Real Time Distributed Control Systems using RTAI

Lorenzo Dozio
Dipartimento di Ingegneria Aerospaziale
Politecnico di Milano, via La Masa, 34
20158, Milano, Italy
dozio@aero.polimi.it

Paolo Mantegazza
Dipartimento di Ingegneria Aerospaziale
Politecnico di Milano, via La Masa, 34
20158, Milano, Italy
mantegazza@aero.polimi.it

Abstract

The paper outlines the design and implementation of the Real Time Application Interface (RTAI) for Linux, as used for high performance local/distributed control systems implemented on low cost off the shelf general purpose computers. Its native lean real time middleware layer is described along with its use in an advanced tool to easily manage and monitor complex networked control systems.

1. Introduction

The range of applications requiring a distributed real time operating system (DRTOS) to timely exchange data, events, and commands by using Ethernet and/or standard buses, has grown very quickly in the recent years. The distribution of computational resources is appealing for the design, development and implementation of effective, high cost/performance, high demand real time simulations of dynamic systems and their controllers such as those found in aerospace applications, just to cite our main field of interest.

Enthusiasm about these technologies collides with two kinds of connected issues. First many open buses are limited in performances and the question "Can Ethernet be Real Time?" will likely remain unanswered even when the Petabit per second transfer rate will be reached. This leads to the need of a viable definition of what a real time system is, especially when a digital controller is in the play. The second issue involves whether the implementation of DRTOSes should be based on application specific hardware/software or on off the shelf technologies. Taking the side of Custom Off The Shelf (COTS) solutions seems to bring with it the need of a real time middleware that can relieve developers from the daunting network programming and tools, thus taking complexity out of system production, from data-type specifications through applications debugging and end user configurations.

The paper addresses such issues by describing the solutions adopted in the design and implementation of RTAI. Section 2 presents the real time "philosophy" behind RTAI, introducing a peculiar classification of real time control applications based on low cost COTS hardware. Sections 3-5 give an overview of RTAI core concepts and a sketchy general overview of its services. Section 6 is focused on the RTAI capability to generate and distribute real time applications, with an explanation of its specific Remote Procedure Call (RPC) based middleware. Finally section 7 demonstrates RTAI middleware at work in a tool, RTAI-Lab, affording a structured framework to design, build, run and monitor any suite of RTAI-based single/multitasking local/distributed simulations and control systems.

2. High performance control systems and low cost off the shelf general purpose computers

The term "real time" can have significantly different meanings, depending on the audience and application at hand. The computer science literature generally divides real-time systems in two main categories: soft and hard.

A soft real time (SRT) system is characterized by its ability to execute a task according to a desired time schedule on the average. A video display is usually taken as a typical SRT example. It is clear that, because of the human eye dynamics, the loss of an occasional frame will not cause any perceived system degradation, providing the average case performance remains acceptable.

Hard real time (HRT) systems instead embody guaranteed timing, cannot miss deadlines and must have bounded latencies, whose level depends on the particular application at hand. A typical example of an HRT system consists of a controlling system (computer) and a controlled system (plant). It is imperative that the state of the plant, as perceived by the controlling system, is consistent with the actual plant state, within an acceptable error margin (noise level). Moreover timing correctness requirements arise for the control actuations, which have to be carried out according to the sampling rate for which the discrete time control system has been designed.

There is no doubt that control systems must be HRT and, generally speaking, hard real time constraints can be met with strict determinism only by specific dedicated Central Processing Units (CPU), e.g. Digital Signal Processors (DSP) and DSP like microcontrollers, having a guaranteed interrupt latency in the range of a single/few execution cycles. General Purpose CPUs (GPCPU), i.e. any brand used in workstations, desktops, personal computers and their industrialized clones, are in principle rather unsuitable for hard real time applications. In fact Virtual Memory and its related Memory Management Unit context switches, high dependence of performances on multi level caches, possible bus arbitration from “intelligent” input/output subsystems, high depth paralleled piped execution and speculation, subject them to many non deterministic latencies and jitter, that can largely exceed even the longest instruction execution time. Modern GPCPUs are in this respect often worse than their fore parents of thirty years ago, that were a few orders of magnitude less powerful in average throughput [1].

It is thus important to get both an operational view of the hard real time definition and an appropriate perspective of why GPCPUs are usable indeed for high demanding control systems. A well programmed modern GPCPU achieves double precision floating point capabilities from a few to many hundreds millions Floating Point Operations (multiply-add) per Seconds (FLOPS), without any special programming trick, by using plain high level languages. Giga FLOPS peaks are achievable by using vector computation units (Single Instruction Multiple Data), that come for free with many recent GPCPUs. So it allows the implementation of fairly complex control schemes. Add the possibility of easily achieving even higher performances using low cost Multi Processors (MP) or networked GPCPUs to realize that such a solution can meet high requirements with unprecedented cost/performance figures.

Taking a high precision complex shape profiling machine as an example, let us now suppose that its controller runs over a single GPCPU with a Real Time Operating System (RTOS) which allows locking their whole code and data into memory. Moreover we assume that to meet high demanding profiling specifications some form of active structural control to compensate for machine frame compliance and vibrations is required, so the sampling rate should be as high as 10 KHz. If our application is run standalone on a GPCPU and its execution timing checked, we will see latencies in the range of few microseconds. Such a situation is rarely met in real applications as often our controller needs to communicate data through a fast link, to receive commands and to supervise its operation. It is not uncommon that some data logging to a local disk is required also, not to speak of more complex data visualization and monitoring. So the control action has to work cooperatively in a prioritized fully preemptable multitasking execution environment. Because of the previously hinted GPCPUs architectural features, such an execution mix can lead to non deterministic latencies, here called jitter, that could now and then pessimistically peak up to 30 microseconds (worst case). At a first sight a jitter amounting to 30% of the sampling period will impede classifying such a system as hard real time. Nonetheless if we dare trying it and look at the finishing and tolerances of our machined items we will verify that they meet our top expectations, around the clock, seven days a week. A comparison against any equivalent dedicated DSP implementation will show no difference in results. The reason is clearly in the fact that our hard real time specifications must be related to the actual system bandwidth, and only indirectly to the sampling rate. Roughly speaking it means that our 10 KHz controller will likely be controlling a 1 KHz bandwidth system for which rare worst case 3% timing uncertainties will produce measurements and command errors that are guaranteed to be filtered, down to an acceptable noise level, by the inherent system low pass attenuation. We will not enter into more technical issues related to adding fault tolerance and outliers correction by using dynamic observers, but cannot hasten from noting that the high FLOPS throughput of today GPCPUs could ease their implementation thus allowing either a further jitter noise reduction or the extension of their use to a somewhat wider bandwidth.

What just said makes it clear how hard real time should be specified and why GPCPUs can be used for highly demanding control systems. It should be remarked also that such a point of view carries over to latencies related to distributed communications and makes blurred any distinction between soft and hard real time. In fact mating the classification of real time applications to the bandwidth of a system makes the transition from soft to hard real time mainly related to the cut off frequency of interest. So there is no clear cut between hard and soft real time, but simply a continuous transition from low to high bandwidths. When we say that soft real time performances can be satisfied “on the average” we simply acknowledge that averaging is just a kind of digital low pass filtering acceptable for the bandwidth at hand. Despite such a point of view in the followings we will continue using the terms hard/soft for reason of convenience.

It is worth noting that what said above can be easily verified by using even the lowest performance personal computers one can buy at any general store today, thus vividly emphasizing COTS hardware advantages and the idea that the future of high performance computing is more and more being fostered by kids playing computer games (exaggerated but not totally untrue hype). What said can be safely extended to lower end GPCPUs, such as those we could roughly classify as old plain Pentium class, sold no more for personal computers but still used in industrial applications. In practice, once the design specifications and performances are scaled accordingly, anything previously said should be acceptable for any 32 bits GPCPUs marketed today.

At this point it must be added that the many RTOSes supporting GPCPUs often promote themselves on the base of performance figures related to task switching times, and their likes, that are totally meaningless in an environment in which their actual performance are dominated by the cited architectural features, thus making meaningless task switching times and worst instructions count based latencies, nowadays often at a sub microsecond level.

3. RTAI overview

The Real Time Application Interface (RTAI) [2] project began at the "Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano" in 1996/97 with the aim of providing a hard real time extension to Linux [3]. It stemmed from full awareness of the high cost performances advantages brought by COTS hardware and from the need of making available a low cost effective tool to support a varied set of internal research activities related to advanced active controls for generic aeroservoelastic systems and real time simulation, with and without hardware in the loop, extended to include: large space structures, structural acoustics, flexible manipulators. Its goal was to make it possible their development, implementation and testing on standard 32 bits personal computers (PC) and COTS data acquisition (DAQ) boards, by using high level language programming tools possibly evolving into automatic code generation, so that anybody, including graduating students, could proceed to their implementation with relative ease and speed in building it all.

RTAI is integrated into Linux through a text file containing a set of changes to its kernel source code, known as a patch, and a series of add on programs expanding Linux to hard real time by installing a generic Real Time Hardware Abstraction Layer (RTHAL). RTHAL performs three primary functions:

- Gathers all the pointers to the time critical kernel internal data and functions into a single structure, to allow the easy trapping of all the kernel functionalities that are important for real time applications, so that they can be dynamically substituted by RTAI when hard real time is needed; generally speaking such kernel functionalities include all functions and data structures required to manipulate the hard interrupt flag, external interrupts and internal traps/faults, the system call, programmable interrupt controller and hard timers.
- Reworks the related Linux functions, data structures and macros to make it possible to use them to initialize RTHAL pointers for normal Linux operations.
- Changes Linux to use what pointed in RTHAL for its operations.

Linux is a dynamically expandable kernel and new functionalities can be added to any running instance of it by linking a relocatable object code, called module in Linux jargon. Such a feature is used to extend Linux to hard real time by an RTAI specific module that steals from Linux all the RTHAL related objects and emulates them in software while letting Linux continue working unchanged. After that only hard real time activities can freely use the hardware and have full priority and preemption authority on Linux. It must nonetheless be noted that the hardware is unique and must be shared with Linux anyhow. It is thus possible that a Linux interrupt happens within hard real time activities and they must be registered and pended for subsequent soft dispatching and processing without loosing any of them [4].

4. RTAI meets ADEOS

The ideas exploited by RTAI/RTHAL have been known to and used by the operating system community for quite some time. They have progressed and matured into more comprehensive generalized conceptions that have made it possible structuring effective hierarchical operating systems consisting of different layer of operating systems [5]. A free open source software (FOSS) implementation of such concepts, called Adaptive Domain Environment for Operating Systems (ADEOS) [6], is now available. In its full breadth and scope the purpose of ADEOS is to provide a flexible environment for sharing hardware resources among multiple operating systems, or among multiple instances of a single operating system. The ADEOS nanokernel, i.e. the scheduler of the different operating systems instances opens a full range of new possibilities, notably in the fields of MP clustering, patchless kernel debugging, middlewares and real-time systems addition to general purpose operating systems. To this end ADEOS enables multiple prioritized domains to co-exist simultaneously on the same hardware. To share the hardware among the different operating systems, ADEOS implements a pipeline scheme which allows them to virtualize all what RTAI gathers into its RTHAL. Every domain has an entry in the pipeline and each event that comes in the pipeline is delivered to the registered domains according to their respective priority. In order to achieve hard real time determinism, RTAI over ADEOS is the highest priority domain which always processes interrupts before the Linux domain, thus serving any hard real time activity either before or fully preempting anything that is not hard real time.

RTAI has then been ported onto ADEOS to allow the RTAI development team the possibility of getting rid of kernel patching and maintenance while exploiting a more structured and flexible way to add a future multi faceted real time environment to Linux.

5. RTAI services

With the (RTHAL/ADEOS) RTAI core module installed Linux can extend its execution domain to hard real time. Nonetheless at such a point there is little that can be done but fully preempting Linux at the bare interrupt handling level. It thus arises the need of providing scheduling services to be executed in hard real time, mated to efficient communication tools to allow interacting with standard non real time Linux scheduled tasks. It is noticed that such a separation conception does not enforce any undue constraint as it is usually adopted by control system designers, even if a true native RTOS is available, because it allows a cleaner, easier to develop and more robust implementation of complex high performance control systems. Thus instead of seeing the separation of hard and soft real time applications as a constraint, we take it as a good hard real time design practice to be adhered anyhow and embed it into our Linux hard real time extension.

RTAI does provide the required scheduler along with a wealth of services. The RTAI scheduler is fully preemptible and can schedule directly from within interrupt handlers so that Linux has no chance of delaying any RTAI hard real time activity. Before giving a short overview of the RTAI scheduler and related services, it is important to anticipate the important fact that it allows to symmetrically work inter-intra user and kernel space, by using the same APIs everywhere. So any communication and interaction between the two lands is simple in RTAI, down to full hard real time interrupt handling in user space. It is possible to implement hard real time multitasking applications in kernel space also, by either using standard kernel threads or RTAI proper tasks. The possibility of embedding a control system as part of the kernel allows achieving maximum execution efficiency. In fact RTAI proper kernel tasks add further efficiency in kernel space since they can avoid using the memory management unit of GPCPUs, with a saving in task switching time that can be significant on low end GPCPUs. Thank to such a feature a control system can begin being, wholly and safely, designed in user space and then migrate to become an integral component of the kernel, in part or as a whole, when it is required to achieve the best performances on the GPCPU at hand. This is a relatively original feature of RTAI/Linux. RTAI has always seen a strict separation of kernel and user space as a drawback for hard real time applications and so it has always striven for their easy integration. This is part of the general philosophy behind RTAI: afford mechanisms and not policies; meaning that policies must be wholly in the hand of users. Thus an RTOS must provide just adequate mechanisms to make it easy implementing any policy without any undue constraint.

To allow for such a working freedom RTAI makes available a scheduler that integrates any schedulable object, be it Linux (user space processes-threads, kernel threads) or RTAI (its proper kernel tasks), either UP or MP. MP scheduling is carried out in the so called Multi Uniprocessor Scheduling (MUP). i.e by imposing that any schedulable object is assigned to a specific CPU from its very creation thus achieving better real time performances because of a more efficient caching. No restriction whatsoever is imposed to schedulable objects and they can interact without any constraint, irrespective of what CPU they are running on in the MP case. Another important feature of the RTAI scheduler is the possibility of choosing between either a base periodic timing, with a fixed assigned time resolution tick, the approach mostly used in RTOSes, and an arbitrary timing, allowing scheduling a task at the resolution of the available clock by firing a one shot timer at the time instant imposed by the highest priority task waiting on the timed list. The one shot mode avoids any compromise on the least scheduling resolution, thus giving an almost continuous time resolution, while the periodic mode requires to have any task timed at an integer multiple of the basic timer period. However we must recall what pointed out in the introduction and avoid any illusion that on GPCPUs a task can be scheduled with a nanosecond precision. It is also important to note that the MUP scheduler uses independent per CPU timers and each of them can run independently from any other, so each timer mode of operation can be freely assigned, e.g. there can be periodic and one shot timers and periodic timers need not to run at the same period. Under MP it is also possible to handle external interrupts either in a symmetric way or to force them to a specific CPU or CPU cluster.

It might be interesting to note that task switching time, measured for a semaphore wait-signal on a prioritized semaphore list containing thirty waiting tasks are: 900 ns for Linux processes/kernelthreads and 350 ns for RTAI proper kernel tasks, on a 1 Ghz PIII. Even by pessimistically taking such figures as the worst case RTAI intrinsic latency and longest interrupt disabling it should be noted that they are substantially negligible compared with worst case architectural jitter measured at the mere interrupt acknowledge level, i.e. at the execution of the very first instruction of the interrupt service routine, that can easily skyrocket to more than 10 us under heavy computational and IO load. Once more a clear evidence of the pros and cons affecting real time on GPCPUs.

RTAI schedulers make available the following scheduling policies:

- Fully preemptable First In First Out (FIFO), for voluntary co-operative scheduling. Under FIFO scheduling there is a support function to help meeting periodic tasks deadlines with statically assigned priorities according to the RMS concept.

- Round Robin (RR), like FIFO but only up to a certain allowed per task time slot, after which the CPU is tentatively handed over to any equal priority task waiting on the ready list.
- Early Deadline First (EDF), to dynamically assign priorities in order to meet periodic tasks end of execution deadlines. Effective but difficult to use as it requires that the user assigns a relatively good estimate of the execution time required by each periodic task

Let's now have a quick overview of the services made available by RTAI schedulers recalling, once more, they are symmetrically available inter-intra kernel/user space and pointing out that, even if not cited explicitly, all synchronization and communication functions are fitted with both various conditional and timed out executions mode, using either relative delays or absolute times:

- Basic task management: time management and conversions, dynamic priority assignment, scheduler policy assignment, scheduling locking/unlocking, counting suspend/resume to avoid trivial deadlocks, task yielding, busy, absolute and relative timed suspensions, specific support for periodic execution.
- Memory Management: shared memory for inter tasks, inter-intra user/kernel space data sharing, dynamic memory allocations.
- Object registration, to allow an easy reference of RTAI objects across applications by using short alphanumeric mnemonic identifiers.
- Semaphores: wait, send, broadcast on: counting, binary and resources with full priority inheritance to avoid priority inversion.
- Conditional variables: wait, signal, broadcast, equivalent to the related POSIX APIs, but with an RTAI specific implementation.
- Bits synchronization: likely an inappropriate RTAI jargon for multi events/flags/signals synchronization, i.e. semaphore like operations with different logical masking/unmasking on a set of bits at call and return time.
- Mailboxes: send, receive of messages with multi readers/writers capability, messages queued in either FIFO or priority order. It is possible to use overwriting and urgent sends, broadcast a single message to all task waiting to receive on a mailbox queue, preview any message before reading it.
- Direct Intertask Messages, either in the form of asynchronous: send, receive, or with synchronous remote procedures calls: rpc, receive, return. Messages can be either a single 32 bits value or any size, with the possibility of being previewed before receiving them. RPCs implement priority inheritance to avoid priority inversion and can integrate such a feature with resource semaphores.
- Tasklets and Timers, for prioritized executions of either asynchronous event (tasklets) or time (timers) driven non blocking tasks. Timers allow a lean implementation of flexible timing policies without using a fully featured RTAI task.
- User Space Interrupts (USI) handling, allows to implement interrupt/drivers management directly in user space, making it easier their development and test, eventually going to kernel space with much less troubles and only if needed.
- Watchdog monitor, to help supervising task execution and avoid locks due to inappropriate timings.
- Basic hard real time POSIX supports and message queues.

To be usable in practice RTAI needs clearly also some support for peripherals. So it comes with a real time serial and parallel port driver while drivers for a large variety of data acquisition plug-in boards can be used by means of an RTAI specific interface to the FOSS COMEDI project [7].Communication buses are supported by other FOSS developer [8] along with the real time networking to be described later on.

6. Distributed RTAI

To allow native usage of any of the RTAI APIs on distributed local/remote control nodes, RTAI provides its own small (20KBs) and effective real time middleware layer, called net_rpc. It integrates distributed and local applications by just adding a node/port identifier in front of any RTAI function call. In such a way any application can be run on a single machine or on many networked machines without changing a single line of its source code.

This implementation is nothing but an RTAI specific lean implementation of the RPC idea. The somewhat redundant net_ term in front of it being used just because the acronym "rpc" is used also for RTAI native local inter task messaging APIs. The basic specification for making it easy imposes using just the name of any already available function by substituting the function name prefix rt_... with RT_... and adding two initial arguments more, i.e. the node and the port on the remote machine that will execute rt_... functions that became RT_... Using a null node forces a local execution of the corresponding rt_...function. In this way any application dynamically reverts to a standard local implementation automatically, in a more effective way than using the local host emulation. Naturally it is also possible to run a distributed application wholly in local mode by just setting all the nodes to the local host; less effective than using a null node but useful for developing and testing distributed applications on a single machine. The only change in the formal arguments, with respect to the usual normal local "rt_..." usage, is related to any possible argument associated with a time value. It must be expressed in nanosecs always, instead of the RTAI standard internal timer units count. In this way one is not required to know the frequency of any timer and be sure that a correct timing will be assured irrespective of the node onto which it will run.

The specific implementation adopted in the net_rpc scheme uses a stub (buddy, proxy, agent) task to execute the remote APIs. Stubs and ports are statically connected to guarantee maximum determinism and all services are executed in a strict synchronous request_service /return_result mode. So before doing anything remotely a task has to ask for a port at its remote peer. Port requests/releases are not real time operations, they must be carried out before beginning any true real time work. Because of this assumption they are not time critical and can be managed and timed out in soft real time mode by exploiting Linux directly. Ports are owned by a task and a task can own multiple ports having different identifiers. A port can be associated to a mailbox to support partial asynchronous services, as explained later on. Full asynchronous execution modes require buffering remote requests. They could be easily implemented by mating the stub to a mailbox but are assumed to have little to do with real time. Nonetheless a single asynchronous execution can be requested for non blocking APIs, e.g. semaphore signals, remote tasks resumes, asynchronous inter task messaging, by simply setting the port to its negative number. However any following request to the same node and port can be forwarded only when any previous asynchronous remote API has completed its execution. The remote support stub will answer anyhow and there can be instances requiring the returned values not to be discarded. As previously said net_rpc allows recovering asynchronous returns from remote calls by attaching a mailbox to the port request and a function is provided to inquire about the execution status of the remote service, to synchronize and ascertain the completion of asynchronous executions when it is possible to discard returned values. However if some logging/monitoring of asynchronous calls is required anyhow it is simple to provide a server task that reads from and process the port mailbox in accordance with what is needed by the application at hand. It is important to note that remote calls which make sense being used asynchronously can be called from interrupt handlers also, thus providing the possibility of remote interrupt servers.

An important features of net_rpc is its capability of dynamically expanding its services at run time so that any application specific API available for local operation mode can easily become a distributed service by simply setting up a module containing a table of the new entries and by wrapping the related calls according to the simple net_rpc rules described before. When such an extension module is installed its entries table simply expands already existing net_rpc calls and the new services can be transparently used in a distributed way, without any recoding and without any need to become involved in low level network APIs.

The implementation of low level net services can be anything. The actual readily available support is real time UDP, through dedicated connections and switches. Sharing Linux networking is possible but real time communications must then be forgotten. The non strict determinism of Ethernet UDP protocol is known but using a dedicated high speed hardware guarantees acceptable performances for low to medium bandwidths, say up to 2 KHz, distributed controllers "almost mathematically" and with standard 10/100 Mb ethernet cards and switches. Up scaling to GigaEthernet or Scalable Coherent Interface (SCI) adapters can afford real time determinism for higher bandwidths at an increased cost.

At the moment a true real time network supports come to RTAI from the evolution of an external FOSS project RTNet [9]. Beside plain usage of a standard switch-hub based Ethernet support it features also the implementation of various real time policies to arbitrate the Ethernet in such a way to achieve a stricter guaranteed deterministic networking [10].

7. RTAI-Lab

An existing comprehensive application exploiting RTAI own middleware layer is RTAI-Lab. This tool provides a common structured framework to design, build, run and monitor any suite of RTAI-based single/multitasking controllers and real time simulators, either specifically coded in a high level procedural language, typically C/C++ (some Fortran possible), or automatically generated by proprietary MATLAB/Simulink/Real-Time-Workshop [11], simply RTW in the following, and/or fully FOSS Scilab/Scicos/CodeGen, simply Scicos [12]. The target real time code(s) can be generated and executed in a local/distributed way, as well as the monitoring interface can be executed locally/remotely. The interface to the real time applications involves changing the selected tunable parameters on the fly, scoping and logging selected signals, generic multidimensional data logging, and performance supervision. The building/running and interfacing frameworks represent the two main aspects of RTAI-Lab, which will be separately outlined in the following discussion.

Another freely available, albeit not at a source code level, powerful automatic code generator specifically aimed at highly distributed control systems, i.e. Syndex [13], is being screened for a possible integration into RTAI-Lab. One of the key features of Syndex is its ability to optimize the distribution of controllers on available CPUs, satisfying implementation constraints and taking into account communication delays. Instead RTW/Scicos generated code must be specifically distributed by the controller designer as described below.

It is believed that automatic control system generation from visual block schemes cannot meet the performances achievable by direct coding. In fact the intrinsic generalization of such procedure restricts the level of the final optimization of the whole assembled code, even if a single built-in piece of code implementing a specific operation could be highly efficient. Nonetheless the high computational power afforded by today GPCPUs makes it possible to avoid caring of some efficiency loss in favor of a faster and shorter development cycle allowed by the tighter design, simulation and control system implementation that comes with an effective integrated Computer Aided Control Systems Design (CACSD) environment.

RTW is an automatic C language code generator for Simulink. Under Simulink it is possible to create, simulate and analyze complex hybrid (continuous/discrete, linear/nonlinear) dynamic systems by simply connecting functional blocks, mostly available from various preconfigured libraries, within a friendly graphical user interface. Furthermore, being Simulink part of MATLAB problem solving environment, it shares the same straightforward integration of computation, monitoring and visualization. One of the main advantages of RTW consists in its fully configurable code generator that specifies how to transform a Simulink blocks model into a C code, allowing to produce a target software for virtually any hybrid operating system and platform onto which MATLAB can be run, thus allowing not only code generation but even full real time simulation with hardware in the loop.

Scicos has similar capabilities but can generate only discrete time controllers that can be easily run in real time thanks to RTAI-Lab. So even if Scicos allows to simulate whatever system, it is not capable of true real time simulations. However almost real time simulations could be pursued by having its discrete controllers run under RTAI-Lab interfaced to the hybrid system simulation capability of Scicos. If no hardware in the loop is needed, it is possible to test the controller with performances similar to RTW.

The setting up of control systems within these environments is performed in three main steps. First, the designer creates a Simulink/Scicos block diagram that represents the system implementing the chosen control strategy. The diagram is composed by blocks coming from the built-in library, along with specific RTAI and Digital Acquisition (DAQ) blocks providing the interface to RTAI-Lab monitoring and to the I/O support, respectively. The DAQ support comes both with RTAI specific drivers and with a COMEDI based general interface, thus allowing to easily access a very large number of COTS DAQ hardware. The second step involves the C code generation. The process is straightforward and the user has to choose only the right template makefile for the target language compiler to have the control programming automatically generated with a mouse click. At this point the generated code should be compiled by using the mating generated makefile and linked with the RTW/Scicos specific interface code to the RTAI-Lab framework, called rt_main, to obtain the real time target executable. The code building is such that the final executive can be run along with a set of options that allow choosing between soft/hard real time execution modes, periodic/oneshot timing, internal/external timers, infinite/finite final time and other running options. The executable is a standalone code which can be put on any machine running the same version of RTAI used to build it.

As said above rt_main creates a suitable frame onto which the generated control system runs. Taking the RTW development environment as an example, this architecture involves a task that initializes all the real time stuff and manages the start/stop/final-time events of the real time executive, a task that creates a double way communication channel from/to its RTAI-Lab interfacing counterpart, and as many hard timed tasks as the number of different sampling rates of the corresponding RTW model, scheduled in rate monotonic mode.

What described above may be considered the standard development and execution procedure, since both RTW and Scicos do not provide a native support to automatically and generically distribute the generated code. This limit is partially overcome by RTAI-Lab which supplies the controller designer with a specific library of I/O blocks embedding RTAI net_rpc services into the code generators, thus adding the capability to generate a distributed control system within RTW/Scicos. To set up such a system the designer has to create as many block diagrams as the number of the machines onto which the final code will be downloaded and run. Then he/she should generate and compile each block to obtain the corresponding set of executables. Each model contains the I/O blocks to send to and receive messages from other peers of the distributed real time control system thus allowing distributed applications in an easy way. Note that rt_main does not contain any direct call to remote RTAI functions and thus it remains unchanged when used in a distributed context. In fact the networking layer is provided by the net_rpc module and the specific RT_... calls come from the RTW/Scicos generated code via the net_rpc I/O library. It should however be remarked that the user is responsible of implementing his/her own distributed policy by using the comprehensive APIs set provided by net_rpc, no automatic generation of a distributed application being provided at the moment.

Let us now talk about the second aspect of RTAI-Lab:

the local/remote interface to the real time executives called (x)rtailab, where the x prefix stands for the graphical mode to be run under X terminals. The basic concept of RTAI-Lab is to allow two separate systems, the host and the targets, to communicate. In a remote implementation, the host is the machine where the RTAI-Lab interface is executing in soft real time, the targets are the machine where the generated hard real time codes run. The host send/receives messages using net_rpc requesting the targets to accept parameters changes and to send signal data for graphical displaying and file logging. Such a scheme is implemented also by the Matlab native external mode, which is based on a client/server architecture: MATLAB/Simulink is the client and the target is the server. External mode works by establishing a communication channel between Simulink and the RTW generated code. This channel is implemented by a low-level transport layer that handles physical transmission of messages. Both Simulink and the generated model code are independent of this layer. The transport layer and the code directly interfacing to the transport layer are isolated in separate modules that format, transmit, and receive messages and data packets. As said above, in RTAI-Lab this layer is fully integrated into the real time code via net_rpc without the need to adapt and link it with an ad hoc transport layer implementation. Moreover the adopted scheme allows to communicate with a network of PCs running the distributed control system. This basic framework can be extended by running on the same host many RTAI-Lab sessions, thus monitoring and interfacing many sets of targets simultaneously. To allow interoperability of local/remote/distributed usage, (x)rtailab allows to configure a server by setting some identifiers, as well as the remote node addresses. The identifiers refer to the real time objects of the executables which are devoted to the host interface, i.e. the rt_main interface task and the scope and data log mailboxes, and thus permit to distinguish and separate corresponding environments of RTAI-Lab sessions simultaneously running on the same machine. Moreover, since the interface is common, (x)rtailab is able to communicate both with all controllers. either generated by RTW or Scicos, that are running on the same machine.

8. Conclusions

Issues related to the design and implementation of high performance distributed control systems have been discussed in relation to the solutions and the services provided by RTAI. Hard real time specifications have been related to the actual system bandwidth to justify the viability of using COTS hardware for high demand applications. The main features of the simple, small and effective middleware layer provided by RTAI have been outlined. Its remote procedure call middleware has proven suitable in supporting the development and implementation of distributed control systems and adequate in relieving its user from low level network programming. Such features have been vividly demonstrated in a readily available advanced tool (RTAI-Lab) aimed at supporting a more and more simple and effective development, management and monitoring of complex distributed control systems, including appropriate interfaces to effectively exploit widely available proprietary and FOSS automatic code generators.

9. References

[1] A.S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, NJ, 2001.

[2] http://www.aero.polimi.it/~rtai

[3] http://www.kernel.org

[4] D. Stodolsky, J.B. Chen, and B.N. Bershad, "Fast Interrupt Priority Management in Operating System Kernels", *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, San Diego, CA, USA, 1993.

[5] D. Probert, J.L. Bruno, and M. Karaorman, "SPACE: A New Approach to Operating System Abstraction", *Proceedings of the International Workshop on Object-Orientation in Operating Systems*, Palo Alto, USA, 1991.

[6] http://www.freesoftware.fsf.org/adeos

[7] https://cvs.comedi.org/comedi

[8] http://www.port.de/engl/canprod/sw_linux.html

[9] http://www.rts.uni-hannover.de/rtnet

[10] S. Schneider, G. Pardo-Castellote, and M. Hamilton, "Can Ethernet Be Real Time?", Real-Time Innovations, Sunnyvale, CA, http://www.rti.com/products/ndds/ethernet.pdf

[11] http://www.mathworks.com

[12] http://www.scilab.org

[13] http://www-rocq.inria.fr/syndex