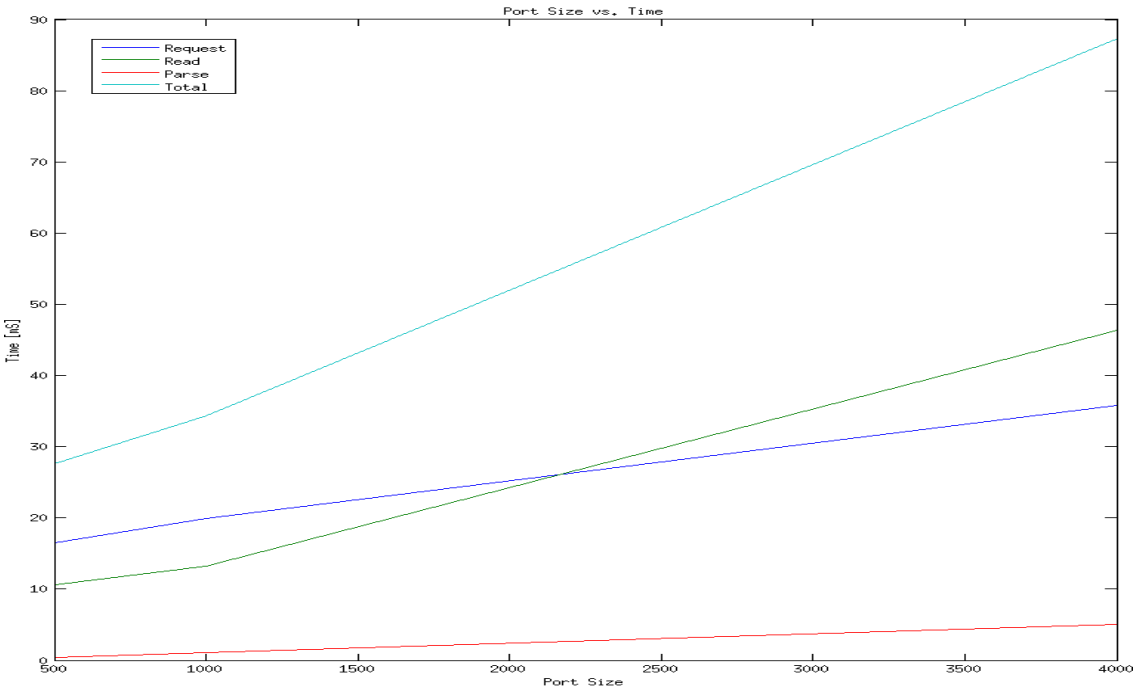

Figure 6.11: Total time for each stage to retrieve data over *dedicated socket* on *localhost* as a function of the port size.

"Port" (ms of data)	Request [ms]	Read [ms]	Structure [ms]	Total [ms]
500	16	11	1	28
1000	17	13	2	32
4000	36	46	4	86

Table 6.2: Total time for each stage to retrieve data over *dedicated socket* on *localhost* as a function of the port size (values from Figure 6.11).

To make consistent comparisons between the version described in Section 5.4.4 and this new one described above, as the ports size is not fixed with the asynchronous *audio stream server*, the parameters and the time instants to request data were chosen so as to match the synchronous version, along Table 6.3.

Chunks of 50 ms are considered, with sampling frequency Fs=44100 Hz, so that `N` writes as `N=NumberOfChunks*(SamplesPerChunk*2(channels))+1(position)+2`.

"Port" (ms of data)	Samples(N)	Bytes
500	44112	176448
1000	88222	352888
4000	352882	1411528

Table 6.3: Bytes sent as a function of the port size.

6.6 Further Results

Timings over *localhost* look promising as shown in Figure 6.11. However, the *audio stream server* and its clients will be running on different computers. Therefore, the same experiment was held over the *CNRS network* and on a *local network* built around a *switch*.

Figure 6.12 shows the outcome of an experiment conducted in the *CNRS network*. The communication is slower than on *localhost*. More details can be read from Table 6.4.

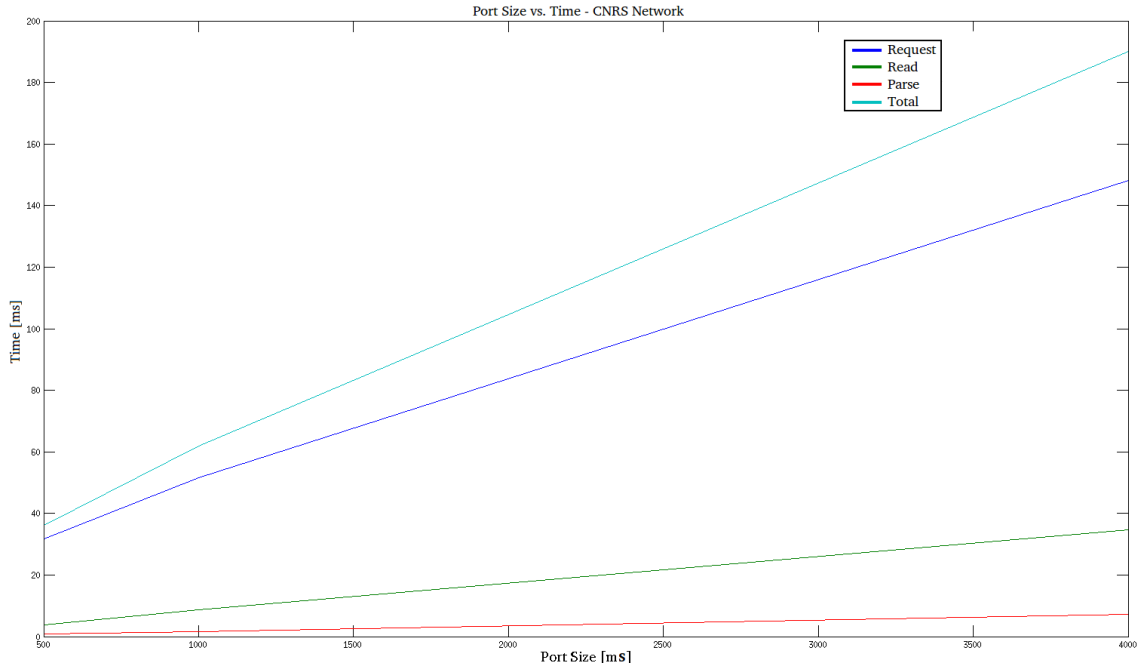

Figure 6.12: Total time for each stage to retrieve data over *dedicated socket* on *CNRS Network* as a function of the port size.

However, these timings improved over the *Local Network*, almost matching the ones over *Local Host* as it is shown in Figure 6.13 and more detailed in Table 6.5.

"Port" (ms of data)	Request [ms]	Read [ms]	Structure [ms]	Total [ms]
500	32	4	1	37
100	48	10	3	61
4000	145	37	4	186

Table 6.4: Total time for each stage to retrieve data over *dedicated socket* on the *CNRS Network* as a function of the port size.

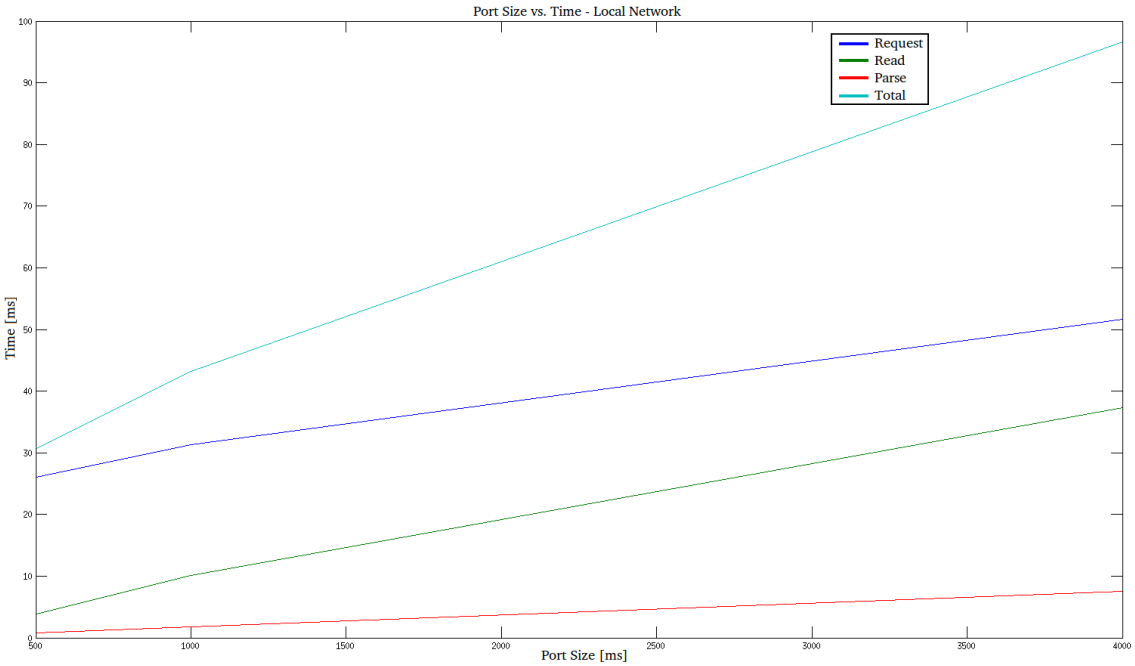

Figure 6.13: Total time for each stage to retrieve data over *dedicated socket* on *Local Network* as a function of the port size.

"Port" (ms of data)	Request [ms]	Read [ms]	Structure [ms]	Total [ms]
500	27	4	1	32
100	30	10	3	43
4000	52	36	8	96

Table 6.5: Total time for each stage to retrieve data over *dedicated socket* on *Local Network* as a function of the port size.

7 Virtual environment for the deployment system based on *MORSE*

7.1 Introduction

The cognitive components of the Two!Ears development architecture need to be tested while the functional layer of the deployed architecture is being implemented. This need also appears in large-scale experiments in robotics, and this is the reason why the development of *MORSE* was launched some years ago at *CNRS*. *MORSE* (*M*odular *O*pen*R*obots *S*imulation *E*ngine) is a generic simulator for robotics. It focuses on realistic 3D simulation of small to large environments, indoor or outdoor, with one to tens of robots, including *visual rendering* and *physics simulation*. Nowadays, there is a community of about 100 users, and about 15 developers worldwide keep enhancing the software.

The user describes the scene to be simulated using the *python API* in a small *Python* script. There, the robot(s) and the environment have to be addressed. *MORSE* provides several command-line tools to create stubs. The process is very fast to get a first running simulation.

MORSE comes with a set of standard *sensors* such as cameras, laser scanners, GPS, odometry to name a few, and *actuators* such as speed controllers, high-level waypoint controllers or generic joint controllers. It also incorporates *robotic bases* such as quadrotors, ATRV, generic 4 wheel vehicle, or the *PR2*. New ones can be added easily.

MORSE rendering is based on the *Blender Game Engine*. This OpenGL-based Game Engine supports shaders and multi-texturing, provides advanced lighting options, and uses the state-of-the-art *Bullet library* for physics simulation.

The user can select the level of realism of the simulation. For example, to work on vision, the user needs accurate camera sensors but may not care too much about the realism of the motion controller. *MORSE* lets the user define how realistic the components of the robot should be.

MORSE also supports two different strategies for handling time: *best effort*, that tries to keep up with the simulation, at the cost of dropping frames if necessary, or *fixed step*

to ensures the simulation accuracy. In this last case, *MORSE* exports its own clock, so that external time-dependent modules can be adjusted. *MORSE* also complies with HLA specifications.

MORSE does not make any assumption on the user's architecture. At the moment, it supports four open-source middlewares (*ROS*, *YARP*, *Pocolibs* and *MOOS*). It also supports a simple *socket*-based protocol for easy integration in another language/toolbox.

The following sections describe the simulation environment for rendering scenes and mechanical robot behavior with *MORSE*.

MORSE was chosen because it can be interfaced transparently with *GenoM3/ROS*. It can be easily linked to the cognitive level thanks to the *genomix matlab bridge* and visual processing can be efficiently implemented in *GenoM3*. Therefore, the whole architecture stays the same either in the virtual or real environment.

Even though *MORSE* does not provide auditory information, the complete simulation system will be performed on two *parallel* "virtual worlds". The visual information will be from *MORSE* and the auditory information will come from the WP1-framework.

7.2 Describing a Toy Scenario

A simple *Python* script has been defined so as to describe a first simulation *scenario* for Two!Ears. It includes a robot along with its sensors and actuators.

7.2.1 The Robot

Two robots have been simulated so far: a full *PR2* able to navigate and to control the motion of its torso, arms and head; a "head-on-a-stick" type system, that is, a head-and-torso-simulator (HATS; used model: *KEMAR*) endowed with cameras for stereoscopic vision[1], allowing rotation motion of the dummy head to actively explore the environment. This last device is shown in Figure 7.1.

As *ROS* is the middleware underlying the Two!Ears deployment system, a *ROS* interface was included in the *Python* script to control the *PR2* or *KEMAR* robot.

1 A design from scratch was made, due to Copyright issues.

Figure 7.1: Virtual *KEMAR* HATS

7.2.2 Sensors

Three sensors have been included in the description of this scenario, for the *PR2*:

Odometry An odometry has been fitted to the robot so as to estimate its position over time. The name "`odom`" is specified as the *ROS* topic (*i.e.*, port) that publishes the corresponding data.

Laser Scan One laser scan for distance measurement at every pointing direction is needed. Its range, resolution and scan window are specified as well as the name "`base scan`" for the *ROS* topic where corresponding data is published.

"RGBA" cameras Two such cameras (*RGBA* being a type defined in *MORSE*) have been mounted on the *PR2* head. Their focal length, as well as their width and height resolution in pixels can be set. "`Video camera`" was the chosen name for the *ROS* topic where the data of the cameras are published.

For the *KEMAR*, the same *RGBA* cameras have been included and placed in the positions of the eyes.

7.2.3 Actuators

The *PR2* has been equipped with two actuators.

Motion XYW This actuator enables the motion of the base of the *PR2*. The related *ROS* topic was named "`cmd vel`".

Keyboard This actuator has been added so as to drive the robot manually. It is especially useful to build a map offline by using SLAM functions in the so-called `navigation stack` of the robot, that is, in the collection of *ROS* `nodes` (*i.e.*, modules) for navigation. This map is a prerequisite to any positioning/navigation of the *PR2* in the environment.

For the *KEMAR*, no actuator has been included as its rotational dof is straightly controlled from the aforementioned *GenoM3* module.

7.2.4 The Environment

The environment where the simulation takes place is a *.blend* file. It is designed in Blender, and must be specified in the description of the scenario.

For both robots, a default scenario included in *MORSE* was chosen as the environment. It emulates the robotics hall of *CNRS*, where some Two!Ears large scale experiments will take place.

7.3 *GenoM3-MORSE* Integration

A virtual robot in *MORSE* can be accessed (control and data) from within a *ROS* module thanks to the *ROS* support provided by *MORSE*. As *MORSE* and *GenoM3* create one *ROS node* each, it is straightforward to access from a *GenoM3* module a *ROS topic* (port) associated to *MORSE*, so as to control a robot. To retrieve data from the sensors, a *GenoM3* module has to subscribe to the specific *MORSE*'s *ROS topic* where the required data is published. This is the case for the *KEMAR*, which is a simple robot.

The *PR2* is a much more complex robot which works with *ROS* actions[2] for interfacing with preemtable tasks (moving the base of the *PR2* or performing a laser scan). The middleware-independance inherent to *GenoM3/ROS* implies that *ROS* actions should not be called directly from a *GenoM3/ROS* component. This is the reason why an additional *Python script* has been implemented, as shown in Figure 7.2. This new *Python script* creates a new *ROS node*. The described implementation to control a virtual *PR2* within *MORSE* is to write in a *GenoM3/ROS* module an interface enabling a user to send the required parameters to the *Python* script. This *Python* script encapsulates them into *ROS* action goals and writes them on the targeted *ROS topic*.

The first step to control the *PR2* was through the *ROS* `navigation stack`. After that, a

2 `http://wiki.ros.org/actionlib`

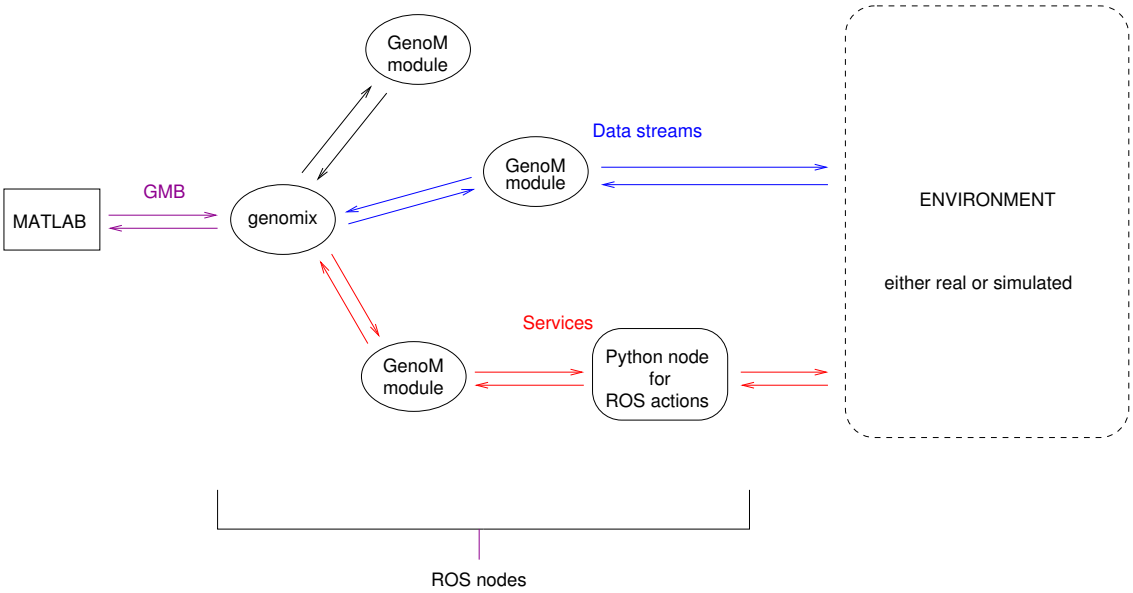

Figure 7.2: Final Architecure to integrate *MORSE* and *GenoM3/ROS*

simple *GenoM3* module was coded to send the desired position of the joints (both arms and torso, and the pan and tilt on the head) to the *Python* script in charge of the control. This allowed the user to control separately each part of the *PR2*.

Below are the different activities in the *GenoM3/ROS* module listed that control the virtual *PR2* on *MORSE*.

- Move head;
- Move left arm;
- Move right arm;
- Move torso;
- Navigate.

Figure 7.3 shows the *PR2* controlled with the *GenoM3* module through the *genomix matlab bridge*.

For the control of the neck rotation of the *KEMAR* HATS, a much simpler *GenoM3* module was coded. It fills directly a specific *ROS topic* with the required angular position and velocity. It simulates the activities described in section 6.4 for *Position Control* and *Velocity Control*.

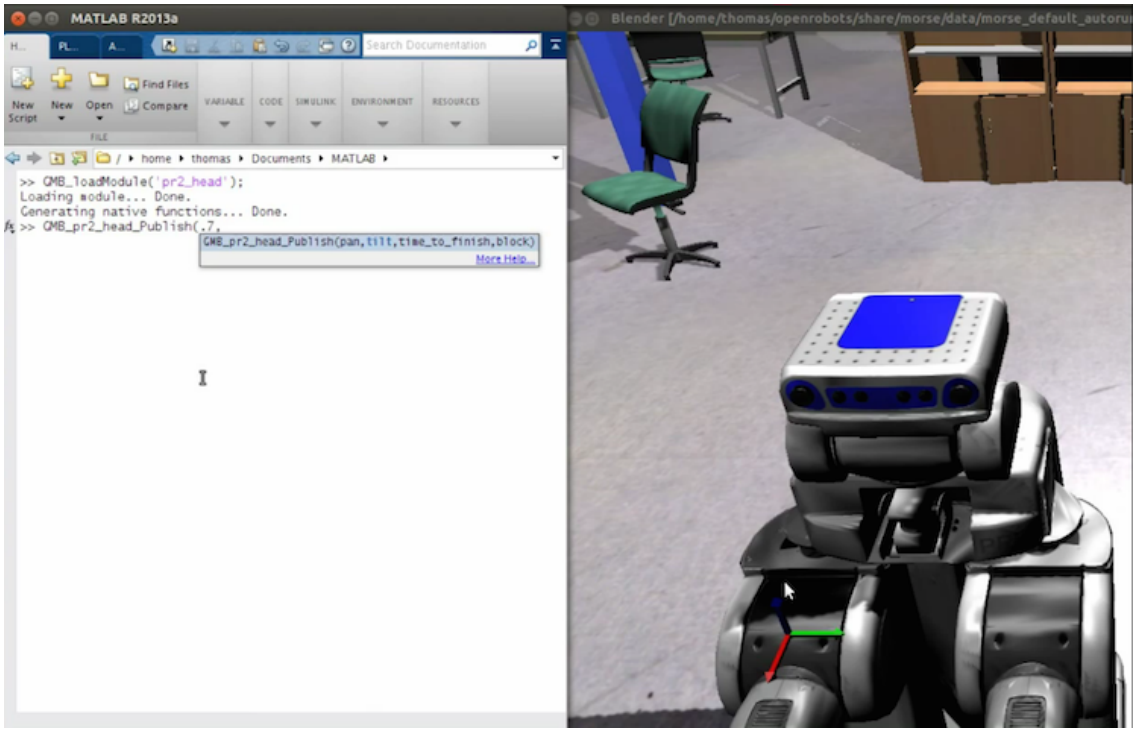

Figure 7.3: The *PR2* on *MORSE* controlled with the *GenoM3* module through the *genomix matlab bridge*

As mentioned previously, user-defined cameras, such as the *RGBA*, have been mounted on the virtual robots by means of the *Python* script that describes the simulation's scenario. Depending on their parametrization, the size of the published data changes. For instance, if `width=256` and `height=256`, then the output buffer (on its *ROS topic*) is made of `256*256*4=262144` elements, each one refering to the R, G, B and A for each pixel. A *Python* script is not necessary to access these values as a the *ROS topic* can be accessed via a *GenoM3/ROS* module. Note that even though two *RGBA* cameras were included in the simulation, the real implementation will entail two *RGB* cameras. Therefore, the data related to A is not be used in image processing algorithms.

It is important to remind that the way how the two virtual robots described here are controlled is completely reproducible for the real *KEMAR* and *PR2*.

8 Ongoing work and short-term prospects

This chapter summarizes ongoing work as well as planned work in the short term. First, the main issues are described, which are related to *KEMAR* head-and-torso-simulators. These essentially consist in further instrumentation and in collecting new head related transfer functions (HRTFs). Then, software developments concerning vision are briefly mentioned, and work at the intersection of WP2 and WP5 is discussed.

8.1 Issues related to the *KEMAR* HATS

8.1.1 Design of *GenoM3* control modules for other motorized *KEMAR* HATS

Before the beginning of the project, URO and TUB had designed a system for the control of the neck rotational degree-of-freedom of their own *KEMAR* HATS, based on a *POWERCUBE* rotary actuator. As this is very noisy while in motion, a new solution was proposed in Chapter 6 for Two!Ears. Unfortunately, due to the differences between the former and our most recent models of the *KEMAR* HATS, this new solution cannot be brought to their HATS. So, a *GenoM3* skeleton, similar to the one shown in Appendix 9.5.2, was coded and distributed to URO and TUB. In the short term, they will include, in collaboration with CNRS, their own low-level libraries. The aim is obviously to access all the *KEMAR* HATS of the consortium with the same interface, that is, the same *.gen* file gathering the same internal data structure, services and ports, so that only the underlying hidden code differs. Similar integrations with other binaural heads from USFD and RUB are under study.

8.1.2 Design of a new HRTF database for the motorized *KEMAR* HATS

The motorization of the *KEMAR* HATS is functional, and provides a control of the relative angle between its head and its torso. However, binaural methods based on HRTF measurements cannot fully be applied, *e.g.*, for source localization or separation.

Indeed, existing KEMAR HRTF databases, such as the *CIPIC HRTF database*[1], the *MIT MediaLab database*[2], or the *TUB database*[3] assume that the head and torso are sticked to each other. So, acoustic measurements will be conducted for a dense grid of relative head-torso angles values, and the corresponding extended HRTF database will be disseminated publicly.

8.1.3 Instrumentation of the *KEMAR* head with stereovision

The *KEMAR* head must be equipped with a stereroscopic sensor. As mentioned in Section 4.2.3 of Deliverable D3.2, several sensors can be envisaged with the following requirements and options:

- passive stereovision is preferred (no random light is projected on the scene by the visual sensor);
- the stereoscopic sensor must provide both appearance-based and 3D data;
- the depth accuracy of the 3D measurements must be about 10 cm;
- the maximum 3D range may be within $5 - 7$ m;
- the sensor resolution and field of view depend on the demonstration scenarios.

A first analysis of disparity as a function of the depth is proposed in Figure 4.10, page 49 of Deliverable 3.2.

Three preliminary operations have been launched.

Tests with the dual stereo pair of the PR2 The head of the *PR2* robot embeds a *wide stereo camera* and a *narrow stereo camera*. The size and baseline of each pair may constitute an admissible option for their mounting on the *KEMAR* head (see `http://pr2s.clearpathrobotics.com/wiki/PR2%20Manual/Chapter9#Head_Cameras`). Some evaluations have begun to characterize experimentally each stereoscopic pair in view of the above criteria, by using *GenoM3* and *ROS* modules available at *CNRS* for image acquisition and processing on the *PR2* robot.

1 V.R. Algazi, R.O. Duda, D.M. Thompson and C. Avendano, "The CIPIC HRTF Database", IEEE Workshop on Applications of Signal Processing to Audio and Electroacoustics, New Paltz, NY, 2001, `http://interface.cipic.ucdavis.edu/sound/hrtf.html`.

2 `http://sound.media.mit.edu/resources/KEMAR.html`.

3 H. Wierstorf, M. Geier, A. Raake and S. Spors, "A Free Database of Head-Related Impulse Response Measurements in the Horizontal Plane with Multiple Distances", AES 130th Convention, 2011, `https://dev.qu.tu-berlin.de/projects/measurements/wiki/2010-11-kemar-anechoic`.

"Glasses" fitting closely on the KEMAR face Two preliminary designs of plastic "glasses" which could fit closely on the *KEMAR* face and serve as a support of the stereoscopic pair—with a baseline of approximatively 9 cm—have been obtained on the basis of the CAD model of the *KEMAR* head. Once they are refined and if they prove useful, they will be manufactured by 3D printing. The first design looks like a filled "mask" (Figure 8.1). From this model, the design of "glasses" shown in Figure 8.2 has been obtained. In both cases, an aluminium part supporting the cameras would be attached to the front part in order to keep the stereo rig calibration insensitive to temperature variations.

Spare cover to support a stereo rig In case anthropomorphic vision cannot be installed on the *KEMAR* head (including on "glasses"), a clone of the *KEMAR* cover has also been molded (Figure 8.3). The aim is to attach on it a support for a stereo rig without altering the genuine *KEMAR* head.

8.1.4 Porting of the *KEMAR* head on *PR2*

The porting of the binaural/audiovisual *KEMAR* head (w/o the torso) on the *PR2* robot is in progress. Its position must be high enough with respect to the *PR2* torso and shoulders so as to minimize their effects on the sensed signals. So, it was decided to mount it on the top of the *PR2* head. Importantly, the mechanical assembly of the motorization system (Figures 6.7 and 6.8 page 47) has been designed so as to come to an easy and repeatable install.

As the *KEMAR* head momentum exceeds the admissible physical limitations on the *PR2* payload, and in view of the fact that the tilt of the *PR2* head can take important values during initialization or emergency stop, it was decided to freeze this degree-of-freedom. As already mentioned, tilt does not bring much to active motions. Contacts with *ClearPath Robotics* have been taken so as to solve related technical problems, such as cancelling the servocontrol of the tilt motion at startup while it is mechanically locked. The *KEMAR* head will be mounted on the *PR2* head once the firmware is updated by ClearPath Robotics so that only translation and azimuthal degrees-of-freedom are kept.

In humans, it is known that the presence of a torso has the effect of increasing the acoustic pressure in the vicinity of the ear up to frequencies of around 2 kHz Algazi *et al.* (2001), with an effect greater for frontal sources than for lateral ones. Conversely, the shoulders affect the same frequency range mainly for sources emitting from lateral positions. It is then clear that the shoulders and torso both contribute to spatial effects. This should be reproduced on the robotic system.

As a preliminary solution, an original *PR2* cover has been designed, see its 3D picture in Figure 8.4, and parts in Figure 8.5. This cover wraps around the robot upper part and

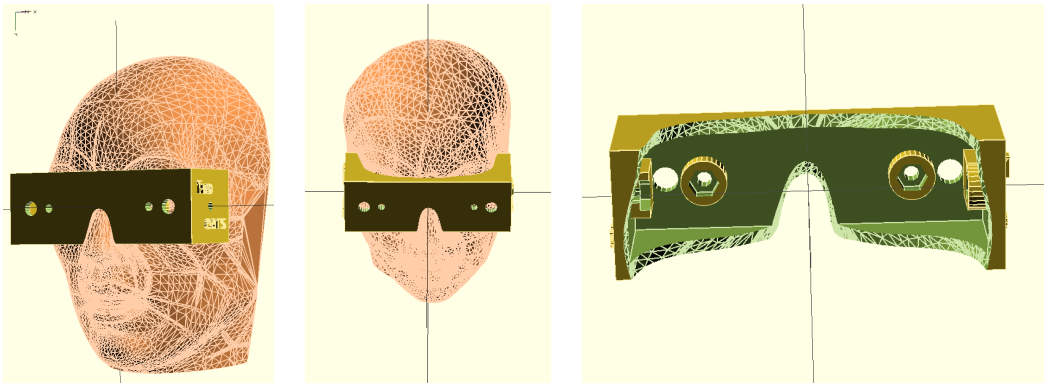

Figure 8.1: Preliminary designs of a "mask" to support cameras on *KEMAR* head.

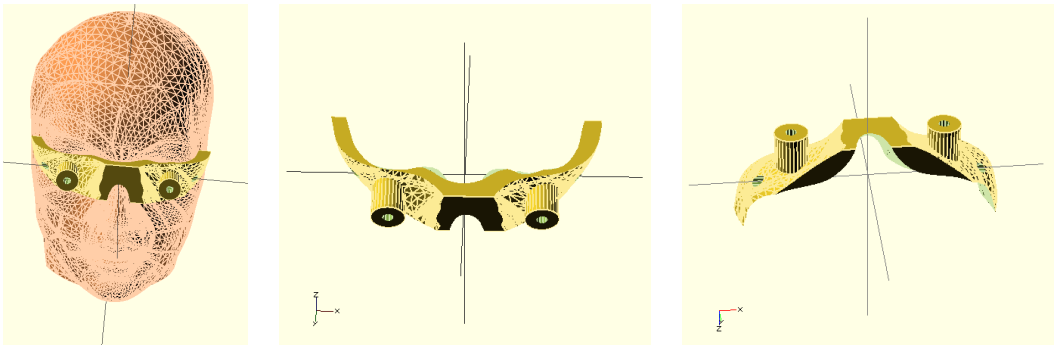

Figure 8.2: Preliminary designs of "glasses" to support cameras on the *KEMAR* head.

Figure 8.3: Molding of another cover for the *KEMAR* head, which could support a stereo rig.

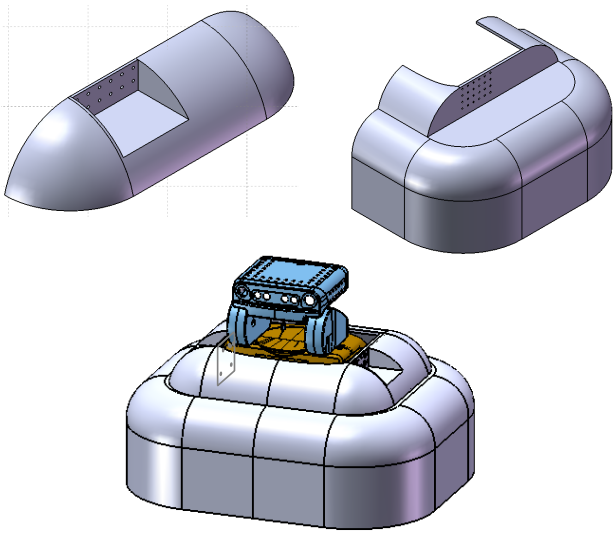

Figure 8.4: 3D view of the preliminary design of the *PR2* cover.

two arms, which are not used in the project. Additionally, it can allow to minimize the "self-noise" originating from the fans of the cooling system positionned at the bottom-back of the robot. The cover is made of two parts in order to ease its fastening and removal. All mechanical connections between these two parts and the robot are made of soft plastic to avoid the propagation and the possible amplification of vibrations during robot motion. The cover material has not been definitely chosen. A balance between weigtht, rigidity, resistance, cost and ease of manufacturing has to be found. The final design is expected between months 13 and 18.

Note that the HRTFs of the *KEMAR* head will be significantly modified, and will have to be re-identified.

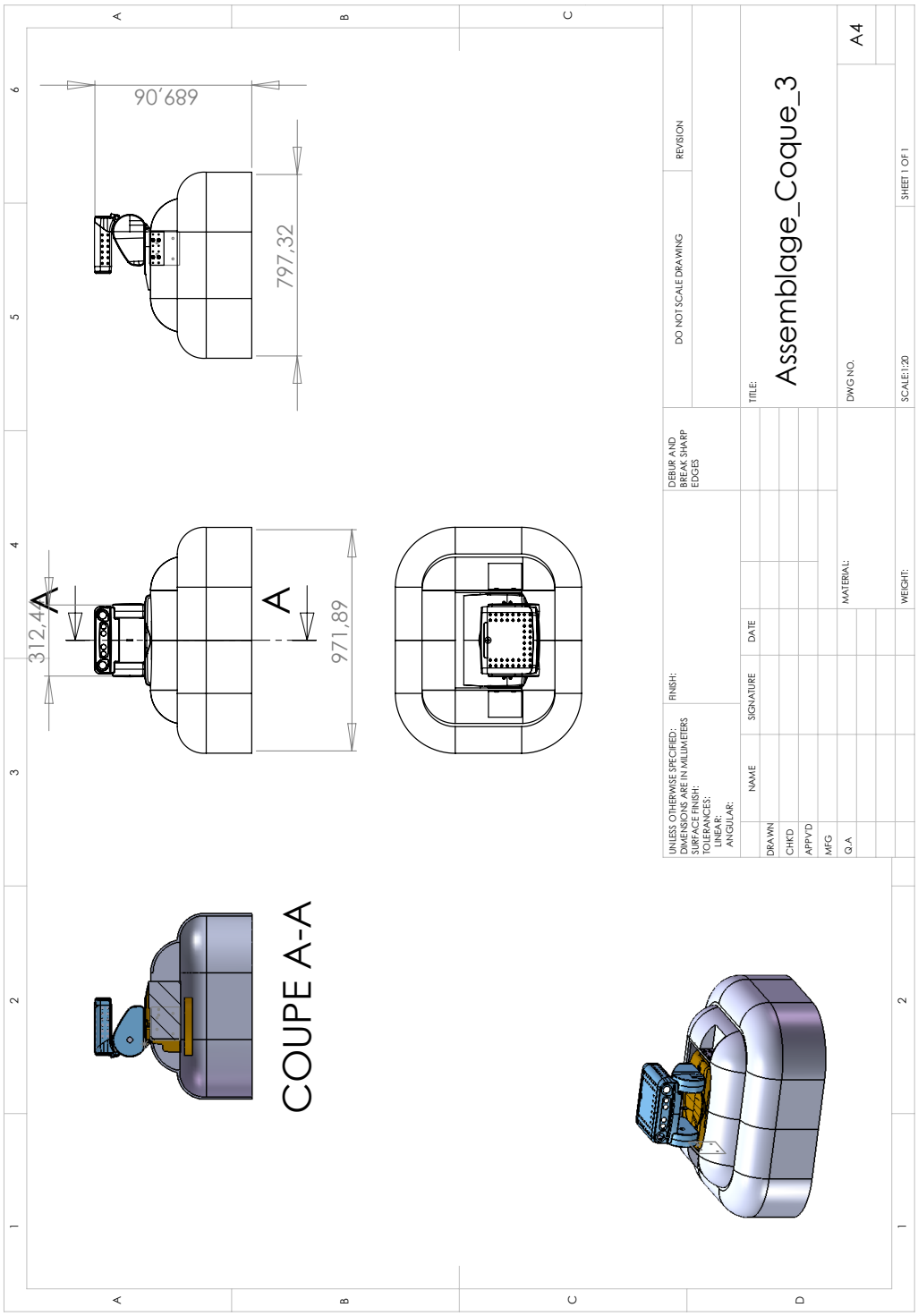

Figure 8.5: *PR2* cover mechanical plan.

8.2 Other issues

8.2.1 Visual data acquisition, streaming and processing

GenoM3 modules are needed for calibration, video acquisition and time-stamped streaming. In view of the experience of *CNRS* on this aspect (availability of similar modules for many types of cameras, etc.), and as TWO!EARS does not raise additional contraints on such functions, they should be shortly deployed.

Functions for visual perception of persons and objects will be deployed on the basis of existing algorithms developed locally at *CNRS* and after extensive evaluation of available off-the-shelf *ROS* stacks (collections of modules). These functions will first concern detection, segmentation and tracking. Semantic labeling will come during the second part of year 2.

8.2.2 Work at the intersection of other workpackages and WP5

During year 1, extensive evaluations of the automatic generation of standalone *C* code were conducted, by means of the *MATLAB coder*, for a subset of functions from the *MATLAB*-based *Auditory Modeling toolbox*, which have also been integrated in the *Auditory Front End* (AFE) of TWO!EARS. The aim was to recast in *C/C++* some of the WP2-related low-level audio processing functions developed in *MATLAB*, and to further encapsulate the obtained library into *GenoM3* modules for real time, concurrent execution with other functions. Unfortunately, it was shown that this automatic transcription process generates a large number of files, and often fails.

The selection of parts from the WP2 development system and their transcoding into *GenoM3* modules from scratch will be conducted in year 2, as it is an essential need to adress large scale experiments with good performance.

Discussions about whether auditive (from WP2) or visual (from WP3) functions could be embedded in a hardware-software "System-on-a-Programmable-Chip" (SoPC) architecture for high-performance data acquisition and low-level processing are underway within the consortium. Preliminary evaluations of the Zedboard development board were conducted in order to prepare hardware-software co-design of such smart sensors.

8.3 Scientific work

Last, *CNRS* has been designing a three-layer strategy—developed independently of the project—to source localization from a binaural head by combining the binaural perception and the sensor motion: (A - "short-term detection") estimation of the spatial arrangement of active sources—possibly with the detection of their number—from the analysis of the binaural stream over small time snippets; (B - "audio-motor binaural localization") assimilation of these data over time and combination with the motor commands of the sensor, so as to get a first level of active localization; (C - "information-based feedback control") feedback control of the sensor motion so as to improve the fusion performed in stage (B).

The two first layers have been prototyped in *MATLAB* and implemented in *C/C++*. They are in the process of being encapsulated into *GenoM3* and extensively evaluated, so as to be brought to TWO!EARS.

This work contributed to the publications Portello *et al.* (2014a,b), Blauert *et al.* (2014) and to the submissions Blauert *et al.* (submitted, 2014), Bustamante *et al.* (2015, submitted, invited paper).

9 Appendix

This appendix contains several complement references to the main text. In particular, additional information about *GenoM3*, such as installation instructions and examples of modules, is provided.

9.1 The BSD 3-Clause License

The BSD-3 Clause Licence, used for the copyright of developed material, is shown below. The template used for this licence was taken from the url: `http://opensource.org/licenses/BSD-3-Clause`.

```
Copyright (c) 2014, LAAS/CNRS
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
```

```
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

9.2 Guidelines to *GenoM3* install and associated tools

9.2.1 Introduction to the installation process

The best way to install *GenoM3*, the *genomix* server, the *Tcl* client and even the `demo` module (Section 4.3.3) is through *robotpkg*, a compilation framework for installing robotic software, initiated at *CNRS*. The recommended setup is to have *robotpkg* in `/home/username/robotpkg` and to install any robotic software in a folder `/home/username/openrobots` on the Hierarchical File System. This location ensures that the user can install and run software without root privileges.

9.2.2 Installation instructions

The following simple procedure was defined for installing *ROS* middleware, together with the *GenoM3* generator of modules and associated tools of the Two!Ears software architecture.

```
            *** INSTALLATION INSTRUCTIONS ***

CONTENTS
1. INSTALL ROS
2. INSTALL ROBOTPKG
3. INSTALL GENOM3
4. INSTALL GENOMIX AND TCL CLIENT
5. COMPILE AND RUN A GENOM3 COMPONENT
6. INSTALL MORSE

NOTE: You need a Unix system, e.g. Ubuntu-12.04 (other systems have not been
      tested).
```

```
1. INSTALL ROS ------------------------------------------------------------------

  Prerequisites: none.

  Install ROS groovy, you only need the ros-dekstop package. For Ubuntu-12.04
  this is done as follows (instructions from
  http://wiki.ros.org/groovy/Installation):
    > sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu precise main" >
                                        /etc/apt/sources.list.d/ros-latest.list'
    > wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
    > sudo apt-get update
    > sudo apt-get install ros-groovy-desktop

  Setup your environment for ROS. With bash, edit your shell startup file
  (~/.bashrc) and add this line:

    source /opt/ros/groovy/setup.bash


2. INSTALL ROBOTPKG -------------------------------------------------------------

  Prerequisites: none.

  Configure robotpkg.
    > cd
    > git clone git://git.openrobots.org/robots/robotpkg
        (if you cannot clone the repository, try using this url:
        > git clone https://git.openrobots.org/robots/robotpkg.git)
    > cd robotpkg/bootstrap
    > ./bootstrap --prefix ${HOME}/openrobots

  Checkout robotpkg/wip.
    > cd ~/robotpkg
    > git clone git://git.openrobots.org/robots/robotpkg/robotpkg-wip wip

  Setup your environment for robotpkg. With bash, edit your shell startup file
  (~/.bashrc) and add those lines:

    export INSTALL_DIR=$HOME/openrobots
    export PATH=$PATH:$INSTALL_DIR/bin:$INSTALL_DIR/sbin
```

```
  export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$INSTALL_DIR/lib/pkgconfig
  export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$INSTALL_DIR/src/ros-nodes:
                                                          $INSTALL_DIR/share
  export PYTHONPATH=$PYTHONPATH:$INSTALL_DIR/lib/python2.7/site-packages:
                                      $INSTALL_DIR/lib/python3.2/site-packages


3. INSTALL GENOM3 ------------------------------------------------------------

  Prerequisites: (1.) & (2.).

  Installation of GenoM3 is made by installing the demo-genom3 software that
  needs all the GenoM3 dependencies.

  Add these lines to ${HOME}/openrobots/etc/robotpkg.conf (add them anywhere in
  the file, but near the beginning makes more sense):

    PKG_OPTIONS.demo-genom3= codels
    PKG_OPTIONS.demo-genom3+= pocolibs-server pocolibs-client-c
    PKG_OPTIONS.demo-genom3+= ros-server ros-client-ros ros-client-c
    PREFER_ALTERNATIVE.ros = groovy

  Then install the demo-genom3 software:
    > cd ~/robotpkg/wip/demo-genom3
    > make update

  During the install, robotpkg may encounter missing system dependencies. They
  are normally easily installed with "apt-get install <package>" (Ubuntu and
  Debian likes), where <package> is suggested by robotpkg in its error message.
  After any new dependency installation, just repeat the "make udpate" command
  until it succeeds. If eveything goes well, you should see the message
  "Done install for demo-genom3-1.1~...".


4. INSTALL GENOMIX AND TCL CLIENT --------------------------------------------

  Prerequisites: (1.) & (2.) & (3.).

  Install wip/genomix and one last package in wip/tcl-genomix that can be used
  to control the component, the TCL client:
    > cd ~/robotpkg/wip/genomix
    > make update
```

```
    > cd ~/robotpkg/wip/tcl-genomix
    > make update

  If you succeeded, you should have at least these packages installed:
    genom3-2.99.22
    tcl-genomix-1.2
    genom3-pocolibs-1.5
    genom3-ros-1.7
    genomix-1.4
  Check with this command:
    > robotpkg_info -I '*genom*'


5. COMPILE AND RUN A GENOM3 COMPONENT ----------------------------------------

  Prerequisites: (1.) & (2.) & (3.) & a GenoM3 component ready to be compiled.

  These guidelines are for compiling a GenoM3 component with ROS templates.
  (More information: https://git.openrobots.org/projects/genom3/wiki/Compiling).

  Go to the directory where the genom module you want to compile is. Let's say
  the module name is foo for this example. The folder should at least contain a
  foo.gen description file and a folder codels/ with the code.
  To compile the module, enter the following commands:
    > genom3 skeleton -i foo.gen
    > ./bootstrap.sh
    > mkdir build
    > cd build
    > ../configure --prefix=$INSTALL_DIR --with-templates=
                                          ros/server,ros/client/c,ros/client/ros
    > make
    > sudo make install

  To run the module, first launch roscore:
    > roscore &
  Then run it:
    > foo-ros -b
  (The -b option is for running the module in background).


6. INSTALL MORSE -------------------------------------------------------------
```

```
Prerequisites: (1.) & (2.).

There are several ways to install Morse, these guidelines will install it
through robotpkg. (More information:
http://www.openrobots.org/morse/doc/latest/user/installation.html).

Morse uses Blender as graphics engine, you need to get it first:
  > sudo apt-get install blender

Install Morse with ROS support.
Open the file ~/robotpkg/simulation/morse/Makefile and find the line:
  CMAKE_ARGS+= -DBUILD_ROS_SUPPORT=OFF
Turn ROS support ON by replacing OFF by ON. Then, you can install morse:
  > cd ~/robotpkg/simulation/morse
  > make update

During the install, robotpkg may encounter missing system dependencies. They
are normally easily installed with "apt-get install <package>" (Ubuntu and
Debian likes), where <package> is suggested by robotpkg in its error message.
After any new dependency installation, just repeat the "make udpate" command
until it succeeds.

For ROS support, a few other packages need to be installed (More information:
http://www.openrobots.org/morse/doc/latest/user/installation/mw/ros.html).
The following instructions will download files and install the packages. For
each instruction block, you can choose a place to download the files before
completing the instructions. At the end, if your installation succeded, you
can choose to remove the downloaded files if you do not want to keep them.

  > wget http://python-distribute.org/distribute_setup.py
  > sudo python3 distribute_setup.py

  > git clone git://github.com/ros/rospkg.git
  > cd rospkg
  > sudo python3 setup.py install

  > git clone git://github.com/ros-infrastructure/catkin_pkg.git -b 0.1.9
  > cd catkin_pkg
  > sudo python3 setup.py install

  > git clone git://github.com/ros/catkin.git
  > cd catkin
```

```
    > sudo python3 setup.py install

  You can check if the installation is OK by running:
    > morse check
  You should see "Your environment is correctly setup to run MORSE."
```

9.3 Examples for *GenoM3*: a session with the *Tcl* client and a toy module

9.3.1 Sample of a session with the *Tcl* client

Here is an example of blocking and non-blocking calls on the `demo` module (Section 4.3.3) with the *Tcl* client.

```
# Calling the GotoPosition service in a blocking way
eltclsh > ::demo::GotoPosition 1
# The service blocks the calling routine, until it is done. Its output is then
# returned (the GotoPosition actually has no output, so nothing is returned).

# Calling the GotoPosition service in a non-blocking way
eltclsh > ::demo::GotoPosition 1 &
::demo::0 # Returned is a command for retrieving the output of this call
          # later. The 0 in ::demo::0 is the request ID associated to this
          # request.
eltclsh > # The calling routine can enter new commands directly.

# Making a new non-blocking call
eltclsh > ::demo::GotoPosition 0 &
::demo::1 # As the output of service 0 has not been retreived, the request ID
          # 0 is not available. This call takes request ID 1.

# Getting the output of service 0
eltclsh > ::demo::0
# Again, as the GotoPosition service has no output, nothing is printed.
# The request ID 0 has now been cleaned. If a new call is made, it will take
# request ID 0 as it is now available.
# If the command to get the output of a service is called but the service is
# not done yet, a message tells that the request is still in progress. The
# request ID is not freed and the command is still available until the service
```

```
# is done. To sum up, a called service can have two status: "sent" or "done"
# ("done" can also be "error", meaning that the service is done and did not
# ended properly. It is the case when the service is interrupted by another one
# for instance).
```

9.3.2 A toy module

Features

Consider the design of a simple `countserver` (server) module that increments a counter on a regular time basis, and another `countclient` (client) module that connects to it, gets the value of the counter, and displays its value.

The specification of the server module is available in the file `countserver.gen`. The provided services are as follows. The counting activities run within a task named `count_task` which initializes the internal counter at 0 and then executes the codels on a periodic basis (500 ms).

`CountStart` An activity that triggers the counter from its current value.

`CountFrom` An activity to count from a value passed as input parameter of this service.

`CountStop` A function that interrupts the `CountStart` or the `CountFrom` activity. The two activities also interrupt each other. Note that the corresponding code is automatically generated from the *.gen* specification.

`SetSpeed` An attribute to set the counting speed. Four paces are available: the counter can be incremented or decremented by 1 or by 10 at each period.

`GetSpeed` An attribute to get the current counting speed.

The `countserver` module has an `out port` that publishes the current value of the counter, see file `countserverinterface.gen`. The `countclient` module is specified in `countclient.gen`. It has an `in port` which must be connected to the `out port` of `countserver`.

The `countclient` module just prints the received value on the screen, and proposes no service. Its internal task, `display_task` wakes up at every 1000 ms. This way, the effect of the selection of the respective periods of the two modules can be easily seen (*e.g.*, so that only one value of the counter out of two is displayed).

Specification files

Server: `countserver.gen`

```
/*****************************************************************************
countserver/countserver.gen: The .gen file for the counting module.
It declares a counting task with a period of 500 ms. The services CountStart
and CountFrom allow to control the behaviour of the counter. The values for the
counter and its speed are stored in the IDS.
*****************************************************************************/

#include "countserverinterface.gen"

component countserver {
    /* Module properties */
    version "1.0";
    email "tforgue@laas.fr";
    lang "c";
    provides countserverinterface;

    /* IDS declaration */
    ids {
        countserverinterface::Counter idsCounter;
        countserverinterface::Speed idsSpeed;
    };

/* Exceptions declaration */
exception INVALID_SPEED;
exception INVALID_INITIAL_VALUE;

/* Execution task declaration */
task count_task {
    period 500ms;
    stack 4000;
    codel <start> start_count_task(inout ::ids, port out CounterPort)
                                                        yield ether;
    };

/* Attributes declaration */
```

```
attribute SetSpeed (in idsSpeed) {
    doc "Sets current speed value";
    validate ControlSpeed(local in idsSpeed);
    throw INVALID_SPEED;
    };

    attribute GetSpeed (out idsSpeed) {
    doc "Gets current speed value";
    };

    attribute GetInitial (out idsCounter.initial) {
    doc "Gets counter's initial value";
    };

    attribute GetCurrent (out idsCounter.current) {
    doc "Gets counter's current value";
    };

/* Services declaration */
activity CountStart () {
    task count_task;
    doc "Starts the counter";
    interrupts CountStart, CountFrom;
    codel <start> csStart() yield exec;
    codel <exec> csExec(inout ::ids, port out CounterPort) yield exec, stop;
    codel <stop> csStop() yield ether;
    };

activity CountFrom (in short initial_value) {
    task count_task;
    doc "Starts the counter from given value";
    validate ControlInitValue(in initial_value);
    throw INVALID_INITIAL_VALUE;
    interrupts CountStart, CountFrom;
    codel <start> cfStart(in initial_value, inout::ids,
    port out CounterPort) yield exec;
    codel <exec> cfExec(inout ::ids, port out CounterPort) yield exec, stop;
    codel <stop> cfStop(inout ::ids, port out CounterPort) yield ether;
    }
    ;
    function CountStop () {
    doc "Stops the counter";
```

```
    interrupts CountStart, CountFrom;
    };
};
```

Server: `countserverinterface.gen`

```
/*****************************************************************************
countserver/countserverinterface.gen: port and structures declarations.
The port publishes the current value of the counter and its initial value.
*****************************************************************************/

interface countserverinterface {
    /* Structures declaration */
    struct Counter {
    short initial;
    short current;
    };

enum Speed {
    FASTBWD, SLOWBWD, SLOWFWD, FASTFWD
    };

/* Ports declaration */
port out Counter CounterPort;
};
```

Client: `countclient.gen`

```
/*****************************************************************************
countclient/countclient.gen: The .gen file for the display module.
It declares a displaying task with a period of 1000ms. This module has no
services, it simply runs a state-machine with codels for the task directly.
*****************************************************************************/

#include "../countserver/countserverinterface.gen"
```

```
component countclient {
    version     "1.0";
    email       "tforgue@laas.fr";
    lang        "c";
    uses        countserverinterface;

    task display_task {
        period      1000ms;
        stack       4000;

        codel <start> start_display_task() yield exec;
        codel <exec> exec_display_task(port in CounterPort) yield exec, ether;
  };
};
```

9.4 Standalone client

In preparation to writing a *MATLAB* client of *GenoM3* (section 4.2.2), consider the design of a simple standalone client program using the generic *C* client library. The main aspects of the program are the following:

- The program is a shell program interacting with the user: it prompts him/her to choose a service he/she wants to call, among those proposed by the loaded modules.
- The module's dynamic libraries are loaded with the `dlopen(3)` function.
- The main function is made of an event loop, polling on a set of file descriptors (with `poll(2)`), one for each loaded module, and an additional one for the keyboard input from the user.
- Inputs to services are written by the user as *JSON*[1] objects.
- Simple callbacks functions are declared for being executed when a service is sent and when it is done.

1 JavaScript Object Notation. c.f. `http://json.org/`

9.5 Specification (*.gen* files) of the developed *GenoM3* modules

9.5.1 *audio stream server*

```
component capture {
    version     "1.1";
    lang        "c";
    require     "genom3 >= 2.99.24";
    provides    captureinterface;

    /* ---Exceptions declaration--- */
    exception INVALID_CHUNK_TIME;
    exception ERROR_SEQUENCE_ENOMEM;

    /* ---IDS declaration--- */
    native alsaParams_t;
    ids {
        string              device;
        unsigned long       transfer_rate;
        unsigned long       chunk_time;
        unsigned long       Port_chunks;
        alsaParams_t        params;
        capture::chunk_t    current_chunk;
    };

    /* ---Task declaration--- */
    task retrieve_data {

        codel <start>   start_retrieve_data(inout ::ids) yield ether;
        codel <stop>    stop_retrieve_data(inout ::ids) yield ether;
    };

    task socket {
        period      5ms;
        priority    200;
        stack       4000;

        codel <start>   sInitModule() yield ether;
    };
```

```
    /* ---Services declaration--- */
    activity StartCapture(
        in string device =                "hw:1,0"    : "Name of the sound device",
        in unsigned long transfer_rate =  44100       : "Sample rate in Hz",
        in unsigned long chunk_time =     50          : "Size of transfer chunks in
                                                        miliseconds",
        in unsigned long Port_chunks =    20          : "Size of the Port in number
                                                        of chunks") {
        task    retrieve_data;
        throw INVALID_CHUNK_TIME, ERROR_SEQUENCE_ENOMEM;
        validate controlChunkTime(local in transfer_rate, local in chunk_time);

        codel <start> scStart(inout ::ids, local in device,
                              local in transfer_rate, local in chunk_time,
                              local in Port_chunks, port out Port)
                              yield exec, ether;
        async codel <exec> scExec(inout ::ids, port out Port) yield exec, stop;
        codel <stop> scStop(inout ::ids) yield ether;
    };

    function StopCapture() {
        interrupts StartCapture;
    };

    attribute GetCaptureConfig(out device, out transfer_rate,
                               out chunk_time, out Port_chunks);

    activity DedicatedSocket() {
        codel <start>           initModule() yield ether, recv;
        async codel <recv>      Transfer(in ::ids, port in Port) yield recv, ether;

        task    socket;
    };

    activity CloseSocket(){
        codel <start>           closeSocket() yield ether;
        task socket;
    };
};
```

9.5.2 *KEMAR* motorization

```
component kemar {
    version     "1.0";
    lang        "c";
    require     "genom3 >= 2.99.24";

    task motion {
        codel <start>   motionStart() yield ether;
        codel <stop>    motionStop() yield ether;
    };

    /*Current Position: Displays on screen and/or publshes the current
    position on a port*/
    activity CurrentPosition() {
        task motion;

        codel <start>           cpStart() yield recvCP;
        async codel <recvCP>    cpWaitForData() yield recvCP, ether;
    };

    /*Control in Position: Move Absolute Position*/
    activity MoveAbsolutePosition(in double target, in double velocity) {
        task motion;

        codel <start>           mapStart() yield sendMAP;
        codel <sendMAP>         mapSend(in target, in velocity) yield recvMAP, ether;
        async codel <recvMAP>   mapWaitForData() yield sendMAP, recvMAP;
    };

    /*Control in Position: Move Relative Position*/
    activity MoveRealtivePosition(in double target, in double velocity) {
        task motion;

        codel <start>           mrpStart() yield sendMRP;
        codel <sendMRP>         mrpSend(in target, in velocity) yield recvMRP, ether;
        async codel <recvMRP>   mrpWaitForData() yield sendMRP, recvMRP;
    };
};
```

9.5.3 *KEMAR* and *audio stream server* merged

```
component capture {
    version     "1.0";
    lang        "c";
    require     "genom3 >= 2.99.24";
    provides    captureinterface;
    codels-require "elmo-axis-libs";

    port out kemar::state currentState;
    port out kemar::position_audio_indexes_movement Indexes;

    /* ---Exceptions declaration--- */
    exception INVALID_CHUNK_TIME;
    exception ERROR_SEQUENCE_ENOMEM;

    /* ---IDS declaration--- */
    native alsaParams_t;
    ids {
        string              device;
        unsigned long       transfer_rate;
        unsigned long       chunk_time;
        unsigned long       Port_chunks;
        alsaParams_t        params;
        capture::chunk_t    current_chunk;
    };

    /* ---Task declaration--- */
    task retrieve_data {

        codel <start>   start_retrieve_data(inout ::ids, port out StateCapture)
                        yield ether;
        codel <stop>    stop_retrieve_data(inout ::ids) yield ether;
    };

    task socket {
        period      5ms;
        priority    50;
        stack       4000;

        codel <start>   sInitModule() yield ether;
```

```
};

task motion {
    codel <start>   motionStart(port out Indexes) yield ether;
    codel <stop>    motionStop() yield ether;
};

task state {
    period                  2ms;
    priority                400;
    stack                   4000;

    codel <start>           stateStart() yield sendS;
    codel <sendS>           sSend(port out currentState) yield recvS, sendS;
    async codel <recvS>     sWaitForData() yield sendS, recvS;
};

/* ---Services declaration--- */
activity StartCapture(
    in string device =                "hw:1,0"    : "Name of the sound device",
    in unsigned long transfer_rate =  44100       : "Sample rate in Hz",
    in unsigned long chunk_time =     50          : "Size of transfer chunks in
                                                    miliseconds",
    in unsigned long Port_chunks =    20          : "Size of the Port in number
                                                    of chunks") {

    task    retrieve_data;
    throw INVALID_CHUNK_TIME, ERROR_SEQUENCE_ENOMEM;
    validate controlChunkTime(local in transfer_rate, local in chunk_time);

    codel <start> scStart(inout ::ids, local in device,
            local in transfer_rate, local in chunk_time,
            local in Port_chunks, port out Port, port out StateCapture)
            yield exec, ether;
    async codel <exec> scExec(inout ::ids, port out Port) yield exec, stop;
    codel <stop> scStop(inout ::ids, port out StateCapture) yield ether;
};

function StopCapture() {
    interrupts StartCapture;
};
```

```
attribute GetCaptureConfig(out device, out transfer_rate,
                           out chunk_time, out Port_chunks);

activity DedicatedSocket() {
    codel <start>           initModule() yield ether, recv;
    async codel <recv>      Transfer(in ::ids, port in Port) yield recv, ether;

    task    socket;
};

activity CloseSocket(){
    codel <start>           closeSocket() yield ether;
    task socket;
};

/*Homing: Calls kemarHoming(h) and k=KemarStructInit(h)*/
activity Homing() {
    task motion;

    codel <start>           hStart() yield sendH;
    codel <sendH>           hSend() yield recvH, ether;
    async codel <recvH>     hWaitForData() yield sendH, recvH, ether;
};

/*Current Position: calls kemarGetInfo*/
activity CurrentPosition() {
    task motion;

    codel <start>           cpStart() yield sendCP;
    codel <sendCP>          cpSend(port out currentState, port in Port)
                                        yield recvCP, sendCP, ether;
    async codel <recvCP>    cpWaitForData() yield sendCP, recvCP;
};

/*Stop Current position*/
activity StopCurrentPosition() {
    task motion;

    codel <start>           scpStart() yield ether;
};

/*Set Velocity: calls kemarSetGearVelRadS*/
```

```
    activity SetVelocity(in double velocity) {
        task motion;

        codel <start>           svStart(in velocity) yield ether;
    };

    /*Move Absolute Position: calls kemarSetGearVelRadS, kemarSetGearPosAbsRad,
    kemarWaitMsgValid*/
    activity MoveAbsolutePosition(in double target) {
        task motion;

        codel <start>           mapStart() yield sendMAP;
        codel <sendMAP>         mapSend(in target, port out Port, port out Indexes) yield
                                                                    recvMAP, ether;
        async codel <recvMAP>   mapWaitForData() yield sendMAP, recvMAP;
    };

    /*Move Relative Position: calls kemarSetGearVelRadS,
    kemarSetGearPosRelRad, kemarWaitMsgValid*/
    activity MoveRealtivePosition(in double target) {
        task motion;

        codel <start>           mrpStart() yield sendMRP;
        codel <sendMRP>         mrpSend(in target) yield recvMRP, ether;
        async codel <recvMRP>   mrpWaitForData() yield sendMRP, recvMRP;
    };

    /*Control in Speed (Reads Velocity to set): calls kemarSetGearVelRadS*/
    activity ControlInSpeed(in double velocity) {
        task motion;

        codel <start>           cisStart() yield sendCIS;
        codel <sendCIS>         cisSend(in velocity) yield recvCIS, ether;
        async codel<recvCIS>    cisWaitForData() yield sendCIS, recvCIS;

    };
};
```

9.5.4 Virtual *PR2* on *MORSE*

```
component pr2_full{
version "1.0";
lang "c";
require "genom3 >= 2.99.24";

    port out subscriber::head Port_head;
    port out subscriber::position Port_gotoposition;
    port out subscriber::l_arm Port_l_arm;
    port out subscriber::r_arm Port_r_arm;
    port out subscriber::torso Port_torso;
    port in subscriber::sensor_msgs__Image Image;

    exception   INVALID_TORSO;

    task PublishPort{
        period      20ms;
        priority    200;
        stack       4000;

        codel <start>   InitModule() yield ether;
    };

    activity move_head(in double pan, in double tilt,
                       in double time_to_finish){
        codel <start>   cMove_Head(in pan, in tilt, in time_to_finish,
                                   port out Port_head) yield ether;
        task PublishPort;
    };

    activity go_to_position(in double x, in double y, in double w){
        codel <start>   cGo_To_Position(in x, in y, in w,
                                        port out Port_gotoposition)
                        yield ether;
        task PublishPort;
    };
    activity move_l_arm(in double l_shoulder_pan_joint,
                        in double l_shoulder_lift_joint,
                        in double l_upper_arm_roll_joint,
                        in double l_elbow_flex_joint,
```

```
                    in double l_forearm_roll_joint,
                    in double l_wrist_flex_joint,
                    in double l_wrist_roll_joint,
                    in double time_to_finish){
    codel <start>   c_Move_Left_Arm(in l_shoulder_pan_joint,
                                    in l_shoulder_lift_joint,
                                    in l_upper_arm_roll_joint,
                                    in l_elbow_flex_joint,
                                    in l_forearm_roll_joint,
                                    in l_wrist_flex_joint,
                                    in l_wrist_roll_joint,
                                    in time_to_finish,
                                    port out Port_l_arm)
                    yield ether;
    task PublishPort;
};
activity move_r_arm(in double r_shoulder_pan_joint,
                    in double r_shoulder_lift_joint,
                    in double r_upper_arm_roll_joint,
                    in double r_elbow_flex_joint,
                    in double r_forearm_roll_joint,
                    in double r_wrist_flex_joint,
                    in double r_wrist_roll_joint,
                    in double time_to_finish){
    codel <start>   c_Move_Right_Arm(in r_shoulder_pan_joint,
                                     in r_shoulder_lift_joint,
                                     in r_upper_arm_roll_joint,
                                     in r_elbow_flex_joint,
                                     in r_forearm_roll_joint,
                                     in r_wrist_flex_joint,
                                     in r_wrist_roll_joint,
                                     in time_to_finish,
                                     port out Port_r_arm)
                    yield ether;
    task PublishPort;
};

activity move_torso(in double torso, in double time_to_finish){
    codel <start>   c_Move_Torso(in torso, in time_to_finish,
                                 port out Port_torso) yield ether;

    validate    controlTorso(in torso);
```

```
        task PublishPort;
        throw   INVALID_TORSO;
    };

    task VideoCamera_RGBA{
        period      49ms;
        priority    200;
        stack       4000;

        codel <start>   InitModule() yield ether;
    };

    activity receive_image(){
        codel <start>   vcRGBAReceive(port in Image) yield start;
        task VideoCamera_RGBA;
    };
};
```

9.5.5 Virtual *KEMAR* on *MORSE*

```
component morse_kemar{
version "1.0";
lang "c";
require "genom3 >= 2.99.24";
    codels-require "opencv";

ids
    {
        double currentPosition;
};

    port in subscriber::sensor_msgs__Image CameraL;
    port in subscriber::sensor_msgs__Image CameraR;
    port in subscriber::geometry_msgs__PoseStamped RobotPose;
    port out subscriber::geometry_msgs__Twist RobotMotion;

    task ReceiveImage{
        period      20ms;
        priority    200;
        stack       4000;
```

```
    codel <start>   InitPublishPort() yield ether;
};

task Motion
{
    period      20ms;
    priority    300;
    stack       4000;
    codel <start>   InitMotion() yield ether;
};

task Position
{
    period      1ms;
    priority    100;
    stack       4000;
    codel <start>   InitPosition(inout ::ids) yield ether;
    //codel <getpos>  pGetPosition(port in RobotPose, inout ::ids) yield getpos;
};

activity Receive(){
    codel <start>   rReceive(port in CameraL, port in CameraR) yield start;
    task ReceiveImage;
};

activity Rotate(in double z){
    codel <start>   mRotate(in z, port out RobotMotion, port in RobotPose) yield ether;
    task Motion;
};

/*Control in Position: Move Absolute Position*/
activity MoveAbsolutePosition(in double target, in double velocity) {
    task Motion;

    codel <start>           mapStart() yield sendMAP;
    codel <sendMAP>   mapSend(in target, in velocity, port in RobotPose,
    port out RobotMotion) yield sendMAP, recvMAP, ether;
    async codel <recvMAP>   mapWaitForData(in target, in velocity,
    port in RobotPose, port out RobotMotion) yield sendMAP, recvMAP;
};

/*Control in Position: Move Relative Position*/
```

```
    activity MoveRelativePosition(in double target, in double velocity) {
        task Motion;

        codel <start>           mrpStart(in target, in velocity,
        port in RobotPose, port out RobotMotion) yield ether;
    };

    /*Control in Speed*/
    activity ControlInSpeed(in double velocity) {
        task Motion;

        codel <start>           cisStart(in velocity, port out RobotMotion)
                                yield ether;
    };
};
```

9.6 Low-level libraries used for the control of the *KEMAR*

The `Kemar` library		
Function	Description	Calls
`kemarHoming`	Initalizes a homing proccedure to detect RLS (Reverse Limit Sensor) and FLS (Fordward Limit Sensor)	
`kemarHomingRegConfig`	Sets registers for homing	
`kemarInit`	Initializes CAN BUS, Harmonica structure, Starts the motor and Initializes Homing	`socketcanInit`, `socketcanEnd`, `harmonicaInitCtrl` and `harmonicaStart`
`kemarSwitchesInit`	Sets internal bits as RLS and FLS	
`kemarWaitMsgValid`	Wait loop and requests messages generation	`socketcanReceiveMsgWait`
`kemarStructInit`	Initializes the Kemar Head structure (programming)	
`kemarStructEnd`	Ends the communication with the motor controller	
`kemarSetGearPosAbsRad`	Sets an absolute position in Radians for Control in Position	

`kemarSetGearPosRelRad`	Sets a relative position in Radians for Control in Position	
`kemarSetGearVelRadS`	Sets a speed in Radians/Sec for Control in Position and Speed	
`kemarSetEncPosAbsIncr`	Sets an absolute position in Increments for Control in Position	
`kemarSetEncPosRelIncr`	Sets a relative position in Increments for Control in Position	
`kemarSetEncVelIncrS`	Sets a speed in Increments/Sec for control in Position and Speed	
`kemarSetGearPosAbsVelRadS`	Sets an absolute Position in Radians and Velocity in Radians/Sec for control in Position	
`kemarSetGearPosRelVelRadS`	Sets a relative Position in Radians and Velocity in Radians/Sec for control in Position	
`kemarSetEncPosAbsVelIncrS`	Sets an absolute position in Increments and Velocity in Increments/Sec for Control in Position	
`kemarSetEncPosRelVelIncrS`	Sets an relative position in Increments and Velocity in Increments/Sec for Control in Position	
`kemarSetMotionType`	Sets motion type dynamically	
`kemarGetInfo`	Requests position, speed and status	

Table 9.1: The `Kemar` library

The `Harmonica` library		
Function	Description	Calls
`harmonicaInit`	Initializes communication with the motor controller	`socketcanTransmitMsg`
`harmonicaInitCtrl`	Requests initialization of the motor controller	
`intprtSetInt`	Send a command to the byte interpreter	`socketcanTransmitMsg`
`harmonicaSetMotionType`	Sets the bytes according to control in *Position* or *Velocity*	
`harmonicaRequestStatus`	Requests status from the motor controller	
`estimVel`	Computes a velocity estimation, based on the position difference	
`evalStatusRegister`	Interprets the contents of the controller's status register	
`evalMotorFailure`	Interprets motor failure error codes	
`harmonicaEnd`	Ends the communication with the motor controller and frees memory	
`harmonicaStart`	Starts the motor controller	
`harmonicaStop`	Shuts down the motor controller	
`harmonicaReset`	Resets the motor controller	
`harmonicaSetGearPosVelRadS`	Requests a given position and velocity and asks for data updates	`socketcanTransmitMsg`
`harmonicaSetGearVelRadS`	Requests a given velocity and asks for data updates	`socketcanTransmitMsg`
`harmonicaSetGearVelIncr`	Requests a given velocity and asks for data updates	`socketcanTransmitMsg`
`harmonicaGetPosRad`	Requests for current position	
`harmonicaDecode`	Decodes CAN messages sent by the motor's controller	`canMsgGetInt`
note: The `Harmonnica` library interfaces with the *ELMO Harmonica* motor controller		

Table 9.2: The `Harmonica` library

The `Socketcan` library		
Function	Description	Calls
`socketcanInit`	Initializes the communication with the CAN bus controller	
`socketcanEnd`	Ends the communication with the CAN bus controller	
`socketcanTransmitMsg`	Send a message on the BUS	
`socketcanReceiveMsgWait`	Waits for a message and recieves it	
`socketcanReceiveMsg`	Non-Blocking checks for next message and recieves it	
`canMsgSet`	Imitializes a CAN message	
`canMsgGetInt`	Extracts a little-endian 32 bit intiger from a CAN message at given offset	
`canMsgGetFloat`	Extracts a float from a CAN message at given offset	
`canMsgShow`	Displays the raw contents of a CAN message on stdout	
note: The `Socketcan` library interfaces with the linux socket CAN layer		

Table 9.3: The `Socketcan` library

Bibliography

Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (**1998**), "An Architecture for Autonomy," *Int. Jour. on Robotics Research* **17**, pp. 315–337. (Cited on page 11)

Algazi, V., Duda, R., Thompson, D., and Avendano, C. (**2001**), "The CIPIC HRTF database," in *IEEE ASSP Workshop on Appl. of Signal Processing to Audio and Acoustics*, pp. 99–102. (Cited on page 67)

Blauert, J., Kolossa, D., and Danès, P. (**2014**), "Feedback Loops in Engineering Models of Binaural Listening," in *67th meeting ASA*. (Cited on page 72)

Blauert, J., Kolossa, D., and Danès, P. (**submitted, 2014**), "Feedback Loops in Engineering Models of Binaural Listening," *JASA EL* . (Cited on page 72)

Bustamante, G., Portello, A., and Danès, P. (**2015, submitted, invited paper**), "A Three-Stage Framework to Active Source Localization from a Binaural Head," in *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP'2015)*. (Cited on page 72)

Ingrand, F. and Ghallab, M. (**2015, in press**), "Robotics and Artificial Intelligence: a Perspective on Deliberation Functions," *AI Communications* . (Cited on page 9)

Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., and Ingrand, F. (**2010**), "GenoM3: Building Middleware-independent Robotic Components," in *IEEE Int. Conf. on Robotics and Automation (ICRA'2010)*, Anchorage, AK. (Cited on page 11)

Portello, A., Bustamante, G., Danès, P., and Misfud, A. (**2014**a), "Localization of Multiple Sources from a Binaural Head in a Known Noisy Environment," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Chicago, Il. (Cited on page 72)

Portello, A., Bustamante, G., Danès, P., Piat, J., and Manhès, J. (**2014**b), "Active Localization of an Intermittent Sound Source from a Moving Binaural Sensor," in *Proc. Forum Acusticum*, Kraków, Poland. (Cited on page 72)