On the Design and Evaluation of XBotCore, a Cross-Robot Real-Time Software Framework

Luca Muratore, Arturo Laurenzi, Enrico Mingo Hoffman, Alessio Rocchi, Darwin G. Caldwell, Nikos G. Tsagarakis

Advanced Robotics Department (ADVR), Istituto Italiano di Tecnologia, Genova, Italy
School of Electrical and Electronic Engineering, The University of Manchester, M13 9PL, UK

Abstract—In this work we introduce XBotCore (Cross-Bot-Core), a lightweight, Real-Time (RT) software platform for EtherCAT-based robots. XBotCore is open-source and is designed to be both a RT robot control framework and a software middleware. It satisfies hard RT requirements, while ensuring 1 kHz control loop even in complex Multi-Degree-Of-Freedom systems. It provides a simple and easy-to-use middleware Application Programming Interface (API), for both RT and non-RT control frameworks. This API is completely flexible with respect to the framework a user wants to utilize. Moreover it is possible to reuse the code written using XBotCore API with different robots (cross-robot feature). In this paper, the XBotCore design will be described and the experimental results on the humanoid robot WALK-MAN and on the bi-manual platform CENTAURO, both developed at the Istituto Italiano di Tecnologia (IIT), will be presented.

Index Terms—Software Architectures, Control Architectures and Programming, Real-time systems and embedded systems, Humanoid Robots.

1 INTRODUCTION

ONE of the main challenges when developing a complex robotic system is the design and the implementation of a software architecture, essential for the interaction and the coordination of hardware and control modules. Ever more frequently a robot control system has to be able to perform critical tasks in an autonomous way, satisfying hard RT requirements, i.e. it must guarantee predictable response times. Furthermore it is necessary to have a software middleware capable of abstracting the complex hardware (e.g. actuators and sensors) of the robot providing a simple, standardized API to control the system. The robotics middleware should be modular, easy-to-use, robust, reliable, easy to maintain, efficient, flexible and should provide support for multi-threading [1]. As a distributed control system the hardware components of the robot have to communicate using a field-bus system with RT communication capabilities: we selected EtherCAT (Ethernet Control Automation Technology), an industrial protocol built on the Ethernet (IEEE 802.3) specifications that assures:

- high transmission rate
- minimum roundtrip (reaction) time, w.r.t. other industrial protocols (e.g. CAN, Profinet, etc.) [2]
- precise synchronization (< 1µs) by exact adjustment of Distributed Clocks
- flexible topologies: Line, Star, Tree, Daisy Chain + Drop Lines can be used in any combination
- easy configuration and implementation
- cost effectiveness

EtherCAT combines an efficient and relatively high speed message transmission, with the predictability imposed by a master/slave medium access control policy. All the message reception, data processing and frame retransmission operations are made "on the fly" by the slave nodes, without any extra delays. Special hardware components, embedded in the slave’s Ethernet interface, are responsible for these operations. RT scheduling is essential for precise robot control period, especially for high-frequency (e.g. 1 kHz): there are several operating systems or platforms which support RT operation, like Windows CE, INtime, RTLinux, RTAI, Xenomai, QNX, VXWorks. We selected a Linux based RTOS because we
want to avoid a licensed product that does not give us 
the possibility to modify the source code depending on our 
system. Xenomai is our choice because its design considers extensibility, portability and maintainability as well as low latency [3], furthermore it has been already used successfully in many robotics hardware.

XBotCore is not specific to a single robot or to a class of 
robots: its implementation is flexible, generic and cross-
robot. Furthermore it does not depend on any existing software 
platform, but it gives to the user the opportunity to easily 
integrate any RT or non-RT framework.

The rest of the paper is organized as follows: Section II 
presents the related work, while section III introduces the 
design goals of XBotCore. Section IV describes in details 
the XBotCore main components and section V presents the 
experimental trials and results. Finally section VI addresses 
the conclusions.

2 RELATED WORK

In [4] a low level control framework, called OROCOS (Open 
Robot Control Software), is introduced, which provides a set of 
components for RT control of robotic systems. It relies on the 
Common Object Request Broker (CORBA) architecture, 
that allows inter-process and cross-platform interoperability for 
distributed robot control. We decided not to depend on any 
Inter-Process-Communication (IPC) framework in order to 
avoid increasing the complexity of the software platform.

Very similar to OROCOS is OpenRT-M [5], developed in 
Japan from 2002 under NEDOs (New Energy and Industrial 
Technology Development Organization) Robot challenge pro-
gram. It is based on CORBA, so similar considerations as 
for OROCOS can be made w.r.t. the software complexity; 
moresover part of OpenRT-M documentation is in Japanese.

YARP (Yet Another Robot Platform) [6] and ROS (Robot 
Operating System) [7] are popular component-based frame-
work for IPC that do not guarantee RT execution among mod-
ules/nodes. It is essential for us to have a component responsible for the RT control of the robot, making these frameworks only viable as external (high-level) software frameworks.

PODO [8] is the framework used by KAIST in HUBO 
during the DRC (Darpa Robotics Challenge) Finals. Its con-
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.

Figure 1. The diagram shows XBotCore components interactions: an EtherCAT master RT thread communicates with the EtherCAT slaves and with the Plugin handler RT thread which schedules execution of a set of RT plugins. The Communication Handlers non-RT threads allow interfacing with non-RT external Software Frameworks

Considering the above limitations, we started developing 
XBotCore from scratch, in order to have a reliable RT control 
framework without depending on complex IPC framework.

3 DESIGN GOALS

The design of a software platform that lies at the foundations of a complex system, such as a robotic system, is the most 
crucial phase in the software development process. XBotCore 
was designed to be both an RT control system and an easy-
to-use, flexible and reusable middleware for RT or non-RT 
tasks. XBotCore design goals are the following:

- **Hard RT control system**: it must perform computation within predictable timing constraints
- **High control frequency**: robotics applications may require high frequency control loops, e.g. 1 KHz or 500 Hz for RT Pattern Generator for Biped Walking or haptics applications
- **Cross-Robot compatibility**: it has to work with any kind of EtherCAT-based robot, without any code modification. It is crucial to be able to reuse the software platform with different robots, or different part of the same robot
- **External Framework integration**: it has to be possible to use XBotCore as a middleware for any kind of external software framework (RT or non-RT)
• **Plug-in Architecture**: users and third parties should be able to develop their own modules. In a robotic system platform we need an highly expandable software structure.

• **Light-weight**: we don’t want too many dependencies on other libraries, it should be easy to install and set up. Moreover we expect to run XBotCore on embedded PCs with low performance requirements in terms of memory and CPU. We therefore need a small footprint and to avoid high CPU usage.

• **Simplicity**: it must be simple. Complex systems may have unneeded and over-engineered features. For robotics application we need the full control over the software platform. *KISS* ("Keep It Simple, Stupid") principle is essential; simplicity is a key goal in XBotCore design and unnecessary complexity should be avoided.

• **Flexibility**: XBotCore has to be easily modified or extended in order to be used in applications or environments other than those for which it was specifically designed.

• **Open-source**: open-source software provides transparency in the software implementation since any developer can study and modify the code, eventually to the benefit of the robotics community. Moreover a flexible license is essential for the free distribution of XBotCore in other open-source projects.

4 **XBotCore**

As shown in Figure 1 XBotCore consists of 5 main components: EtherCAT master, Plugin Handler, XBotCoreModel, RT and non RT middleware API and Communication Handlers.

4.1 EtherCAT master

XBotCore is designed for EtherCAT based robots: we expect a network of EtherCAT slaves in the system, i.e. the electronic boards responsible for motor control and sensors data acquisition. The EtherCAT protocol has a master/slave medium access control policy that works in this way: there is a single master node on the network that has the right to initiate data transfers; this node sends an Ethernet frame to the slave nodes. Each slave node extracts the data from the frame addressed to it, puts some new data in the frame and then sends the frame to the next slave. The frame arrives back to the master node confirming the correctness of the transmission.

XBotCore EtherCAT master implementation is developed starting from the SOEM (Simple Open EtherCAT Master) library, an open source implementation, meant to be highly portable on a variety of embedded platforms (HW and RTOSe) [12]. The structure of the data flowing in the EtherCAT network is called PDO (Process Data Object) and it has two different sub-structures: PDO RX: master input, slave output e.g. link position, motor position, motor velocity, torque, temperature etc. and PDO TX: master output, slave input e.g. position reference, torque reference, gains etc.

![Figure 2](image)

**Figure 2.** XBotLoggingPlugin UML class diagram: it derives from XBotControlPlugin implementing the init control plugin(), control loop() and close() pure virtual functions.

Furthermore the XBotCore EtherCAT master provides an asynchronous API to the higher level components in order to read/write the PDO data.

4.2 Plugin Handler

The Plugin Handler is the main component of the RT plugin architecture: it is an RT thread responsible to start all the loaded plugins, execute them sequentially (as in [13]), and close them before unloading them. A Plugin is a class inheriting from the abstract class XBotControlPlugin. The Plugin implementation is compiled as a shared object (.so). It is possible to dynamically load and unload one or more plugins in the Plugin Handler. Writing a Plugin is straightforward for the user, as he just needs to implement three basic functions:

- an **init control plugin()** function that will be called only once by the Plugin Handler in order to initialize the variables of the Plugin
- a **control loop()** function which will be executed in the run loop of the Plugin Handler
- a **close()** function that will be called in the Plugin Handler closure phase

In Figure 2 the UML class diagram of a simple RT plugin to log data is shown. The Plugin implementation is compiled as a shared object (.so). There is the possibility to dynamically load and unload one or more plugin in the Plugin Handler: it
4.3 XBotCoreModel

XBotCore implies a novel approach to the configuration of low-level control systems by using modern description formats such as URDF (Universal Robotics Description Format) and SRDF (Semantic Robotic Description Format), traditionally used for high-level software components (e.g., ROS nodes). Its main feature is to be a cross-robot software platform: thanks to the abstractions provided by the XBotCoreModel class it is possible to control different robots or different parts of the same robot without code modifications. In fact the API provided to control the robot is dynamically built starting from the URDF and SRDF of the robot. Modifying the SRDF, removing for example a kinematic chain (e.g., the torso of the robot), results in a different API for the user that is compatible with the available/desired parts of the robot to control. The same happens when the URDF is modified, e.g., when working with a different robot.

4.4 RT and non-RT middleware API

XBotCore is also a middleware that provides the user with both RT and non-RT APIs. The RT API is suitable for the RT plugins that will run in the Plugin Handler: it works using a shared memory communication mechanism with the low level RT EtherCAT thread. The interfaces implemented by the RT API are: IXBotJoint (abstraction of the robot joints), IXBotFT (abstraction of the robot Force/Torque (F/T) sensors) and IXBotIMU (abstraction of the robot IMU sensors).

The non-RT API implements similar interfaces (i.e., IXBotJoint, IXBotFT and IXBotIMU), but it uses XDDP (Cross Domain Datagram Protocol) Xenomai pipes in order to have asynchronous communication between RT and non-RT threads. It is crucial to have a lock-free interprocess communication (IPC) in a robotic system: RT control threads are able to exchange messages with non-RT communication threads without any context switch.

4.5 Communication Handlers

None of the above-mentioned software components give the possibility to communicate with the external modules/hosts outside the robot: for this purpose a robotic system needs to be equipped with a set of non-RT threads communicating with a remote pilot station or cloud services. In XBotCore the XBotCommunicationHandler component is provided with the CommunicationInterface abstract class: instances of classes inheriting from it implement the functions to send the robot state from the non-RT API to the chosen communication framework and receive the reference provided by the external framework to send to the robot using the non-RT API.

XBotCommunicationHandler is a thread that provides an XDDP handler with the ready-to-use XBotCore non-RT API for the CommunicationInterface(s). The run loop of this component is quite simple: it updates the internal robot state using the XDDP pipe with the non-RT robot API, it sends the robot state to all the communication frameworks implemented as CommunicationInterface(s), it receives the new reference from the requested “master” CommunicationInterface (we avoid to have multiple external framework commanding the robot) and it sends the received reference to the robot using the XDDP non-RT robot API.
Figure 5. XBotCore validation experiment set 1 setup: WALK-MAN needs to remove a set of objects in order to perform the valve turning.

API. In Figure 3 it is shown the communication mechanism and the threads deployed inside XBotCore.

It is pretty straightforward to implement a new CommunicationInterface: XBotCore 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 XBotCore is one of the key feature offered by the XBotCommunicationHandler.

5 EXPERIMENTS

5.1 Experiments description

To validate and evaluate the performance of the XBotCore software platform, we performed two sets of experiments: in the experiment set 1 we used the WALK-MAN [14] robot, a full-size humanoid with 33 DOFs (Degree-Of-Freedoms), 4 custom F/T sensors and 1 VN-100 imu. The WALK-MAN head is equipped with 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 [15] (Generic Yarp Module), a component model to easily develop software modules for robotics leveraging the YARP ecosystem: YARP Based Plugins for Gazebo Simulator [16] were used to validate the control modules in simulation. Whole-body control and inverse kinematics are solved through the OpenSoT control framework [17]. Figure 4 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 XBotCore: ArmarX [18] perceptual pipeline for hierarchical affordance extraction [19], OpenSoT previewer based on the MoveIt! ROS library 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 5 the experimental setup is shown.

In [20], ArmarX was integrated with the robot software environment YARP: in this way we took advantage of the built-in YARP CommunicationInterface for the external software framework integration with XBotCore.

In the experiment set 2 we performed an RT end-effector Cartesian Control on two different robots: the aforementioned WALK-MAN and CENTAURO (in Figure 6). CENTAURO [21] is a high performance human size and weight compatible bi-manual manipulation platform with 15 DOFs. Each arm has 7 DOF, instead 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: one, called IKCommunication to receive the end-effector pose from the XBotCommunicationHandler (with the built-in ROS CommunicationInterface) through the XDDP pipes and OpenSoTRTIK to solve the inverse kinematics. We focused our evaluation on the overhead introduced by the IKCommunication RT plugin that is exploiting two communication mechanism offered by XBotCore: XDDP to receive the data from the non-RT layer and XBotSharedMemory to communicate these data to the other RT plugin(OpenSoTRTIK).
5.2 Results

In the set 1 we analyzed XBotCore performance in terms of control period of the RT plugin XBotCommunicationPlugin and CPU usage: during the experiments, each millisecond, we recorded all the data flowing from the EtherCAT master to the EtherCAT slaves and vice versa, thanks to the XBotCore low-level logging tool.

In Figure 7 we show the RTT (Round Trip Time) measured by the EtherCAT master during the set 1 experiments in the worst-case scenario, i.e. while the robot was performing the manipulation actions: it is clear that the control period is always below the 1000 µs (i.e. 1 kHz control frequency) even if the RT system is communicating with the high-level software components through XBotCore built-in YARP Communication Interface non-RT threads.

In Figure 8 a comparison is presented between XBotCore 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 XBotCore when the robot is performing the manipulation task as described above, is only 1.2% (in average). Furthermore it is clear that the CPU usage of XBotCore is very low (always ranging from 11.7% to 14.2%).

In the set 2 we focused our attention on the communication overhead introduced by XBotCore: 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 pose 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.

6 CONCLUSIONS

In this work we introduced an RT, robot-agnostic software platform called XBotCore.

XBotCore works with any EtherCAT based robot and it has a plugin-based software architecture: it is easy and not time-consuming to write a new RT control plugin. As a final remark we showed that external frameworks (e.g. ROS or YARP) are easy to integrate in XBotCore, thanks to his middleware API.

We evaluated XBotCore performance on the humanoid robot WALK-MAN and on the bi-manual robot platform CEN-
TAURO, showing the cross-robot capability of the framework and that it can ensure 1 kHz RT control loop during complex manipulation tasks, while communicating with the non-RT external frameworks. Moreover we demonstrated that XBotCore is light-weight: the CPU usage is very low, both when the robot is idle and when it is executing a task; also the communication mechanisms offered to read data from the non-RT layer (XDDP pipes) and to communicate between RT plugins (XBotSharedMemory) introduce very small overhead in terms of time (in average less than 2 μs in our experiments). XBotCore is released free and open-source at https://github.com/ADVRHumanoids/XBotCore

ACKNOWLEDGMENTS

The authors would like to thank Barry Lennox, Lorenzo Natale, Ali Paikan, Alberto Cardellino and Alessio Margan for the support. The authors want also to thank Tamim Asfour, Peter Kaiser and Markus Grötz from High Performance Humanoid Technologies Lab (H2T) in KIT, for their collaboration in the preparation of the experiment set 1.

REFERENCES