IST-2001-37652

Hard Real-time CORBA

Title

D3.5
Robot Control Testbed
Hard real-time implementation

Authors

Klas Nilsson (klas@cs.lth.se)
Anders Blomdell (andersb@control.lth.se)

Reference IST37652/076 Deliverable 3.5
Date 2003-10-23
Release 1.0
Status Final
Clearance Consortium

Partners

Universidad Politécnica de Madrid
Lunds Tekniska Högskola
Technische Universität Wien
SCILabs Ingenieros

Summary Sheet

IST Project 2001-37652
HRTC
Hard Real-time CORBA

D3.5 Robot Control Testbed
Hard real-time implementation

Abstract:

The robot control testbed for evaluation of hard real-time CORBA has been implemented based on hard real-time communication via the ThottleNet real-time Ethernet protocol. An ABB Irb-2000 robot that has been reconfigured to permit control experiments is used. The control computers include an external VME-based computer system with Motorola PPC processors. These are connected via Ethernet to both dedicated ETRAX processors for hard real-time sensing and actuation, and also to host computers handling external soft real-time sensors. The platform permits testing of different timing and real-time communication principles.

HRTC Partners:

Universidad Politécnica de Madrid
Lunds Tekniska Högskola
Technische Universität Wien
SCILabs Ingenieros.

Release Sheet (1)

Release:	**0.1 Draft**
Date:	2003-09-27
Scope	Initial version
Sheets	All

Release:	**0.2 Draft**
Date:	2003-10-12
Scope	Revised version
Sheets	All

Release:	**1.0 Final**
Date:	2003-10-23
Scope	Final version
Sheets	All

Table of Contents

1 Introduction 5
2 Using the ThrottleNet real-time Ethernet protocol 5
3 The Linux Real-Time Application Interface (RTAI) 8
4 User-space ORB and kernel-space applications 9
5 The hybrid HRTC communication approach 11
6 References 13

1 Introduction

The testbed implementation for non real-time testing included development of a virtual testbed, a PCI-based controller for an ABB Irb-2400 robot, and the VME-based system for the Irb-2000 robot that was selected for the full implementation for hard real-time purposes. In that system, it is the servo control of the robot joints that comprises the innermost control loop including distributed objects and real-time communication. That is, external sensor based control (based on stereo vision in this case) is based on soft real-time communication/computing is not the primary concern here. The focus is on the hard real-time communication in the context of the testbed application. Thus it is the nodes Sensor, Controller, Actuator and GlobeThrottle that are covered here, see Figure 1.

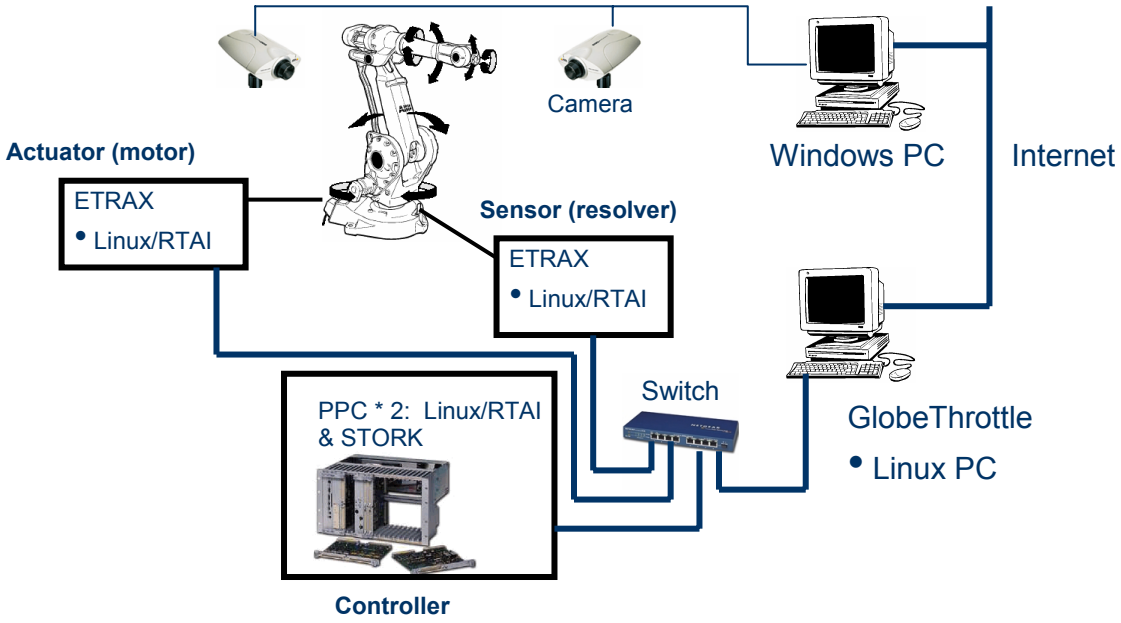

Figure 1 The testbed including ThrottleNet communication.

2 Using the ThrottleNet real-time Ethernet protocol

Basic idea of ThrottleNet (TN) is to accomplish hard-RT communication over a local switched Ethernet area (RTLAN, see Figure 2) by restricting the amount of network traffic each sender may generate. It is possible to guarantee an upper hard bound on the network latency since a switch provides point-to-point communication between any two single nodes in the RTLAN, and if the traffic to any single node always stays below the buffering available in the switch, no network packages are lost [Mart2002]. The bandwidth limitation can in technical terms be called throttling, hence the name ThrottleNet.

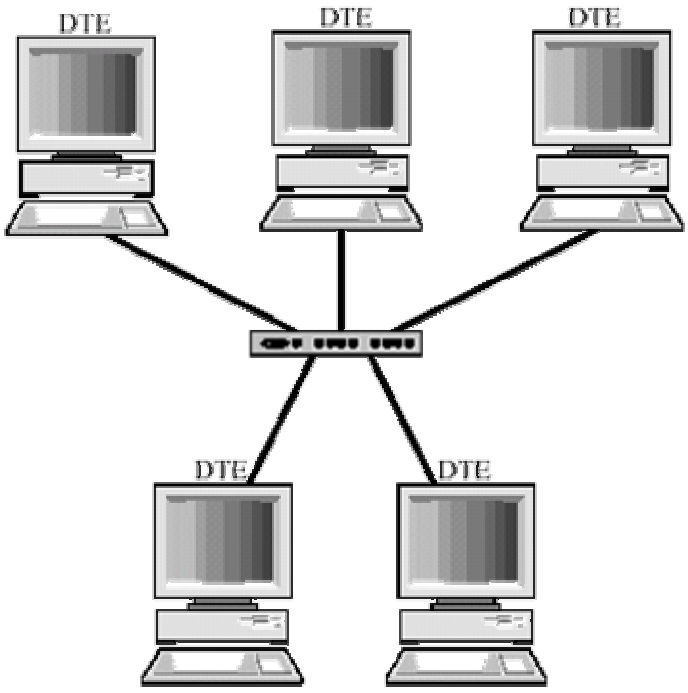

Figure 2 Computers connected via switched Fast-Ethernet (100Mbit/s), forming a local real-time network.

Assuming no clock synchronisation and no time-triggered schedule for the network traffic, the hard-RT guarantees as well as the resulting amount of jitter come from a buffering/scheduling analysis. The works case for the traffic to a target node (T) is when all other nodes (that are ever communicating with T) want to send at exactly the same time. If the switch buffer is sufficient to hold all these Ethernet packages, it will always be able to do so, and we get a hard upper bound on the maximum communication delay. For this hard-RT property to hold, it is also required that

- The communication is full duplex, which switches today typically are.
- The delay caused by the buffering the main source of non-determinism, which is the case according to our measurement on switches.
- The traffic is periodic, or non-periodic with a lower bound on the time between any two attempts to communicate.
- Broad-cast and multi-cast traffic is not permitted, that is, all traffic is point-to-point.
- Keep-alive packets are sent (say, once per minute) to each node so the switch does not forget the MAC address.

An example of a useful (low-cost) switch is the DES 1016D from D-Link. The HRT RCT is implemented such that the above assumptions hold.

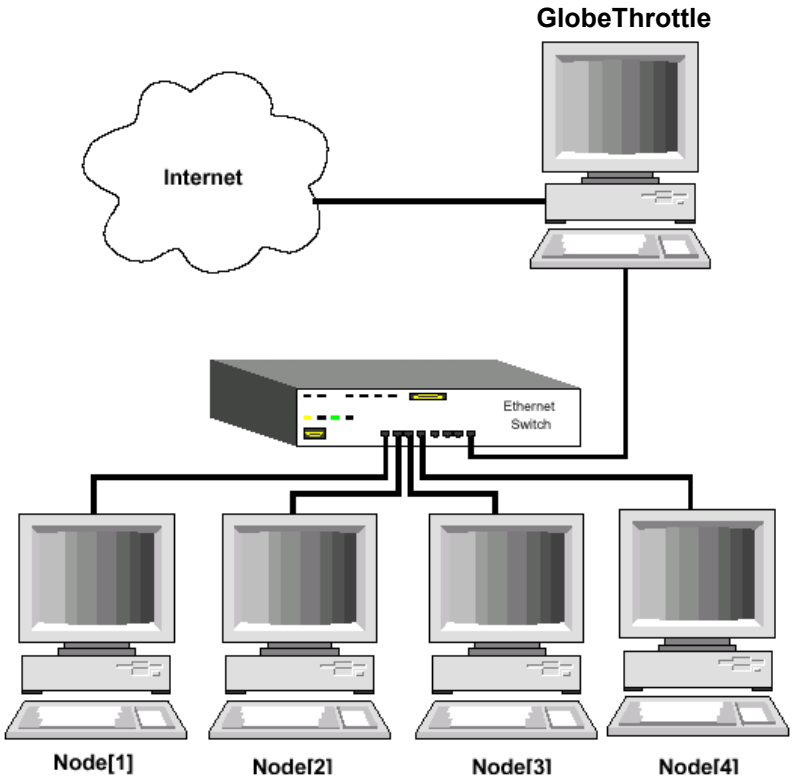

Figure 3 TrottleNet connected via the GlobeThrottle to Internet.

To reduce the jitter, clock synchronisation and traffic scheduling could be done similar to the scheduling of TTP, but that is future work. For the RCT and the servo control it contains, the communication bandwidth is more important, and the reason (apart from cost) that we do not use TTP/C in the RCT. From a CORBA point of view, assuming traffic scheduling and pluggable transports, the application level can be made independent of the protocol.

The key argument for using TTP/C is dependability; in the RCT the robot simply stops if one connection is lost, but for a safety critical vehicle application (e.g. the braking or steering) that would not be acceptable. But for an application that is not safety critical, the hardware we use can be purchased in any computer shop near you.

There is a special GlobeThrottle node, called the GlobeThrottle as shown in Figure 3 and Figure 1, which takes care of the connection to ordinary networks (LAN or the Internet). Broad-cast and multi-cast traffic from outside the TN to nodes inside the TN are converted by the GlobeThrottle to uni-cast traffic to each of the involved TN nodes.

By reserving Ethernet bandwidth for non-RT traffic, and by using special device drivers, the ordinary TCP/IP non-RT traffic will still work within the TN.

3 The Linux Real-Time Application Interface (RTAI)

The RTAI provides a valuable combination of predictable execution of real-time applications (in kernel space) and a full-featured operating system (in user space). The predictable execution is what we need for hard real-time applications, while a rather complete OS (Linux in this case) is what we need for CORBA support. The kernel-space functionality is quite limited or different compared to the Linux in user space, and therefore we need to do implementations on both levels as described in the following sections.

On the kernel level, the hard real-time software for communication and control needs the claimed predictability of RTAI. That included the device drivers for Ethernet and Throttlenet. However, during implementation and test of the TN device drivers, the experienced problems with getting the software to work was traced down to the (according to our opinion) strange handling of interrupts. In the interrupt service routine (ISR) taking care of the Ethernet network interrupt, a new available packet is signalled via a semaphore to the consuming thread. Clearly, the interrupt routine should complete its execution before permitting a context switch to the signalled thread. However, according to the standard RTAI implementation, the signalled thread starts to run (until blocking) in the context of the ISR. This in turn resulted in problems with predictability in general and with getting the TN devices to work in particular.

There are several enhancements of RTAI that should preferably be done, but to give an example how it can look, the following is the first and foremost problem with the TN implementation was reported to the RTAI community as:

```
I'm having problem with interrupt 18 (ethernet) on a VME2600 (ppc)
system. If the interrupt is only used from RTAI everything works all
right, likewise if it is only used from Linux. But after loading a
driver in Linux space, the interrupt is unusable (blocked) in RTAI.

Here is why:

  After boot irq_desc[18].status (arch/ppc/kernel/irq.c) starts with
  the value 0x00000040 (IRQ_DISABLED = FALSE). Now it is OK to use
  the interrupt from RTAI if I acknowledge the interrupt with
  rt_unmask_irq(18), a call which eventually ends up in openpic_end_irq
  (arch/ppc/kernel/open_pic.c).

  BUT: when I load a Linux driver, it calls setup_irq(18), setting
  irq_desc[18].status to 0x00000040 (which is OK) and on unloading it
  calls free_irq(18), setting irq_desc[18].status to 0x00000042
  (IRQ_DISABLED = TRUE). This is OK for Linux drivers, since next
  call to setup_irq(18) reenables the interrupt.

  If I now try to load a RTAI driver, (arch/ppc/rtai.c),
  irq_desc[18].status is not changed (still 0x00000042; IRQ_DISABLED =
  TRUE), and subsequent calls to rt_unmask_irq(18) are not honored,
  since openpic_end_irq sees that the irq_desc[18].status is
  IRQ_DISABLED, and hence it does not enable the interrupt.
```

```
  To me it seems that the solution is to let:
    rt_request_global_irq  call   request_irq
  and
    rt_free_global_irq     call   free_irq
```

To avoid this problem, the following patch was submitted:

```
--- arch/ppc/rtai.c.orig        Mon Sep 15 03:37:35 2003
+++ arch/ppc/rtai.c     Mon Sep 15 03:29:36 2003
@@ -768,6 +768,11 @@
 static unsigned long irq_action_flags[NR_IRQS];
 static int chained_to_linux[NR_IRQS];

+void dummy_handler(int irq, void *dev_id, struct pt_regs *regs)
+{
+  printk("dummy_handler should never be called\n");
+}
+
 int rt_request_global_irq(unsigned int irq, void (*handler)(unsigned int irq))
 {
        unsigned long flags;
@@ -804,6 +809,7 @@
         */
        IRQ_DESC[irq].handler = &real_time_irq_type;
 #endif
+       request_irq(irq, dummy_handler, SA_SHIRQ, "RTAI fix", &dummy_handler);
        hard_unlock_all(flags);

        return 0;
@@ -848,6 +854,7 @@
        }

        flags = hard_lock_all();
+       free_irq(irq, &dummy_handler);
        IRQ_DESC[irq].handler = &trapped_linux_irq_type;
        if (global_irq[rirq].mapped == RTAI_IRQ_MAPPED_TEMP)
                unmap_ppc_irq(irq);
```

This code fragment also illustrates a typical situation in platform development; The solution consists of just a few lines of code, but those lines can take weeks to find out. The RCT contribution to Linux-based real-time are made available as free software on the Internet, which pays back in terms of improved Linux support for our hardware.

4 User-space ORB and kernel-space applications

To achieve hard real-time for the entire application, the ORB (or vital parts of it) must run in an RTOS (RTAI in our case). If that is not the case, no matter how well the code is tuned and timed, the execution of the hard real-time threads can be delayed by the scheduling decitions done by the non-RT OS. Clearly, this is independent of the properties of any hard real-time communication (such as TTP or TN); the execution platform also has to provide real-time guarantees. Still, there will be activities and communication links that do not require real time communication, then typically using TCP/IP.

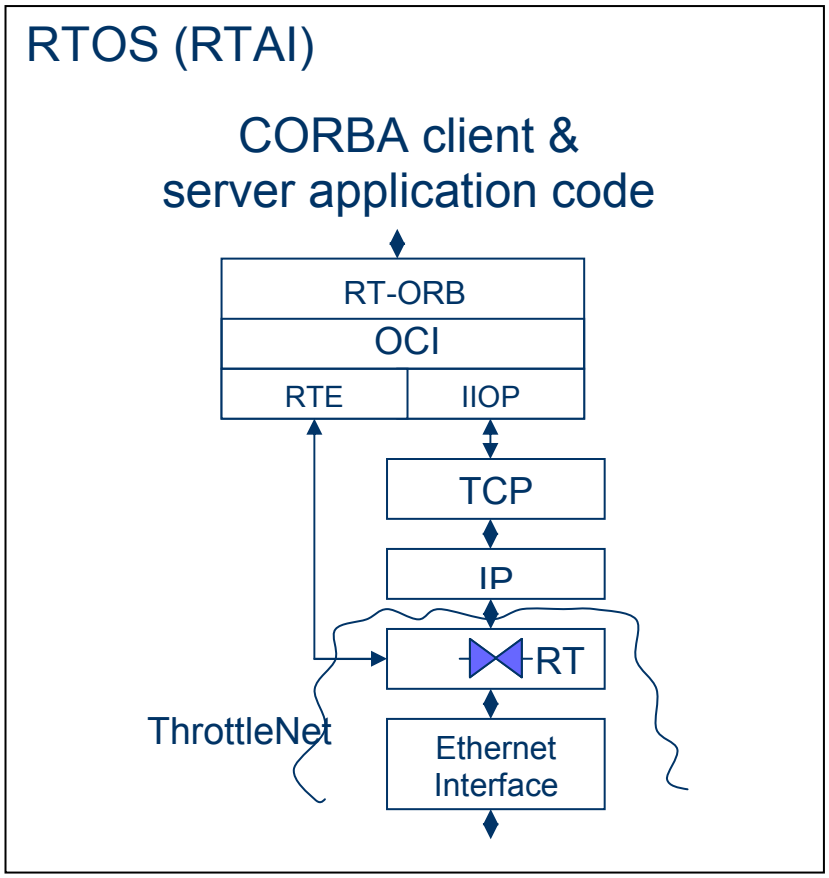

Figure 4 Communication for non- and hard-RT communication.

With a combined need for hard RT and non-RT, we can make use of the OCI specification, and the support for it in the ICa ORB. This means, assuming TN for hard RT communication, that we have a structure as depicted in Figure 4.

However, to accomplish CORBA support on top of an RTOS such as RTAI is beyond the scope of this project. In fact, it can even be questioned if this is reasonable at all; the required engineering efforts to port and to maintain such an ORB may be too extensive, at least compared to other possible approaches that we are to explore here. Thus, here we have to approach HRT CORBA without porting the ORB to RTAI. However, in the testbed the ETRAX and PPC application code need to run in RTAI, to obtain the desired timing properties, so we need some kind of hybrid approach.

5 The hybrid HRTC communication approach

The main idea of the HRTC hybrid RT approach is to let hard RT threads acquire hard RT communication in a CORBA-compatible manner by obtaining a handle to the hard RT transport via a call of the user-level ORB.

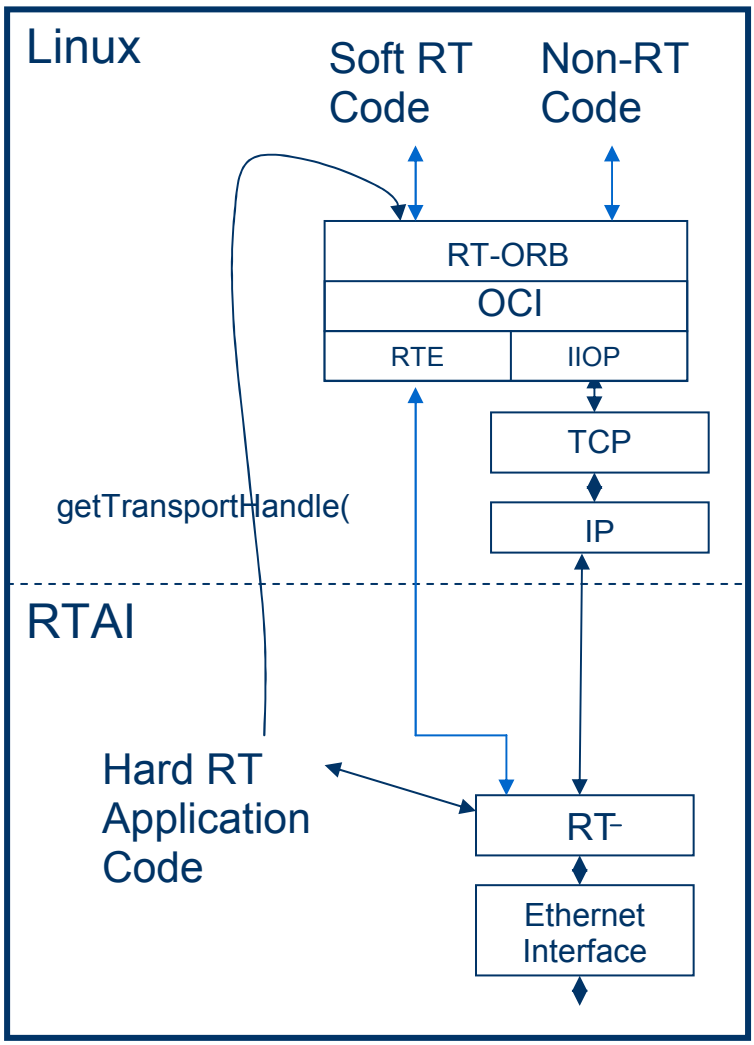

Figure 5 Hybrid kernel-space user-space solution.

The characteristics of this HRTC solution is that a full-featured OS (Linux) is combined with hard real-time support for threads needing that, see Figure 5. Note that even if the handle/descriptor for the HRT transport is used locally in kernel-space without involving the ORB for each transfer, the ORB is still aware of the transport and its usage. That is, first the ORB is responsible for establishment f the connection and then the ORB can communicate with the GlobeThrottle (over a non-RT transport) to handle on-line resource utilisation. Hence, CORBA-level resource awareness is maintained.

Also note the similarities with TTP/C and the engineering principles used for time-triggered distributed components, compared to RT-CORBA: There is no need for priorities or the mapping, propagation, lanes, etc. Instead, distributed components are designed to be event driven and accepting some timing jitter (as in the RCT), or components are designed to be time driven and communicating over so called temporal firewalls (corresponding to TTP or a scheduled TN, which is outside the scope of this work). In both cases, the scalability of CORBA is improved since global properties are managed globally (in the GlobeThrottle or in the TTP configuration) and non-scalable features such as priorities are kept locally.

As mentioned, sending and receiving data on the hard RT level is done directly via the handle obtained from the ORB. Note that since data transfer is done directly (connection-based unicast) between the two communicating nodes that are configured from the ORB-level, there is no need to send complete GIOP messages over the RT channels. Instead, full object references are only needed when obtaining the connection, which is then working with small packets (configurable but as a default in the current RCT implementation it is set to the minimum Ethernet packet size of 64 bytes) that are transmitted within one Ethernet frame. Omitting type definitions and auxiliary functions, the ThrottleNet header file for the RT-level contains the following:

```
throttlenet_receive_socket
    throttlenet_open_receive_socket(char *name, int size, int period);

void
    throttlenet_close_receive_socket(throttlenet_receive_socket socket);

throttlenet_transmit_socket
    throttlenet_open_transmit_socket(char *name, int size, int period);

void
    throttlenet_close_transmit_socket(throttlenet_transmit_socket socket);

int throttlenet_receive(throttlenet_receive_socket socket,
                        void *buf, int buf_length);

int throttlenet_transmit(throttlenet_transmit_socket socket,
                         void *buf, int buf_length);

int throttlenet_tunnel_transmit(void *buf, int buf_length);
```

These functions reflect that each distributed hard-RT object refers location-transparently (within the TN area) to other objects by name when opening the connection, and for ensuring the predictable transport the message size and period are also provided. The *tunnel* refers to a connection from the hard-RT level to the GlobeThrottle node. That tunnel is used by a separate throttlenet_tunnel kernel module that emulates the Ethernet connection for the TCP/IP protocol for the non-RT application of the node. The non-RT traffic

coming via the tunnel is split to fragments fitting into TN packets and put into the available TN communication slots, and then sent to the GlobThrottle node where a tunnel_handler (omitted in the header file above) collects TN non-RT packets and restores the TCP/IP content. The networking on the GlobThrottle node then remains unchanged on the Linux user-space level, and standard routing as setup in Linux on the GlobThrottle is used also for routing the traffic to other nodes in the TN area. Hence, non-RT traffic within the TN area always goes via the GlobThrottle node, which may look like a detour but in this way all the routing/handling of non-RT traffic comes for free (no extra code or programming, which also simplifies dissemination of the technique), and no additional scheduling of many-to-many connection is needed.

Several extensions for scheduling of time-driven communication and application support are possible, but outside the scope of the current project. Even if full implementation and packaging of TN was carried out within the HRTC efforts and driven by the HRTC needs, experiences from the HRT RCT implementation makes us believe that ThrottleNet will be used both independently of CORBA and together with CORBA via the OCI. The current implementation of TNIOP should, however, be enhanced to better handle the distribution of GIOP messages on TN packets (Ethernet frames); the current implementation uses OCI only on the byte-stream level.

On the hard RT level, on the other hand, the robot joints can be controlled with a sampling/communication frequency of up to 10 kHz. Typically we use 8 kHz which is the frequency of the force sensing. Since sensor and actuator values for all joints of the robot fit into one TN/Ethernet packet, most of the bandwidth is left for non-RT traffic, and due to the throttling the timing of the RT transport is superior even without exact scheduling the traffic.

6 References

[CORB] Common Object Request Broker Architecture (CORBA/IIOP) Specification 3.01, Object Management Group, Needham, MA, U.S.A., 2002, http://www.omg.org

[RTCORB] OMG: Real-Time CORBA 1.0 Specification, http://www.omg.org

[RTAI] Real-Time Application Interface: www.rtai.org

[Mart2002] A. Martinsson: “Scheduling of realtime traffic in a switched Ethernet network”, Master thesis, Dept. of Automatic Control, 2002.