XBotCore: A Real-Time Cross-Robot Software Platform

Luca Muratore[1] [2], Arturo Laurenzi[1], Enrico Mingo Hoffman[1], Alessio Rocchi[1], Darwin G. Caldwell[1] and Nikos G. Tsagarakis[1]

***Abstract*— In this work we introduce XBotCore (*Cross-Bot-Core*), a light-weight, Real-Time (RT) software platform for EtherCAT-based robots. XBotCore is open-source and is designed to be both an 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 and architecture will be described and experimental results on the humanoid robot WALK-MAN [17], developed at the Istituto Italiano di Tecnologia (IIT), will be presented.**

I. 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 [4].

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, Profibus, etc.) [13]
- precise synchronization ($\ll 1\mu$s) 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 [2], 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.

II. Related Work

In [3] 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** [1], developed in Japan from 2002 under NEDOs (New Energy and Industrial Technology Development Organization) Robot challenge program. It is based on CORBA, so similar considerations as for OROCOS can be made w.r.t. the software complexity; moreover part of OpenRT-M documentation is in Japanese.

YARP (Yet Another Robot Platform) [11] and **ROS** (Robot Operating System) [14] are popular component-based framework for IPC that do not guarantee RT execution among modules/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 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 [7].

[1]Advanced Robotics Department (ADVR), Istituto Italiano di Tecnologia, Genova, Italy

[2]School of Electrical and Electronic Engineering, The University of Manchester, M13 9PL, UK

{luca.muratore, arturo.laurenzi, enrico.mingo, alessio.rocchi, darwin.caldwell, nikolaos.tsagarakis}@iit.it

In [16] an RT architecture based on OpenJDK is introduced (used by IHMC during the DRC Finals). Nevertheless, to their own admission [9], 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, we started developing *XBotCore* from scratch, in order to have a reliable RT control framework without depending on complex IPC framework.

III. 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
- **1 kHz control frequency**: robotics applications may require high frequency control loops, e.g. 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

IV. 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.

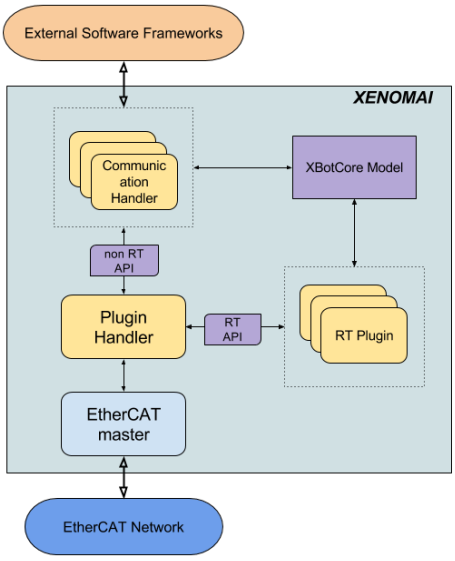

Figure 1. The diagram shows *XBotCore* components interactions: an EtherCAT master RT thread communicates with the EtherCAT slaves and with the Plugin handler RT thread which schedules execution of a set of RT plugins. The Communication Handlers non-RT threads allow interfacing with non-RT external Software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

A. EtherCAT master

XBotCore is designed for EtherCAT based robots: we expect a network of EtherCAT slaves in the system, e.g. the electronic boards responsible for motor control and sensors data acquisition. The EtherCAT protocol has a master/slave medium access control policy: *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 RTOSes) [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.

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

B. 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 [19]), and close them before unloading them. A *Plugin* is a class inheriting from the abstract class *XBotPlugin*. 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 functions:

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

- a `close()` function, called when the *Plugin Handler* wants to remove the plugin

C. XBotCoreModel

XBotCore implies a novel approach to the configuration of low-level control systems by using modern description formats such 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.

D. 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), `IXBotChain` (abstraction of the robot kinematic chain), `IXBotRobot` (abstraction of the robot) and `IXBotFT` (abstraction of the robot Force/Torque (F/T) sensors).

The non-RT API implements similar interfaces (i.e. `IXBotJoint` and `IXBotFT`, 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 IPC in a robotic system: RT control threads are able to exchange messages with non-RT communication threads without any context switch.

E. Communication Handlers

A robotic system has to communicate with the external world using a set of non-RT threads: in *XBotCore* the `XBotCommunicationHandler` class is provided; instances of classes inheriting from `XBotCommunicationHandler` run in non-RT threads, from which developers have access to ready-to-use non-RT API functions.

It is pretty straightforward to implement a new set of Communication Handlers: *XBotCore* provides built-in support for YARP and ROS communication frameworks.

V. Experiments

A. Experiments description

To validate and evaluate the performance of the *XBotCore* software platform, we performed a set of experiments on the WALK-MAN robot, a full-size humanoid with 33 DOFs (Degree-Of-Freedoms) and 4 custom F/T sensors. 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 are based on GYM [5] (Generic Yarp Module), a component model to easily develop software modules for robotics leveraging the YARP ecosystem: Yarp Based Plugins for Gazebo Simulator [6] were used to validate the control modules in simulation. Whole-body control and inverse kinematics are solved through the OpenSoT control framework [15].

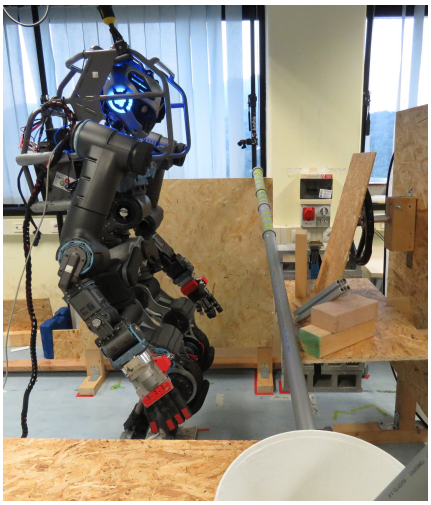

Figure 2. *XBotCore* validation experiments setup: WALK-MAN needs to remove a set of objects in order to perform the valve turning

In the evaluation different high-level software frameworks were successfully integrated on top of *XBotCore*: *ArmarX* [18] perceptual pipeline for hierarchical affordance extraction [10], Open-SoT previewer based on the *MoveIt! ROS* library for motion feasibility analysis and collision checking and a manipulation GYM module, Open-SoT based, using the *YARP* communication framework.

The experiments were carried out in a DRC-inspired scenario targeting the removal of debris in front of a valve. In Figure 2) the experimental setup is shown.

B. Results

We analyzed *XBotCore* performance in terms of control period of the RT plugins 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 an *XBotCore* low-level logging tool.

In Figure 3) we show the control period measured during the 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 1000μs (i.e. 1 kHz control frequency) even if the RT system is communicating with the high-level software components through YARP `XBotCommunicationHandler` non-RT threads.

In Figure 4) 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 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%).