

# A COMPACT SOFTWARE FRAMEWORK FOR DISTRIBUTED REAL-TIME COMPUTING

Oliver Wulf, Jan Kiszka, Bernardo Wagner

Institute for Systems Engineering

Real-Time Systems Group

University of Hannover

Appelstrasse 9A, D-30167 Hannover, Germany

{wulf,kiszka,wagner}@rts.uni-hannover.de

## Abstract

The amount of software in automatic control and mechatronic systems has been continuously growing in the last years. Within this domain, it is also getting more and more important to have software architectures and frameworks which allow efficient application development. On the other hand, size and computational overhead plays an important role on embedded controllers. For this reason, we have developed a compact software framework for distributed real-time computing which supports a component based system design. For all components, a small number of required common methods is defined. The framework provides uniform communication between the components based on package mailboxes. These mailboxes allow inter-component communication between components on the same system, real-time communication between distributed systems, and also communication between heterogeneous real-time and non real-time operating systems. The current implementation is used for control of autonomous service robots which are equipped with one or two Linux RTAI embedded PCs and a Windows laptop with a Java user interface. The communication and synchronisation between the two embedded PCs is done via real-time Ethernet using RTnet. The non real-time communication is managed by a TCP/IP router. This paper describes the concepts of the framework, the integration of RTnet, and the experiences we have with our robotics application.

## 1 Introduction

This paper describes a software framework which is used for the control of mobile service robots. The software project started in 2001 with the design of a software architecture for autonomous robots. The design fulfils the needs of robotics software but at the same time addresses general problems within modern automatically controlled systems. A complete service robot is a complex system which needs to combine a number of sensor inputs with actuators for driving or manipulation. To control the robot in an autonomous way, computationally expensive algorithms for sensor data fusion, environment perception and navigation need to be processed. Furthermore, hard real-time capabilities are needed to control the system in a safe and deterministic way. As service robots are about to operate in human environments, new ways of man-machine interaction need to be found. To allow research in this area, the software framework needs to be open for new

technologies like multimedia, wireless networking, or remote control via Internet.

Several approaches exist which cover either generic or robotic-specific software frameworks. A middleware that can be used for distributed computation is CORBA [4]. This specification, originally designed for non real-time systems, has been extended for real-time and embedded systems (RT-CORBA [5]). An implementation of RT-CORBA on top of a hard real-time operating system is challenging [6] and has not yet been completed for Open Source RTOSes like RTAI [7] which is used in our robot systems. The OROCOS project develops an Open Source robot operating system [1]. It is designed to satisfy the requirements of future robotics and control software. As it is based on CORBA, OROCOS is lacking a real-time capable inter-component communication mechanism. Therefore, it requires a concentration of real-time tasks on single computers. Furthermore, there are a number of other robot software projects, e.g. SmartSoft [2] or Dave's Robotic Operating System

[3], that do not fulfil the need for distributed real-time communication.

For this reason our paper describes the pragmatic design and implementation of a compact software framework for distributed real-time computing. The described solution is open to be used for various problems in automatic control, but the low complexity and the small overhead make it easy to implement and suitable for embedded PCs.

## 2 Component-Based Framework

The design of our component-based framework is divided into two parts. The first part which is described in section 2.1 introduces components and inter-component communication. On top of this basic concept, the framework specifies a number of definitions and tools that are described in section 2.2.

### 2.1 Framework Architecture

The software framework is based on two principal constituents: *modules* and *packages*. A common way to reduce the complexity of large systems is to divide the whole system into less complex components with a well defined function. These components are called modules within the described software framework. Modules are programs that can be executed individually as they have got no shared functions or memory with other modules. Communication between modules is only possible through packages. The advantage of this module definition is the complete encapsulation of the inner structure. Thus modules can be changed or replaced without the need to change the whole system. Furthermore, the encapsulation allows modules to be executed on distributed systems and allows modules to be implemented and executed with different programming languages and on different operating systems. Figure 1 illustrates an example taken from our robotics application.

The framework defines packages for prioritised hard real-time communication between modules. These packages are used as commands or replies of remote procedure calls (RPC) or they contain unidirectional data or events. For transmitting packages from one module to another, the software framework provides a uniform mechanism. The addressing is based on system wide unique mailbox names. All information like the position of the addressed module, available communication media, or the target operating system is hidden from the user. A description of this uniform communication is given in section 3.

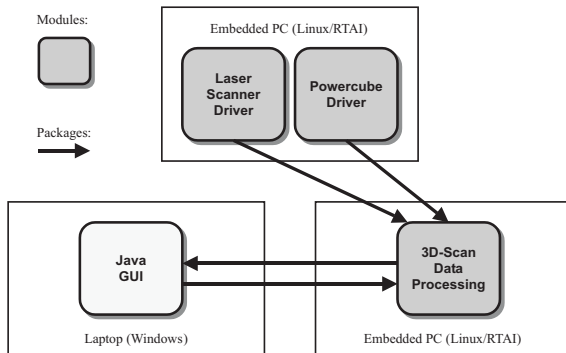


FIGURE 1: *Distributed Software Modules*

### 2.2 Basic Definitions and Services

In addition to the basic module and package concept, the software framework contains a number of definitions and tools that allow efficient development of complex robotics software systems.

#### 2.2.1 Module Structures and Interfaces

The communication framework itself lays no constraints on the internal structure of the modules, but, within the software architecture, it is useful to define a small set of common module functions and a common way to exchange data. Due to these definitions, most modules are similarly structured and based on a common template, which allows software reuse and creates easier understandable code. The common functions are *moduleOn*, *moduleOff* and *moduleGet-Status*. Modules can either be enabled, disabled, or in an error state.

The data flow between modules can be realised according to the polling model if the queried module supports the *getData* function. A module can furthermore implement the *getContinuousData* function, which allows continuous data acquisition at a specified frequency according to the publish/subscribe model.

As modules share no common memory and provide no directly callable functions, inter-module function calls are realised as remote procedure calls (RPC). Each RPC consists of sending a command package, waiting for the reply package, and handling the appropriate exceptions. As these repetitive constructions worsen the readability of the code, the RPC mechanism is encapsulated in so-called proxy functions which can be used like normal function calls.

#### 2.2.2 Naming and Prioritisation

To support the mailbox and therefore also module name assignment, the software architecture divides

the 32 bit mailbox namespace into four 8 bit sections: *system id*, *class id*, *instance id*, and *local id*. Each module is defined by its *class id* and *instance id*. The *local id* is used to differentiate between multiple mailboxes handled by the same module. The *system id* is reserved to address corresponding mailboxes located on different robots. This multi-robot communication is in principal possible but not practically used yet.

The assignment of task and message priorities turned out to be a problem while implementing the growing system. The absence of rules for assigning appropriate priorities and the lack of a system wide overview lead to inconsistent priorities. Therefore, both the module and package priorities are derived from the module's class id. This allows a centralised consistent management of all priorities, which is implemented by using a single configuration file.

### 2.2.3 Global Time Base

To allow accurate sensor data fusion, all data sets acquired from a sensor are stored together with a time stamp. If sensors are attached to different computers, it is necessary to have a common time base. For this reason, the framework provides a synchronisation mechanism that is based on a time master. The time master broadcasts its current time with a frequency of 1 Hz. The communication medium for this time broadcast needs to be hard real-time capable like the CAN bus. Based on the broadcasted master time and the respective local time, every node can calculate its time offset and thus the system time.

### 2.2.4 Console Output Service

One practical problem that occurs when developing distributed systems is the scattered output of information and debug messages on various screens or different remote consoles. For this reason, the software framework includes a Generic Distributed Output System (GDOS) that collects all messages and displays them in a window of the graphical user interface. To improve the clearness, all messages contain information about the module that printed the message and the message type. Based on this information, it is possible to filter messages issued by specified modules or to adjust the level of detail.

## 3 Uniform Communication

For transmitting packages from one module to another, the software framework provides the two functions `packageSend()` and `packageReceive()` which encapsulate the transport mechanism. This allows uniform communication which means that the user

can send the package to an abstract mailbox name without the need to know where the addressed module is running and which communication media is available. The necessary information is provided by the framework. The inner structure of this uniform communication mechanism will be described in this section.

### 3.1 Package Format

Packages consist of a package head with a fixed size of 16 bytes and a package body with variable length (see figure 2). The head contains the name of the destination mailbox (*to*), the reply address (*from*), and the package body length (*dataLen*). Furthermore, the package *type* is encoded. This type can either be a positive command number or a negative reply- or data-type that is unique in combination with the sender- and receiver-module. The *id* field is used to distinguish between two or more packages of the same type. Applied to the RPC algorithm, this *id* is used to find the reply package that corresponds to the send command. Finally, the *flags* are used to specify package priority and to encode the package byte order (little-endian, big-endian).

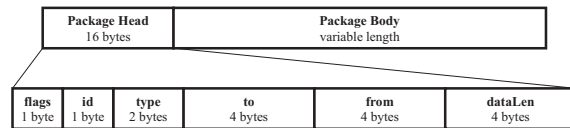


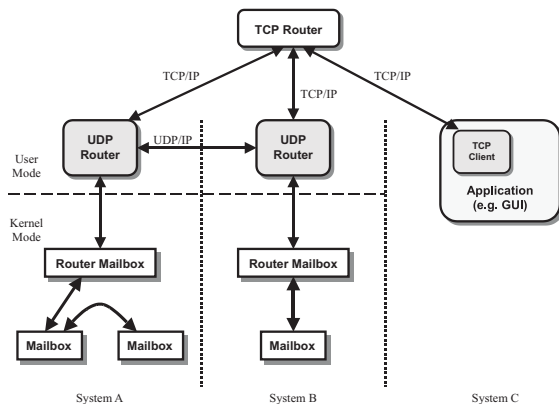
FIGURE 2: *Package Format*

### 3.2 Package Transmission

The communication mechanism is divided into three layers. The lowest layer is the local package transfer via a mailboxes mechanism. Based on prioritised buffers, the mechanism allows hard real-time package transfer for modules that are running on the same computer. All other communication layers use these mailboxes indirectly. Mailboxes are initialised by the package receiver in order to allocate the required buffer memory. The size of the allocated memory block is the maximum package size times the maximum number of storable packages. The fixed slot size is required to avoid memory fragmentation due to the prioritised read access. Furthermore, the mailbox is registered to a list of all locally available mailboxes. If a module sends a package to a local mailbox, the package is copied into a free slot of the mailbox. The transmission call is non-blocking and will return with an error if the mailbox buffer is full. Packages that are addressed to a mailbox which is not in the local mailbox list will be transferred automatically to the next higher layer of the communication mechanism.

To receive packages, the framework provides receive functions which are either blocking with an optional timeout or non-blocking. The receive functions return the package with the highest priority first.

The middle layer of the communication mechanism is used for data exchange between computers that are connected to the same local area network (LAN). A so-called UDP Router which is implemented as a user-space real-time application (LXRT) receives locally undeliverable packages over a central router mailbox (see figure 3). If the destination is reachable over the LAN, the router forwards the packages using UDP/IP. Packages are transmitted in soft real-time as long as the traffic on this network is limited. The UDP Routers of a LAN form a peer-to-peer network which is defined statically by configuration file. Within our robotics application this middle layer is used to connect embedded controllers that are located on one the same robot.



**FIGURE 3:** *Package Routing*

For package exchange with modules that are not in the LAN, a TCP/IP connection via a central TCP Router is available. This communication is not real-time capable but it allows easy access to a broad variety of communication media like wireless networks or the Internet. Furthermore, it is used to connect other non real-time components like Java modules. The TCP Router which plays a server role is connected to the TCP Clients running on each computer in the system (see figure 3). All packages that can not be delivered on the low or middle layer of the communication mechanism will automatically be send to this top layer. The TCP Router transfers the packages based on a dynamic routing table to the appropriate client. In our robotics application, the TCP Router is used for attaching the graphical user interface via wireless networks or Internet. A further application scenario is inter-robot communication.

## 4 Real-Time Communication over Ethernet

As discussed in section 3, the first framework version already supports package exchange over UDP/IP but, because standard non real-time network services were used, without guaranteeing bounded transmission delays. Only a global time stamp was distributed over CAN in order to synchronise the clocks of all processing nodes. But as more and more applications, for example high-speed acquisition of 3-dimensional laser scan images, demand both determinism and high bandwidth, a new transport medium is required. Ethernet is the preferred choice, because its hardware components are cheap and common, it provides typical bandwidths of 10 to 100 MBit/s, and it is already integrated into our robot platforms.

### 4.1 RTnet

RTnet is an Open Source project which provides temporally deterministic communication over standard Ethernet. Instead of modifying the hardware of the network interface controller (NIC) or using expensive real-time enabled industrial switches, RTnet achieves bounded transmission delays only by means of software protocols. RTnet was originally started by David Schlee as a network stack for real-time Linux [8]. The project was restarted by the current maintainers at the Real-Time Systems Group with the aim to port it to current Linux versions and to add the missing support for deterministic media access control. RTnet is runs on top of RTAI.

Currently available commercial approaches for a deterministic Ethernet extension either require modified NICs or switches [9][10][11] or they only provide soft real-time support [12]. Furthermore, many of their specifications are not fully published. The attempt to define a standard industrial real-time Ethernet [13] did not succeed. Academia real-time Ethernet protocols like [14] or [15] are still lacking or have just started the implementation on standard RTOSes.

#### 4.1.1 Network Structure

In Figure 4, a typical setup of a RTnet network is illustrated. As RTnet uses a software-based media access control, only nodes are allowed within a physical network segment which conform to the respective protocol. To enable stations which are only connected via RTnet to access the Internet or Intranet, non real-time traffic is tunnelled through the real-time domain. Thus, remote administration, moni-

toring, and maintenance are still possible even when using just a single Ethernet link to attach a station.

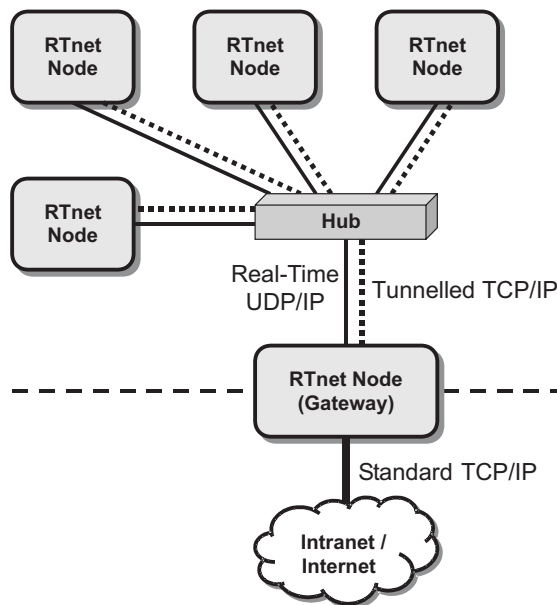


FIGURE 4: Typical RTnet Network

#### 4.1.2 Internal Design

Figure 5 shows the components of the RTnet protocol stack and their integration into the Linux/RTAI system. The Ethernet NICs are managed by real-time enabled adapter drivers. As RTnet widely preserves the network driver model of Linux, all of the currently available drivers have been ported from the original Linux version with comparably small effort. RTnet already supports most of the popular NICs like Intel EtherExpress PRO 100, Real-Tek 8139, DEC 21x4x, and several others. Suitable for real-time networking are all chip-sets which do not require intolerable long hardware synchronisation phases within the critical paths of the interrupt handler and the transmission routine.

The media access, i.e. the permission to start a NIC driver's packet transmission, is controlled by a protocol layer called RTmac. As an optional module, it can be unplugged if a deterministic network is available, e.g. two cross-connected stations, and only real-time data is exchanged. In the current implementation, RTmac includes a basic TDMA access protocol. Every station has a fixed time slot within the elementary TDMA cycle. One station takes the role of the master and periodically sends a synchronisation packet to signal the beginning of a new cycle. As a byproduct, the TDMA master also distributes a global time stamp. For future versions of RTnet, it is planned to include also alternative access protocols, e.g. token-based methods.

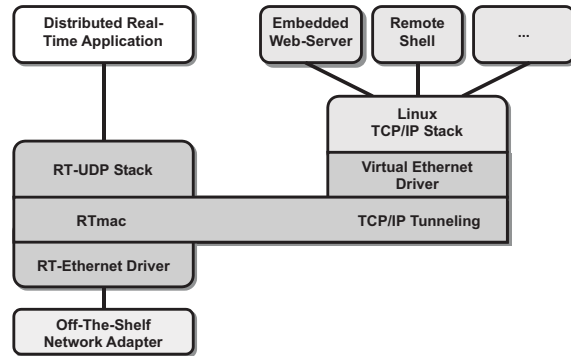


FIGURE 5: RTnet Protocol Stack

RTmac provides a prioritised queue for outgoing packets in order to synchronise them according to the media access control protocol. 32 different levels are available, the lowest one is reserved for non real-time packets. Such packets are generated by the virtual Ethernet device driver (VNIC), which is also a component of RTmac. The VNIC enables the real-time-safe tunnelling of non real-time Ethernet protocols like TCP/IP, IPX, AppleTalk, etc. through a RTnet network.

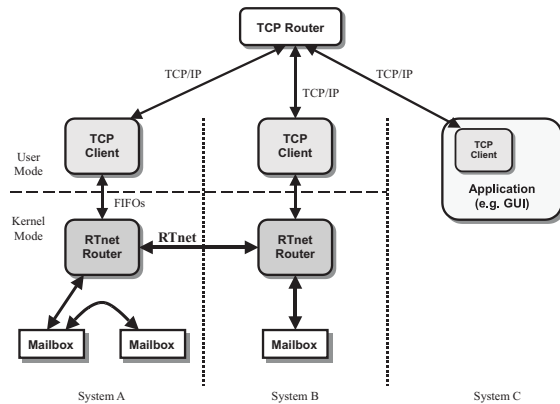
RTnet implements a modular UDP/IP protocol stack including basic ICMP. Fragmentation of IP packets is supported, but with the limitation that only fragments arriving in ascending order are accepted. The address resolution protocol (ARP) has been modified to suppress its automatic operation. Instead, during the configuration phase, all resolutions have to be explicitly triggered. To ease the development of real-time applications, the RTnet API conforms to the BSD socket interface. Besides UDP sockets, the API also supports packet sockets which allow to implement further Ethernet protocols as real-time applications.

A critical part of the RTnet stack with respect to real-time requirements is the buffer management. Due to the demand of strictly bounded execution time, a dynamic allocation of input and output buffers is forbidden. Instead, RTnet provides a multiple-pool-based allocation mechanism for its fixed-sized buffers. If using just a single global pool, the communication system can quickly lock up when an application or a system component fails to fetch all incoming data on time or sends out too many packets at once. This is especially problematic when allowing non real-time traffic tunnelling. Therefore, RTnet implements one pool for each critical component, like NIC receive pools, user socket pools, or VNIC pools. On the one hand, the initial creator of outgoing packets has to take the required buffer from its own pool. On the other hand, when a filled

packet is handed over to a consumer (e.g. a socket), an empty buffer has to be taken from the consumers pool and transferred to the pool of the producer (NIC driver).

## 4.2 Porting the Package Service to RTnet

The usage of RTnet by the package communication mechanism allows both deterministic and also more efficient data exchange between different embedded PCs. As shown in figure 6, the old LXRT-based UDP Router is now divided into a kernel-mode RTnet router and a so-called TCP Client which runs as a normal Linux application.



**FIGURE 6:** *Advanced Communication Architecture*

The RTnet router integrates the mailbox management which was a separated module beforehand and the interface to the RTnet stack. As the TDMA media access control protocol of RTnet already comes with global time stamp support, the former CAN-based service which is described in section 2.2.3 becomes also part of the RTnet router.

Again, every package which is not locally addressed is sent over UDP/IP the destination real-time node, but now over the deterministic transport service of RTnet. By dropping the router mailbox and avoiding the LXRT interface, several copy steps can be saved. Packages are either copied into the local mailbox or directly transferred to the network stack. The heads of packages arriving over RTnet are retrieved by the read-ahead mechanism of the BSD socket API (MSG\_PEEK flag), evaluated, and then the whole packages are copied within a single step into the destination mailbox. To map the 8-bit-wide package priority values to the 31 levels available with RTnet, the highest 4 bits of the selected class id (see section 2.2.2) are used to set the priority of outgoing packages.

Packages with unknown routes are sent over a FIFO to the TCP client. This component simply forwards

every package between the central TCP Router and the local RTnet router over a non real-time TCP link. Furthermore, upon startup, the TCP Client reads the configuration file of the node, interprets it, and sends the real-time routing information over the FIFO to the RTnet router module. Afterwards it registers itself with the central TCP Router.

## 5 Current System

The first version of the software framework presented here has been successfully used for more than a year now to control three different mobile robots. The sizes of the robots range from 50 cm length and 25 kg weight to 180 cm length and 350 kg weight (see figure 7 for the latter system). The robots are equipped with a number of different sensors including GPS receivers, gyroscopes, laser and sonar range sensors, and 3D laser range scanners. For sensor data fusion and for control, the robots are equipped with one or two embedded PC's with 166 MHz to 700 MHz Pentium processors. Every robot is connected to a Windows laptop via WLAN which is used to run a graphical user interface and debugging tools.



**FIGURE 7:** *Outdoor Robot DORA*

### 5.1 Real-Time Execution Platform

Until recently, the RTAI extension deployed for the robots was installed on the embedded PCs either on top of a normal SuSE distribution or a minimised Linux system. The SuSE systems were stored on notebook hard disks and were also hosting most of the software development tools. On the other hand, the mini-system provided only a runtime environment which fitted on a 8 MB compact flash disk.

To standardise the used embedded platforms with all required peripheral interfaces, the concept of a so-called Scalable Processing Box (SPB [16]) has been

designed and implemented by an associated research group at the Learning Lab Lower Saxony, University of Hannover. Several boxes containing either Pentium I PC/104 or Pentium III EBX boards (see figure 8) have been assembled and are now being successively introduced on the robots. The software distribution which is part of the SPB concept contains both a small Linux/RTAI runtime system which is executed on the boxes and a basic development environment installable on typical Linux distributions. The future versions of the runtime system will also come with an elementary robotic library derived from the software framework presented in this paper.



FIGURE 8: A Scalable Processing Box

## 5.2 Practical Software Structuring

To write efficient software for this broad variety of devices, hardware abstraction becomes an important topic. Therefore, the driver modules are divided into abstract sensor and actuator classes. These classes must include a control interface and well-specified data input and output interfaces. In practice this means that one path planning algorithm can be used to control three different mobile platforms or that one localisation algorithm can work with the input data of four different laser range sensors types.

Except for the drivers, all other parts of the software are hardware and thus robot independent. These modules are grouped according to their functions. The middleware group contains all generic modules of the presented framework. Modules for localisation and mapping algorithms are part of the navigation group. The perception group covers all modules for sensor data processing. The user interface modules form another group. Similar to the hardware abstraction for driver modules, abstract module interfaces have been defined for these groups. This allows, for example, to change selectively only the filter module of a complex 3D perception algorithm.

## 5.3 Graphical User Interface

Another important part of the software is the graphical user interface (GUI) which is implemented in Java and can thus be executed on Windows or Linux desktop computers (see figure 9). The independence of programming languages and operating systems provided by the framework allows to use powerful tools and platform for developing and running the GUI. With the GUI being an active module within the framework, the system hosting it is increasing the computational power of the whole robot system. This also means that the GUI causes no effects on the embedded PCs.

The GUI itself is designed as modular as the whole framework. It consists of two parts, a module monitor on the left side and a module workspace. The module monitor gives a quick overview over the loaded modules and continuously monitors their status (enabled, disabled, error). More detailed information about each module is displayed in a window within the workspace. The GUI components are implemented separately for each module class and can thus be adapted individually to the module characteristics.

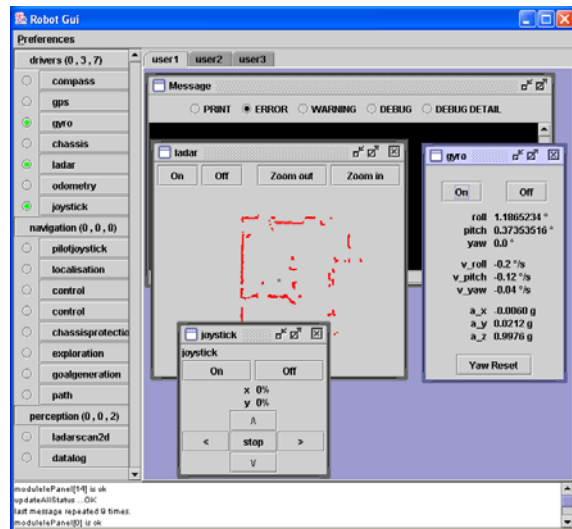


FIGURE 9: Java-based User Interface

## 6 Summary and Outlook

This paper described a compact software framework which is used to control autonomous service robots. It is based on distributable and system independent components. The framework provides a uniform communication mechanism. Besides the currently used implementation of this mechanism, the integration of hard real-time Ethernet using RTnet was described. The real-time part of the framework and the control software is running on top of Linux/RTAI.

The framework is connecting the robot control with a Java-based graphical user interface which is mainly running on Windows systems.

Future work will concentrate on the integration of the framework into the Scalable Processing Box. One goal is the publication of an Open Source robotic library based on the existing software. Furthermore, a configuration distribution system for RTnet and thus the framework is going to be implemented which will be usable by interconnected SPBs.

## References

- [1] OROCOS, 2003, *Open Robot Control Software*, [www.orocos.org](http://www.orocos.org)
- [2] Christian Schlegel, Robert Wrz, 1999, *The Software Framework SmartSoft for Implementing Sensorimotor Systems*, IN PROCEEDINGS OF THE IEEE/RSJ CONFERENCE ON INTELLIGENT ROBOTS AND SYSTEMS, pp 1610–1616, Kyongju, Korea.
- [3] David Austin, 2003, *Dave's Robotic Operating System*, [www.dros.org](http://www.dros.org)
- [4] Object Management Group, 2000, *The Common Object Request Broker: Architecture and specification (CORBA 2.4.1)*, [www.omg.org](http://www.omg.org)
- [5] Douglas C. Schmidt, Fred Kuhns, 2000, *An Overview of the Real-time CORBA Specification*, IEEE COMPUTER SPECIAL ISSUE ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING.
- [6] Ricardo Sanz, et al., 2003, *An Experiment in Distributed Objects for Real-time Process Control*, IN PROCEEDINGS OF THE IEEE CONFERENCE ON ENGINEERING TECHNOLOGIES AND FACTORY AUTOMATION, pp 664–668, Lisbon, Portugal.
- [7] P. Mantegazza, E. Bianchi, et al., 2000, *RTAI: Real-Time Application Interface*, Linux Journal.
- [8] LinuxDevices, 2000, *Lineo announces GPL real-time networking for Linux: RTnet*, [www.linuxdevices.com](http://www.linuxdevices.com)
- [9] BERNECKER + RAINER Industrie-Elektronik Ges.m.b.H., 2002, *ETHERNET POWERLINK, White Paper, Version 0005*, [www.ethernet-powerlink.org](http://www.ethernet-powerlink.org)
- [10] Beckhoff Industrie Elektronik, 2002, *Ethernet communication in real-time*, PC-Control issue 2 (corporate magazine), [www.pc-control.net](http://www.pc-control.net)
- [11] Joachim Feld, Ralph Bsgen, 2003, *Echtzeit am Ethernet mit PROFINet V2.0*, atp Heft 1.
- [12] Real-Time Innovations, Inc., 2001, *RTPS Wire Protocol Specification Version 1.0*, [www.ida-group.org](http://www.ida-group.org)
- [13] IAONA e.V., 2003, *Industrial Automation Open Networking Alliance*, [www.iaona-eu.com](http://www.iaona-eu.com)
- [14] Paulo Pedreiras, Lus Almeida, Paulo Gai, 2002, *The FTT-Ethernet Protocol: Merging Flexibility, Timeliness and Efficiency*, EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS.
- [15] Jos Mara Martnez, Michael Gonzlez Harbour, and J. Javier Gutierrez, 2002, *A Multipoint Communication Protocol based on Ethernet for Analyzable Distributed Real-Time Applications*, 1ST INTERNATIONAL WORKSHOP ON REAL-TIME LANs IN THE INTERNET AGE.
- [16] P. Hohmann, U. Gerecke, B. Wagner, 2003, *A Scalable Processing Box for Systems Engineering Teaching with Robotics*, INTERNATIONAL CONFERENCE ON SYSTEMS ENGINEERING, Coventry, UK.