We are IntechOpen, the world’s leading publisher of Open Access books. Built by scientists, for scientists.

5,600
Open access books available

138,000
International authors and editors

170M
Downloads

154
Countries delivered to

TOP 1%
Our authors are among the top 1% most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™
Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected. For more information visit: www.intechopen.com
Chapter

XBot: A Cross-Robot Software Framework for Real-Time Control

Luca Muratore, Arturo Laurenzi and Nikos G. Tsagarakis

Abstract

The widespread use of robotics in new application domains outside the industrial workplace settings requires robotic systems which demonstrate functionalities far beyond that of classical industrial robotic machines. The implementation of these capabilities inevitably increases the complexity of the robotic hardware, control a and software components. This chapter introduces the XBot software architecture for robotics, which is capable of Real-Time (RT) performance with minimum jitter at relatively high control frequency while demonstrating enhanced flexibility and abstraction features making it suitable for the control of robotic systems of diverse hardware embodiment and complexity. A key feature of the XBot is its cross-robot compatibility, which makes possible the use of the framework on different robots, without code modifications, based only on a set of configuration files. The design of the framework ensures easy interoperability and built-in integration with other existing software tools for robotics, such as ROS, YARP or OROCOS, thanks to a robot agnostic API called XBotInterface. The framework has been successfully used and validated as a software infrastructure for collaborative robotic arms as KUKA lbr iiwa/lwr 4+ and Franka Emika Panda, other than humanoid robots such as WALK-MAN and COMAN+, and quadruped centaur-like robots as CENTAURO.

Keywords: software architecture for robotics, real-time control, cross-robot framework, humanoid robotics, hardware abstraction layer, XBot, ROS

1. Introduction

Nowadays effective robotic solutions targeting new applications outside the traditional industrial environment, are supposed to operate in partially known spaces with unforeseen uncertainty and increased variability in the application tasks. Hence, to be effective, they have to adapt rapidly and seemly their functionalities in these demands, leading to an increase of the complexity in each layer of the robotic system, from the hardware to the high level control.

To tackle this, several software frameworks for robotics have been developed in the past twenty years, as stated in [1], aiming to provide flexible infrastructures, which not only permit the seamless integration of new functionalities and interfaces in the robotic system, but also ensure standardization, easy tracking and maintenance of the software development, despite the increased complexity. Apart from dealing with the software complexity, these frameworks have to provide hard Real-Time (RT) performance, ensuring predictable response times [2] as required in critical tasks when robots need to perform in autonomous mode, responding to
disturbances and interacting with the physical environment during the execution of
a task. Thus, a vital feature of a software framework for robotics is the Real-Time
saferness and scheduling, essential for precise robot control, especially when dealing
with high frequency and low jittering control cycles.

Furthermore, a software middleware needs to abstract the complex hardware
(e.g. actuators and sensors) of the robot providing an easy-to-use, standardized
Application Programming Interface (API). As a matter of fact, a robot can be
considered a distributed system composed of a set of hardware devices communi-
cating through a fieldbus. The fast prototype and development of control and
application software which can be shared, ported and reused in various robotic
platforms with minimum effort, is another fundamental requirement for the soft-
ware architecture. An important component needed to achieve this goal is the
Hardware Abstraction Layer (HAL), which can be incorporated to mask the phys-
cical hardware differences and limitations (e.g. control frequency, kinematics/
dynamics model, actuators type and size, sensors, etc) varying from one robot to
another. The HAL can provide a relatively uniform abstraction layer that assures
portability and code re-use: it permits the development of control modules that can
be easily ported from one robot to another.

The existing robotics software frameworks address different needs and require-
ments, therefore one of the key aspects for a brand-new middleware is the interop-
erability with well-known and established robotic software platforms.
Interoperability should ideally allow users to execute existing software without the
necessity of (i) changing the current code and (ii) writing hand-coded “bridges” for
each use case [3].

In this chapter the XBot software framework is presented. The development of
the XBot was driven by the need to provide a software framework that abstracts the
diverse variability of the robotic hardware (effectively becoming a cross robot
platform framework), providing deterministic hard Real-Time (RT) performance,
incorporating interfaces that permits it to integrate state of art robot control frame-
works and achieve enhanced flexibility through a plug-in architecture.

2. Related works

In this section the state of the art of robotic software architectures will be
analyzed.

In [4] the low level control framework, called OROCOS (Open Robot Control
Software), is introduced, which provides a set of components for RT control of
robotic systems. OROCOS relies on the Common Object Request Broker (CORBA)
architecture, that allows inter-process and cross-platform interoperability for dis-
tributed robot control. Depending on any Inter-Process-Communication (IPC)
framework can critically increase the complexity of the software platform. Despite
OROCOS is used in a fair number of robotics projects, the framework maintenance
as well as the community looks not being very active anymore 1.

Very similar to OROCOS is OpenRT-M [5], developed in Japan from 2002
under NEDO’s (New Energy and Industrial Technology Development Organiza-
tion) “Robot challenge program”. It is based on CORBA, so similar considerations as
for OROCOS can be made with respect to the software complexity; moreover part
of OpenRT-M documentation is in Japanese.

1 In particular we refer to the discontinuity in maintaining the framework under last versions (≥3.X) of
Ubuntu Xenomai, where the OROCOS porting is still experimental.
YARP (Yet Another Robot Platform) [6] and ROS (Robot Operating System) [7] are popular component-based frameworks for IPC that do not guarantee RT execution among modules/nodes. It is essential for a robotic system to have a component responsible for the RT control of the robot, making these frameworks only viable as external (high-level) software frameworks. It is worth to mention that a new ROS version, called ROS 2 [9], has been released: it is still in an early stage development phase, so it cannot be used in real-world scenario [3].

PODO [8], is the framework used by KAIST in HUBO during the DRC (Darpa Robotics Challenge) Finals. Its control system has RT control capabilities and its inter-process communication facilities are based on POSIX IPC; moreover it uses a shared memory system called MPC to exchange data between processes in the same machine. This heterogeneous system has the potential to cause confusion as it is unclear which architectural style must be used to communicate with a specific component [9].

In [10] an RT architecture based on OpenJDK is introduced (used by IHMC during the DRC Finals). Nevertheless, to their own admission [11], none of the commercially available implementations of the Java Real Time Specification had the performance required to run their controller. Existing Real-time Java Support is insufficient.

Considering the above limitations, summarized in Table 1, the XBot [12–14] was developed from scratch, in order to have a reliable RT control framework with HAL support and without depending on complex IPC frameworks.

### Table 1.
Summary of the features of the available software frameworks for robotics: The XBot was developed from scratch given the limitations and the missing features of the presented existing framework.

<table>
<thead>
<tr>
<th>Framework</th>
<th>RT</th>
<th>HAL</th>
<th>IPC Complexity</th>
<th>Ready-to-Use</th>
<th>Community</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROS</td>
<td>No</td>
<td>Yes</td>
<td>Low</td>
<td>Yes</td>
<td>Big and active</td>
</tr>
<tr>
<td>YARP</td>
<td>No</td>
<td>Yes</td>
<td>High</td>
<td>Yes</td>
<td>Medium and active</td>
</tr>
<tr>
<td>OROCOS</td>
<td>Yes</td>
<td>No</td>
<td>Very High</td>
<td>Yes</td>
<td>Medium and inactive</td>
</tr>
<tr>
<td>OpenRT-M</td>
<td>Yes</td>
<td>Yes</td>
<td>Very High</td>
<td>Yes</td>
<td>Part of docs in Japanese</td>
</tr>
<tr>
<td>ROS 2</td>
<td>Yes</td>
<td>Yes</td>
<td>High</td>
<td>No</td>
<td>Small and Active</td>
</tr>
<tr>
<td>PODO</td>
<td>Yes</td>
<td>Yes</td>
<td>Very High</td>
<td>No, not available</td>
<td>KAIST group only</td>
</tr>
<tr>
<td>IHMC OpenJDK</td>
<td>Yes, low performance</td>
<td>No</td>
<td>Medium</td>
<td>Yes</td>
<td>Small</td>
</tr>
<tr>
<td><strong>XBot</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Low</td>
<td>Yes</td>
<td>Small</td>
</tr>
</tbody>
</table>

YARP (Yet Another Robot Platform) [6] and ROS (Robot Operating System) [7] are popular component-based frameworks for IPC that do not guarantee RT execution among modules/nodes. It is essential for a robotic system to have a component responsible for the RT control of the robot, making these frameworks only viable as external (high-level) software frameworks. It is worth to mention that a new ROS version, called ROS 2 [9] has been released: it is still in an early stage development phase, so it cannot be used in real-world scenario [3].

PODO [8], is the framework used by KAIST in HUBO during the DRC (Darpa Robotics Challenge) Finals. Its control system has RT control capabilities and its inter-process communication facilities are based on POSIX IPC; moreover it uses a shared memory system called MPC to exchange data between processes in the same machine. This heterogeneous system has the potential to cause confusion as it is unclear which architectural style must be used to communicate with a specific component [9].

In [10] an RT architecture based on OpenJDK is introduced (used by IHMC during the DRC Finals). Nevertheless, to their own admission [11], none of the commercially available implementations of the Java Real Time Specification had the performance required to run their controller. Existing Real-time Java Support is insufficient.

Considering the above limitations, summarized in Table 1, the XBot [12–14] was developed from scratch, in order to have a reliable RT control framework with HAL support and without depending on complex IPC frameworks.

### 3. XBot framework

The development of XBot was driven by the need to provide a software infrastructure that abstracts the diverse variability of the robotic hardware (effectively becoming a cross-robot framework), provides deterministic hard Real-Time (RT) performance, incorporates interfaces that permit its integration with state of art robot control frameworks and achieves enhanced flexibility through a plug-in architecture.
In the next sections, the XBot design goals and the software architecture insights are going to be described following a bottom-up approach going from the hardware towards the high-level control of the robotic system.

### 3.1 Design goals

The considerations and limitations of the existing frameworks, described in detail in the previous section, motivated the development of the XBot framework bearing in mind that the design of a software platform, which lies at the foundations of such complex and diverse robotic systems, is the most crucial phase in the software development process. XBot was designed to be both an RT control system and a user friendly, flexible and reusable middleware for RT and non-RT control software modules. XBot was developed starting from the following design goals and features:

- **Hard RT control performance**: it must perform computation inside specific timing constraints with minimum timing jitter. There are several operating systems or platforms which support RT operation, including Windows CE, INtime, RTLinux, RTAI, Xenomai, QNX and VXWorks. Xenomai\(^4\) [15], a Linux based Real-Time Operating System (RTOS) was selected to avoid a licensed product that does not give the possibility to modify and adapt the source code to fit it to the specifications of the system. Moreover Xenomai satisfies the requirements for extensibility, portability and maintainability as well as ensuring low latency as stated in [16, 17].

- **High control frequency**: robotics applications may often require high frequency control loops, e.g. RT pattern generator for bipedal walking, impedance regulation controllers or force feedback modules.

- **Cross-Robot compatibility**: it should be possible to use it with any robot, without code modification. It is crucial to be able to reuse the software platform with different robots, or subsystems of the same robotic platform.

- **External Framework integration**: it should be possible to use XBot as a middleware for any kind of external software framework (RT or non RT) without tailored software or specific bridges for every different case.

- **Plug-in Architecture**: users and third parties should be able to develop and integrate their own modules. In a robotic system platform a highly expandable software structure is needed.

- **Light-weight**: small number of dependencies on other libraries, it should be easy to install and set up. It is expected to run XBot on embedded PCs with low performance requirements in terms of memory and CPU. Therefore, it should demonstrate a small footprint to avoid high CPU usage.

- **Simplicity**: it must be simple. Complex systems may have unneeded and over-engineered features. For robotics application full control over the software platform is required. KISS (“Keep It Simple, Stupid”) principle is essential and unnecessary complexity should be avoided.

\(^4\) https://xenomai.org/
• **Flexibility**: XBot has to be easily modified or extended to be used in systems and applications other than those for which it was specifically designed.

Finally, the XBot software framework was not developed to address the requirements of a specific robotic platform, instead its implementation is flexible, generic and cross-robot. Furthermore it does not directly depend on any existing software or control platform, but it provides to the user the functionality to easily integrate any RT or non-RT framework. To obtain the above features, a user of the XBot with a generic robotic platform to control, has to provide a set of configuration files, mainly related to the kinematics and dynamics of the robot, the control plugins to execute, the HAL implementation, the high level communication framework and the kinematic/dynamic engine to use.

### 3.2 Software architecture

As presented in **Figure 1**, the XBot software architecture is composed of different components, described in detail within the following sections. In particular the design choices, from a software engineer point of view, are the results of the design goals described in Section 3.1. To avoid scheduling issues and keep the complexity of the software infrastructure as low as possible only two RT threads and one non-RT thread are currently employed in the framework as presented in **Figure 2**. The RT layer contains the R-HAL (Robotics Hardware Abstraction Layer) to assure cross-robot compatibility and seamless porting of the higher level code from the simulation to the real robot. The communication mechanism employed in this layer is a shared memory one using basic synchronization methods (i.e. mutex and condition variables). On top of the R-HAL, the **Plugin Handler** is designed using a

---

**Figure 1.**
XBot software architecture: components overview and interaction. From the bottom the R-HAL and the plugin handler inside the RT Xenomai layer are presented, with the shared memory communication between them. The non-RT layer and the external software integration component of the XBot is represented by the Communication Handler which is able to communicate with the robot/simulation through a XDDP mechanism which assures lock-free IPC. Thanks to the XBotInterface on the left, all the layers of the framework have an uniform way (through an API) to send commands and receive state from the robot.
component-based software design paradigm by featuring a clear component concept (the plugin as a shared library) with well-defined structure and communication interfaces. In order to be able to assure external software integration, the communication with the non-RT layer, represented by a standalone component called Communication Handler, is implemented using a lock-free Inter Process Communication (IPC) mechanism based on XDDP (described in details in the following sections). The same concept is applied to communicate with other RT frameworks (e.g. OROCOS), using a mechanism based on IDDP. The idea of the NRT Deployer came from the need to have a behavior similar to the one of the Plugin Handler but completely in the non-RT layer. Finally a standard way of communicating with the robot regardless of its specific structure (humanoid, quadruped, manipulator, etc), and also independently of the particular software layer that the user wants to operate within, is provided by mean of the XBotInterface.

### 3.2.1 R-HAL

The Cross-Robot compatibility feature is achieved through the development of a suitable hardware abstraction layer [18], which enables the user to efficiently port and run the same control modules on different robots, both in simulation and on the real hardware platforms. The main goal of this software component is to provide an independent layer in between the robot hardware and the high-level control, enabling the seamless integration of new actuators, sensors or other hardware components.

Concerning the threads configuration, XBot employs a separate thread to execute the low-level robot control loop and permits to realize separate controllers with different frequencies. The synchronization between the Plugin Handler thread and the R-HAL thread is implemented using condition variables, assuring the safe access of the shared data structures.

XBot currently supports EtherCAT (for robots like WALK-MAN, CENTAURO and COMAN+), Ethernet (for COMAN), and KUKA LWR 4/KUKA LBR arm based robots [19–21]. The possibility to simulate the robot and its controllers behaviors prior to testing on the real hardware is essential, especially when dealing with complex robotic systems. To achieve this we provide an R-HAL implementation for the well known Gazebo simulator environment Figure 3. In particular we rely on the Gazebo ModelPlugin class to be part of the Gazebo internal loop.

---

5 http://gazebosim.org/
3.2.2 Plugin handler

The main component of the XBot architecture is called PluginHandler and it is represented in Figure 1 with the dark pink color. The software design of this component relies on two core requirements for a robotic system (described in 3.1): the RT control and the highly expandable software structure. To achieve this the PluginHandler is implemented using a single RT thread running at high frequency (e.g., 1 kHz) and it is responsible for the following actions with the order they appear below: load the set of plugins requested by the user from a configuration file, initialize all the loaded plugins, and start them upon user request, execute the plugins that have been started sequentially, reload and reinitialize a plugin upon user request, close and unload all the loaded plugins. In Figure 4, the UML state diagram representing the life-cycle of a plugin is presented.

The Plugin implementation is compiled as a shared object library (.so). In details a Plugin is a simple class inherited from the abstract class XBotControlPlugin; this means that writing a Plugin is straightforward for the user, as the only need is to implement three basic functions:

- an init_control_plugin() function, which is called by the PluginHandler after the plugin is loaded/reloaded and is useful to initialize the variables of the Plugin
- a control_loop() function, which is called in the run loop of the PluginHandler after the plugin is started
- a close() function, which is called in the PluginHandler closing phase

After the design and the implementation of the latency-free, hard real-time layer the next significant feature is accompanied by the implementation of flexible
3.2.3 Communication handler

The above mentioned software components do not give the possibility to communicate with external modules/hosts outside the robot: for this purpose the software framework of a robotic system should incorporate a set of non-RT threads that permit the communication of the system with remote pilot stations or cloud services. *XBots* provides this with the implementation of the Communication Handler component (represented in Figure 1 with the yellow color) that is a non-RT thread exploiting an XDDP (Cross Domain Datagram Protocol) handler with the ready-to-use *XBots* non-RT API for a set of components called CommunicationInterface(s). The non-RT API uses XDDP Xenomai pipes to achieve asynchronous communication between RT and non-RT threads. A lock-free inter-process communication (IPC) is employed to permit the RT control threads to exchange messages with the non-RT communication threads without any context switch. The execution loop of the Communication Handler thread is responsible for updating the internal robot state using the XDDP pipe with the non-RT robot API, sending the robot state to all the communication frameworks implemented as CommunicationInterface(s), receiving the new reference from the “master” CommunicationInterface (to avoid having multiple external frameworks commanding the robot) and finally for sending the received reference to the robot using the XDDP non-RT robot API.

It is relatively straightforward to add a new CommunicationInterface in the framework: *XBots* provides built-in support for YARP and ROS communication.
frameworks, meaning that the end-users has YARP control board wrappers / analog sensors and ROS joint state / command messages already available. Interoperability for YARP/ROS framework and XBot is one of the key feature offered by the Communication Handler.

In the specific ROS case, XBot provides two families of interfaces:

1. a joint space interface, consisting of standard ROS topics that are advertised/subscribed by the Communication Handler itself in order to publish the robot state (including sensors) and accept commands

2. a set of tools for using a subset of ROS inter-process communication capabilities from the RT domain

ROS-powered robots expose to their users an interface that is mainly based on topics. For instance, the robot joint state is usually published to a /joint_state topic through sensor_msgs/JointState messages. The same kind of message can be published by the user on a /command topic in order to control the robot.

Inside XBot a similar interface to the ROS middleware is offered, the only difference lies in the message type being used. Broadly speaking, XBot uses an extended joint state message that make it possible to perform more flexible control of the robot than is allowed with standard ROS.

The XBot framework provides also the integration with any external RT software framework (e.g. OROCOS) thanks to the use of the IDDP (Intra Domain Datagram Protocol) pipes for the RT inter-process communication.

Inside the Communication Handler component, the XBot framework provides the user with the possibility of running non-RT plugins that are useful for performing Input/Output operation from the non-RT layer to the RT layer. The so called IOPlugins are very similar to the Plugin used in the RT layer, in fact the implementation is compiled as a shared object library (.so). In detail, an IOPlugin is a simple class inheriting from the abstract class XBot::IOPlugin, which gives to the user simple access to the shared memory component and the pipes to communicate between the non-RT layer and the RT one. As for the standard Plugin, XBot provides a ready-to-use skeleton (simple script to run) for the user.

4. Experimental validation

In this section the results of the validation of the overall framework is going to be presented with particular focus on the flexibility in terms of integration with different robots and external software frameworks, and also on the overhead introduced by the XBot while assuring predictable response time at high frequency (i.e. 1 kHz) control loop.

4.1 Experimental setup

To evaluate the performance of the XBot software platform, two sets of experiments were performed: in the experiment set 1 the WALK-MAN [22, 23] robot was used, a humanoid with 33 Degree-Of-Freedom, 4 custom F/T sensors and 1 VN-100 imu. The WALK-MAN vision system is composed of a CMU Multisense-SL sensor that includes a stereo camera, a 2D rotating laser scanner, and an IMU. The robot control modules were based on GYM [24] (Generic Yarp Module), a component model to easily develop software modules for robotics leveraging the YARP ecosystem: YARP Based Plugins for Gazebo Simulator were used to validate the control
modules in simulation. Whole-body control and inverse kinematics are solved through the OpenSoT control framework [25]. Figure 5 reports a representation of the software components in use for experiment set 1.

In the set 1 evaluation different high-level software frameworks were successfully integrated on top of XBot: ArmarX [26] perceptual pipeline for hierarchical affordance extraction [27], OpenSoT previewer based on the MoveIt! ROS library.

Figure 5. XBot validation experiment set 1 software components: How they are allocated in the two WALK-MAN embedded PCs (i.e. N-RT WALKMAN EXP and RT WALKMAN EXP).

Figure 6. XBot validation experiment. On the left set 1 setup: WALK-MAN needs to remove a set of objects in order to perform the task of turning the valve. On the right CENTAURO bi-manual robot platform used in experiment set 2.
for motion feasibility analysis and collision checking and a manipulation GYM module, OpenSoT based, using the YARP communication framework.

The set 1 experiments were carried out in a DRC-inspired scenario targeting the removal of debris in front of a valve. In Figure 6 the experimental setup is shown.

In [28], ArmarX was integrated with the robot software environment YARP taking advantage of the built-in YARP CommunicationInterface for the external software framework integration with XBot.

In the experiment set 2 an RT end-effector Cartesian Control on two different robots was performed: the aforementioned WALK-MAN and CENTAURO (in Figure 6). CENTAURO [20, 29, 30] upper body is a high performance human size and weight compatible bi-manual manipulation platform with 15 DOFs. Each arm has 7 DOF and the trunk has 1 DOF that permits the yaw motion of the entire upper body and extends the manipulation workspace of the robot.

In the set 2 experiments two RT Plugins were used: the first one, called IKCommunication to receive the end-effector pose from the Communication Handler (with the built-in ROS CommunicationInterface) through the XDDP pipes and OpenSoTRTIK to solve the inverse kinematics. The evaluation was focused on the overhead introduced by the IKCommunication RT plugin that exploits two communication mechanism offered by XBot: XDDP to receive the data from the non-RT layer and XBotSharedMemory to communicate these data to the other RT plugin (OpenSoTRTIK).

4.2 Results

In the set 1, XBot performance in terms of control period of the RT plugin XBotCommunicationPlugin and CPU usage were analyzed: during the experiments, each millisecond, all the data flowing from/to the R-HAL were recorded, using the XBot RT low-level logging tools.
Figure 7. In Figure 7 the RTT (Round Trip Time) measured by the EtherCAT master implementation of the R-HAL during the set 1 experiments in the worst-case scenario is shown, i.e. while the robot was performing the manipulation actions: it is

Figure 8. Experiment set 1: XBot CPU core comparison: Robot idle vs. robot running the experiments. The difference between the two bold lines represents the actual overhead introduced by the middleware when executing the control modules described in the experimental setup for set 1.

Figure 9. Experiment set 2: Communication overhead (RT - RT and N-RT - RT) introduced by XBot. Experiment results on both WALK-MAN and CENTAURO are shown.

In Figure 7 the RTT (Round Trip Time) measured by the EtherCAT master implementation of the R-HAL during the set 1 experiments in the worst-case scenario is shown, i.e. while the robot was performing the manipulation actions: it is
clear that the mean control period is below the 1000 μs (i.e. 1 kHz control frequency) even if the RT system is communicating with the high-level software components through XBot the built-in YARP CommunicationInterface non-RT threads. Only two of the RTT measurement (over 200000) were above the requested control period because of missing PDO (Process Data Objects) round in the beckhoff\textsuperscript{6} chip responsible for the EtherCAT communication.

In Figure 8 a comparison is presented between XBot CPU usage while the robot is idle (i.e. not moving, nor communicating with external software frameworks) and when the set 1 manipulation experiments are running: the CPU core usage overhead introduced by XBot when the robot is performing the manipulation task as described above, is only 1.2% (on average). Furthermore it is clear that the CPU usage of XBot is very low (always ranging from 11.7% to 14.2%).

In the set 2 the focus was placed on the communication overhead introduced by XBot: both the XDDP pipes (communication between RT and non-RT layers) and the XBotSharedMemory (RT plugins communication) are taken into account. As shown in Figure 9 the mean execution time of the IKCommunication RT plugin is around 1.2 μs for both WALK-MAN and CENTAURO experiments. This means that it is possible to send end-effector reference poses and receive back the robot state from a non-RT framework, while controlling the robot (at 1 kHz in the experiments) using a RT plugin implementing the IK (OpenSoTRTIK in the experiments), with negligible overhead (Figure 10).

\textsuperscript{6} https://www.beckhoff.it/
5. Conclusions

In this chapter the XBot\textsuperscript{7} RT software architecture was presented. It provides to
the users a software infrastructure which can be used with any robotic systems
enabling fast and seamless porting of the code from one robot to the other, requir-
ing no code changes, assuring flexibility and reusability. The implementation of
the framework ensures easy interoperability and built-in integration with other existing
software tools for robotics, such as ROS, YARP or OROCOS. The component-based
development of the XBot includes a Robotic Hardware Abstraction Layer (R-HAL)
interface and a set of ready-to-use tools to control robots either within a simulation
environment or the real hardware.

The framework has been successfully used and validated as a main software
infrastructure (\textbf{Figure 5}) for humanoid robots such as WALK-MAN (result of
WALK-MAN EU FP7 project\textsuperscript{8}, notably XBot received the EU innovation radar
award in this context\textsuperscript{9},) and COMAN+ (result of COGIMON EU H2020 project\textsuperscript{10})
or for quadruped centaur-like robots as CENTAURO (result of the CENTAURO EU
H2020 project\textsuperscript{11}). Moreover the cross-robot functionality has been exploited to
develop both RT and non-RT control modules not only for the above mentioned
robots, but also for commercial robotic systems such as KUKA LBR, KUKA 4+ or
Franka Emika Panda, or other humanoid robots like COMAN or iCub.

Regarding the simulation part, XBot enables the direct porting of the control mod-
ules tested in the simulator to the real hardware using the same interfaces and without
requiring any code modifications. The built-in simulator supported in the framework
is Gazebo, but there is the option to support other simulation environments (as it
happened inside the CENTAURO H2020 project with the VEROSIM simulator\textsuperscript{12}).

XBot currently relies on a dual-kernel approach using Xenomai, which performs
better than PREEMPT_RT\textsuperscript{13}, both in terms of system predictability and absolute
latencies. Nevertheless Xenomai in the long term can introduce disadvantages by
making the software development more complex, which means harder maintain-
ability and lower portability.

Further development of the framework will target to provide synchronized
distributed execution of multiple RT threads in multiple computational units. In fact
currently the Plugin Handler is only able to execute a set of plugins in sequence,
without any concurrency. This makes the maintenance of the framework easier, but
restricts the performance in terms of computation power. Moreover the current
architecture is characterized by a unique point of failure since both the R-HAL
thread and the Plugin Handler (which executes RT plugins) thread run in the same
process. In fact, there is the possibility that a misbehaving RT plugin might cause
memory corruption, or crash altogether, causing also the R-HAL to crash. Currently
only expert users are allowed to load their RT plugins in the Plugin Handler, but it is
desirable to eventually separate the R-HAL and Plugin Handler either in two
different processes or in two different machines to improve isolation.

\textsuperscript{7} https://github.com/ADVRHumanoids/XBotControl
\textsuperscript{8} https://www.walk-man.eu/
\textsuperscript{9} https://www.innoradar.eu/innovation/30632
\textsuperscript{10} https://cogimon.eu/
\textsuperscript{11} https://www.centauro-project.eu/
\textsuperscript{12} https://www.verosim-solutions.com/en/
\textsuperscript{13} PREEMPT_RT was introduced to have RT capabilities in the Linux kernel avoiding the adoption of a
dual-kernel.
Acknowledgements

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 101016007 CONCERT, and the European Union’s Horizon 2020 Research and Innovation Programme (H2020-ICT-2019-2/ 2019-2023) under grant agreement No. 871237 SOPHIA and the Italian Fondo per la Crescita Sostenibile – Sportello “Fabbrica intelligente”, PON I&C 2014 - 2020, project number F/190042/01-03/X44 RELAX.

Author details

Luca Muratore*, Arturo Laurenzi and Nikos G. Tsagarakis
Humanoids and Human Centered Mechatronics (HHCM), Istituto Italiano di Tecnologia, Genova, Italy

*Address all correspondence to: luca.muratore@iit.it
References


