FP7-ICT-2013-C618075 Two!Ears Project

Deliverable 5.2: Second Intermediate Report on Hardware/Software Integration and Robotics Test Bed

WP5 *

November 30, 2015

* 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:	D5.2
Document title:	Second Intermediate Report on Hardware/Software Integration and Robotics Test Bed
Version:	1

Delivery date:	30th November 2015
Actual publication date:	30th November 2015
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, Antonyo Musabini, Jonathan Piat, Ariel Podlubne, Bertrand Vandeportaele
Reviewer(s):	Dorothea Kolossa, Bruno Gas

Contents

1 Executive Summary

The computational framework of auditory perception and experience designed in Two!Ears is implemented 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 is in charge of 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. Each of them must be accompanied by a comprehensive real time software architecture, entailing a modular low "functional" layer, where components run concurrently under severe time and communication constraints, and a high "cognitive" layer, where decisional processes take place. All the functional modules must be systematically submitted to extensive tests.

This deliverable documents the progress made during Year 2 towards the deployment of two robotics test beds, each one entailing the mounting on a differential wheeled robot of the *KEMAR* head-and-torso simulator endowed with a controllable neck degree-of-freedom and an anthropomorphic stereoscopic visual sensor. One of these is complete, up to the delivery of the sensor lenses, and thus provides translational degrees-of-freedom for long-range navigation as well as multimodality. The companion real time software architecture has reached a mature, stable state, and is comprehensive enough to target ambitious experiments transverse to all work packages. Its prominent elements are described, namely: an improved, self-sufficient, *MATLAB* bridge to connect the functional layer with multiple *MATLAB* cognitive processes; custom integrations of the low-level functions for locomotion and sensor handling under standard interfaces, so as to enable a transparent interchange of the robot; off-the-shelf widely used software for simultaneous localization and mapping (SLAM) and planned/reactive navigation; components for audio and visual streaming (including slight improvements w.r.t. Year 1); more involved sensory/sensorimotor functions for multiple people detection/tracking, learning/detection/segmentation of objects, sound source audio-motor binaural localization (azimuth & range) and binaural audio based sensorimotor feedback control. The deployment of the functional layer has followed the following rules, layed during Year 1: the selection of the celebrated *ROS* middleware; the use of off-the-shelf *ROS*-compliant software iff it is suitable and has been successfully tested by the robotics community; the design of components specific to the project by means of the model-driven middleware-independent *GenoM3* framework, for an improved

robustness, sustainability, and code reusability. All the described implementations have remained compliant with virtual simulations on *MORSE*.

Subsets of the real time software architecture have been successfully ported to TurtleBot robots used in the "robotics challenge" during the Two!Ears Summer School in September 2015. This deliverable also reports low-cost, MEMS microphone based, spherical binaural heads with system-on-chip acquisition and streaming, which were specifically designed and deployed for this event.

Last, a proof of concept has been set up, for the transcoding of the Auditory Front End (developed in WP2 for *MATLAB* based low-level audio processing) into a *GenoM3/ROS* functional module with improved tasks concurrency. This work will be continued during the first half of Year 3.

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 mobile 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 conducted 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, starting from hardware and going to software.

Chapter 3 reminds fundamental elements of the **robotics software architecture**. General considerations are first reviewed on how to bridge the gap between the Two!Ears conceptual model and the functional and cognitive layer of the real time software architecture supporting the deployed test beds. Elements of *ROS* and *GenoM3* are also included so as to make the report self-contained. Then, recent upgrades to this software architecture are described. These consist in a minor upgrade to the binaural audio stream server, and a fully redesigned, ergonomic and optimized *GenoM3/ROS-MATLAB* bridge, which can constitute a valuable open-source alternative to very recent proprietary *ROS-MATLAB* solutions.

Chapter 4 reports the work conducted on **hardware**. The mounting on the differential wheeled robot "Jido" (*CNRS*) of the motorized *KEMAR* head-and-torso simulator (HATS) is first explained. A similar test bed will shortly be available at *UPMC*. The chapter follows by the incorporation of an anthropomorphic stereoscopic visual sensor on this HATS. To ease the reading, **companion components of the software functional layer** are also described. These address issues such as: low-level locomotion, teleoperation and sensor handling; simultaneous localization and mapping (SLAM), path planning, localization and trajectory execution with reactive obstacle avoidance; visual data streaming, multiple people detection and tracking, as well as visual based learning, detection and segmentation of objects.

Chapter 5 describes **specific components of the functional layer for audio and audio-motor functions**. First, insights are reported to the transcoding of the Auditory Front End (AFE, developed in WP2 for *MATLAB* based low-level audio processing) into a *GenoM3/ROS* component with improved tasks concurrency. This will constitute the last part to be integrated in the deployment system. Then, audio-motor binaural azimuth & range localization, as well as binaural audio based sensorimotor feedback are presented, along their theoretical developments reported in Deliverable 4.2@month24. Their implementations are outlined, together with experiments on the Two!Ears test bed.

Last, Chapter 6 sketches the **ingredients that have been specifically developed for the "robotics challenge" of the Two!Ears Summer School** organized at *CNRS* in September 2015.

Appendix 7 concludes the manuscript.

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 the following items.

> *Design and deployment of an anthropomorphic binaural head mounted on a pan-tilt unit and installed on a PR2 mobile robot* This subtask has been completed with changes. Instead of mounting a head on a pan-tilt unit, the neck of the *KEMAR* head-and-torso simulator (HATS) has been endowed with a controllable azimuthal degree-of-freedom, as explained in Deliverable 5.1@month12. In addition, due to severe dependability problems on the two *PR2* owned by *CNRS*, distinct differential wheeled robots were selected instead at *CNRS* and *UPMC*, so as to carry the whole HATS. This is argued and explained in Chapter 4.

Equipment of this binaural head with a stereoscopic pair of cameras This subtask will be completed as soon as missing lenses are delivered to *CNRS*. Importantly, all the software part is functional. Tests could be performed because *CNRS* has been using similar micro-cameras for several months (but with fish-eye lenses, which are not suited two the needs of TWO!EARS).

Data acquisition and processing, to compute high-quality low-level audio or visual cues Several versions of a binaural audio stream server have been designed, so that a very good level of efficiency, versatility and ease-of-use has been reached. As for vision, in addition to a generic component proposed in Year 1 for acquisition, calibration, data streaming and point cloud computation from stereoscopic sensors, an off-the-shelf component has also be used. Indeed, though less versatile, is is better suited to the selected microcameras.

"System-on-a-programmable-chip" based integrated audio/audiovisual sensor No specific need of such an integrated sensor has been identified for the TWO!EARS test beds. Nevertheless, a *ROS*-compliant system-on-chip prototype of a binaural audio acquisition and streaming system for MEMS microphones has been designed and manufactured into five copies for the needs of the Two!Ears Summer School on Active Machine Hearing.

Less capable but more transportable HATS-based system The initially planned "wheel-chair"-type platform as a back-up to the test beds of *CNRS* and *UPMC* was finally not needed, due to the successful deployment of an alternative system at *CNRS* and the design of an additional system at *UPMC*, both fully compatible with the TWO!EARS software. So, this subtask has been removed, with no consequence on the project.

The end of Task 5.1 has been slightly delayed as *UPMC* will receive its TWO!EARS robot by December 2015.

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. A working full software architecture was expected at Year 2, which includes main functional and cognitive modules. This task has been completed. Within the functional layer, remaining work will be limited to the transcoding of the *MATLAB* based Auditory Front End developed in WP2 into a single *GenoM3/ROS* component for task concurrency and guaranteed computation time. Importantly, though this transcoding is planned to be ended by the first semester of Year 3, ambitious experiments transverse to all work packages can already be envisaged. Their pace may just have to be slowered in order to allow some time-consuming low-level computations.

Last, extensive tests have been conducted, along the requirements of **Task 5.3 — Modular tests and evaluations**, so that the current state of the deployment system can be deemed robust and sustainable. As aforementioned, the subsets of the real time software architecture needed for the TWO!EARS Summer School "robotics challenge" have been successfully ported to TurtleBot robots.

3 Reminder on the Robotics Software Architecture and Upgrades

This chapter first recalls essential aspects of the Two!Ears Robotics Software Architecture, so as to make the Deliverable self-contained. Then, we briefly describe important upgrades brought during Year 2. Further detailed information can be found in the Two!Ears documentation, partly included in Appendix 7.1.

3.1 Reminder on the Two!Ears deployment system

The Two!Ears computational framework of auditory perception and experience entails low-level audio processing (developed in WP2), high-level feature extraction and reasoning (WP3), and includes various sorts of feedbacks (WP4). The *development system*, primarily implemented in *MATLAB*, enables the testing of these elements on simulated data, *e.g.*, generated in WP1. The work package WP5 is in charge of the synthesis of a *deployment system* enabling the testing of concepts and algorithms against real-life scenarios defined in WP6 and in connection with WP1. This deployment system is based on robotics test beds endowed with mobility and multimodality. It is grounded in a comprehensive real time software architecture, built on the top of their instrumentation and of the encapsulation of their basic capabilities into standard interfaces. From a robotics viewpoint, this architecture involves two layers:

- The *functional layer* is composed of components which can run concurrently under severe time and communication constraints. These 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 implementation languages are *C* or *C++*, on the top of a dedicated software called middleware.

- Higher in the architecture, the *decisional/cognitive layer* hosts deliberation primitives (learning, goal reasoning, task planning, deliberate action/perception and monitoring). These abilities take place at a more abstract level, under lighter time constraints.

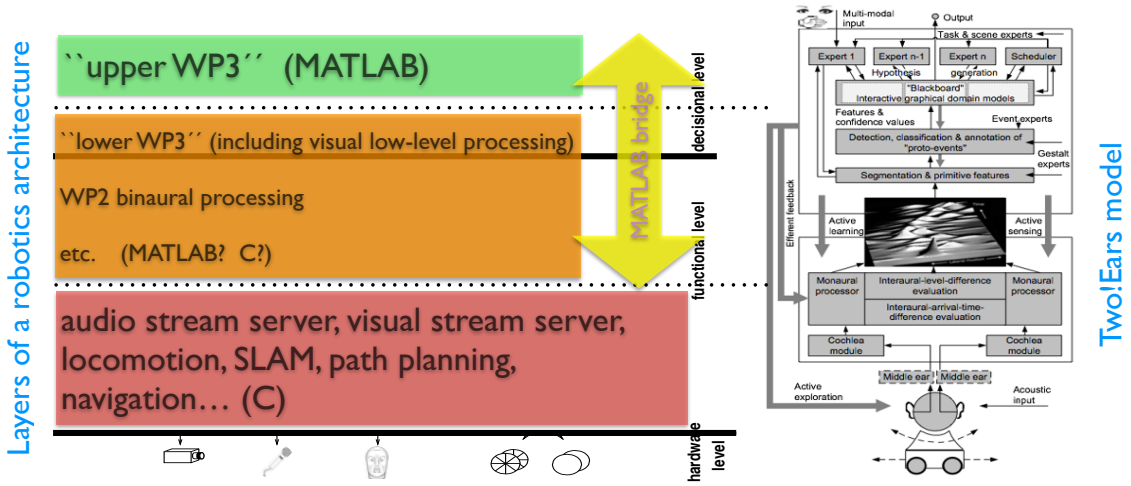


Figure 3.1: From the TWO!EARS computational model (right) to a real time robotics software architecture (left).

> They are typically implemented under an interpreted language: symbolic reasoning system, supervisor, etc. Quite uncommonly in comparison with robotics, *MATLAB* has been selected in TWO!EARS.

Ideally in robotics, a virtual environment simulator is also used, in which virtual platforms are accessed in exactly the same way as real robots, through similar interfaces. This enables the testing of decisional/cognitive concepts in fully controlled experimental conditions, assuming perfect perception and mobility, prior to integrating the comprehensive architecture on the real robots. The generic simulator *MORSE* (Modular OpenRobots Simulation Engine)[1], which development was initiated at *CNRS* and is now taken over by a worldwide team, has been selected, see Chapter 7 of Deliverable D5.1@month12.

In Year 1, it was advocated that functions of the TWO!EARS conceptual framework for robot locomotion, streaming of binaural audio or visual signals, as well as localization, navigation and SLAM, should come into the functional layer, and that the cognitive part of WP3 straightly takes place within the deliberative layer. An intermediate set of abilities can be identified in-between, whose *MATLAB* implementation can be used in the short term, but which are planned to be incorporated into the functional layer in order to tackle meaningful scenarios with maximum responsiveness. These include the Auditory Front-End (AFE) developed in WP2 for monaural and binaural processing, and visual low-level processing (*e.g.*, object detection and segmentation, human detection and tracking). Figure 3.1 outlines how to match the TWO!EARS model with a real time

1 `https://www.openrobots.org/wiki/morse/`.

robotics software architecture. Note that a specific bridge was developed during Year 1 between the upper decisional and lower functional layers, *i.e.*, between *MATLAB* and the middleware supporting the functional layer.

This chapter is organized as follows. Section 3.2 recalls the basics of component-based architectures, as well as the celebrated *ROS* middleware and *GenoM3* framework, which are the cornerstone of the deployment system. Then, Section 3.3 describes updates to the streaming of binaural signals. Finally, Section 3.4 details a major upgrade in the interface between *ROS* and *MATLAB*.

3.2 General aspects

3.2.1 A component-based software architecture

In robotics, *component-based architectures*, where components are concurrent and independent processes, have become the *de facto* standard. Each software component is dedicated to a given task, from low-level control to high-level processing. Components of the functional layer communicate with each other with the help of a software piece called the *middleware*. Two essential concepts are involved:

Data flow refers to the exchange of data between components. Data routing from one component to another is ensured by the middleware.

Control flow denotes calls to *services* that components typically provide to modify their behavior. The availability of a component's service is also handled by the middleware.

Calls to services can be emitted by any other component of the functional layer—referred to as a Remote Procedure Call (RPC). They can also be emitted by a piece of software of the decisional layer (software monitoring the state of the robot, supervisor,...) or by a user by means of a generic interpreter. Figure 3.2 illustrates these concepts on a toy model.

Component-based software architectures offer great benefits in robotics (Brooks *et al.*, 2005), addressing typical issues such as modularity (so that the architecture can be distributed over a network of host machines), re-usability (common components can be used across robots without having to recode them), scalability, and even formal proofs of dependability.

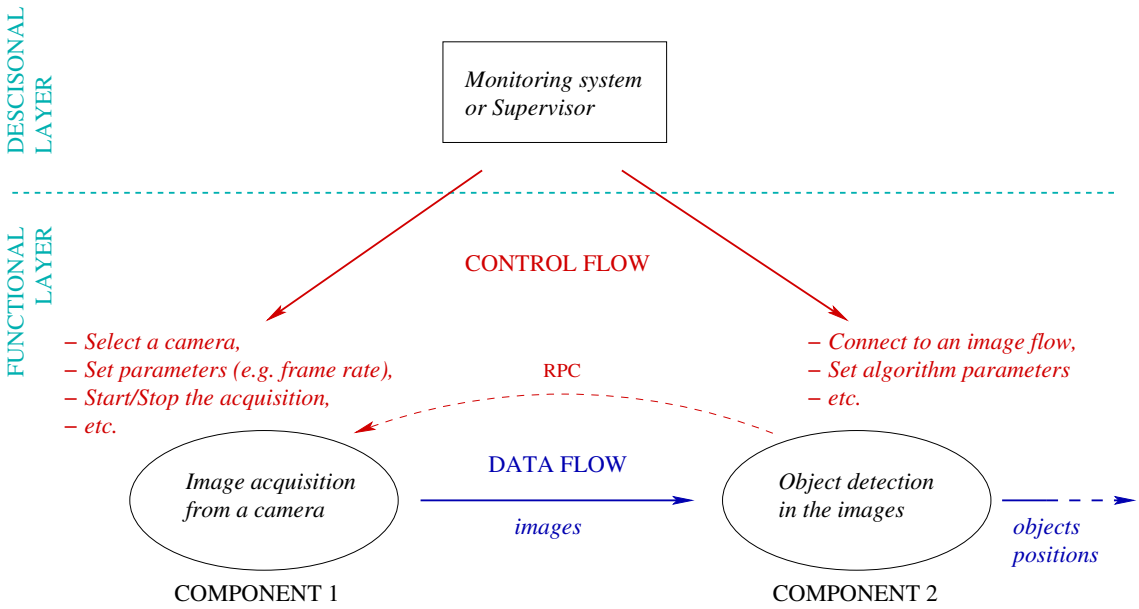


Figure 3.2: A simple component-based architecture to perform object detection in images. Two components are involved: one acquires images from a camera and streams them; the other runs an object detection algorithm. The data flow from the first component to the second one is shown in blue. Both components provide services that the decisional layer can call, shown in red. A Remote Procedure Call (RPC) is also illustrated here by a dashed red arrow.

3.2.2 *ROS*, a software platform for robotics

ROS (Robot Operating System) is a widely known software platform in robotics. It not only provides a middleware, but also implements a wide range of commonly-used functionalities into software components (such as localisation, mapping, path planning, obstacle avoidance, *etc.*), with a build system and a packaging system for easy compilation and installation. As claimed by the growing *ROS* community, *ROS* was built from the ground up to encourage collaborative robotics software development. This makes *ROS* a common choice as a robotic software platform, as it is for Two!Ears.

ROS embraces the principles of component-based software architectures, allowing concurrent and distributed computation, software reuse and rapid testing. The main *ROS* terminology is summarized here:

Nodes Software components using *ROS* middleware are called *ROS nodes*.

Topics and messages Data flows are called *topics*. A node that outputs data *publishes* on a topic. A node that inputs data *subscribes* to a topic. The data elements flowing on topics are called *messages*. Each message is made of various data fields forming part

of a data structure called *message type*. As a given topic only carries one message type, the term *topic type* is equally used.

Services and actions Nodes can provide *services* to control them. A service may take input parameters at its invocation, and may return output parameters upon its completion. Services that take a long time to execute (*e.g*, image acquisition) are rather defined as *actions*, which provide feedback mechanisms during their execution.

Software in *ROS* is organized in *packages*. A package can contain *ROS* nodes, useful datasets, configuration files, etc. *ROS* packages are themselves organized into *stacks*, which are the primary mechanism in *ROS* for distributing software. For instance, the *ROS navigation* stack contains many packages dedicated to the navigation of a mobile base in a learnt map of its environment.

The officially supported platform for *ROS* is *GNU/Linux Ubuntu*. Different versions of *ROS* exist, with compatibility restrictions on *Ubuntu* versions[2]. During Year 1, *ROS groovy* on *Ubuntu* 12.04 was used, as it was the configuration of the target *PR2* robot at that time. Now, using the *PR2* is no longer considered (*cf.* Section 4.1.1), and the target robot runs the latest stable version as of today, *i.e.*, *ROS indigo* on *Ubuntu* 14.04 (*cf.* Section 4.1.3).

3.2.3 *GenoM3*, a tool to develop robotic components

The process of developing robotic components can be significantly improved by the mean of a tool called *GenoM3* (Generator of Modules, version 3). As a result of two decades of research on real time architectures for autonomous systems (Alami *et al.*, 1998)(Mallet *et al.*, 2010), *GenoM3* brings valuable properties to robotic components:

Middleware independence Components developed with *GenoM3* are middleware independent, *i.e.*, they are not tied to a specific middleware and can be compiled for different middleware solutions without changing their source code. A clear separation of concerns between the algorithmic core and the middleware is thus conducted, helping towards the good design of robotic components.

GenoM3 can create components for the *ROS* middleware. In this case, the program built by *GenoM3* is a genuine *ROS* node.

Model-driven design *GenoM3* emphasises the clear definition of robotic components by adopting a model-driven approach. A *GenoM3* component is first defined by a description file, called the *dotgen* file, with the `.gen` extension. This file gathers

2 *cf. ROS* Enhancement Proposal 3: `http://www.ros.org/reps/rep-0003.html`.

in a single place all the definitions related to the component's interface, needed to interact with it, specifically:

- its *ports* in charge of data coming in and out of the component (analog to *ROS* topics);
- its *services*, which are either called *functions* for small operations which should be executed and finished almost instantaneously (analog to *ROS* services), or *activities* for operations that need time to perform (analog to *ROS* actions).

Functions and *activities* implement the algorithmic core of the component. They are made of atomic, non preemptable routines called *codels* (for "code elements"). A *function* consists of a single codel, while an *activity* is defined by a finite state machine, with one codel per state, running in a *task* that can be declared periodic or aperiodic, synchronous or asynchronous. Any activity can be interrupted during state transitions by any other function or activity, depending on their relative priorities.

On the basis of the model described in the *dotgen* file, *GenoM3* automatically generates real time code for tasks sequencing, as well as skeletons of codels run by the services. So, the developer just has to fill them with its algorithms. The corresponding algorithmic core is written in separate *C* or *C++* source files, possibly referring to external libraries.

Powerful framework *GenoM3* facilitates the development of essential features for robotic components, such as the definition of finite state automata with an optional clock as mentioned above, task concurrency and memory sharing between concurrent tasks, clean interruption mechanisms, and efficient error handling. It results in highly robust, sustainable, reusable and middleware-independent robotic components. Though not used in Two!Ears, *GenoM3* can also be coupled with formal validation and verification tools[3].

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

3.3 Audio streaming

3.3.1 Updates to the *Binaural Audio Stream Server*

The *Binaural Audio Stream Server* (*BASS*) is a *GenoM3* component in charge of acquiring binaural audio data from any ALSA[4]-compliant hardware sound device, and of making it available to other components of the software architecture.

BASS offers services to parameterize/start/stop the acquisition, and streams the captured data to an output port. In its capturing state, the sound device periodically delivers chunks of new data to the *BASS* component. Their size, commonly given in amount of frames[5], is set before starting the acquisition. *BASS* then pushes every new chunk on its port, so that a sliding window of the most recent data is published. For instance, the port can be configured as a FIFO[6]-like buffer which contains the last two seconds of acquired signals.

BASS was introduced in Deliverable D5.1@month12, and is now fully documented in the online Two!Ears documentation (Section 1.2.1 in Appendix 7.1). The following minor changes have been brought to Year 1 version, so that the software has reached a definitive, mature state, fitting the objectives of versatility, performance and ease-of-use:

- The component offers a new service—named `ListDevices`—to list the available ALSA sound cards on standard output. Finding the proper card identity is thus made easier.
- In order to enable its clients to keep track of the data and detect any frame loss, the component used to publish on its port an index indicating the number of chunks streamed since the beginning of the acquisition. This index is now expressed in number of frames, rather than chunks, which eases data integrity checking by the clients. The frame index is coded on 64 bits in order to prevent any overflow.
- In addition to the frame index, the possibility of adding accurate time stamping has also been examined. The ALSA Application Programming Interface (API) provides this feature for hardware devices that support it. The packaged version of ALSA on *Ubuntu* 14.04 (`alsa-lib` version 1.0.27) allows time stamping with the system's boot time as origin. If Epoch time encoding turns out to be needed, for instance for

4 The Advanced Linux Sound Architecture (ALSA) is a part of the *Linux* kernel, providing drivers for audio devices.
5 Here, a frame is defined as a pair of left and right samples at a common sampling time.
6 First In, First Out (FIFO).

data synchronisation with other systems, one solution can be to upgrade ALSA to the last version (`alsa-lib` version 1.0.29), which includes this type of time stamps.

- The component's inner terminology has been revisited, including its name. Previously called the *audio stream server*, the new name *BASS* emphasizes its use for binaural audio capture.

Aside from the genuine *Binaural Audio Stream Server*, a specific version of the component was also setup on a *Raspberry PI* on the occasion of the Two!Ears summer school, so as to acquire audio signals from MEMS microphones instead of an ALSA-compliant sound card, see Chapter 6.

3.3.2 Elements for clients of *BASS*

As recalled above, *BASS* publishes on its output port a sliding window of the most recently acquired audio data. A typical client performs block-based processing (as stated in Section 2.2 of Deliverable D2.2@month12), *i.e.*, it regularly fetches a new block from the port and processes it, then fetches the next block and so on[7]. All requested blocks usually have a fixed size, but this is not mandatory. Each block has a start date and an end date, that can be clearly defined thanks to the frame index introduced above. When reading the port to get a new block, the client must ensure two points:

1. Two consecutive blocks must be contiguous, *i.e.*, the start date of a block must just follow the end date of its predecessor so that no frames are dropped between them. To achieve this, the client's fetching period must be at most the duration of one block. Otherwise, an *overrun*[8] will eventually occur, leading to data loss (*e.g.*, an overrun will occur if the client requests blocks of 50 ms but only reads the port every 70 ms).

2. The new block must have the requested size. A single access to the port may not enable the client to get a full block. This happens if the end date of the requested block is in the future. So the client must internally save fetched parts until the block can be fully rebuilt. Naturally, the requirement, discussed in the above item, of no dropped frames between consecutive blocks also applies between consecutive parts of a single block.

A generic algorithm was written, allowing a client to fetch a block of a given size, from

7 These blocks are defined at the client level, somewhat independently of the nature of the chunks delivered by the hardware device.

8 An overrun occurs when the FIFO on the port throws away data that were not received by the client yet.

a given start date. It has been implemented in a sample *GenoM3* component named *BASC (Binaural Audio Stream Client)*, and presented in the Two!Ears documentation (Section 1.2.2 in Appendix 7.1).

3.4 Bridging *ROS* and *MATLAB*

3.4.1 Brief assessment of existing solutions

While being a popular need, the interface of *MATLAB* and *ROS* is a complex task that many projects have tried to take on[9]. It appears that none of them has gained notable popularity, mainly due to installation and usability concerns (Corke, 2015). Three main approaches have emerged:

The MEX approach wraps the *ROS C++* API in *MEX* files. This approach too often leads to compile-time and runtime errors due to incompatibilities between the versions of libraries (*e.g.*, *boost*) and *C++* compilers respectively used by *ROS* and *MATLAB*. These can be solved at the cost of rebuilding *ROS* with the same version of compilers and libraries used by *MATLAB*.

The Java approach uses the *ROS Java* API. This makes use of the possibility to write *Java* code in *MATLAB*. Though this API is not officially supported in *ROS*, it is mature enough. This approach leads to a close integration between *MATLAB* and *ROS*, benefiting for instance from automatic conversion of *Java* data types in the *MATLAB* workspace. The resulting solution is also cross-platform.

The bridging approach places a software interface between *ROS* and *MATLAB*. In this approach, *MATLAB* is not directly interfaced with *ROS*. This solution provides more flexibility, and can be easily made cross platform. The communication between *MATLAB* and the interface can rely on a protocol such as *TCP/IP*.

In January 2014, The MathWorks™ provided official *ROS* support through the *ROS I/O Package*. This *Java* based solution had notable drawbacks, such as the impossibility to call *ROS* services[10]. This is why we decided during Year 1 of the project to design a custom solution, using tools provided by *GenoM3*. This bridging approach is recalled in Section 3.4.2. In early 2015, The MathWorks™ released the *Robotics System Toolbox*

9 A *ROS* wiki page used to list existing solutions (`http://wiki.ros.org/groovy/Planning/Matlab`), but is far from being up-to-date. Also, a mailing list was setup for this topic (`https://groups.google.com/forum/#!forum/ros-sig-matlab`), but has only little activity.

10 *cf. ROS I/O Package* Getting Started Guide `https://fr.mathworks.com/hardware-support/files/ros-io-package-getting-started-guide.pdf`.

for *MATLAB* R2015a or later, featuring *ROS* support. This toolbox, also *Java* based, replaces the former *ROS I/O Package*[11], and circumvents most of its drawbacks. From our side, we improved our approach using *GenoM3* tools. Our resulting new bridge and the *Robotics System Toolbox* are compared in Section 3.4.3.

3.4.2 The solution developed during Year 1 and its upgrade

Our first solution to bridge *ROS* and *MATLAB* was called the *genomix matlab bridge*. The principle was to use *genomix*, a generic server that can receive *HTTP GET* requests, for controlling *GenoM3* components of the software architecture and reading their data flows. Compared to a *Java* based approach, this choice allows to take full advantage from the model-based design of *GenoM3* components, bringing useful information about the components directly into *MATLAB*. For instance, *MATLAB* functions to call/read each service/port of a component are automatically created and documented according to the component's model[12]. Figure 3.3 shows the interface of *MATLAB* and *ROS* components using *genomix*, along with other elements introduced further below.

Typically, two outputs are generated during the build of a *GenoM3* component, namely, the component itself and a middleware-independent *C* client library. This library provides functions for calling services of the component and reading its ports. *genomix* implements an *HTTP* interface to call functions of any component's client library. It provides high genericity through *JSON* serialization of data structures. The *genomix matlab bridge* relied on this interface. *HTTP* communication was ensured by the *Instrument Control Toolbox*[13], and *JSON* serialization used the *JSONlab*[14] open-source toolbox. A *MEX* based implementation called *matlab-json*[15] was also tested, but led to no significant performance increase.

With this solution, only *GenoM3* components could be accessed from *MATLAB*. Handling standard *ROS* nodes of the architecture implied the design of a specific *GenoM3* client, to be then accessed via the *genomix matlab bridge* as any other *GenoM3* component. To overcome this limit, an additional server called *rosix* was developed during Year 2. *rosix* implements the same *HTTP* interface as *genomix*. It uses the *ROS Python* API to call

11 The *ROS I/O Package* (for *MATLAB* R2014b or earlier) is no longer available, as stated by The MathWorks™ in `http://www.mathworks.com/matlabcentral/answers/195837-why-am-i-not-able-to-find-the-ros-i-o-package-previously-available-on-matlab-central`.

12 This information, including the name of services, their input and output type and description, comes directly from the component's *dotgen* file.

13 *cf.* `http://www.mathworks.com/products/instrument/`.

14 *cf.* `http://iso2mesh.sourceforge.net/jsonlab`.

15 *cf.* `https://github.com/christianpanton/matlab-json`

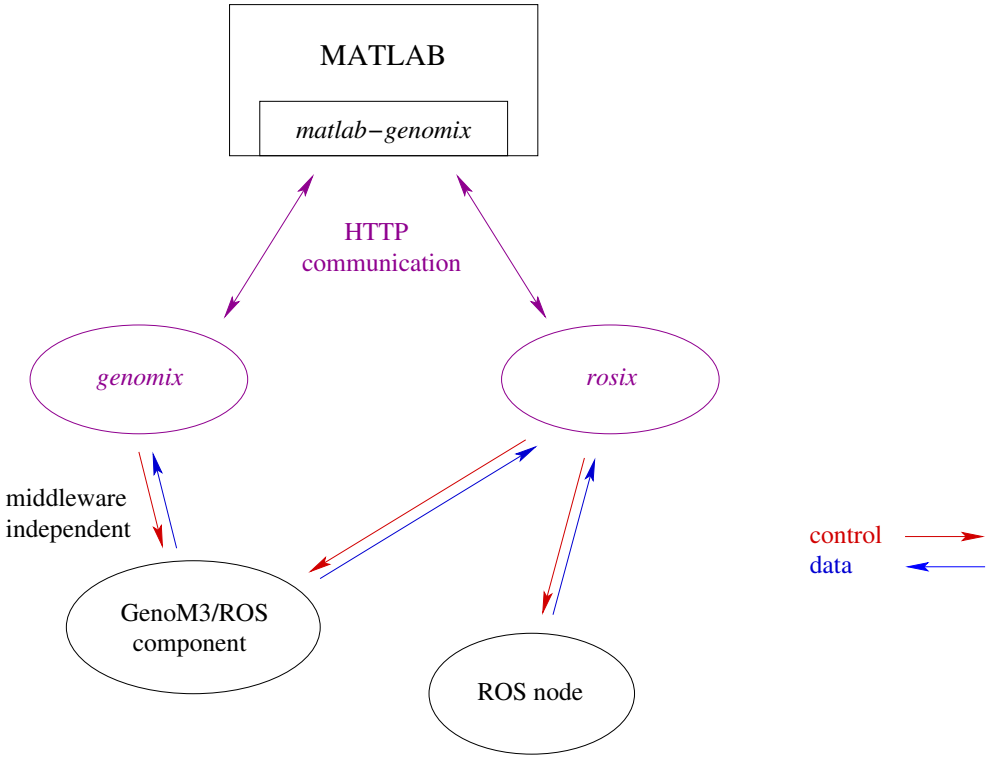


Figure 3.3: Use of *genomix* and *rosix* to bridge *MATLAB* and *ROS*. *genomix* allows to control *GenoM3* components and read their data flows independently of the middleware, while *rosix* can control and read data from any *ROS* node of the functional layer. *matlab-genomix* can be a client of any *genomix* or *rosix* server.

services and read data from any *ROS* node[16].

In addition, a new *MATLAB* client of *genomix* or *rosix* was designed, called *matlab-genomix*. It replaces the *genomix matlab bridge* with a better design that fully implements the *HTTP* interface, without relying on third-party toolboxes (such as the *Instrument Control Toolbox* or *JSONlab*). *HTTP* communication and *JSON* serialization are now coded in C and wrapped in *MEX* files. The *matlab-genomix* client currently runs on *GNU/Linux* and *Mac OS*. In the same way as *GenoM3/ROS* and native *ROS* components can be distributed across several Central Processing Units (CPUs), their clients via *matlab-genomix* can consist in one or several instances of *MATLAB*, running on one or several CPUs. For *Microsoft Windows* support, the socket interface will be adapted to use the *Windows Sockets* API.

16 Note that *GenoM3* components compiled for a *ROS* architecture are in fact *ROS* nodes, so they can also be accessed through *rosix* like any other node.

A sample use of *matlab-genomix* to access the *Binaural Audio Stream Server* from *MATLAB* can be found in the TWO!EARS documentation (Section 2.1 in Appendix 7.1).

3.4.3 Comparison between the *Robotics System Toolbox* and *matlab-genomix*

We conducted a comparison of what we consider to be the two currently prominent solutions for the integration of *ROS* features in *MATLAB*, namely, the *matlab-genomix* client or the *Robotics System Toolbox*. Some prior facts are reported on Table 3.1.

One major difference is that the *Robotics System Toolbox* considers *MATLAB* to be a component of the software architecture, whereas *matlab-genomix* acts as a supervisor (*cf.* Figure 3.2). Indeed, *GenoM3* discourages the use of Remote Procedure Calls, to guarantee that components can be controlled and will not interfere with the system. This also increases components reusability, as they do not depend on third-party services (Mallet *et al.*, 2010). As a result of being a supervisor, the *matlab-genomix* client cannot directly publish data on a *ROS* topic. Instead, the supervisor can request one component to publish some data on a topic, by making a call to a service (the data being passed as a parameter of the service).

Another notable difference lies in data handling. In the *Robotics System Toolbox*, strong data typing enables fast data transfer and computation, but also requires data structures to be known *a priori*. As a consequence, custom message types other than standard *ROS* messages are not directly supported by the *Robotics System Toolbox*. A separate interface must be installed in order to integrate them[17]. The *matlab-genomix* client does not have this issue, as any data structure is encapsulated in a *JSON* object easily parsed into a *MATLAB* structure.

17 The interface is available at `http://www.mathworks.com/matlabcentral/fileexchange/49810-robotics-system-toolbox-interface-for-ros-custom-messages`.

ROS support from the *Robotics System Toolbox*	*ROS* and *GenoM3* support from *matlab-genomix* with *genomix* and *rosix*
Proprietary and closed-source, developed by The MathWorks™.	Free and open-source, developed by *CNRS*.
For *MATLAB* ≥ R2015a.	For any *MATLAB* version.
Works on all operating systems supported by MATLAB.	*Microsoft Windows* will be supported soon.
Can publish data on *ROS* topics. *MATLAB* is considered to be a component of the software architecture.	Cannot publish data directly. *MATLAB* is seen as a supervisor.
Message types must be known *a priori*, custom messages are only possible through a separate interface.	Highly generic, data structures are de-serialized from *JSON* objects.
Strong data typing enables faster data transfer.	Data marshalling requires extra processing.
Solution for *ROS* middleware only.	Middleware-independent solution with *GenoM3* components.

Table 3.1: Summary of differences between *matlab-genomix* and the *Robotics System Toolbox* for *ROS* support in *MATLAB*.

3.5 Installation and license

3.5.1 Installation

The installation process for the needed tools of the robotic software architecture is simple and detailed in the Two!Ears documentation (Section 1.1.2 in Appendix 7.1):

- *ROS* is installed following the standard procedure on *GNU/Linux Ubuntu*.
- *GenoM3*, *genomix*, *rosix* and *matlab-genomix*, are installed through the compilation framework and packaging system *robotpkg*[18].
- *GenoM3* components such as *BASS* are compiled from source (using the *Autotools*).

3.5.2 License

Most robotic software are released under permissive[19], BSD-like licenses. *ROS* core packages for instance have the BSD 3-Clause License. In Section 2.3 of Deliverable D5.1@month12, it was stated that software from Work Package WP5 would be distributed

18 *cf.* `http://robotpkg.openrobots.org/`.

19 A permissive license allows software to be redistributed with restricted access to the possibly modified code.

under this same license. The BSD 2-Clause License[20] has been eventually choosen, because the third clause does not bring notable benefit.

Selecting a permissive license allows any other software piece to integrate or link to software from Work Package WP5 with minimal requirements. Other Work Packages can select a copyleft[21] license without any legal issue, as Work Package WP5 will not link to this software.

20 The license template is available at `http://opensource.org/licenses/BSD-2-Clause`.

21 A copyleft license requires that redistributed software remains free and open-source, and any modification or extension made to the software preserves the original rights.

4 Hardware and associated low-level software components

4.1 Binaural mobile robots

4.1.1 Discard of the *PR2* robot

The *PR2* was initially selected as the Two!Ears robotics test bed because it is an open, versatile, and rather widely disseminated platform (*CNRS* and *UPMC* respectively own 2+1 units). However, retrospectively, it has appeared to be very fragile. Since 2011, when *CNRS* received the first robot, more than forty problems have been reported on its two platforms. Quite often there have been issues with the caster control boards, arm control boards, EtherCAT hub, actuators, sensors or batteries. If such events kept on occuring, this would limit the robots availability and maybe jeopardize the project. In addition, the ClearPath Inc. support hotline fees, fixing costs and shipping costs are significant.

Besides, when the temperature in the four-caster pseudoholonomic base gets high, the associated fan (situated close to the ground) becomes very noisy. This known sporadic event implied the design of a noise cancelling cover, outlined in Deliverable D5.1. Nevertheless, the pause of experiments until the fan stops rotating could never be ruled out definitively. Last, in spite of the dissemination of the *PR2* robot advocated in the application and of the fact that it is in some sense the priviledged *ROS*-based platform, the question has remained whether it is really the right test bed for Two!Ears. Its mechanical structure is very involved, and its arms are useless to demonstrate the project goals. The mounting of the *KEMAR* head on the *PR2* head is possible only if it is separated from the torso, what significantly breaks human-like binaural perception. These points were briefly discussed during Year 1 review.

Consequently, the *PR2* has been discarded. It will be replaced by two robots introduced below, namely a working platform at *CNRS* and a platform to be shortly available at *UPMC*. Importantly, these test beds have similar kinematics, up to size: both of them are non-holonomic, with two driving wheels, and will carry a *KEMAR* Head And Torso Simulator (HATS). Although their low-level locomotion software differ from each other,

their encapsulation into *ROS* nodes will result in a standard interface, so that both can be accessed in exactly the same way from all the other functional modules and from the cognitive level of the software real time architecture. This genericity will ensure a smooth sharing of software between *CNRS* and *UPMC* during Year 3, and will enable reproducible research inside and outside the consortium. The wide variety of sensors (laser, vision, 3D,...) that can be used for *ROS* off-the-shelf Simultaneous Localization and Mapping (SLAM) and navigation components even provides some degrees-of-freedom in the equipment of the two platforms.

4.1.2 Off-the-shelf *ROS* stacks for SLAM and navigation

The nature and scenarios of Two!Ears imply that the robot can move around safely and autonomously. It must be able to localize itself in the environment and navigate between any set of location (x, y, θ) coordinates. This implies the planning of a reference path in free space, complying with the robot's kinematic contraints, as well as its reflexive execution while avoiding unexpected or moving obstacles (people, etc.). Such algorithms are now state-of-the-art, so that we used the off-the-shelf *ROS navigation* stack[1], which gathers some popular achievements. This set of packages takes as input the odometry, laser data, and a goal pose, and outputs velocity commands sent to the robot mobile base.

For the *ROS navigation* stack to work properly, the changes of frames describing the important aspects of the geometry of the robot must be described in the Unified Robot Description Format (URDF)[2], which is published by the *ROS* node *robot_state_publisher*[3] as a tree in the TF[4] format. This implies the identification of the locations (position and attitude) with respect to the robot's base frame of the laser rangefinders.

The *ROS navigation* stack can be broken into three main packages:

- *amcl* is the odometry and laser based localization algorithm;
- *map_server* provides a node that broadcasts map data on the *ROS* topic called `/map`;
- *move_base* includes a global planner to compute an admissible path between two locations, and a real time, reflexive, execution of such a path which includes the

1 `http://wiki.ros.org/navigation`
2 `http://wiki.ros.org/urdf`
3 `http://wiki.ros.org/robot_state_publisher`
4 *ROS* package that maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

detection of unexpected obstacles as well as local motion strategies to avoid them.

Figure 4.1 shows these components running on the robotic platform from *CNRS*, presented later in Section 4.1.3. Among the running components, one can see the `/amcl`, `/map_server`, `/move_base_node` and `/robot_state_publisher` nodes.

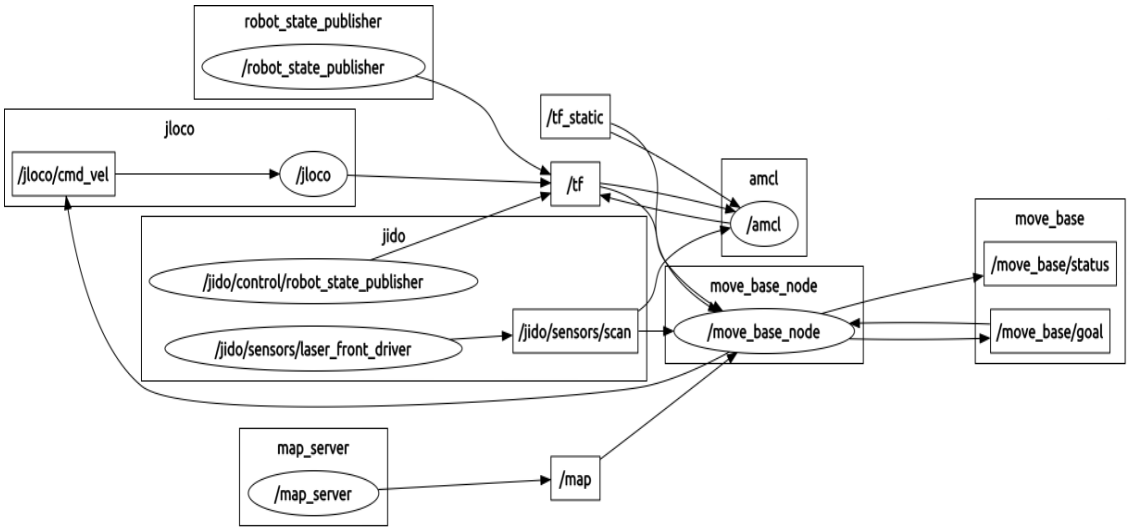


Figure 4.1: Interaction between components during a navigation session. This sample graph was obtained on the robotic platform from *CNRS*. *ROS* nodes and topics are respectively represented by ellipses and boxes. Bigger boxes that surround nodes and topics are namespaces that gather closely related resources.

Two costmaps[5] are used to store information about the world. One is meant for global planning, by creating long-range plans over the entire environment, and the other is used for local motion and obstacle avoidance. Therefore, four groups of parameters need to be configured.

- *Common costmap parameters* include parameters to set the thresholds on obstacle information (e.g. range) and on the area (named *footprint*) of the robot.
- *Global costmap parameters* include the coordinate frame the global costmap runs in, the reference frame of the base of the robot used by the global costmap and the update frequency of the costmap.
- *Local costmap parameters* include the coordinate frame the local costmap runs in, the reference frame of the base of the robot used by the local costmap, as well as

5 A costmap is a 2D map that maintains information about where the robot can navigate in the form of an occupancy grid.

the size (width and height in meters) and resolution (meters/cell) of the costmap grid (each cell of this costmap being associated with an occupancy probability of an obstacle).

- The *motion parameters* include the admissible maximum velocity of the mobile base and the tolerances to reach the goal given a plan and a costmap.

A Map building

Once all aforementioned parameters have been well tuned, a map of the environment can be built. A popular SLAM algorithm is implemented in the off-the-shelf *ROS* package called *gmapping*[6] included in a *ROS* stack dedicated to SLAM. It is a *ROS* wrapper for OpenSlam's Gmapping[7]. It relies on a Rao-Blackwellized particle filter assimilating laser and odometer measurements and combining them with the movement of the robot (Grisetti *et al.*, 2007).

Once *gmapping* is launched, the robot must be driven around the environment, *e.g.*, with the help of the joystick. The consequent map can be viewed in real time on the 3D visualization tool for *ROS* named *rviz*[8], as shown in Figure 4.2. Once the user is satisfied with the generated map, by checking in *rviz* that the area where the robot must navigate has been entirely mapped and all the static obstacles (*e.g.*, tables, chairs,...) have been detected, he/she stores it in a file for further use by the localization and navigation algorithms.

gmapping requires to finely tune some parameters (out of a total of 36), such as the maximum range of the laser sensor, the dynamic noise of the prior motion model, the number of beams to skip in each scan to reduce computation time, the number of particles and the resolution of the map (meters per grid). Some of these parameters depend on the size of the environment. For example, the resolution of the map plays a key role when the robot has to plan a path going through a door. If it is too coarse, both sides of the door may be mapped too close to each other, leaving no room for the robot to go through. The admissible maximum velocity has to be reduced as well in a small environment so as to prevent the robot from hitting a wall or having to recalculate a trajectory in case of overshooting, *i.e.*, when it undergoes a turn and cannot follow a given path.

6 `http://wiki.ros.org/gmapping`
7 `http://openslam.org/gmapping.html`
8 `http://wiki.ros.org/rviz`

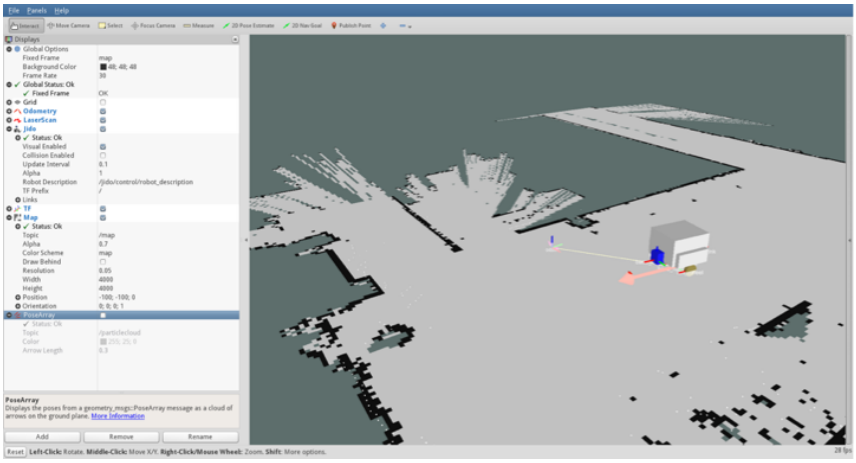


Figure 4.2: Side view of the map in *rviz*.

B Localization and Autonomous Navigation

The off-the-shelf *amcl*[9] (for Adaptive Monte Carlo Localization) *ROS* package considers a robot moving in a 2D environment which map has been built with *gmapping*. It implements a stochastic estimation of the pose of the robot into this map by means of a particle filter, on the basis of the time record of odometer and laser measurements. Three categories of parameters are entailed in the configuration of *amcl*: overall filter (minimum and maximum allowed number of particles,...), laser model (minimum and maximum scan range to be considered,...) and odometry model (differential *vs* omnidirectional, etc.)

The *ROS navigation* stack executes the following steps to safely navigate to a given location, *i.e.*, to a (x, y, θ) coordinate tuple expressed in the world frame. First, the robot is localized in the environment map. If necessary, it executes a "recovery behavior", *i.e.*, a 360° rotation around the vertical axis, to fulfill its localization. Then, the global planner computes the admissible shortest path from the current location to the goal. A trajectory controller ensures that the computed path is followed. It also runs a laser based obstacle detection algorithm. If a detection occurs, then it brings local changes to the trajectory planned by the global planner. In case of failure, the robot can even move backwards.

The *rviz* graphical visualization interface allows the user to define a goal in an intuitive way. Internally, it publishes such a tuple on a specific *ROS* topic from the *ROS navigation* stack called *move_base/goal*. For the Two!Ears project, the goal situations of the robot

9 `http://wiki.ros.org/amcl`

must be often defined at the *MATLAB* level. Therefore, a *GenoM3* component, named *sendPosition*, has been coded, which provides the user with the possibility to control the robot either in *absolute* mode (*i.e.*, with respect to the world frame) or in *relative* mode (*i.e.*, with respect to its current location). In the first case, the coordinates $(x, y, \theta)_{\text{absolute}}$ entered by the client are just published on the aforementioned *ROS* topic. If the robot is controlled in relative mode, then its current location $(x, y, \theta)_{\text{current}}$ is combined with the relative goal $(x, y, \theta)_{\text{relative}}$ in order to deduce its desired absolute location $(x, y, \theta)_{\text{absolute}}$ along the following equations:

$$\begin{bmatrix} x_{\text{abs}} \\ y_{\text{abs}} \\ \theta_{\text{abs}} \end{bmatrix} = \begin{bmatrix} x_{\text{cur}} \\ y_{\text{cur}} \\ \theta_{\text{cur}} \end{bmatrix} + \begin{bmatrix} \cos\theta_{\text{cur}} & -\sin\theta_{\text{cur}} & 0 \\ \sin\theta_{\text{cur}} & \cos\theta_{\text{cur}} & 0 \\ 0 & 0 & -1 \end{bmatrix} \times \begin{bmatrix} x_{\text{rel}} \\ y_{\text{rel}} \\ \theta_{\text{rel}} \end{bmatrix} \tag{4.1}$$

4.1.3 Robot at *CNRS*: *JIDO*

A Hardware

The currently fully functional robot, named *JIDO*, is based on a MP-700 mobile platform from *Neobotix*[10]. This non-holonomic *differential wheeled robot*[11] was part of the *CNRS* fleet, but had not been used for some years. It has three caster wheels, two in the front and one in the back to improve horizontal stability. Its maximum payload of 300 kg allows the mounting of the *KEMAR* HATS on top of it, as shown on Figure 4.3.

The driving motors and their relative encoders are connected to an Harmonica controller[12] similar to the one used for the motor of the neck of the HATS. As slipping is reduced and the encoder's resolution is high, the robot odometry is fairly accurate. *JIDO* is also equipped with two SICK LMS200 laser range finders[13], one in the front and one in the back.

To fit the needs of Two!Ears, *JIDO* has been remanufactured. The important upgrades are as follows.

- The power distribution board was changed, and the batteries status was checked.

10 `http://www.neobotix-robots.com/mobile-robot-mp-700.html`
11 A differential wheeled robot is a mobile robot whose locomotion is based on two separately driven wheels placed on either side of its body.
12 `http://www.elmomc.com/products/harmonica-main.htm`
13 `http://sicktoolbox.sourceforge.net/docs/sick-lms-technical-description.pdf`

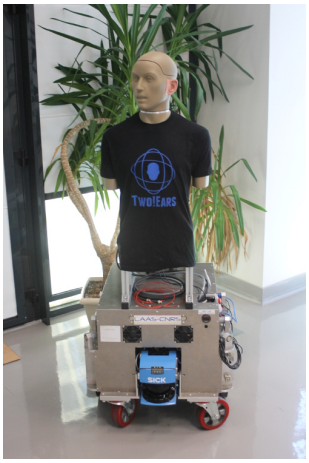


Figure 4.3: *KEMAR* HATS on *JIDO*.

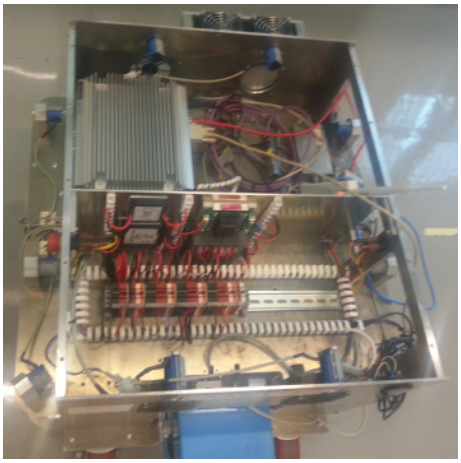


Figure 4.4: Re-wiring of *JIDO*.

- The robot was totally re-wired, with new power converters, cables, connectors, etc. (Figure 4.4).
- The three caster wheels were replaced.
- A mechanical adaptater was designed and installed, so as to carry the *KEMAR* HATS (Figure 4.3).
- A new fanless computer was inserted, featuring an Intel® Core™ i7 CPU E610 2.53 GHz processor and 4 GB of RAM. Two Control Area Network (CAN) bus interfaces were added, so as to connect sensors and actuators.

B Software: a custom *ROS* stack for *JIDO*

As the whole Two!Ears deployment system uses *ROS*, the base computer on *JIDO* runs *ROS indigo* on *GNU/Linux Ubuntu* 14.04, and a *ROS* stack named *jido_ros* was developed for low-level functions specific to this platform. Packages from the *jido_ros* stack comply with the standard interface of the *ROS navigation* stack for odometry, laser data, navigation goals and velocity commands. This allows clients to control the platform in a highly generic way. The packages constituting the *jido_ros* stack are listed below.

jido_description includes an XML representation of the robot's model in the Unified Robot Description Format (URDF).

jloco provides a node to drive the wheels. It reads velocity commands from a *ROS* topic, and publishes the robot odometry on another one.

jido_base is the main package of the *jido_ros* stack, providing launch files to quickly bring the base in an operational state. These files launch nodes provided by other packages either from *jido_ros* or standard *ROS* stacks. Notably, they load the drivers for the laser rangefinders.

jido_teleop provides a node that receives inputs from a joystick and turns them into velocity commands. It publishes the commands on the topic subscribed by the `/jloco` node, letting the joystick control the motion of the base.

jido_2dnav handles navigation by executing functions of the off-the-shelf *ROS navigation* stack described in Section 4.1.2.

Figure 4.1 introduced earlier shows components involved in a navigation session of *JIDO*. Among them, one can notice the `/jloco` node that drives the wheels of the robot, and the `/jido/sensors/laser_front_driver` node in charge of streaming laser data.

4.1.4 Forthcoming binaural robot at *UPMC*

In parallel to the development of *JIDO* at *CNRS*, *UPMC* purchased (on its own funds) a new non-holonomic differential wheeled robot dedicated to the Two!Ears project. Its mobile base is endowed with batteries, an embedded computing unit and sensing capabilities (proprioception, laser rangefinder) for autonomous navigation. It can move with a maximum linear speed of about 1m/s, and a rotational speed of 0.5 rad/s. It will carry the *KEMAR* HATS of *UPMC*, equipped with the neck motorization system. Similarly to *JIDO*, all the hardware needed to control the neck rotation, the sound acquisition, the microphone conditioning, etc. will be embedded on this platform. Like many other recent commercial robots, the low-level software will be implemented on *ROS*, so that many standard *ROS* packages (for navigation, SLAM, etc.) can be used on the top of it. The robot, shown on Figure 4.5, is expected to be shipped at the beginning of December 2015. It will then be shortly made available to the consortium for experiments.

4.1.5 Condition for an omnidirectional head

The two mobile robots presented in Sections 4.1.3 and 4.1.4 are non-holonomic differential wheeled bases with two degrees of freedom. For any one of these mobile bases, the non-holonomic constraint imposes a zero linear velocity along the axis of the wheels, *i.e.*, a

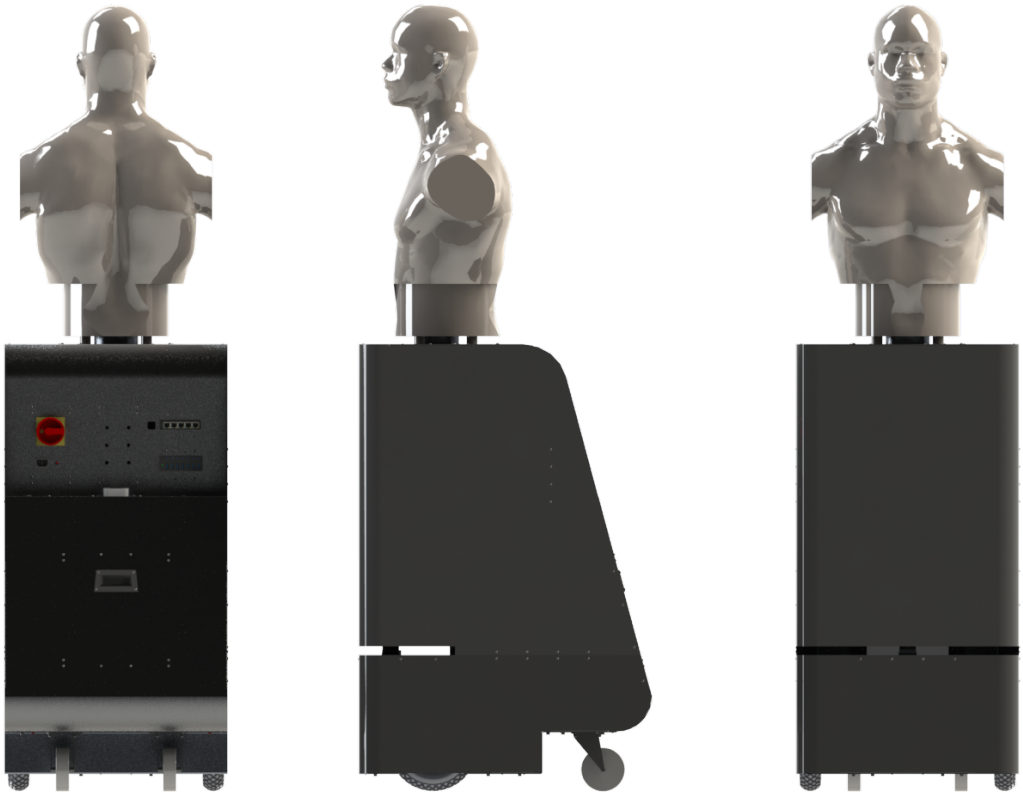


Figure 4.5: 3D preview of the robot which will be available at *UPMC* in December 2015.

zero velocity along the normal to the trajectory. We thus define v_{base}, the linear velocity tangent to the trajectory (*i.e.*, perpendicular to the axis of the wheels), and ω_{base}, the angular velocity around the vertical axis.

The *KEMAR* HATS is mounted on top of the base. In Year 1, a system made of a motor, an encoder and a micro-controller was designed and set up inside the torso to control the rotation of the neck. Associated software for its position and velocity control was encapsulated in a dedicated *GenoM3* component. This endows the overall system with an additional degree of freedom. The angular velocity of the head with respect to the base and torso is noted ω_{head}.

Some applications, such as the active localization further exposed in Section 5.2, require to express the velocity vector of the *KEMAR* head in its own frame (H, x_H, y_H, z_H), *cf.* Figure 4.6. We define v_y, the linear velocity of the head along the inter-aural axis, v_z, its linear velocity towards the boresight direction, and ω_x, its angular velocity around the

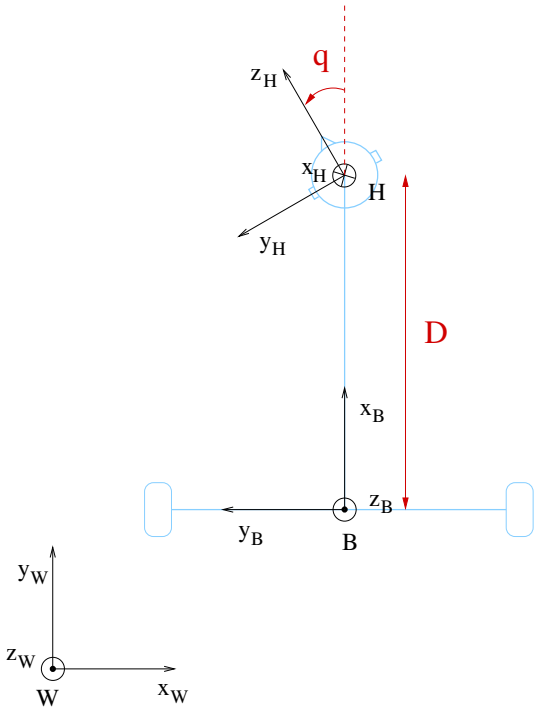


Figure 4.6: Top view of the robot. The base and the head appear in blue. W, B, H despict the world, the base and the head respectively. Points B and H are defined here in a common horizontal plane, with D the distance between them.

vertical axis. These are velocities relative to the world but expressed in the head frame[14]. An omnidirectional control of the head (v_y, v_z, ω_x) implies a joint motion of the mobile base and the *KEMAR* neck, through the three control inputs $v_{base}, \omega_{base}, \omega_{head}$. In some sense, the head is driven, and the base follows so as to carry it. Classical computations lead to (Cadenat, 1999):

$$\begin{bmatrix} v_y \\ v_z \\ \omega_x \end{bmatrix} = \underbrace{\begin{bmatrix} -\sin q & D\cos q & 0 \\ \cos q & D\sin q & 0 \\ 0 & -1 & -1 \end{bmatrix}}_{J(q)} \begin{bmatrix} v_{base} \\ \omega_{base} \\ \omega_{head} \end{bmatrix}, \tag{4.2}$$

where q is the angle of the head with respect to the torso, and D is the distance from the midpoint of the wheels to the rotation axis of the head, *cf.* Figure 4.6. The determinant of

14 As a first approximation, the rotation axis of the head is assumed vertical and included in the midperpendicular plane of the wheels. An accurate identification of the transform between the base frame and the head frame will be refined shortly, but the principle remains the same.

$J(q)$ is D, what gives a necessary and sufficient condition for the head omnidirectionality: $D \neq 0$, *i.e.*, the head rotation vertical axis must not pass through the midpoint of the wheels.

Equation 4.2 allows to compute the theoretical velocity commands $(v_{base}, \omega_{base}, \omega_{head})$ leading to any targeted head velocity (v_y, v_z, ω_x). However, in practice, some limits exist. In particular, the rotation of the head is constrained by left and right hardware stops, required to endow the neck with a human-like rotation and to avoid harms on the microphone cables. So, the resulting saturation on ω_{head} must be taken into account when piloting the head.

Note that the locations (position and attitude) with respect to the robot's base frame of the *KEMAR* torso and head (as a function of the neck azimuth degree-of-freedom) are incorporated in the Unified Robot Description Format (URDF) mentioned above.

4.2 Incorporation of the visual modality on the *KEMAR* HATS

This section describes the final design and current state of implementation of a visual system on the *KEMAR* HATS, and associated software. We decided to endow our HATSs with anthropomorphic stereoscopic vision in a non-intrusive way, by fixing cameras on 3D-printed glasses. In addition, visual functions on people and objects have also been implemented (Section 4.2.2).

4.2.1 Image acquisition by a stereo camera

A Active *vs* passive image sensors

Image sensors can fall into two main categories. Active image sensors have recourse to a modulated light source and observe the reflected light, while passive image sensors observe light regardless of its source.

On the one hand, when the aim is to recover geometric information about the scene, the accuracy and repeatability of passive systems are sensitive to many parameters. First, such systems have to be well calibrated. Second, for stereoscopic systems, textured images taken under good lighting conditions are necessary for an accurate triangulation. On the other hand, active systems such as Kinect-like 3D sensors are known to be more robust as they generate controlled lighting and do not need particular texture in the scene. They also require less complex software.

Some promising 3D point clouds based algorithms were tested on data generated by means

of 3D sensors, see Section 4.2.2. Nevertheless, the incorporation of vision on the *KEMAR* head has been selected to be in an anthropomorphic configuration. Therefore, point clouds will be generated by a stereoscopic sensor, described below.

B Hardware

B-1 Cameras The cameras chosen for the stereo vision system are the UI-3241LE-C-HQ μeye cameras[15] from *IDS* (Figures 4.7 and 4.8), with a USB 3.0 interface. They are endowed with a 1.3 megapixel, 1/1.8" CMOS sensor, with a 1280 x 1024 resolution. They provide the option of selecting either a rolling shutter (for extremely low-noise, high-contrast images) or a global shutter (for capturing moving objects). The optical size is 6.784 mm x 5.427 mm and the pixel size is 5.3 μm. Their size is $36.0 \times 36.3 \times 20.2$ mm (H×W×L) and their mass is 12 g.

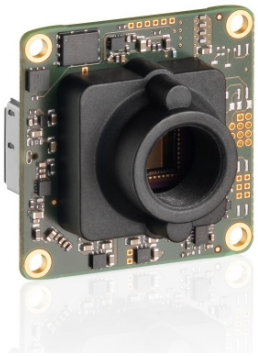


Figure 4.7: *Front side* of the *μeye camera* UI-3241LE-C-HQ.

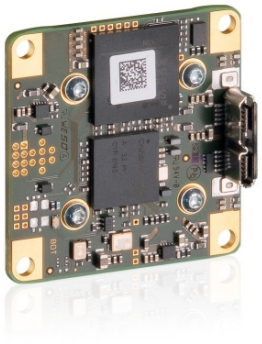


Figure 4.8: *Back side* of the *μeye camera* UI-3241LE-C-HQ.

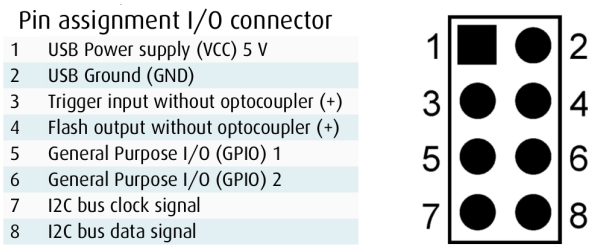
The manufacturer provides an API along with the drivers. It contains a vast number of functions to access all the parameters and functions of the *μeye camera*[16], as well as its configuration.

B-2 Hardware synchronization By default, the *μeye camera* runs in *freerun mode*. In this mode, the camera triggers itself at the frequency set by the user. For accurate 3D reconstruction by stereovision, images have to be synchronized, which requires to run

15 `https://en.ids-imaging.com/store/ui-3241le.html`
16 `https://en.ids-imaging.com/manuals/uEye_SDK/EN/uEye_Manual/index.html?c_programmierung.html`

the cameras in *external trigger mode*. In this mode, cameras are triggered by a periodic *external signal*.

Each camera is endowed with eight accessible pins for hardware synchronization (Figure 4.9). One camera, termed as the *master*, can deliver the trigger signal to both itself and the other camera, termed as the *slave*. The trigger signal is usually sent through the *flash output*. On the selected camera models, a General-Purpose Input/Output (GPIO) can be used instead, with a Pulse Width Modulation (PWM) signal generated within the master camera and routed to both cameras, as shown in Figure 4.10. This setting is preferred over the *flash output* signal, providing a highly accurate number of frames per seconds determined by the frequency of the PWM.



Pin assignment I/O connector	
1	USB Power supply (VCC) 5 V
2	USB Ground (GND)
3	Trigger input without optocoupler (+)
4	Flash output without optocoupler (+)
5	General Purpose I/O (GPIO) 1
6	General Purpose I/O (GPIO) 2
7	I2C bus clock signal
8	I2C bus data signal

Figure 4.9: *µeye camera* UI-3241LE-C-HQ pinout.

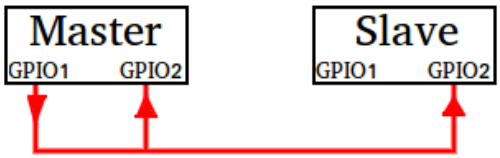


Figure 4.10: Connections to use GPIOs and PWM for external trigger.

B-3 Lenses The lenses for the cameras will be either the Lensagon B16020S12[17] or the Lensagon BSM12016S12[18]. The first reason to select them is the size, mount type and resolution of the *µeye camera*'s sensor. The second reason concerns the focal length, which has been chosen to get a reduced field of view that corresponds to a high angular precision, as each pixel corresponds to a little angle in the scene. This is to obtain 3D accuracy after triangulation.

17 `http://www.lensation.de/en/shop/detail/6-s-mount-m12x05-lenses/flypage/30-lensagon-b16020s12.html?sef=hcfp`

18 `http://www.lensation.de/en/shop/detail/7-megapixel/flypage/149-lensagon-bsm12016s12.html?sef=hcfp`

The main characteristics of the lenses are shown on Tables 4.1 and 4.2.

Image Format	1/2 inch
Mount Type	S-Mount (M12x0.5)
Focal Length	16 mm
Back Focal Length	12.3 mm
Aperture (F)	2
M.O.D.	0.2 m
Angle of View (diag.)	27.8°
IR correction	No
Weight	4.2 g

Table 4.1: Technical data of the B16020S12 lens.

Image Format	1/2 inch
Mount Type	S-Mount (M12x0.5)
Megapixel	1 MP
Focal Length	12 mm
Back Focal Length	6.54 mm
Aperture (F)	1.6
M.O.D.	0.2 m
Angle of View (diag.)	38.6°
Angle of View (horiz.)	31°
Angle of View (vert.)	23°
IR correction	Yes
Weight	6 g

Table 4.2: Technical data of the BSM12016S12 lens.

At the moment of writing this deliverable, the lenses have not arrived yet at *CNRS*. Tests leading to the final selection will be displayed on `http://homepages.laas.fr/danes/TWOEARS/VisionOnKemar/`.

B-4 3D-printed glasses A pair of glasses was designed so as to perfectly fit on the *KEMAR* face at the level of the eyes, and to incorporate the pair of the aforementioned micro-cameras with their wires and connectors. The 3D design, based on the *KEMAR* CAD model, is displayed on Figure 4.11. The final product, manufactured at *CNRS*, is shown on Figure 4.12.

Each camera has four mounting holes on the corners, as shown in Figures 4.7 and 4.8. The 3D-printed glasses are endowed with matching screw threads, defining the cameras final positions.

This provides a strong and steady structure to mount the cameras in a human-like manner, as decided within the Two!Ears consortium, with an enlarged baseline (≈ 10 cm) for an improved stereovision performance. The influence of this visual sensor on the genuine Head Related Transfer Functions (HRTFs) will be evaluated.

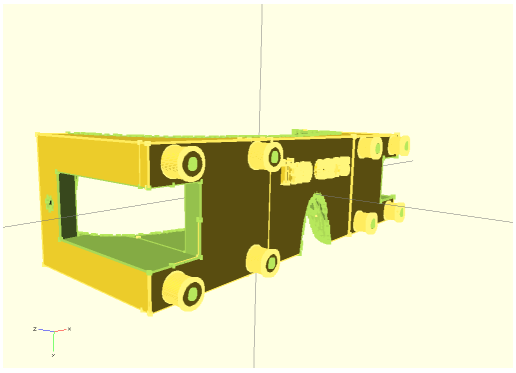


Figure 4.11: 3D design of glasses for the *KEMAR*.

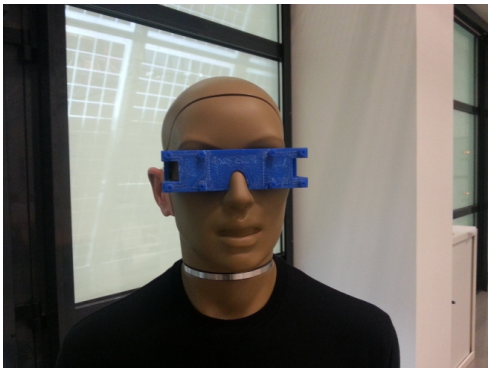


Figure 4.12: 3D printed glasses on the *KEMAR*.

C Low-level software

C-1 Associated *ROS* software To integrate the software for the *µeye camera* to the TWO!EARS architecture, the *ROS* open-source *ueye*[19] package has been used. It provides a driver node for *IDS µeye camera*. The acquisition setting in this *ROS* node uses the flash output for master and the trigger input for slave. Therefore, slight changes were brought to synchronize cameras using the GPIO rather than the *flash output*. As soon as images from master and slave are retrieved by the *ROS ueye* package, they are associated with the same time stamp (current *ROS* time) and then published on a *ROS* topic.

C-2 Calibration The *stereo_image_proc*[20] *ROS* package is used to calibrate a pair of cameras mounted in stereoscopic configuration. It uses the camera drivers to acquire images, and stream them to the vision processing nodes. It also performs rectification[21] of raw stereoscopic images. Disparities and point clouds are obtained and published as well. A diagram can be seen in Figure 4.13.

The calibration is based on the basic *OpenCV*[22] camera calibration algorithm[23], which uses the so-called *pinhole camera model*. In this model, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

19 `http://wiki.ros.org/ueye`
20 `http://wiki.ros.org/stereo_image_proc`
21 Stereo image rectification projects images onto a common image plane in such a way that the corresponding points have the same row coordinates
22 `http://opencv.org/`
23 `http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html`

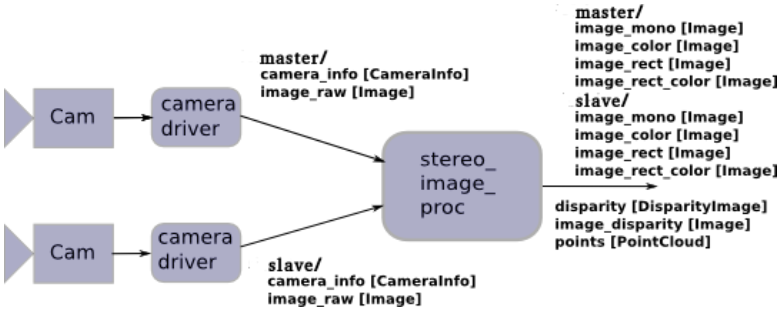


Figure 4.13: Inputs and outputs of the *ROS* node stereo_image_proc

The intrinsic and extrinsic camera parameters are obtained by moving an object with a known geometry and easily detectable feature points, as displayed in Figure 4.14, so that the camera can capture it from different views, as shown in Figure 4.15. Such object is called a calibration pattern. *OpenCV* has a built-in support for a chessboard-like calibration pattern.

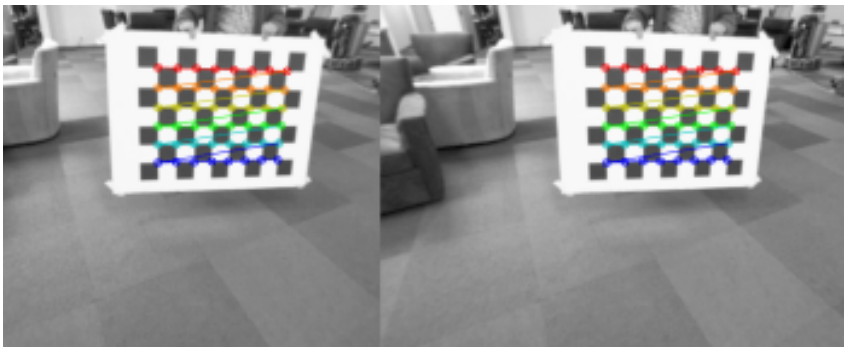


Figure 4.14: Detectable feature points during calibration

4.2.2 Visual functions

A Visual functions on people

The implementation for *detection* and *tracking* of humans is based on the open source "Online Multi-Person Tracking by Tracker Hirarchy" algorithm[24]. It is based on "Tracking-by-detection", which is a commonly used paradigm for multi-person tracking. A classifier is used to detect candidate instances of pedestrians in the current video frame. The

24 `http://cs-people.bu.edu/jmzhang/tracker_hierarchy/Tracker_Hierarchy.htm`

Figure 4.15: Different views of the calibration pattern

resulting detections are linked together, frame-to-frame, to reconstruct the trajectories of pedestrians across time. Generally, the classifier can be trained offline or adapted via an online retraining mechanism and it does not require calibration information (Zhang *et al.*, 2012). In our implementation this is done online.

This algorithm has been encapsulated into a *GenoM3* component with one activity for *human detection*. As the algorithm requires several parameters which can be set by the user, the activity takes them as input. They are described below.

- *frameRate:* The sequence frame rate in frames per second.
- *temporalSlidingWindowSize:* The size of the temporal sliding window (number of frames) for some statistic estimations. The higher it is, the longer it takes to initialize and terminate trackers.
- *detectorFrameRatio:* The frame rescaling ratio for the detector.
- *maxTrackNumber:* The maximum number of tracks in the sequence.
- *maxTemplateSize:* The maximum template size for the tracker. Recommended value: 10 or lower to save computation.
- *expertThreshold:* The minimum template size for an expert tracker. This parameter should be lower than maxTemplateSize.
- *detectionRescalingFactor:* The rescaling factor for the detection boundingbox.
- *trackRescalingFactor:* The rescaling factor for the track boundingbox.

The period of this activity is defined as 1/FPS, where FPS is the frequency of the PWM

used to trigger the cameras. Tests showed that at 30 FPS, no frames are dropped and a successful synchronization is achieved. Therefore, every time a new pair of frames is available on the corresponding *ROS* topic, it is read by this *GenoM3* component.

Then, one frame at a time is passed as a parameter for the detection and tracking function to obtain 2D coordinates of the detected people. With this information, it is possible to obtain 3D coordinates by following a triangulation with the 2D coordinates obtained from the master and slave frames. After this, they are published in a *ROS* topic to be retrieved in *MATLAB*. An example of detection and tracking is shown in Figure 4.16.

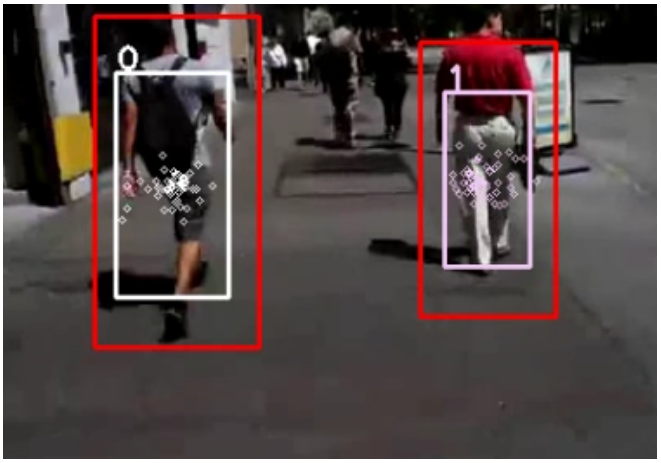


Figure 4.16: People detected and tracked in one image frame.

B Visual functions on objects

Linemod is a multi-modal object detection algorithm. It is implemented in the *OpenCV* library. As such, it relies on various modalities, in particular color gradients and surface normals computed from a RGB-D point cloud (Figures 4.17 and 4.18). Therefore, all the concepts displayed above about calibration are applied here too, as one of the outputs from the calibration shown there is in fact a point cloud.

The *linemod* approach has been developed to detect known textureless objects in cluttered scenes. It uses a template matching approach for object detection. An initial learning step consists in acquiring RGB-D point clouds of the object from various viewpoints. They are called templates and their union is named the object model. For each viewpoint, features (color gradients and surface normals) are extracted from the point cloud and saved as a template. The new templates are added to a database of templates.

Figure 4.17: Example of an RGB-D point cloud.

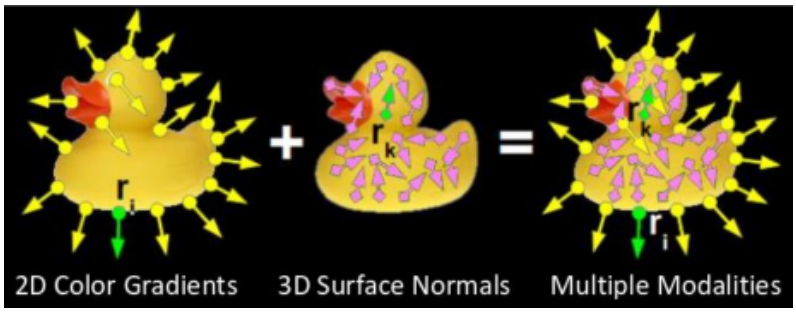


Figure 4.18: Features extracted from a point cloud.

At detection time, the templates are matched against the incoming point cloud with a multi scale sliding window approach.

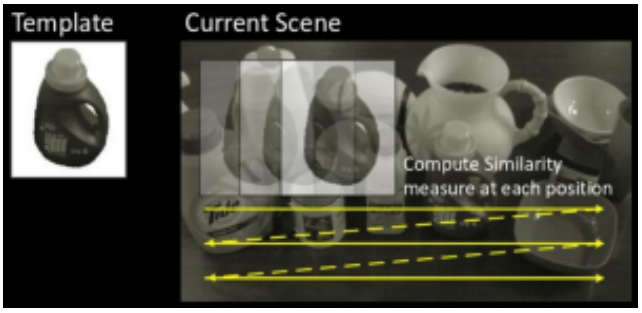


Figure 4.19: Template matched with incoming point cloud.

B-1 Modeling A *ROS* package was developed to build the models of the objects to be detected. It takes a point cloud as argument. It segments the biggest visible plane and

removes it from the point cloud. Anything more than $1m$ above the plane is removed. Points closer than $30cm$ or further than $2m$ are also removed. What is left is just the object.

rqt_reconfigure[25] is a *ROS* Graphical User Interface (GUI) which allows the user to set the parameters of a *ROS* node online. In this case, it is used to limit the area of the field of view of the camera from where the object will be modeled. This is shown on Figures 4.20 and 4.21. In the first figure, the setup is shown. The object to be modeled can be seen, in

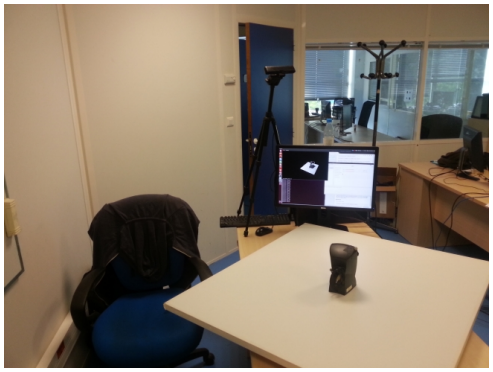


Figure 4.20: Modeling scenario.

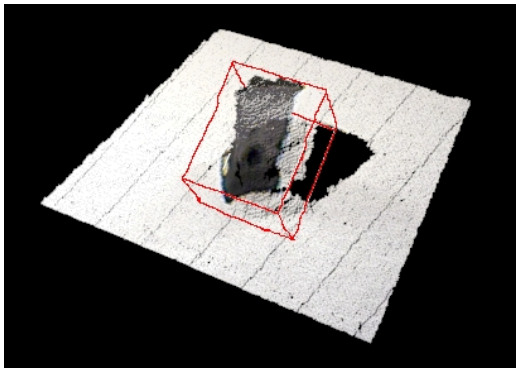


Figure 4.21: Bounding box around the detected object.

this case a computer's loudspeaker. It sits on top of a flat surface which is centered on a turntable. In the second one, the result of the limited area is shown, as well as a bounding box around the point cloud being segmented and modeled.

A fast processing requires a light model. Conversely, a model must also include as many different views of the object as possible for an accurate detection. One strategy to get a light and good model is to use various camera heights with respect to the object.

Even though Figure 4.20 shows the modeling process with the Xtion 3D sensor from Asus[26], the approach remains based on point clouds. Consequently, it can be applied in the same manner with a stereovision system.

B-2 Detection A *ROS* package has also been developed to perform the detection of modeled objects. It takes as arguments a point cloud and the list of all the models that

25 `http://wiki.ros.org/rqt_reconfigure`
26 `https://www.asus.com/3D-Sensor/Xtion_PRO`

have been modeled as described above. It outputs two 3D points in the sensor frame which delimit a bounding box around the detected object. The position of the object in the world frame is obtained based on these two values. Note that the axes of the above bounding box are parallel to the sensor frame, so that no orientation of the object can be extracted. Figure 4.22 shows several objects on a table. The ones that were previously modeled are detected and tracked.

Figure 4.22: Two modelled objectes detected.

5 Components for audio and audio-motor functions

5.1 Bringing the Auditory Front-End into the *ROS* architecture

Many aspects of the Two!Ears architecture are implemented in *MATLAB*. This is especially the case for the higher, cognitive level, taking decisions based on the descriptors extracted from the raw binaural signals. On the other hand, robotic components (such as locomotion, audio acquisition and streaming) are part of the functional level of the architecture, see Chapter 3. As such, all these operations must be performed in real time to provide a robust and reactive system, able to face dynamic acoustic conditions. Consequently, such a functional level is rooted on *ROS*.

The Auditory Front-End (AFE) is placed in-between. It is in charge of turning the monaural/binaural input signals—originating from the functional layer—into auditory cues and higher level features exploited by the decisional layer. For now, a *MATLAB* implementation of the AFE is entailed in the Two!Ears development system. Its modular, object-oriented, design allows great flexibility. It provides bottom-up, signal-driven, processors, together with a manager object to instantiate and route them, see Deliverable 2.2@month12. Top-down feedback is also supported, *e.g.*, by enabling on-the-fly changes in the values of the processor parameters. However, the AFE has not been implemented with real time constraints in mind: no concurrency is available between processors, and guaranteed computation time can hardly be satisfied when extracting a lot of features. Its use in experiments involving reflexive behaviors (emergency routines, obstacle avoidance, audio-motor functions, sensorimotor feedback, etc.) would thus imply to dramatically reduce the pace of the scenarios, at the possible expense of making them unrealistic. To target realistic search and rescue scenarios with maximum reactivity, the Two!Ears deployment system thus requires an implementation of the AFE right at the functional layer, supported by the *ROS* middleware.

5.1.1 *C/C++* implementation of the AFE algorithmic core

To make the AFE processors exchange data with each other through shared memory rather than by time-consuming TCP/IP communication, we have decided to implement the AFE as a single *ROS* node. Various ways to produce the underlying algorithmic core into *C/C++* code are investigated in the following.

A Automatic *C/C++* code generation under *MATLAB*

A first way to implement the algorithmic core of the AFE *ROS* node is to automatically generate *C/C++* code from the original *MATLAB* implementation. *MATLAB* can perform such a task thanks to the *MATLAB Coder toolbox*[1]. This generates readable source code which can then be compiled to form libraries or executable files. Importantly, the resulting *C/C++* code can also be straightly called from within *MATLAB* through *MEX* files, what results in faster execution time. For instance, the *C* transcoding of the `ild` processor from the original AFE toolbox leads to a shortening of the *MATLAB* execution time by a factor of 5. Nevertheless multiple limitations make this solution unworkable.

- The generated code is not optimized. One line of *MATLAB* code is generally replaced by one line of *C/C++* code coupled with dozens of files including type description, while code refactoring could basically lead to far more efficient *C/C++* coding.
- The generated code is not easily maintainable. The code generation system introduces some new, non-standard data types, which are dedicated to a *MATLAB* use. These data types are not well-documented and seem redundant with standard data representation originating from the *C/C++* standard libraries.
- Some specific functions provided by *MATLAB* toolboxes cannot be straightly translated into *C/C++* code. These functions rely on closed-source *MATLAB* libraries which must be linked with the object file obtained with the *C/C++* code. The resulting software cannot then be entirely open-source, since the automatically generated code must be linked against proprietary *MATLAB* libraries.

For all these reasons, this automatic *C/C++* code generation from *MATLAB* has been discarded. Consequently, we decided to recode a specific AFE in *C/C++* from scratch, but keeping highly inspired by the structure of the genuine *MATLAB* implementation. Interestingly, a lot of open source *C/C++* APIs dedicated to audio processing already exist in the literature, which can ease the implementation of basic processing algorithms.

1 `http://fr.mathworks.com/products/matlab-coder/`.

Some are listed below.

B Third-party audio processing libraries

The cornerstone of the proposed *C/C++* implementation of the AFE is the *C++ Standard Template Library (STL)*[2], which is designed to exploit templates and handle generic types. This library is recognized by the Internal Organization for Standardization, and provides a set of common *C++* classes implementing vectors, maps, lists, complex data, etc. Audio signal buffers can be coded by using the circular buffer implementation of the *BOOST* library[3]. Among other features, *BOOST* provides smart pointers to instantiate *C++* objects, which are very useful to prevent memory leak problems. In addition, the *GNU Scientific Library (GSL)*[4] provides a wide range of mathematical routines, like Fast Fourier Transform (FFT) computations based on the Fortan *FFTPACK* library. FFT computations can also be provided by the *FFTW* library[5], which is actually used inside *MATLAB* `fft` and `ifft` functions.

All the above libraries are devoted to generic mathematical or coding considerations. The following ones are more specific to audio processing. The *JUCE* library[6] provides a wide-range of *C++* classes for building rich cross-platform applications. While it can be used for other purposes than audio processing, it provides a large set of functionalities dedicated to audio, like the *audio graph*, which is very similar to the manager object in the *MATLAB* implementation of the AFE. Though very promising, the *JUCE* library mostly proposes single-threaded implementations, what prevents tasks concurrency. In the same vein, the *Audio Processing Framework (APF)*[7] is a collection of *C++* code for multichannel audio applications. Originally developed by URO and TUB, this library proposes bi-quad and cascade filters design, convolution algorithms, multiple input/multiple output audio processors, etc.

For now, all those libraries have been selected since they provide data representations, filters and algorithms which are suited to the re-encoding of the genuine *MATLAB* AFE. The original *MATLAB* implementation of the AFE and some libraries cited here (*GSL*, *FFTW*, *JUCE*, *APF*) are released under the copyleft *GNU General Public License*. So, the same license is given to the *GenoM3/ROS* AFE component.

2 `https://www.sgi.com/tech/stl/index.html`.
3 `http://www.boost.org/doc/libs/1_59_0/`.
4 `http://www.gnu.org/software/gsl/`.
5 `http://www.fftw.org/`.
6 `http://www.juce.com/`.
7 `http://audioprocessingframework.github.io`.

5.1.2 *C/C++* implementation of concurrency between processors

A Overview

A *C/C++* version of the AFE requires a clear definition of how a processor can connect to other processors, how processors can be executed concurrently to optimize computational cost, etc. The AFE structure comes as a tree of processors, an instance of which is shown in Figure 5.1. A processor whose output is routed to another processor's input is henceforth called *parent* while the second one is called *child*. A processor can have multiple children, and multiple parents as well[8]. Only the root processors, *i.e.*, those which have no parents, can read ports of other components of the robotics architecture. From this structure, two kinds of concurrency between processors can be highlighted.

Vertical concurrency While a processor works on a resource delivered by its parent, the parent can already prepare the next resource. This kind of concurrency is outlined in red in Figure 5.1.

Horizontal concurrency Children of a processor are mutually independent and can process concurrently their parent's output. This kind of concurrency is outlined in blue in Figure 5.1.

B Formal design

A processor takes an input resource from its parent and produces an output resource to its children. Each child could make its own local copy of the output resource, but this would lead to high memory needs in a tree that involves many children of a processor. Instead, we propose that children of a same parent share read access to a single memory zone, managed by the parent. In addition, the parent processor owns a private memory zone for its internal computation. With this memory management plan, each processor can be formalized by a state machine, as shown in Figure 5.2. A processor goes through four distinct states, in a loop:

Ready The processor is ready to read a new input resource, coming from its parent.

Process As soon as its parent releases the resource, the processor performs its computation. It reads the input resource from its parent's shared memory zone, and stores the result of the computation—its output resource—in its own private memory zone.

8 If one processor has multiple parents, then the structure of the AFE is not a tree, but an oriented graph. In this case, the only constraint is that the graph, oriented from parent(s) to child(ren), has no cycle.

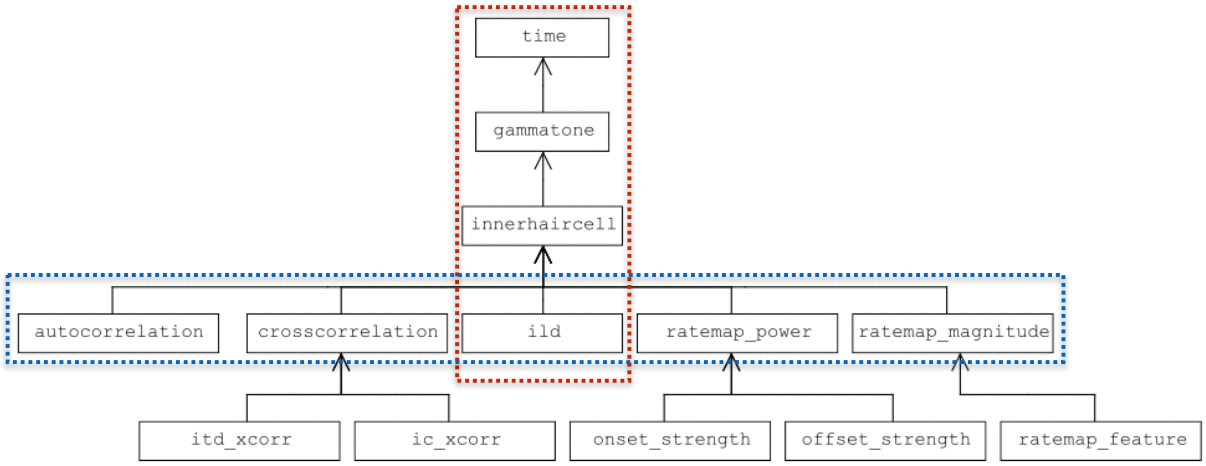


Figure 5.1: Tree of processors. Each processor is represented as a box, which can be connected to one other. In this tree, `innerhaircell` is the child of `gammatone`, and `gammatone` is then the parent of `innerhaircell`. A processor can have multiple children (see the `innerhaircell` processor) and multiple parents (not illustrated here).

Wait The processor stays in a waiting state while its children are still processing the previous output resource it has released. Children lock the processor's shared memory zone.

Release Once all children are done processing the previous output resource, the processor can release the new one: it copies the content of its private memory zone to its shared memory zone.

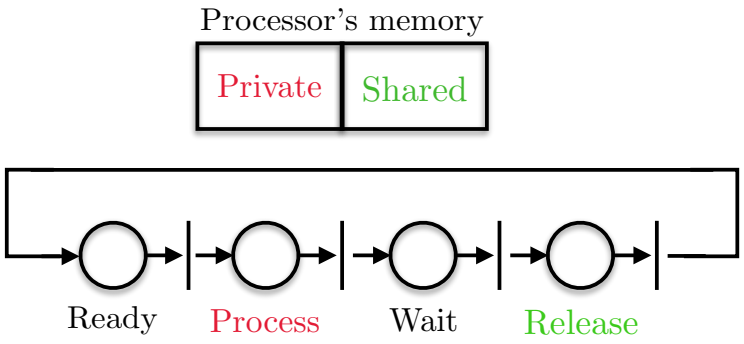


Figure 5.2: State machine and memory management of a processor

On this basis, the two aforementioned kinds of concurrency work as follows.

Vertical concurrency Figure 5.3 illustrates vertical concurrency through adequate synchronization between processors. `processor 2`, whose state machine appears in the middle, has one parent on its left and one child on its right. In this Petri net, interaction between the processors is outlined in red by several synchronization states:

1. While in *Ready* state, `processor 2` needs a token issued after the *Release* state of its parent—`processor 1`—in order to fire the transition to the *Process* state.
2. While in *Wait* state, `processor 2` needs a token issued after the *Process* state of its child—`processor 3`—in order to fire the transition to the *Release* state.

In view of the sample tree in Figure 5.1, this case could be applied for instance to the `time`, `gammatone` and `innerhaircell` processors.

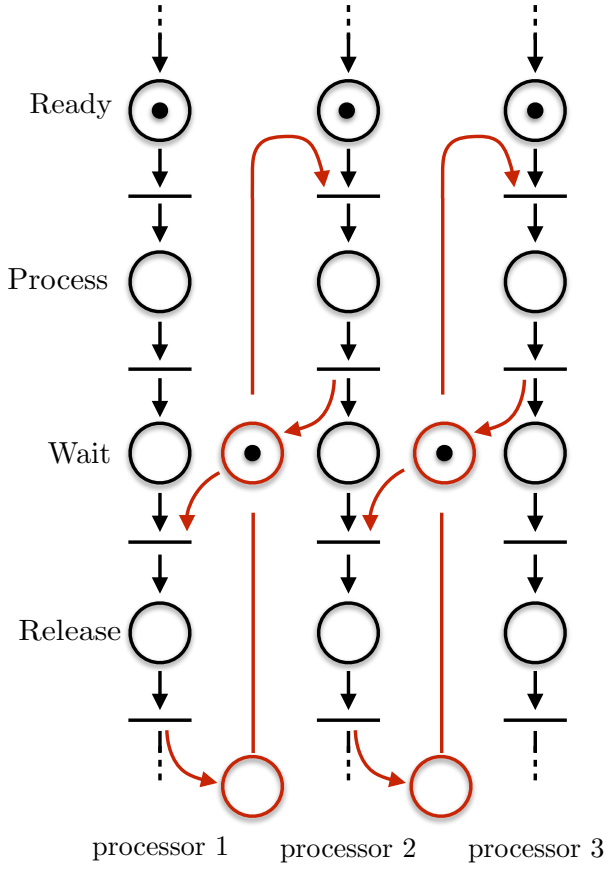


Figure 5.3: Petri net for a serial chain of three processors (vertical concurrency). `processor 1` is a parent of `processor 2`, itself parent of `processor 3`. Tokens are displayed here in the initial marking position.

Horizontal concurrency Figure 5.4 illustrates horizontal concurrency using the same synchronization mechanisms:

1. When the parent processor—on the left—leaves its *Release* state, it issues individual tokens allowing each child—on the right—to fire the transition from *Ready* to *Process* state.

2. Once a child leaves its *Process* state, it issues one token. The parent processor needs as many tokens as it has children (two, here) to fire the transition from *Wait* to *Release* state.

This case could be applied to an `innerhaircell` parent processor with two different `ild` child processors.

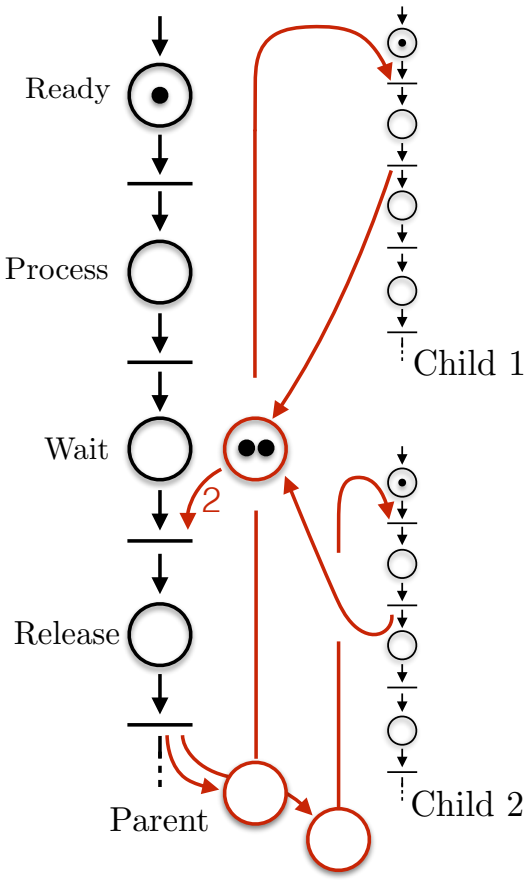


Figure 5.4: Petri net for a parallel chain of two processors (horizontal concurrency). Tokens are displayed here in the initial marking position.

C *GenoM3/ROS* implementation

In the *MATLAB* implementation of the AFE, all processors run in sequence in a single thread. So, any processor must wait until all other instantiated processors have ended up their local processing before it can move on with its next block of data. Contrarily, *GenoM3/ROS* enables concurrent processing. In view of the many concurrency and synchronization properties that must occur within the *ROS* node implementing the AFE, the common approach would be to use a dedicated library to handle these aspects, such as *POSIX Threads*. Nevertheless, we decided as a first step to resort to *GenoM3* as it brings invaluable benefits for rapid prototyping, thanks to the synthetic description of the component in its *dotgen* file and the automatic generation of real time code for the used middleware (Section 3.2.3). Three of them are recalled below.

Task concurrency Each activity of a *GenoM3* component is associated to a task in charge of its progress. The concurrency of multiple tasks entailed in a component is included in the automatically generated real time code, and is transparent to the user. For the *ROS* middleware, this is implemented as the concurrent execution of one thread per task.

We propose an architecture in which the processors are distributed on independent tasks, *i.e.*, independent threads when using *GenoM3/ROS*. Depending on the number of cores, the host machine can either perform parallel processing or task switching to run these threads concurrently.

Specification of an activity as a state machine An activity is specified as a standard finite state automaton, which can include the sequence of states composing its nominal behavior and the transitions relating them, as well as degenerated situations (interruptions, failures, etc.). Changes to this design are effortless as they imply no additional programming.

As it appeared previously, vertical and horizontal concurrency can easily be formalized with state machines. Thus we propose to take advantage of the definition of *GenoM3* activities as finite state automata for a straightforward implementation of the design exposed in Section B.

Memory sharing between concurrent tasks A *GenoM3* component can contain an Internal Data Structure (IDS), defined in its *dotgen* file. Data in the IDS are shared between tasks of the component. Concurrent access to this memory area is safely handled by the automatically generated code.

By nature, vertical and horizontal concurrency reveal the need of synchronization signals between processors. The best way to synchronize state-machines of different

processors is by using *locks* and *semaphores*[9]. As aforementioned, these thread control mechanisms can be handled by a library such as *POSIX Threads*. In order to have a quick and lightweight prototype, we rather attempted an active polling approach. A set of flags is defined in the IDS of the *GenoM3/ROS* AFE component, by which parents notify their children that a new resource is available, and children notify their parents that they are ready to read a new resource. In *Ready* and *Wait* states, processors periodically check a flag and, when it reaches an adequate value, enable the transition to the next state.

The resulting version of the *GenoM3/ROS* AFE component is closely integrated with the state machines which specify the activities of the component. A second ongoing version will include modern thread communication systems through a dedicated API which will define all the communications between processors in a normalized and generic way. But whatever the approach to implement threads control, the architecture, state machines, etc. will remain the same.

5.1.3 A proof of concept

In order to demonstrate the effectiveness of the model, a Proof of Concept (PoC) has been written. It implements the state machine previously exposed. It is described in a *dotgen* file, which is then used to generate the skeleton of the *GenoM3/ROS* AFE component. This node is then connected to the *Binaural Audio Stream Server*. Only one kind of processor is used in this PoC: a basic Butterworth filter, whose parameters can be freely chosen. Its implementation relies on the C/C++ libraries mentioned in Section 5.1.1-B. The PoC relies on the active pooling method for concurrency, with all processors verifying every 100ms the state of the synchronization flags. In short, the PoC is able to:

- choose the origin of the incoming audio chunk, between a real acquisition device (RME Babyface endowed with two microphones as inputs), or simulated data;
- add/modify processors from the process tree: processors can be added at any time to the process tree and connected to any already existing processor; multiple processors can be connected to the same output (i.e. to the same parent) without any problem; the parameters of the processors (in the PoC, the filters parameters) can be changed at any time too;
- plot and listen to the signals in *MATLAB*: all the information present into the buffer

9 A lock will ensure that only one single thread has access to a resource at the same time, while semaphores are kind of switches which can deactivate a thread until it should run. Importantly, a deactivated thread then consumes zero CPU cycle when waiting for new resources.

of any processor can be plotted and listened to directly from the graphical interface.

A screen capture of the PoC GUI is shown in Figure 5.5. A video presenting its functioning is available at the URL `https://goo.gl/djRuuW`.

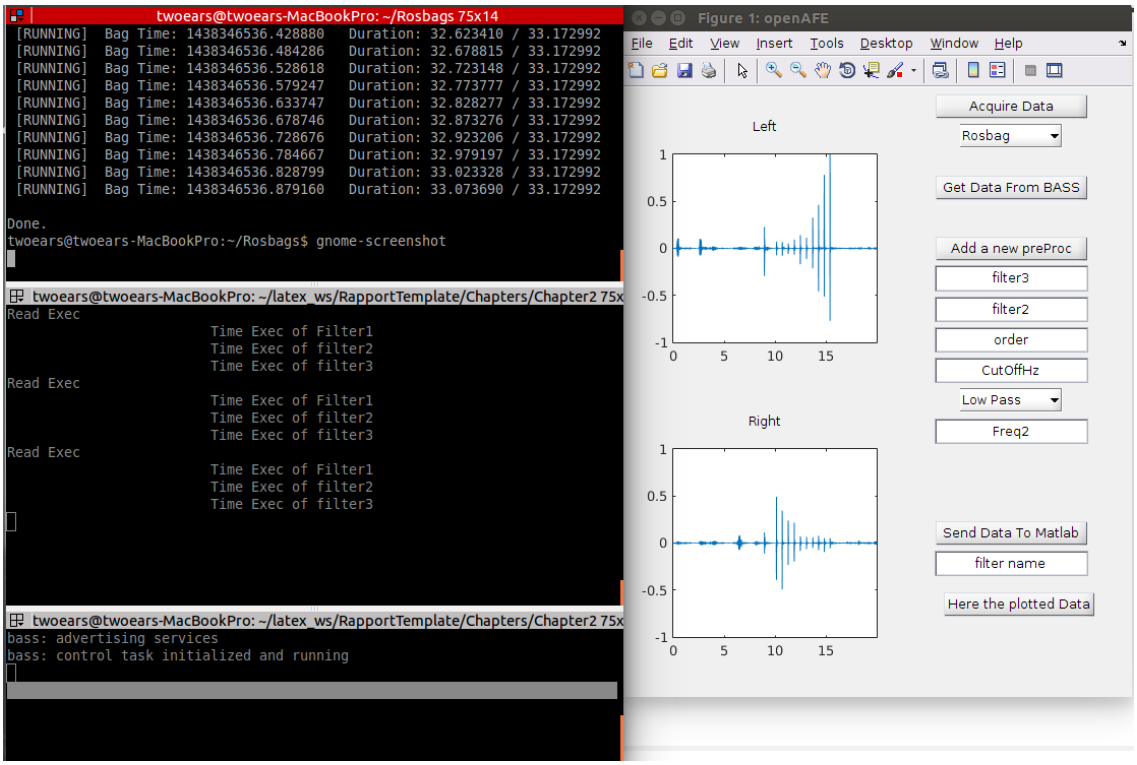


Figure 5.5: The *ROS* AFE Proof of Concept (PoC). Left Side : Terminals. 1st terminal: The audio source simulator. 2nd terminal: the *ROS* AFE node software printing the names of processors every time their task is run. 3th terminal: *Binaural Audio Stream Server*. Right Side: the PoC GUI.

5.2 Active audio-motor and information-based localization

5.2.1 Reminder

A three-stage framework to active binaural localization is described in Section 2.8 of Deliverable 4.2@month24 (Bustamante *et al.*, 2015). As it is situated at the sensorimotor level of the TWO!EARS model, it takes place within the functional layer of the software architecture, and has thus led to the integration of a *GenoM3/ROS* component. The

aim is to jointly process and/or interweave binaural sensing and motor commands of the *KEMAR* head so as to localize one source in the horizontal plane, by disambiguating front from back and recovering its range.

The overall framework is reminded on Figure 5.6. Stage A implements the maximum likelihood estimation of the source azimuth and the information-theoretic detection of its activity from the short-term channel-time-frequency decomposition of the binaural stream (Portello *et al.*, 2013)(Portello *et al.*, 2014a). Stage B assimilates these azimuths over time and combines them with the motor commands into a stochastic filter, leading to the posterior probability density function (pdf) of the head-to-source relative situation (Portello *et al.*, 2012)(Portello *et al.*, 2014b). Stage C provides a feedback controller, which, on the basis of the output from Stage B, can move the head so as to improve the quality of the localization, *i.e.*, of the output from Stage B (Bustamante *et al.*, submitted).

5.2.2 Implementation

The *binauloc GenoM3/ROS* component has been developed from a working *MATLAB* code and successfully tested offline. Its data flow connections with other components of the functional layer are sketched on Figure 5.7. *binauloc* takes as input the audio stream from the *Binaural Audio Stream Server* (*BASS*) and the motor flow from the modules in charge of the displacement of the kemar head and the locomotion of the mobile base. It processes all these data, following the Stage A and Stage B algorithms, so as to compute a Gaussian mixture approximation of the posterior probability density function (pdf) of the head-to-source relative situation (azimuth & range). The weights, posterior means and posterior covariances involved in this approximation are published on a port. Another component has been written to display the result of the localization on

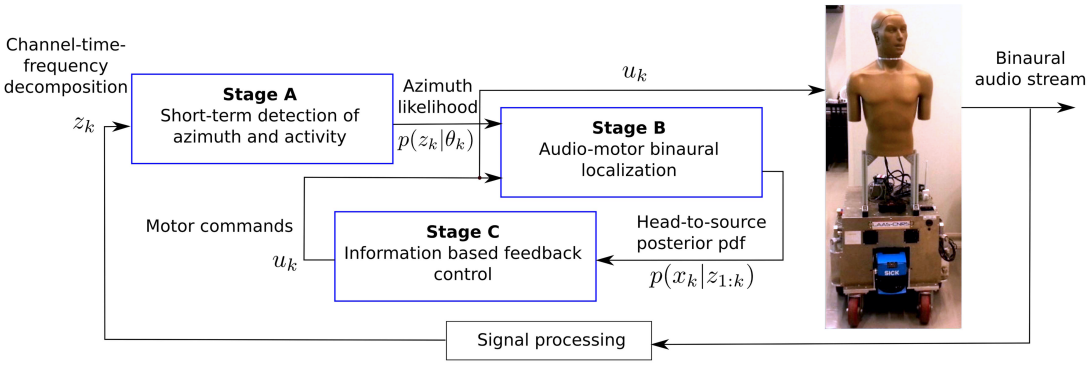


Figure 5.6: Three-stage framework to active binaural localization

the screen from these parameters. It plots the 99%-probability confidence ellipses of the hypotheses constituting the posterior pdf, with a color expressing their weights. *binauloc* also publishes the velocities to be applied on the left and right wheels of the robot and on the neck of the *KEMAR* HATS so as to improve the localization, as per Stage C. As the feedback control problem is stated into the frame related to the *KEMAR* head, these velocities are obtained from (4.2), which explains the incorporation of the angular position from the head.

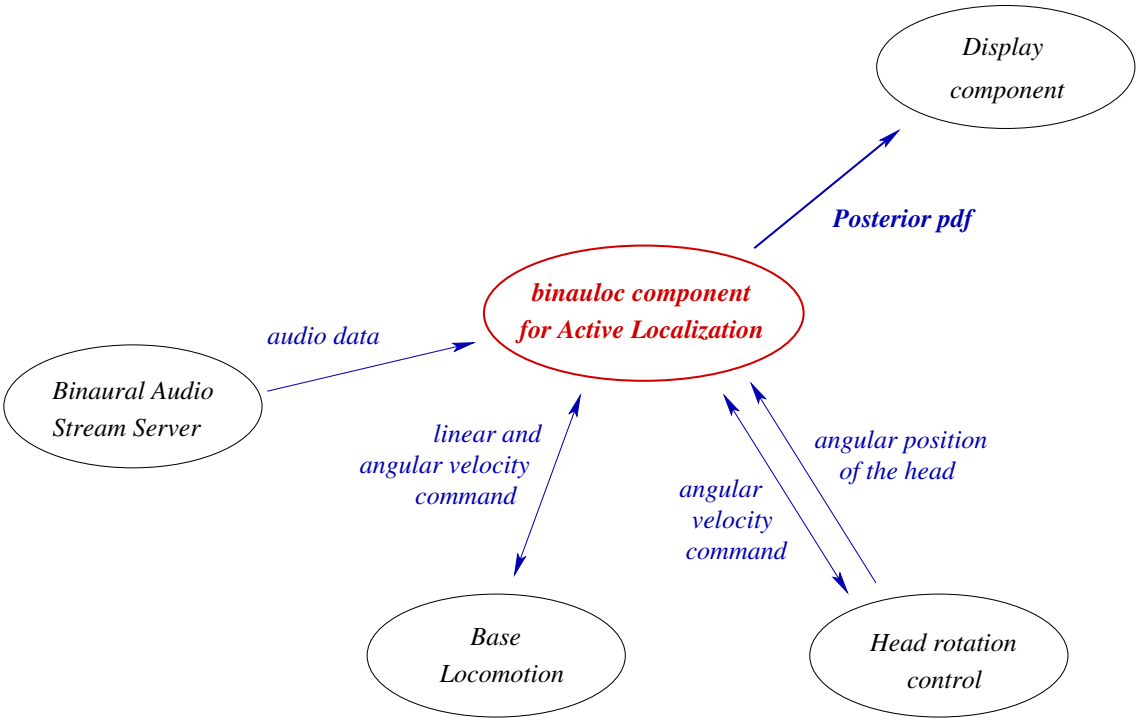


Figure 5.7: The *binauloc GenoM3/ROS* component for active binaural localization and its interaction with other modules of the functional layer.

Some technical details are as follows. Stage A takes as input a 1 second sliding window of the binaural audio stream. It outputs a pseudo-likelihood of the source azimuth every 58 ms. Stage B runs at approximately 6 Hz. The update stage of the underlying stochastic filter entails an approximation of the azimuth pseudo-likelihood produced by Stage A by an unnormalized Gaussian mixture. The variances of the hypotheses of this mixture and the noise statistics of the prior dynamic model have been empirically tuned so as to ensure reproducible and slightly conservative conclusions.

5.2.3 Experiments

Figures 5.8–5.11 show snapshots of live experiments, where the sound source is a white noise signal filtered by a bandpass filter of [1000 Hz, 2000 Hz] bandwidth. The corresponding information measure at the end of each sequence is reported on Figure 2.28 of Deliverable 4.2@month24. Videos are available at `http://homepages.laas.fr/danes/ICASSP2016/`.

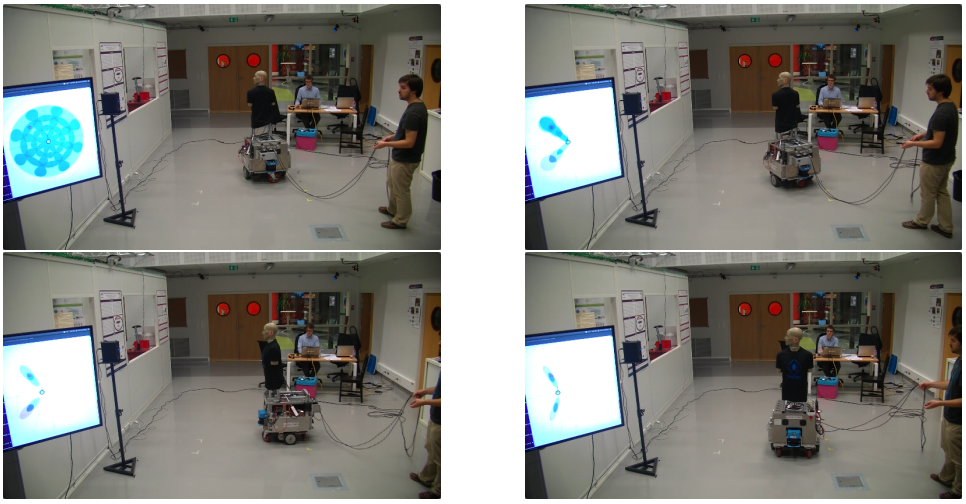


Figure 5.8: Visualization of the posterior head-to-source pdf for an open-loop rectilinear translation motion of the head. The uncertainty is reduced, but front and back cannot be disambiguated.

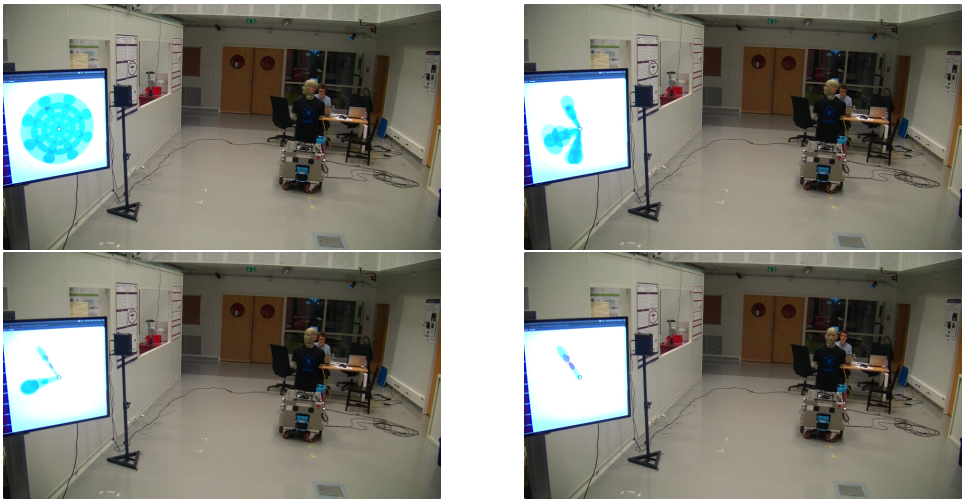


Figure 5.9: Visualization of the posterior head-to-source pdf for an open-loop rotation motion of the head. Front-back ambiguity is removed but range cannot be recovered.

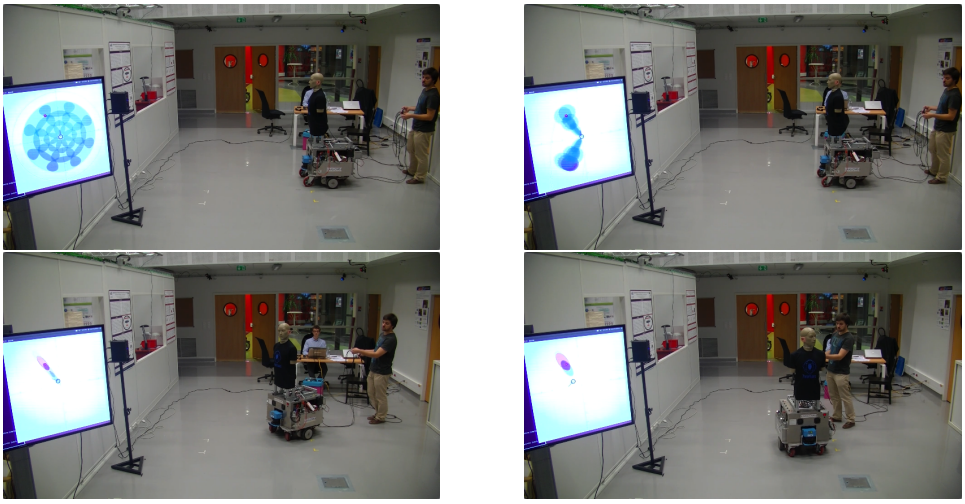


Figure 5.10: Visualization of the posterior head-to-source pdf for an open-loop circular motion. The localization is improved.

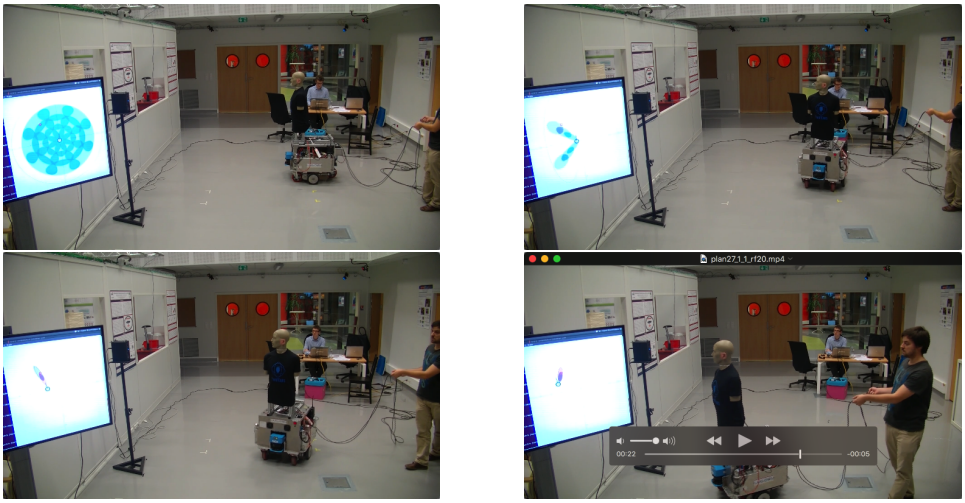


Figure 5.11: Visualization of the posterior head-to-source pdf when the head motion is delivered by the sensorimotor feedback of Stage C. This strategy outperforms the above three other ones.

6 Ingredients for a Binaural Robots Challenge

From September 21st to September 25th 2015, the Two!Ears consortium organized a Summer School on Active Machine Hearing, see its `http://twoears2015.sciencesconf.org` website and the pictures of the event in `https://www.flickr.com/photos/twoearsproject/`. It took place at LAAS-CNRS, Toulouse, France. Its first part was dedicated to theoretical lectures, with programming exercices in-between in order to put theory into practice. A special feature was the introduction of a "robotics challenge". The delegates were put into teams, each of which was provided a similar small (80cm-tall) binaural mobile robot. After getting increasingly familiar with the middleware and with basic functional modules (locomotion, navigation, audio streaming, etc.), they had to integrate during two half days some functions studied during the theoretical part of the training. The aim was to enable the robot to perform a list of milestones composing some active auditory functions.

CNRS was in charge of the local organization of this Summer School. With regard to the robotics challenge, this included:

- renting five commercial robots, their equipment with adequate *ROS* stacks, the design and manufacturing of a small-scale environment, and the development of a real time motion capture and 3D rendering on the *MORSE* simulator;
- the design and manufacturing of spherical binaural heads based on Micro Electromechanical Systems (MEMS) microphones; their mounting on the robots;
- the adaptation of the *Binaural Audio Stream Server* (*BASS*) to the MEMS technology together with (and thanks to) its integration on a system-on-chip architecture.

This chapter summarizes these achievements, which were made possible thanks to the experience gained during the project.

6.1 Robots, Environment and Rendering tools

6.1.1 The selected robots and their associated software

The robotic platform chosen for the TWO!EARS *Summer School* was the low-cost *turtlebot*[1]. It is composed of a Kobuki mobile base[2], a depth "3D" sensor (Asus Xtion[3] or Microsoft Kinect[4]) and a netbook with *GNU/Linux* and *ROS* installed.

Figure 6.1: The *turtlebot* robot

Due to the fact that the *turtlebot* relies on the same *ROS* stacks for SLAM and navigation as *JIDO* does, their use for this platform is carried out exactly as described in Section 4.1.2. However, the input data to the locomotion and navigation algorithms differs between the two robots. The odometry of the *turtlebot* incorporates the data provided by low-quality encoding wheels and by a gyroscope. In addition, the *turtlebot* is not equipped with a laser, so the output from its depth camera is used instead for navigation, and converted into laser-like data[5]. In view of the limited embedded computing power, the frequency to process the incoming data has to be slowed down.

Some of the internal parameters should be re-tuned, due to the distinct size of these robots and of the reduced-scale environment. The locomotion velocity limits in the *ROS navigation* stack should be significantly reduced to prevent the robot from hitting the walls, in view of the fact that depth measurements generated by the embedded 3D sensor are less accurate than with lasers and are provided at a lower rate.

The *SendPosition GenoM3* component described in Section4.1.3–B was reused "as is" to make the robot navigate into the map either in absolute or in relative mode.

6.1.2 The small-scale environment

Considering that the robot's footprint is a disk of 30 cm radius (with only 24 cm between the driving wheels) and its height is 42 cm, its total height with its embedded binaural head can hardly be greater than 80 cm. However, the interaural axis of the embedded binaural

1 `http://www.turtlebot.com`
2 `http://kobuki.yujinrobot.com/home-en`
3 `https://www.asus.com/3D-Sensor/Xtion_PRO`
4 `https://dev.windows.com/en-us/kinect`
5 A *ROS* package is dedicated to this conversion, available at `http://wiki.ros.org/depthimage_to_laserscan`.

sensor should be much higher in order to address a conventional environment, shaped for humans, with openings typically above 1.20 m. This is why a small-scale environment was specifically designed for the TWO!EARS *Summer School* challenge, see Figure 6.2. Its size ($L \times l \times H$) is 5 m×3 m×1 m. It is made out of 16 mm-thick plywood, which leads to clean reverberations. It includes a fairly wide open space as well as a narrower room and a corridor, to increase the variability of experimental conditions as the robot navigates.

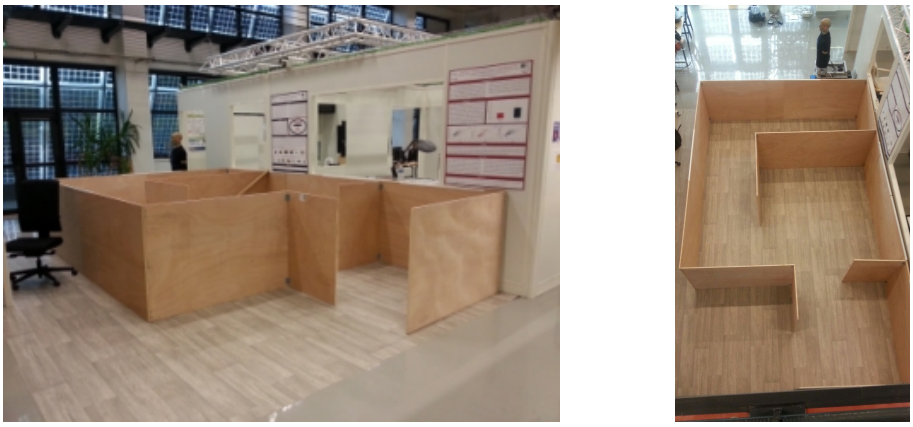


Figure 6.2: Side and top views of the small-scale environment specifically designed for the TWO!EARS Summer School Challenge.

6.1.3 The real time motion capture system and the 3D rendering on *MORSE*

A motion capture system was used to get the position of the *turtlebots* in real time within the reduced-scale environment. It is an *OptiTrack*[6] system deployed in the robotics hall of the *ADREAM* building[7] at *CNRS*. At least 6 infrared cameras, as the one shown in Figure 6.3 were used for the challenge, out of the 28 ones composing the connected network over the *ADREAM* hall. Each camera is endowed with an infrared illuminator around its lens. The tracking is based on a series of passive markers made with a retroreflective material which reflects light with minimum scattering. An algorithm extracts the centroids of their projections in the image planes of the infrared cameras. If each marker is seen by at least two cameras, then its 3D position can be recovered. The *OptiTrack* software allows the user to create "rigid

Figure 6.3: Optitrack camera with 96 infrared leds around its lens.

6 http://www.optitrack.com
7 https://www.laas.fr/public/en/adream

bodies", composed of at least three markers, and to localize them as wholes. Therefore, one rigid body composed of four markers was added on top of each *turtlebot*. The pattern drawn by the configuration of each such rigid body was distinct in order to localize with no ambiguity any set of robots in the environment.

The software installed on the *OptiTrack* server multicasts the locations (positions and attitudes) of the bodies defined by the user. These are defined in the reference frame of the cameras network, which is obtained by calibrating the system. The *GenoM3* component named *optitrack-genom3*, previously developed at *CNRS*, retrieves these data and publishes them on a port. Besides, a virtual 3D environment which matches the small-scale environment described above in Section 6.1.2 was designed in *Blender*[8] and incorporated in *MORSE*[9], the generic simulator for robotics used in the project (Figure 6.4). As objects are placed in *MORSE* with respect to a frame originating at the center of the virtual scene, two bodies composed of three markers were stuck at opposite corners of the small-scale environment in order to compute the rigid transform from the *OptiTrack* reference frame to that frame.

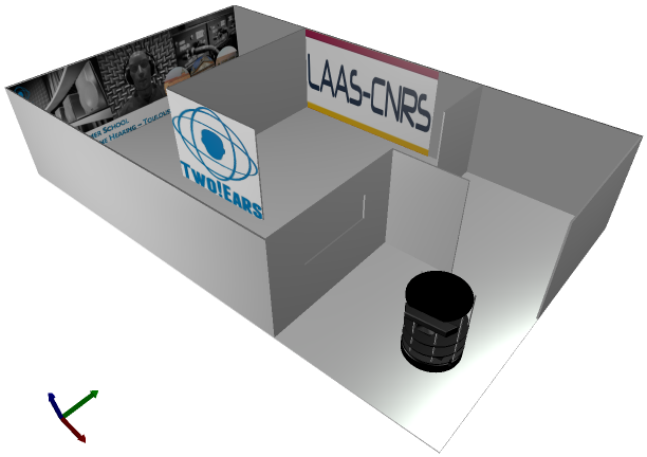


Figure 6.4: 3D rendering of the small-scale environment with one *turtlebot*

The positions of the *turtlebots* present in the scene were read every 100 ms on the port of the *optitrack-genom3* component. Their counterparts expressed in the reference frame of the 3D virtual environment were also published every 100 ms, so that *MORSE* could update the positions of the virtual robots therein. This enabled a smooth 3D rendering of their navigation.

8 `https://www.blender.org/`
9 *MORSE* was introduced in Chapter 7 of Deliverable 5.1@month12.

6.2 Binaural spherical heads based on MEMS microphones

A binaural sensor was designed, on the basis of a polystyren solid ball of 15 cm diameter. It was mounted on a 30*cm*-tall tube, itself attached to the top of the *turtlebot*, see Figure 6.5. So, the height of the interaural axis was about 80*cm* from the ground. Considering that the walls of the small-scale environment are 1*m* height, all the reflections from any sound source located within these walls could be captured without any issue.

Figure 6.5: The binaural spherical head and its mounting on the *turtlebot*

Each of the five binaural *turtlebots* had to embed the *Binaural Audio Stream Server* (*BASS*, cf. Section 3.3) on a compact and cheap processing unit which could cope with MEMS microphone technology and could integrate *ROS* components. This section summarizes how an original system-on-chip solution, entailing a *Logi PI*[10] Field-Programmable Gate Array (FPGA) board plugged onto a *Raspberry PI* 2[11] computer running under *GNU/Linux*, fulfilled these constraints, with a volume of $9\times6\times3\,\mathrm{cm}^3$ and cost less than 150 €.

6.2.1 Low-cost solution for audio acquisition from MEMS microphones

Two *MP34DT01* microphones from STMicroelectronics[12] were selected to equip the spherical head. These are ultra-compact, low-power, omnidirectional, microphones, based on Micro Electromechanical Systems (MEMS) technology. These digital devices are built with a capacitive sensing element and an Integrated Circuit (IC) interface manufactured using a CMOS process. They have an acoustic overload point of 120 dB of sound pressure level (dBSPL) with a 63 dB signal-to-noise ratio and -26 dB relative to full scale (dBFS) sensitivity. We designed two dedicated circuits so as to provide a digital output signal in Pulse Density Modulation (PDM) format. They were assembled into a homemade 1/2-inch diameter cylinder with an adequate connector. The whole design is shown on Figure 6.6.

The MEMS microphones could not be directly connected to the turtlebot's netbook. Indeed, the standard for digital audio interfaces is Pulse Code Modulation (PCM) (Figure 6.10),

10 `http://valentfx.com/logi-pi/`
11 `https://www.raspberrypi.org`
12 `http://www.st.com/web/en/resource/technical/document/datasheet/DM00039779.pdf`

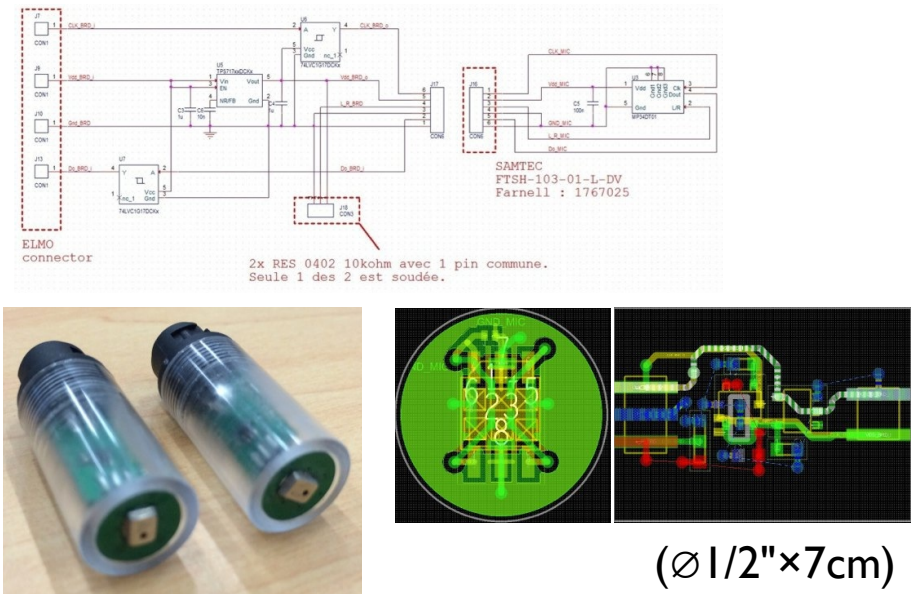


Figure 6.6: Typical 1/2-inch cylinders manufactured at *CNRS*, housing a MEMS microphone, a digital circuit, and a connector. The upper and lower right diagrams portray the electronics of the two small boards constituting the digital circuit.

so a conversion from PDM to PCM was required. This need could be fullfiled by a *Logi PI* on a *Raspberry PI* 2, as detailed below.

The *Raspberry PI* 2 is a low cost, credit-card sized computer[13] that can use a standard USB keyboard and mouse, and be connected to a computer monitor or TV. It embeds a quad-core ARM Cortex-A7[14] CPU @ $900Mhz$ and $1GB$ of RAM. Besides, it includes 4 USB ports, 40 GPIO pins as well as a full HDMI port and Ethernet interface. Persistent storage for the File System can be provided by a SD card, thanks to the presence of a MicroSD slot. Due to the fact that the *Raspberry PI* 2 has an ARMv7 processor, it can run the full range of ARM *GNU/Linux*[15] distributions. Consequently, a dedicated off-the-shell *Ubuntu* image was installed[16]. It combines the 14.04 release with Personal Package Archives (PPA) containing kernels and firmware suited to the *Raspberry PI* 2.

To enable acquisition from the pair of MEMS microphones, a prototyping platform that could be migrated to a System on a Chip (SoC)[17] has been designed, and implemented

13 `https://www.raspberrypi.org`
14 `http://www.arm.com/products/processors/cortex-a/cortex-a7.php`
15 `http://www.arm.linux.org.uk/docs/whatis.php`
16 `https://wiki.ubuntu.com/ARM/RaspberryPi`
17 From now on the term “SoC” will refer as this prototype.

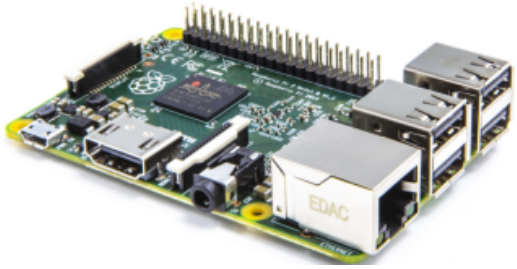
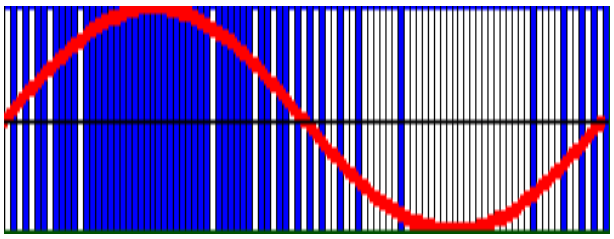


Figure 6.7: The *Raspberry PI* 2

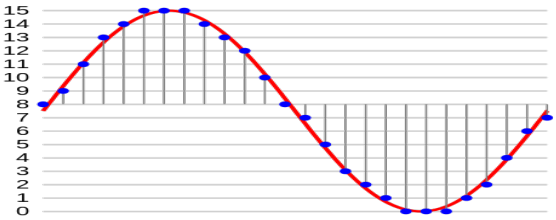
Figure 6.8: The LogiPi.

on the *Logi PI*[18]. The *Logi PI*, a shield for the *Raspberry PI*, features a *Spartan 6 LX9* FPGA from Xilinx[19]. It includes thirty-two GPIOs by means of four Peripheral Module Interface (PMOD) compatible headers. The MEMS microphones are connected to these I/Os.



Credit: *Wikipedia*

Figure 6.9: PDM of 100 samples of one period of a sine wave. White stands for 0 and blue is 1.



Credit: *Wikipedia*

Figure 6.10: Digitization (sampling and quantization) of a sine wave on 4-bit PCM.

The processing chain shown in Figure 6.11 is implemented within the SoC. The PDM to PCM conversion constituting its second block is detailed on Figure 6.12. Its last block, the *Wishbone Bus*, is an open-source hardware computer bus intended to enable the communication between parts of an integrated circuit, *i.e.*, to enable the connection of different cores to each other within a chip.

18 `http://valentfx.com/logi-pi`
19 `http://www.xilinx.com/`

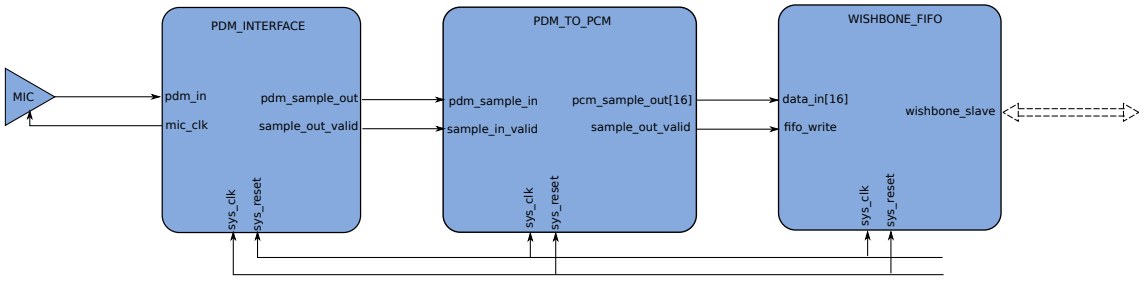


Figure 6.11: Processing chain within the SoC.

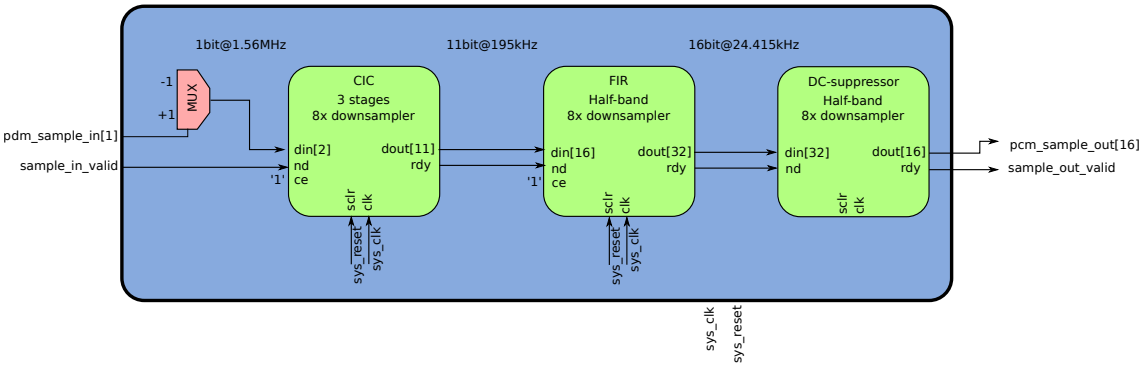


Figure 6.12: PDM to PCM

6.2.2 Integration in the *ROS* architecture

The ability for the SoC to be part of the robotic software architecture was a decisive requirement in adopting the exposed solution. Indeed, audio signals acquired by the MEMS microphones needed to be streamed to other components of the architecture, implying the adaptation of the *Binaural Audio Stream Server* (Section 3.3) to the current system. A robust, thoroughly tested, *ROS* version is available for the *GNU/Linux* version installed on the *Raspberry PI*. Together with the set up of *GenoM3* on the system, it allowed *BASS* to be compiled and run on the *Raspberry PI*. By connecting the *Raspberry PI* and the *turtlebot*'s netbook over Ethernet, the *ROS* architecture could be distributed on both computers, with the *Raspberry PI* in charge of audio streaming and the turtlebot's netbook in charge of locomotion and navigation.

After a successful attempt to acquire audio data from an ALSA device plugged into the *Raspberry PI*, *BASS* was consequently adapted to retrieve audio data through the *Logi PI*. ALSA-related functions of *BASS* were replaced by similar functions using the Serial Peripheral Interface (SPI) of the *Raspberry PI*, enabling data transfer from the *Logi PI* to the *GenoM3* component. The resulting version of *BASS* differs from the original one on the following items:

- The sampling frequency is fixed to 24414 Hz. This constraint comes from the frequency of the oscillator included in the *Logi PI*.
- The size of captured chunks must be a power of 2.
- The audio samples depth is 16 bits, instead of 32 bits within the original version.

With a fully functional *Binaural Audio Stream Server* in the software architecture, the short term azimuth localization of sound sources, presented in Section 5.2, could be integrated as is on the *turtlebot*. The underlying HRTF model was just replaced by a model complying with the geometry of the embedded binaural sensor. It basically consisted in the well-known analytical expressions of the left and right HRTFs to two microphones placed antipodally on a spherical head of adequate size (Morse and Ingard, 1987).

7 Appendix

7.1 Two!Ears online documentation on the robotic architecture

The following pages are an extract of the full Two!Ears documentation, available online at `http://twoears.aipa.tu-berlin.de/doc`. They deal with robotic aspects of the project.

CHAPTER

ONE

ROBOTIC PLATFORM

1.1 Getting started

1.1.1 Introduction

Component-based software architectures in robotics

Robots are highly complex systems that embed numerous sensors and actuators, in the service of a variety of algorithms performing heterogeneous tasks. They often have to deal with severe requirements (timing constraints, limited energy, memory and processing resources, *etc.*) and must show a robust conception as uncertainty about their environment is high, and unexpected events can have critical consequences.

To facilitate the development of robotic software, **component-based architectures**, where components are independent processes, have become the *de facto* standard in robotics. Each software component is dedicated to a given task, from low-level control to high-level processing. Components communicate with each other with the help of a software piece called the **middleware**.

For instance, consider a robot embedding a camera and performing object detection in the images. One component could be in charge of acquiring the images using the camera's driver, and would output them. Another component could input the images and run an algorithm on them to detect objects, producing the detection result as output for any other component in need of this information. Routing data from the output of the first component to the input of the second one is ensured by the middleware, this is called **data flow**.

Components offer **services** to the user to modify their behaviour and adapt to different situations. To follow with the example above, the component acquiring images could provide a service to select which camera to use, another one to configure parameters such as the image size and the number of frames per second, a third one to explicitly request

the start of the acquisition, *etc.* The component detecting objects could have a service to change some parameters in the detection algorithm, another one to choose which image stream to take as input (because there could be several components streaming images from different cameras), *etc.* Making services available to the user is again handled by the middleware, this is called **control flow**.

Note: The *user* mentioned here is not necessarily a physical person. For autonomous robots, it will probably be a detached software piece, supervising the state of the robot and choosing to start a given service to accomplish a new goal. This software piece belongs to the **decisional level**, while other components make up the **functional level**.

Component-based software architectures offer great benefits in robotics, in particular [1]:

- **Modularity**
  - As many operations handled by a robot require to have their own thread of execution (*e.g.* data acquisition for sensors, motion control for actuators), having them in separate programs eases their concurrent execution.
  - The architecture can be adapted to the needs of the robot: adding a new hardware piece such as a sensor will result in running a new software component to drive it.
  - The system can be distributed over a network, as the middleware seamlessly ensures communication between components running on different host machines.
- **Re-usability**
  - Common components can be used across robots without having to recode them from scratch.
  - Components can be packaged and easily shared in the robotics community, where open source software prevails.
  - Re-usable components reduce development cost and time, while improving software quality and sustainability.

ROS, a software platform for robotics

The *previous section* identifies the *middleware* as the software piece ensuring data flow between functional components, and allowing their control.

ROS is a widely known software platform in robotics, providing not only a middleware, but also implementing a wide range of commonly-used functionalities into software components (such as localisation, mapping, path-planning, obstacle avoidance, *etc.*), with a build system and a packaging system for easy compilation and installation. ROS benefits from a large community of users and developers, and runs on many robots today. This makes ROS a common choice as a robotic software platform, as it is for Two!Ears.

ROS embraces the principles of component-based software architectures, allowing distributed computation, software reuse and rapid testing [2]. If you will be a user of a robotic platform running ROS, the core tutorials can help you to get familiar with the ROS environment. The main ROS terminology, introduced in the tutorials, is recalled here:

Nodes Software components using ROS middleware are called ROS **nodes**.

Topics and messages Data flows are called **topics**. A node that outputs data *publishes* on a topic. A node that inputs data *subscribes* to a topic. The data elements flowing on topics are called **messages**. Each message is made of various data fields forming part of a data structure called *message type*. As a given topic only carries one message type, the term *topic type* is equally used.

Services and actions Nodes can provide **services** to control them. Some special services that can take a long time to execute are called **actions**.

[1] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Ore- back. Towards component-based robotics. In *IEEE International Conference on Intelligent Robots and Systems*, pages 163–168, Tsukuba (Japan), 2005.

[2] Jason M. O'Kane. A Gentle Introduction to ROS. http://www.cse.sc.edu/~jokane/agitr/, 2014.

Note: In spite of its name, ROS does not replace, but instead works alongside a traditional operating system. As it provides features such as hardware abstraction and low-level device control [3], ROS has some similarities with an OS (Operating System), hence its name.

GENOM3, a tool to develop robotic components

The process of developing robotic components can significantly be improved by the mean of a tool called GENOM3. As a result of two decades of research on real time architectures for autonomous systems [4] [5], GENOM3 brings valuable attributes to robotic components:

- **Middleware independence**

  Components developed with GENOM3 are middleware independent, *i.e.* they are not tied to a specific middleware and can be compiled for different middlewares without changing their source code.

  This is achieved through the notion of *templates*: a GENOM3 template is a set of instructions which, when applied to the component's source files, automatically generates the code related to middleware communication. A clear separation of concerns between the algorithmic core and the middleware is thus conducted, helping towards the good design of robotic components.

  ROS appears among the middlewares supported by GENOM3. When using ROS templates to compile a GENOM3 component, the resulting program is a genuine ROS node. Only the development process differs from what could be done by writing a ROS node without the GENOM3 tool.

- **Model-driven design**

  GENOM3 emphasises the clear definition of robotic components by adopting a model-driven approach. A GENOM3 component is first defined by a description file, called the **dotgen** file, with the `.gen` extension. This file gathers in a single place all the definitions related to the component's interface, needed by a user to interact with it.

  Each GENOM3 component has its own dotgen file, mainly including the definition of its **services** and its data flows by the mean of **ports** (either from the component to the outside, or the other way round). Each service is defined by a name and a list of input and output parameters with their related data type. Each port is defined by a name, a direction (either in or out), and a data type for the data elements it uses. The dotgen file often include in-line documentation to help the understanding of the component's features (for instance, the role of a given service parameter).

  On the basis of the model specified in the dotgen file, GENOM3 automatically generates real time code as well as skeletons of functions run by the services. So, the developer just has to fill these functions, called **codels** (for code element) with its algorithms. The corresponding algorithmic core is written in separate C or C++ source files or libraries.

- **Powerful framework**

  GENOM3 facilitates the development of essential features for robotic components, such as:

  – the definition of finite state automata with an optional clock,

  – clean interruption mechanisms,

  – efficient error handling.

[3] http://wiki.ros.org/ROS/Introduction

[4] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An Architecture for Autonomy. In *Int. Jour. on Robotics Research 17*, pages 315–337, 1998.

[5] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand. GenoM3: Building Middleware-independent Robotic Components. In *IEEE Int. Conf. on Robotics and Automation*, Anchorage (Alaska), 2010.

If you decide to be a user of GENOM3 components, the tutorial GenoM3 through an example will help you to learn how to read dotgen files. For a developer of GENOM3 components, the documentation defines the whole grammar for writing dotgen files, and how dotgen specifications are mapped into C or C++ codels.

Note: GENOM3 components are often called **modules** (hence the name *Generator of Modules*). The words *module* and *component* refer to the same entity: an independent program that can run on a host machine where the robotic software architecture is distributed.

1.1.2 Installation of the robotic tools

The robotic tools should be installed on any system that hosts components of the software architecture for the *Robotic platform*. This section details the installation process.

Note: For all the guidelines gathered here, we will assume that you are using Ubuntu GNU/Linux as it is the supported distribution for ROS (though any other UNIX platform should be suited to the GENOM3 tools). Many commands given here are intended for the **`bash`** shell. If you use a different shell, you should adapt the bash-related commands accordingly.

Install ROS

The ROS distribution you can install will depend on your Ubuntu version:

- If you have Ubuntu 14.04 LTS or 13.10, follow the ROS Indigo installation instructions.
- Otherwise, follow the ROS Hydro installation instructions.

Install the GENOM3 tools through *robotpkg*

This section will guide you through the installation of the GENOM3 tools. GENOM3 is open-source software (available at https://git.openrobots.org/) and can be compiled from source, but the common installation uses *robotpkg*, a compilation framework and packaging system for robotics software (more information at http://robotpkg.openrobots.org/).

Get *robotpkg* on your system

Note: The following instructions invite you to download two git repositories. If the given URLs using `git://` protocol fail, try `https://` protocol as instructed here and there. If you need an introduction to git have a look at *Git for beginners*.

First, get the `robotpkg` repository in your home folder (you can choose another location, but we recommend this one):

```
cd
git clone git://git.openrobots.org/robots/robotpkg
```

You will also need the wip subset of *robotpkg*, it contains some work in progress that is not officially released, but already available:

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

Next, set the installation path. The tools that you will soon install will be placed in your home folder under a dedicated folder named `openrobots`. Installing robotic components in your home folder ensures that you do not need root privileges for the installation (you can choose another location with a different prefix, but we recommend this one):

```
cd ~/robotpkg/bootstrap
./bootstrap --prefix $HOME/openrobots
```

To finish, update your environment variables to include the installation folder:

Note: In the following commands, note the use of an environment variable `ROBOTPKG_BASE` to indicate your installation path, set to `$HOME/openrobots`. If you have selected a different location at the previous step, you should modify the corresponding command accordingly.

```
echo >> ~/.bashrc
echo '# ROBOTPKG' >> ~/.bashrc
echo 'export ROBOTPKG_BASE=$HOME/openrobots' >> ~/.bashrc
echo 'export PATH=$PATH:$ROBOTPKG_BASE/bin:$ROBOTPKG_BASE/sbin' \
    >> ~/.bashrc
echo 'export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:\
    $ROBOTPKG_BASE/lib/pkgconfig' >> ~/.bashrc
echo 'export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:\
    $ROBOTPKG_BASE/src/ros-nodes:$ROBOTPKG_BASE/share' >> ~/.bashrc
echo 'export PYTHONPATH=$PYTHONPATH:\
    $ROBOTPKG_BASE/lib/python2.7/site-packages:\
    $ROBOTPKG_BASE/lib/python3.2/site-packages' >> ~/.bashrc
echo '# ROBOTPKG END' >> ~/.bashrc
source ~/.bashrc
```

How to install a *robotpkg* package

Your `~/robotpkg` directory contains a tree of packages, grouped into main categories. In the next part, we will invite you to install some of those packages. Here, we expose you the guidelines to follow for each package.

1. **Add options to the package**

   If you are asked to add options `option-1 option-2 ...` to the package `pkg-name`, edit the file `~/openrobots/etc/robotpkg.conf` and add a line (near the beginning for instance) looking like:

   ```
   PKG_OPTIONS.pkg-name+= option-1 option-2 ...
   ```

   You will get a better idea about this step with an actual example in the next part.

2. **Move to the package's directory**

   You will be given the path to the package, such as `robotpkg/category/pkg-name`. Change to this directory:

   ```
   cd ~/robotpkg/category/pkg-name
   ```

3. **Install possibly missing system dependencies**

   Run the following command to list the dependencies for the package you are about to install:

   ```
   make show-depends
   ```

   At the end of the command's output, the dependencies are separated into `Robotpkg dependencies` and `System dependencies`. If any *robotpkg* dependency is missing, it will

be automatically installed. If any system dependency is missing, you need to install it (commonly with **`apt-get`** under Ubuntu). Iterate this step until no system dependency is missing.

You may skip this step, but if a system dependency is missing, the package installation will stop at some point and you will be asked to install it.

4. **Install the package**

   Run the following command:

```
make update
```

   This will download the sources, compile them locally on your system, and install the output files in the `~/openrobots` directory.

Install packages for GENOM3

- Install the package `demo-genom3` from `robotpkg/wip/demo-genom3` with options `codels ros-server ros-client-ros ros-client-c`.

  This package will install the *demo* component. It is a sample component distributed with GENOM3, serving as an example, simply controlling the motion of a fictional robot. The aim of installing the *demo* component is twofold: first, it automatically installs all the dependencies for using GENOM3 (the provided options specifying that we will use the ROS templates); second, you get a GENOM3 component ready to be run to start using the robotic tools.

  For your convenience, here is how applying the *above steps* could look like:

```
# 1. Add options to the package
echo 'PKG_OPTIONS.demo-genom3+= codels ros-server ros-client-ros ros-client-c'\
      >> ~/openrobots/etc/robotpkg.conf
# or better, edit the file manually and add the line near the beginning

# 2. Change to the package's directory
cd ~/robotpkg/wip/demo-genom3

# 3. Install missing system dependencies
make show-depends
# Let's say that `make show-depends` revealed two missing system
# dependencies named 'bison' and 'flex'. Next step would be:
sudo apt-get install bison flex
make show-depends
# No missing system dependency left

# 4. Install the package
make update
```

- Then, install the packages `genomix` from `robotpkg/net/genomix` and `rosix` from `robotpkg/net/rosix`.

  genomix and *rosix* are HTTP servers providing an interface for some clients to control GENOM3 components and generic ROS nodes respectively.

- Then, install the packages `tcl-genomix` from `robotpkg/wip/tcl-genomix` and `matlab-genomix` from `robotpkg/supervision/matlab-genomix`.

  These are clients of *genomix* and *rosix* servers. The *tcl-genomix* client allows to control components using the Tcl language. Its installation is not mandatory (we will rather use the *matlab-genomix* client), but recommended as a common package distributed with GENOM3. The *matlab-genomix* client allows to control components from Matlab.

Note: You need a Matlab installation on your system in order to install the `matlab-genomix` package. If you encounter a missing dependency for `mex` (the MEX compiler from Matlab), you need to add the path to Matlab executables to the `PATH` environment variable. For example, with Matlab R2013a installed in `/usr/local/MATLAB/R2013a`, it would be done with:

```
export PATH=$PATH:/usr/local/MATLAB/R2013a/bin
```

The GENOM3 tools are now installed on your system. If you want to try the Matlab bridge, you can follow the official tutorial using the demo component. You can also follow the instructions below to install *BASS, an audio streaming server component* and later on follow the tutorial *Stream binaural signals from BASS to Matlab*.

Note: The Matlab bridge is installed in `~/openrobots/lib/matlab`. To follow the tutorials using *matlab-genomix*, you need to add this path to the Matlab path.

Install a GENOM3 component from the sources

Instructions

The software part of the Two!Ears robotic architecture includes several GENOM3 components. You may have to install them from their source files. Each GENOM3 component has its own folder, containing a description file named after the component with the `.gen` extension (something like `component.gen`). These steps will install the component in your `~/openrobots` folder:

```
cd path/to/component/folder
genom3 skeleton -i component.gen
./bootstrap.sh
mkdir build && cd build
../configure --prefix=$ROBOTPKG_BASE --with-templates=ros/server,ros/client/c
make install
```

Example: installing the BASS component

BASS is a *component for binaural audio streaming*. The folder for this component is named `bass-genom3`, under the `RoboticPlateform` folder of the software repository. Applying the above commands to install BASS gives:

```
# assuming that you are in the software repository
cd RoboticPlateform/bass-genom3
genom3 skeleton -i bass.gen
./bootstrap.sh
mkdir build && cd build
../configure --prefix=$ROBOTPKG_BASE --with-templates=ros/server,ros/client/c
make install
```

1.2 Audio streaming

1.2.1 BASS, an audio streaming server component

BASS is a GENOM3 component in charge of acquiring binaural audio data from a hardware sound interface, and making it available to other components, henceforth termed as its clients. It relies on the ALSA (Advanced Linux Sound Architecture) library to communicate with the hardware interface, hence working with any ALSA-compliant interface.

The component offers services to start and stop the acquisition of audio data, and streams the captured data on an output port. A sliding window of the most recent data is kept on the port, the size of which can be set at runtime (for instance, the port can be configured so as to keep the last 2 seconds of acquired signals).

The folder containing source files of the component is named `bass-genom3` and is located in the `RoboticPlatform` folder of the software repository. All files that are referred to in this section are in the `bass-genom3` folder.

BASS terminology

This section defines the notions and the vocabulary that BASS uses.

Interface and device They are synonyms for the hardware board in charge of converting analog sound signals into digital streams.

Acquisition and capture They are synonyms for retrieving audio data from microphones through an audio interface.

Binaural audio and channels **Binaural audio** signals consist of two **channels** (like stereo audio), corresponding to left and right ears.

Samples and frames A **sample** is a digital value encoding the signal on one channel at one point in time. A **frame** is a vector of samples, one from each channel, at one point in time (thus for binaural audio, a frame is two samples).

Chunks In capturing state, the sound device regularly delivers blocks of new data to the BASS component. These blocks are called **chunks**. The size of these chunks (commonly given in number of frames) can be selected at the start of a new acquisition, and is fixed until its end.

Note: The above definitions can differ from other applications where the word *frame* may refer to data blocks of a few milliseconds. Here, these blocks are rather called *chunks*, a frame being a single point in time.

Services

The services offered by BASS are defined and documented in the description file `bass.gen`. This section lists them and provides additional details.

- The `ListDevices` service can be called to display on standard output stream (*stdout*) the available sound devices that can be selected for the acquisition.
- The `Acquire` service starts the acquisition of audio data and updates the output port with the captured data (see details about the port in section *Output port*). This service expects 4 input parameters, shown in Table 1.1.

Table 1.1: Input parameters of the Acquire service of BASS

Name	Data type	Default value	Documentation
`device`	`string`	`"hw:1,0"`	ALSA name of the sound device
`sampleRate`	`unsigned long`	`44100`	Sample rate in Hz
`nFramesPerChunk`	`unsigned long`	`2205`	Chunk size in frames
`nChunksOnPort`	`unsigned long`	`20`	Port size in chunks

The `device` parameter is the identifier of a sound device to use. The value for one connected device can be retrieved with the aforementioned `ListDevices` service. The `nFramesPerChunk` parameter is important, as smaller chunks will lead to shorter latency but also higher communication needs between the component and the device. Last, the `nChunksOnPort` parameter sets the number of chunks kept on the port. With the default values given above, 20 chunks of 2205 frames is a total of 44100 frames kept on the port, i.e. 1 second of audio data at the default sample rate.

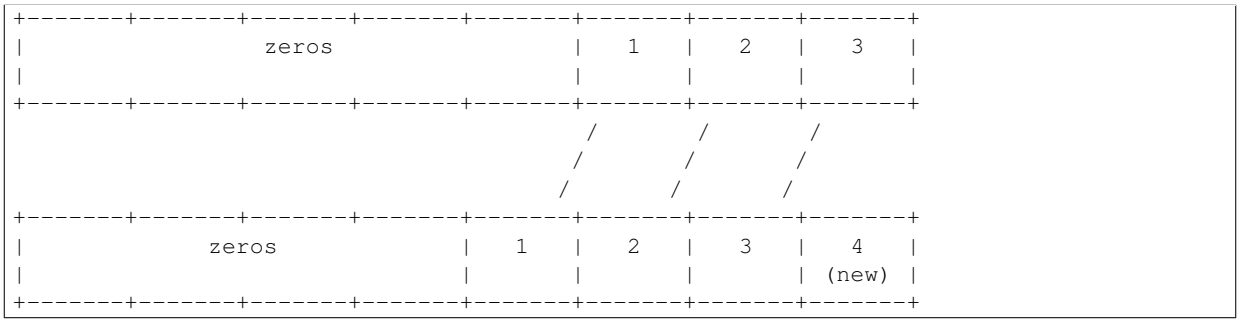
The `Acquire` service can return an exception if the configuration of the interface fails (*e.g.* the requested sample rate is not supported), if a problem occurs during the acquisition (*e.g.* the interface gets unplugged), *etc.* If an exception occurs, the user can get more information by reading the error message flushed on standard error stream (*stderr*).

- The `Stop` service stops the acquisition of audio data. Note that the `Acquire` service also interrupts itself, so a new acquisition with different parameters can directly be started from a running one without having to call `Stop`.

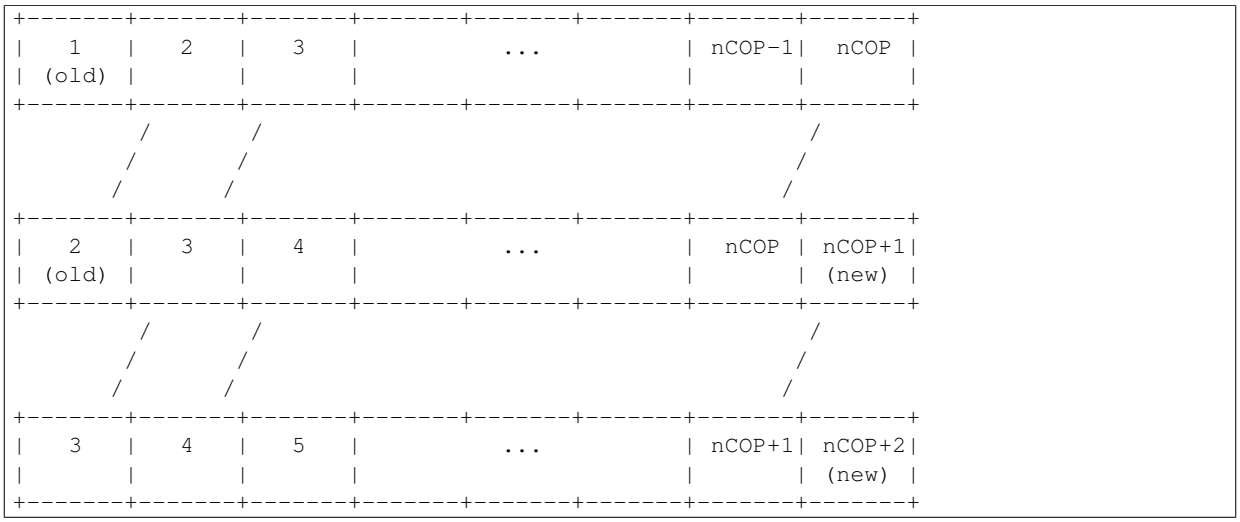
Output port

The data captured by the `Acquire` service are streamed on an output port named `Audio` (defined in file `bassInterface.gen`). They are gathered in two arrays, one for each channel, updated with a FIFO design: every time a new chunk is retrieved from the hardware interface, the content of the arrays is shifted, deleting the oldest chunk of data and making room to the newest one, as detailed below.

At the beginning of the acquisition, the arrays are filled with zeros and the first captured chunks are progressively added. For instance, the state of one array before and after adding the 4th chunk is illustrated here (assuming that the port is longer than 4 chunks):



The length of the port is a round number of chunks, set with parameter `nChunksOnPort` of the `Acquire` service (noted $nCOP$ below). The size of one chunk is also set when calling `Acquire`, with parameter `nFramesPerChunk` (noted $nFPC$ below). Thus, the left and right arrays contain $nFPC * nCOP$ samples each. Once the port is entirely filled with data (all beginning zeros have been erased), the oldest chunk is deleted as a new chunk arrives:



In order to let the clients keep track of the data and detect any loss, the port also publishes an index that indicates the number of frames that have been streamed since the beginning of the acquisition. In other words, it is the index of the last streamed frame, noted `lastFrameIndex`. The data structure of the `Audio` output port (defined in file `bassStruct.idl`) is summarised in Table 1.2.

Table 1.2: Data structure of the Audio output port of BASS

Name	Data type	Comment
`sampleRate`	`unsigned long`	sample rate in Hz
`nChunksOnPort`	`unsigned long`	number of chunks on the port ($nCOP$)
`nFramesPerChunk`	`unsigned long`	number of frames per chunk ($nFPC$)
`lastFrameIndex`	`unsigned long long`	index for tracking data
`left`	`sequence<long>`	audio data from left channel
`right`	`sequence<long>`	audio data from right channel

- The fields `sampleRate`, `nChunksOnPort` and `nFramesPerChunk` are set as input parameters of the `Acquire` service.
- The fields `left` and `right` are dynamic arrays (`sequence<long>` type) of $nFPC * nCOP$ samples. Samples are signed integers coded on 32 bits (`long` type).
- The index for tracking data is stored in the field `lastFrameIndex`. As this index is incremented by $nFPC$ frames every time a new chunk is published on the port, it is important to check that it will not overflow. The index is therefore coded as an unsigned integer on 64 bits (`unsigned long long` type. With a sample rate of 192kHz for instance, the order of magnitude of the index overflow is a million years).

Example of use

The tutorial *Stream binaural signals from BASS to Matlab* is an example of use of BASS, using the *matlab-genomix* bridge. It shows how to invoke its services and how to retrieve the streamed audio data in Matlab.

1.2.2 Writing a client of BASS

This section provides information about designing clients of the BASS component. It first defines a formal algorithm that clients could use, and shows a sample implementation in a GENOM3 component called BASC (Binaural Audio Stream Client).

An algorithm for clients of BASS

A client of BASS can face many different situations:

- It may need just a single block of data (*e.g.* we could think of a client requesting 2 seconds of audio data to learn noise properties), and the block may be longer than the total size of the port.
- It may indefinitely request new blocks of data, with the requirement that they follow each other without frame loss between two consecutive blocks.
- It may request data faster than the port update rate. Or conversely, it may not read the port often enough, leading to data loss. And in case of data loss, it must know how many frames were lost.

Let us define $nFOP$ as the total number of *frames on the port*. The data structure used on the `Audio` port, of type `portStruct`, is recalled in Table 1.3.

Table 1.3: Formal definition of the data type portStruct used on the Audio port of BASS

Field	Data type	Comment
`sampleRate`	unsigned integer	sample rate in Hz
`nChunksOnPort`	unsigned integer	number of chunks on the port ($nCOP$)
`nFramesPerChunk`	unsigned integer	number of frames per chunk ($nFPC$)
`lastFrameIndex`	unsigned integer	index for tracking data
`left[nFOP]`	array of integers	arrays of $nFOP$ samples, with $nFOP = nFPC * nCOP$
`right[nFOP]`	array of integers	

The `left` and `right` fields are arrays of $nFOP$ samples each, updated as FIFOs (*c.f.* *related section in BASS documentation*). What the client need is to copy blocks of given size N from these arrays. Let us define a function `getAudioData`, taking N as input and returning one copied block.

In order not to miss any frame between blocks retrieved with two consecutive calls, the `getAudioData` function also takes both as input and output an index of the *next frame to be read*, noted nfr. For instance, the client get a first block of N frames starting from a given index nfr. The function must return, along with the block of data, the new value of the index, corresponding to the next frame right after the first retrieved block. Then, the client can call the function again with this new value for nfr, so as to get the second block starting from this point.

For the very first block, the client can choose nfr according to the current value of the `lastFrameIndex`.

- If it wants to pick data from the existing frames on the port, nfr is chosen to be less than `lastFrameIndex`.
- If it wants to get fresher data (frames that are not yet on the port but will be published shortly on it), nfr is chosen to be greater than `lastFrameIndex`.

With a call to `getAudioData`, the client requests a block of given size N, but the function may not be able to return a full block of N frames. Indeed, the ending point for the desired block may be a frame that is not yet published by the server. So, the function returns the number n of frames it can get ($n \leq N$). The client can then call the function again and ask for the remaining frames in a loop until the requested block is complete. Here are examples of when this can occur:

- The client requests data more often than they are captured by the microphones (which should be the regular case, because a slower client will end up losing data).
- The client requests a block which is longer than the total number of frames on the port ($N > nFOP$).
- At first call, if the client sets nfr to a greater value than `lastFrameIndex`, then `getAudioData` will not return any available frame ($n = 0$). But the client can keep calling the function in a loop until it gets the requested frames.

Finally, in case of data loss, `getAudioData` also returns the number of frames that were lost. The retrieved block then starts at the first frame that is still available on the port (the oldest one).

Below is the `getAudioData` function written with formal syntax:

```
function: getAudioData
|
| inputs:  integer     N      (number of frames the client wants to get)
|          portStruct  Audio  (data from the the output port of bass)
|
| outputs: integer     n      (number of frames the function was able to get)
|          integer     loss   (number of lost frames, 0 if no loss)
|          array(int)  l[n]   (the retrieved data block from left channel)
|          array(int)  r[n]   (the retrieved data block from right channel)
|
| in&out:  integer     nfr    (index of the Next Frame to Read)
|
| local:   integer     nFOP   (total number of Frames On the Port)
|          integer     lfi    (Index of the Last Frame on the port)
|          integer     ofi    (Index of the Oldest Frame on the port)
|          integer     pos    (current position in the left and right arrays)
|
| algorithm
|  |
|  | nFOP <- Audio.nFramesPerChunk * Audio.nChunksOnPort
|  | lfi <- Audio.lastFrameIndex
|  | ofi <- max(0, lfi - nFOP + 1) //if the acquisition just started and the
|  |                               //port is not full yet, ofi equals 0
|  | /* Detect a data loss */
|  | loss <- 0
|  | if (nfr < ofi)
|  |  | loss <- ofi - nfr
|  |  | nfr <- ofi
|  | end if
|  |
|  | /* Compute the starting position in the left and right input arrays */
|  | pos <- nFOP - (lfi - nfr + 1)
|  |
|  | /* Fill the output arrays l and r */
|  | n <- 0
|  | while (n < N AND pos < nFOP)
|  |  | l[n] <- Audio.left[pos]
|  |  | r[n] <- Audio.right[pos]
|  |  | n <- n + 1
|  |  | pos <- pos + 1
|  |  | nfr <- nfr + 1
|  | end while
|  |
|  | return (l[], r[], n, nfr, loss)
|  |
| end algorithm
|
end function
```

Sample implementation in a GENOM3 component

BASC is a GENOM3 component that acts as a client of BASS. It can be connected to the output port of BASS, and implements the above generic algorithm to retrieve blocks of audio signals. BASC does not perform any processing on the data. It only shows what a GENOM3 client of BASS could look like.

The folder containing source files of the component is named `basc-genom3` and is located in the `RoboticPlatform` folder of the software repository. All files that are referred to in this section are in the

basc-genom3 folder.

Services

BASC runs the above algorithm in a service called GetBlocks, defined in description file basc.gen. It expects the following input parameters:

Name	Data type	Default value	Documentation
nBlocks	unsigned long	1	Amount of blocks, 0 for unlimited
nFramesPerBlock	unsigned long	12000	Block size in frames
startOffs	long	-12000	Starting offset (*past* < 0, *future* > 0)

The first parameter nBlocks sets the number of blocks that the service must get, or can be set to 0 to run the service indefinitely. The second parameter nFramesPerBlock sets the block size (N in the previous section). Last, the startOffs parameter sets the index of the first frame to read, relatively to the current value of the Last Frame Index on the port. For instance, if startOffs is -1000 and lastFrameIndex is 43000 when the service is called, then the first frame to read (nfr in previous section) will have index 42000. With the given default values, GetBlocks will get 1 block of 12000 frames, taking the last 12000 frames on the port.

At any moment, the GetBlocks service can be interrupted by calling a service named Stop.

Execution example

Assume that the GetBlocks service runs at a period of, say, 250ms (defined in the dotgen file). So every 250ms, it calls the getAudioData function, either to request a new block or to complete the current block if the previous call could not return a full one. Depending on the sample rate, if the requested block size (parameter nFramesPerBlock) is too small so that one block is less than 250ms, then the client will eventually loose some frames. On the other hand, if a block lasts more than 250ms, then the client will request data at a rate higher than their update frequency, which should be fine.

This can be tested: assume that the sampling rate is 44100Hz. So, 250ms is 11025 frames. Calling GetBlocks with nFramesPerBlock < 11025 leads to data loss. Following is an example with the *matlab-genomix* client.

```
% The middleware, genomix, bass and basc should be running

% Connect to genomix
>> client = genomix.client

% Load the server BASS and the client BASC
>> bass = client.load('bass')
>> basc = client.load('basc')

% Connect the input port of BASC to the output port of BASS
>> basc.connect_port('Audio', 'bass/Audio')

% Start the acquisition (using default values)
rAcquire = bass.Acquire('-a', 'hw:1,0', 44100, 2205, 20)

%%% EXAMPLE 1: get one block of 2 seconds (88200 frames at 44100Hz)
>> basc.GetBlocks(1, 88200, 0)

% In the terminal where it runs, BASC prints:
%
%    Requested  88200 frames, got  11025.
```

```
%    Requested  77175 frames, got  11025.
%    Requested  66150 frames, got  11025.
%    Requested  55125 frames, got  11025.
%    Requested  44100 frames, got  11025.
%    Requested  33075 frames, got  11025.
%    Requested  22050 frames, got  11025.
%    Requested  11025 frames, got  11025.
%    A new block is ready to be processed.
%
% Each line 'Requested N frames, got n.' indicates the number N of frames
% requested by BASC, and the number n it got in return. The component keeps
% requesting frames until it has formed a block of 88200 frames.

%%% EXAMPLE 2: get unlimited number of blocks, with nFramesPerBlock < 11025
>> rGetBlocks = basc.GetBlocks('-a', 0, 10000, 0)

% After a few seconds, BASC prints:
%
%    Requested  10000 frames, got  10000.
%    !!Lost 1025 frames!!
%    A new block is ready to be processed.
%    Requested  10000 frames, got  10000.
%    !!Lost 1025 frames!!
%    A new block is ready to be processed.
%
% At each attempt to get the following block, some frames are lost because
% BASC does not read the port of BASS often enough.

% Stop the running GetBlocks service
>> basc.Stop()

%%% EXAMPLE 3: get unlimited number of blocks, with nFramesPerBlock > 11025
>> rGetBlocks = basc.GetBlocks('-a', 0, 12000,0)

% Here, as BASC reads the port slighlty faster than its update rate, the
% retrieved block is sometimes incomplete, for instance:
%
%    Requested  12000 frames, got  12000.
%    A new block is ready to be processed.
%    Requested  12000 frames, got  12000.
%    A new block is ready to be processed.
%    Requested  12000 frames, got  11550.
%    Requested    450 frames, got    450.
%    A new block is ready to be processed.
%    Requested  12000 frames, got  12000.
%    A new block is ready to be processed.
%
% The two consecutive lines 'Requested...' show that a first call only get
% 11550 frames, so the component  makes a second request to get the remaining
% part of 12000 - 11550 = 450 frames.

% Stop the running GetBlocks service
>> basc.Stop()

% Kill the components
>> bass.kill()
>> basc.kill()
```

```
% Remove the used objects in Matlab
>> delete(bass)
>> delete(basc)
>> delete(client)

% The remaining processes (the middleware and genomix) can be killed
```

The `getAudioData` function in charge of getting the requested block is written in C in file `codels/basc_read_codels.c`. The codels of the `GetBlocks` service are also written in this file, with comments to explain the overall process followed by BASC.

CHAPTER

TWO

EXAMPLES

2.1 Stream binaural signals from BASS to Matlab

This tutorial shows an example of how to control the BASS component and retrieve audio streams in Matlab, using the *matlab-genomix* bridge.

2.1.1 Preliminary steps

In order to follow this tutorial, you will need:

- A Linux system with the robotic tools and BASS installed (*c.f.* *Installation of the robotic tools*). We will call this system the **BASS host**.
- An ALSA-compliant sound acquisition interface with at least two input channels, and two microphones plugged into it. The interface must be connected to the BASS host.

  Note: Alternatively, if you do not possess an external sound interface but the BASS host has an integrated sound card and microphone, you still might be able to follow the tutorial. Keep in mind though that if there is only one microphone, you will not have a genuine stereo signal, but a simulated one from your mono input.

- A computer with Matlab and the *matlab-genomix* bridge installed. We will call it the **remote client**. The BASS host and the remote client could possibly, but not necessarily, be the same computer.

On the BASS host, open 3 new terminals. In the first terminal, run the command:

```
$ roscore
```

This launches the ROS middleware. ROS nodes can now connect to this node called the ROS master. In the second terminal, run the command:

```
$ genomixd
```

This launches a *genomix* server, now waiting for incoming connections from clients on port 8080 by default. In the third terminal, run the command:

```
$ bass-ros
```

This is the BASS component, now running on the system. The name `bass-ros` specifies that this GENOM3 component uses the ROS middleware. So it is actually a ROS node, connected to the ROS master running in the first terminal.

For the moment, the BASS component is not doing anything. It is waiting for requests from a client (which will be Matlab here) to start services. This is the followed process:

1. The client emits a HTTP message destined for the *genomix* server, requesting to call a service of the BASS component.
2. *genomix* executes the call directed at the BASS component.
3. When the service is completed, BASS returns its output to *genomix*, and *genomix* relays it back to the client.

Keep the third terminal running BASS visible on the screen. When we will call some services, we will notice their effect on the component's standard output stream (*stdout*).

2.1.2 Control BASS to start an acquisition

On the remote client, start a Matlab session and make sure that *matlab-genomix* is in the Matlab path (*c.f. Installation of the robotic tools*).

Connect to *genomix* and load BASS

If you have Matlab on the same computer where the *genomix* server is running, you can simply connect to *genomix* with:

```
>> client = genomix.client
client =

  client with no properties.
```

This will attempt a connection on `localhost:8080` by default. Otherwise if your BASS host and your remote client are two different computers, get the IP address of the BASS host and override the default value with:

```
>> client = genomix.client('xxx.xxx.xxx.xxx:8080') % write the IP address of BASS host
```

Then, load BASS:

```
>> bass = client.load('bass')

bass =

  component with properties:

        genom_state: [function_handle]
               kill: [function_handle]
       connect_port: [function_handle]
    connect_service: [function_handle]
               Stop: [function_handle]
       ListDevices: [function_handle]
    DedicatedSocket: [function_handle]
              Audio: [function_handle]
```

```
    abort_activity: [function_handle]
           Acquire: [function_handle]
       CloseSocket: [function_handle]
```

The returned handle `bass` has a list of properties either corresponding to services (*e.g.* `Acquire`) or ports (*e.g.* `Audio`) of the component.

Get the name of your sound interface

Invoke the `ListDevices` service to get the name of your ALSA device:

```
>> bass.ListDevices();
```

The detected sound devices are listed on the components's standard output stream (*stdout*). On the BASS host, look in the terminal where the component is running, and find a line that matches your interface, something like:

```
hw:1,0 [Babyface2361116] [USB Audio]
```

The leading string, `hw:1,0` in the example, is the name of your ALSA device.

Start an acquisition

We will now use the `Acquire` service to start an acquisition.

Caution: By default, services are invoked synchronously, *i.e.* the command to invoke them only returns after completion of the service. As the acquisition runs indefinitely, the `Acquire` service never completes unless you explicitly stop it. So you must invoke this service asynchronously, *i.e.* the command invoking the service returns immediately and the service output can be retrieved later on. Otherwise you will be blocked in the Matlab command window without control, including stopping the service. If this happens, a solution is to kill the Matlab process and start again.

The service can be invoked asynchronously by providing the `'-a'` option:

```
>> r = bass.Acquire('-a')
 string device: ALSA name of the sound device (hw:1,0) >
```

The `Acquire` service expects input arguments. As we did not passed them to the function directly, they are prompted interactively. Enter values according to your sound interface (see the example below):

- For the `device` parameter, take the value you obtained at the previous step.
- For the `sampleRate` parameter, choose a sampling rate that your device supports. The default value (44100 Hz) is most likely to work.
- For the `nFramesPerChunk` parameter, choose a chunk size that your device supports. Some devices only support powers of 2 (*e.g.* 512, 1024, 2048...), refer to your device manual.
- For the `nChunksOnPort` parameter, choose a value that is big enough so that the output port of BASS streams a few seconds of audio data. For instance, with the default values (44100 Hz for the sampling rate and 2205 frames for the chunk size), keep 80 chunks on the port to have 4 seconds:

$$\begin{aligned} duration &= nChunksOnPort * nFramesPerChunk/sampleRate \\ &= 80 * 2205/44100 \\ &= 4s \end{aligned}$$

```
>> r = bass.Acquire('-a')
 string device: ALSA name of the sound device (hw:1,0) > 'hw:1,0'
 unsigned long sampleRate: Sample rate in Hz (44100) > 44100
 unsigned long nFramesPerChunk: Chunk size in frames (2205) > 2205
 unsigned long nChunksOnPort: Port size in chunks (20) > 80

r =

  request with properties:

       status: 'sent'
       result: []
    exception: []
```

If starting the acquisition succeeded, you should see the status `'sent'` in the returned handle. Otherwise, the status would be `'error'`, check then the error message printed in the terminal on the BASS host. It could be an invalid input parameter.

Note: The parameter prompts like `string device:  ALSA name of the sound device (hw:1,0) >` contains valuable information, *i.e.* the data type of the parameter, its name, a short description and a default value between parenthesis that will be used if you press enter without specifying another value. All this information comes from the dotgen file of the component, and is part of its definition.

2.1.3 Get audio data in Matlab

You can read the output port of BASS, named `Audio`, in Matlab:

```
>> p = bass.Audio()
p =

    Audio: [1x1 struct]

>> p.Audio
ans =

         sampleRate: 44100
      nChunksOnPort: 80
    nFramesPerChunk: 2205
     lastFrameIndex: 251370
               left: {176400x1 cell}
              right: {176400x1 cell}
```

The data structure shown here is retrieved when reading the port with function `bass.Audio()`. The audio signals are stored in the `left` and `right` fields. Note the presence of the index `lastFrameIndex` for keeping track of the data.

If your remote client computer has speakers, you can listen to the retrieved signals:

```
% Speak in the microphones for a few seconds

% Read the last few recorded seconds
>> p = bass.Audio();

% Play the recorded sound, on left channel for instance
>> soundsc(cell2mat(p.Audio.left), p.Audio.sampleRate);
```

Notice how the duration of the sound matches the one you selected with parameter `nChunkOnPort` when starting the acquisition.

2.1.4 End the session

When you are done, you can clear the used objects in Matlab:

```
>> delete(bass);
>> delete(client); % This closes the connection to genomix
```

On the BASS host, you can kill processes `roscore`, `genomixd` and `bass-ros` by typing `Control-c` in each terminal.

List of acronyms

AFE Auditory Front-End . . . 8
ALSA Advanced Linux Sound Architecture . . . 13
API Application Programming Interface . . . 13
CAN Control Area Network . . . 27
CPU Central Processing Unit . . . 17
FIFO First In, First Out . . . 13
FPGA Field-Programmable Gate Array . . . 63
GPIO General-Purpose Input/Output . . . 33
GUI Graphical User Interface . . . 40
HATS Head And Torso Simulator . . . 21
HRTF Head Related Transfer Function . . . 34
IC Integrated Circuit . . . 63
IDS Internal Data Structure . . . 50
MEMS Micro Electromechanical Systems . . . 59
PCM Pulse Code Modulation . . . 63
PDM Pulse Density Modulation . . . 63
PWM Pulse Width Modulation . . . 33
PoC Proof of Concept . . . 51
RPC Remote Procedure Call . . . 9
SLAM Simultaneous Localization and Mapping . . . 7
SPI Serial Peripheral Interface . . . 66
SoC System on a Chip . . . 64
URDF Unified Robot Description Format . . . 22
pdf probability density function . . . 53

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)

Brooks, A., Kaupp, T., Makarenko, A., Williams, S., and Ore-back, A. (**2005**), "Towards component-based robotics," in *IEEE Int. Conf. on Intelligent Robots and Systems*, Tsukuba, Japan. (Cited on page 9)

Bustamante, G., Danès, P., Forgue, T., and Podlubne, A. (**submitted**), "Towards Information-Based Feedback Control for Binaural Active Localization," in *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP'2016)*. (Cited on page 53)

Bustamante, G., Portello, A., and Danès, P. (**2015**), "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 52)

Cadenat, V. (**1999**), "Commande referencee multi-capteurs pour la navigation d'un robot mobile," Master's thesis, Universite Paul Sabatier de Toulouse. (Cited on page 30)

Corke, P. (**2015**), "Integrating ROS and MATLAB," *Robotics & Automation Magazine, IEEE* **22**, pp. 18–20. (Cited on page 15)

Grisetti, G., Stachniss, C., and Burgard, W. (**2007**), "Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters," *Transactions on Robotics, IEEE* **23**, pp. 34–46. (Cited on page 24)

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 pages 11 and 18)

Morse, P. and Ingard, K. (**1987**), "Theoretical Acoustics," Princeton University Press. (Cited on page 67)

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 53)

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 53)

Portello, A., Danès, P., and Argentieri, S. (**2012**), "Active Binaural Localization of Intermittent Moving Sources in the Presence of False Measurements," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vilamoura, Portugal. (Cited on page 53)

Portello, A., Danès, P., Argentieri, S., and Pledel, S. (**2013**), "HRTF-Based Source Azimuth Estimation and Activity Detection from a Binaural Sensor," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Tokyo, Japan. (Cited on page 53)

Zhang, J., Presti, L. L., and Sclaroff, S. (**2012**), "Online multi-person tracking by tracker hierarchy," in *Advanced Video and Signal-Based Surveillance (AVSS), 2012 IEEE Ninth International Conference on*, IEEE, pp. 379–385. (Cited on page 37)