Abstract: Many distributed computer control systems are non-critical in the sense that they have only soft time constraints and no stringent safety requirements. This is the case with certain teleoperated robot control systems, which can be effectively built on top of general purpose hardware and software execution platforms, instead of having to use costly and difficult-to-program real-time computers and operating systems. An approach for developing such kind of applications, based on the use of Linux and Ada on PC architectures, is presented in this paper. Distribution is based on the Distributed System Annex of Ada on top of a TCP/IP LAN. A number of guidelines on how to use them in a proper way are described with reference to an example teleoperation architecture.

Keywords: Soft real-time systems, Ada, Linux, teleoperation systems, robotics.

1. INTRODUCTION

Many real-time control systems have critical safety requirements, which means that failures in the execution of the system may lead to unacceptable consequences, such as economic losses, environmental damages, or even danger to human lives. Therefore, the development of these systems requires appropriate and rigorous development techniques, together with highly reliable hardware and software execution platforms. Moreover, most computer control systems have strict time requirements which must be guaranteed. This implies the use of special real-time operating systems with such special features as high kernel reliability, high resolution timers, or bounded kernel service execution time. As a result, the development of this kind of real-time control systems—usually known as hard real-time systems—tends to be difficult and expensive (Burns and Wellings, 2001).

On the other hand, there are other kinds of control systems which have less stringent requirements (i.e. they are soft real-time systems). These systems can usually tolerate isolated failures in computation or timing without catastrophic consequences.

This is the case for a large class of robot control systems which are supervised or teleoperated by humans. Examples can be found in such application areas as nuclear plant maintenance robots or ship cleaning robots. These kinds of robots usually operate at low speeds, and in the case of a failure in the computer control system, the operator has means for manually stopping the robot in a safe way.

This flexibility enables the development of these type of computer control systems to rely on general-purpose execution platforms which are easier to program and test, and more affordable too, than the specialized platforms which are required for hard real-time systems.

The rest of the paper describes a novel approach for the development of distributed soft real-time control systems.
systems. The approach has been used in the GOYA project (Iborra and Álvarez, 1999), which is aimed at developing a robot for automatic hull blasting and coating removal on ships. The execution platform is based on GNU/Linux and an industrial PC. The programming language is Ada (Ada, 1995). Distribution is based on the Distributed System Annex (DSA) of Ada. The particular implementations are GNAT, the GNU-NYU Ada Translator (Schonberg and Banner, 1994), and GLADE (Pautet and Tardieu, 2001). The project has been used as a test bench to explore the applicability and limits of this approach, and to compare it with other alternatives. As a result, a set of guidelines for developing applications with a sufficient degree of robustness and determinism have been obtained.

2. PROBLEM DEFINITION AND ANALYSIS

2.1 System description

The goal of the GOYA project is to develop an automated and environment-friendly system for hull blasting and coating removal on big ships. The system uses a new technology based on high pressure water jets mounted on a moving cleaning head. This technique is intended to replace the current paint and coating removal methods, based on open grit blasting, which is likely to be banned in the near future because of the air pollution it causes.

One of the technical activities of the project is the development of a robot for maneuvering the cleaning head along the ship hull surface. The robot system is teleoperated, and is composed of the following elements:

- *Hydraulic elevation platform*, which is in charge of moving up and down the rest of the components. It can move the cleaning head on a band between 800 mm and 3300 mm of height, approximately. An additional supplement with a height of 2500 mm will allow to increase the cleaning band. Its motion speed is around 0.07 m/s.
- *Positioning arm*, which moves the cleaning head in a direction perpendicular to the ship surface. Its speed is around 0.1 m/s.
- *Head positioning motor*, which moves the cleaning head in a direction parallel to the ship surface.
- *Head orientation motor*, which moves a pan and tilt head that directs the blasting with an incidence angle of 45° with respect to the ship surface.

The sampling time requirements are in the order of 100-300 ms. These values are derived from the speed of the motors and the refresh rate of the teleoperation system display. Actuator deadlines are taken equal to the sampling periods. The characteristics of the system and the application are such that missing a deadline from time to time does not cause any unacceptable harm. Therefore, the system has soft real-time requirements.

2.2 Analysis of alternatives for the execution platform

The system architecture is shown in figure 1. It is composed of two computer subsystems: the control subsystem, which drives the operation of the robot, and the teleoperation subsystem, which interacts with a human operator, who can see the state of the robot and remotely drive its operation. These subsystems are connected by a dedicated Ethernet LAN with no additional nodes. There is a safety control panel directly connected to the robot which can be used for emergency stop when required.

The control subsystem interfaces directly with the robot sensors and actuators, driving the motors and the start and end of the blasting operation, and executing the control algorithms for all of them. It also accepts commands from the teleoperation subsystem, and sends messages to it through the LAN link.

The teleoperation subsystem displays the state of the robot and accepts and interprets operator commands for the robot. The state is updated on a regular basis in response to periodic commands which are automatically sent to the control subsystem.

Two computers are used because of the different nature of the operations to be performed. Moreover, the functional requirements lead to using two different platforms: one with powerful graphical capabilities, for showing the robot state and the operational environment with enough detail, and another one with the appropriate interface hardware for interacting with the robot. The robot control subsystem runs on an industrial PC, while the teleoperation subsystem is based on a workstation with powerful graphics capabilities. The rest of this paper is focused on the design and implementation of the control subsystem, as the teleoperation subsystem has no special time and safety requirements, and can be developed with general-purpose techniques.
Three alternatives for the basic software platform were explored:

- **Linux (Bovet and Cesati, 2000).**
- **Linux and RTLinux (Barabanov and Yodaiken, 1996).**
- **A dedicated real-time kernel, such as JTK (Ruiz and González-Barahona, 1999) or RTEMS(OAR, 2000), running on the bare hardware.**

The advantages of the first approach are the wide range of available drivers and the possibility of using the same platform for development and execution. The availability of drivers for the most common Ethernet cards and protocols is especially interesting. The main disadvantage of this approach is the lack of a true real-time kernel, although some of the functionality for real-time threads in the POSIX standard is implemented (IEE, 1990).

The other two alternatives rely on real-time kernels, which provide the required functionality for implementing real-time tasks. The RTLinux/Linux configuration is particularly interesting, as it enables the use of a real-time kernel while taking advantage of the Linux drivers. The main drawback of this approach is the perception that RTLinux was not mature enough for an industrial project at the start of this project, and the lack of an Ada compiler for this kernel.

The first configuration with Linux only was finally selected for the GOYA project, after carefully checking that the real-time facilities it provides are sufficient for meeting the functional and timing requirements of the system. In this way, it was not necessary to spend time learning how to use the system or developing complex drivers, and it was possible to use the large amount of information available about this operating system.

### 3. LINUX REAL-TIME BEHAVIOUR

There a number of studies on the possibility of using Linux in the development of real-time systems. One conclusion of these studies is that there are indeed some problems in using Linux as a platform for the design and development of real-time systems. The most important ones are:

- **Linux scheduler:** The basic Linux scheduler is based on the well-known time-sharing technique (Bovet and Cesati, 2000). CPU time is divided into time slices, one for each runnable process. If the process is not terminated or self-suspended when its current slice expires, a process switch occurs and the CPU is granted to another ready process. Processes are ranked according to priority, which is a process parameter which may change dynamically according to complex algorithms. This type of scheduler is clearly not appropriate for real-time systems (Burns and Wellings, 2001).

However, Linux provides another type of scheduler, in compliance with the POSIX standard (IEE, 1990). The alternate scheduler is identified as **SCHED_FIFO**, and it uses a priority-based preemptive method to assign the CPU to a ready process. It is possible to specify the scheduling policy to be used and a fixed real-time priority for a process. Real-time priorities are always more urgent than non-real-time priorities.

- **Timer resolution:** As documented (and measured) the resolution of Linux timers is 10 ms, which is indeed to coarse for fast real-time system, as it can lead to errors of up to 20 ms when programming timer events. However, the time requirements of the GOYA control subsystem are much longer, and it is thus feasible to use Linux timers for its implementation.

- **Dynamic memory:** This feature can increase significantly the execution time of a process, if some portion of its address space is moved onto disk. However, this can be easily overcome by using the **mlockall** system call which disables paging for the calling process.

- **Kernel preemptability:** The Linux kernel is not preemptable. This means that whenever a system call is issued, the kernel executes without preemption, no matter the priority of the runnable user processes. The effect is that real-time processes can suffer undesirable blocking.

- **Interrupt disabling:** The kernel or the device drivers may disable interrupts to prevent their being preempted by an interrupt or another kernel activity. It is clear that while interrupts are disabled, a real-time process will not run, no matter its priority.

The main consequence of this scenario is that real-time processes may suffer undesirable blocking, which causes an increase in their response times. The main problem is that it is difficult to bound the length of the worst case blocking time for a process. This undesirable effect can be reduced by removing from the kernel those modules and device drivers that are not necessary for the execution of the real-time application. In addition, it is convenient to disable all services that are not required.

In order to validate the approach, a number of experiments have been made with the target system. The results indicate that the highest blocking suffered by a real-time Linux process is tolerable, according to the time requirements of the application.

### 4. THE ADA PROGRAMMING LANGUAGE

Ada(Ada, 1995) is the implementation language selected for the GOYA project. It was considered to be the most suitable language because of the following reasons:
Reliability. Ada is a highly reliable language, in which many errors can be found at compilation time, where there are less costly to fix, as opposed to C and especially C++, which very often show complex run-time errors.

Abstraction. Ada is a high level language which includes many concurrency and real-time features as part of the language itself. It also includes an annex for the development of distributed programs. This makes the implementation more reliable, as the compiler is able to make a number of semantic checks. This approach is better than the use of sequential languages such as C and C++, which require handling complex operating system interfaces to provide the same features.

Predictability. Ada’s built-in fixed priority scheduling and intertask synchronization mechanisms enable accurate timing analysis to be performed, and predictable timing behaviour to be enforced.

Availability. GNAT, the GNU/NYU Ada Translator\(^1\), is a high quality development environment for Ada 95, which is available as free software (Schonberg and Banner, 1994). The GNAT environment includes native and cross compilers for many common platforms, as well as a graphic debugger and other useful tools.

Industrial penetration. Although the use of Ada in general industry applications is much less extended than C or C++, it is the first choice language for safety-critical systems.

5. CONTROL SUBSYSTEM DEVELOPMENT

5.1 Design overview

The robot controller is the most relevant subsystem of the teleoperation system from a real-time point of view. The subsystem design relies on four components, one for each of the moving components of the robot, with similar internal structures. A simplified version of this structure is shown in figure 2.

Each component provides a remote interface to the teleoperation system, enabling it to perform a number of operations on the robot components. For example, the remote interface for the elevation platform is defined by the following Ada package:

```ada
package P_Platform.Telemetry_Interface is
  pragma Remote_Call_Interface;
  function Get_State return T_State;
  procedure Stop_Platform;
  procedure Raise_Platform;
  procedure Move_Platform_To (A_Height : T_Position);
end P_Platform.Telemetry_Interface;
```

Whenever a command is issued, the internal state of the robot is changed accordingly. For example, if a command is received to raise the platform, the operating state of the robot is updated and an actuation signal is sent to the elevation motor.

The active parts of a controller component are a couple of tasks. The first one is a periodic task which reads the sensors to get the current position of the platform. This information is processed in order to update the robot state and generate actuation signals, if required. The other one is a sporadic task which drives the platform actuators when required by a teleoperation command or a state change. Both tasks also check the state of the rest of robot components so that a coordinated operation is ensured.

5.2 Implementation details

The purpose of this section is to provide details on how the Ada language and the Linux operating system have been used in order to get a proper behaviour from the application.

5.2.1. Ada process configuration

Each controller subsystem is implemented as an Ada program, which in turn is executed as a Linux process at run time. In order to have a proper behaviour, controller processes are configured as follows:

- The scheduling policy for the process is set to SCHED_FIFO, and the process is assigned a high priority within the allowable range. In this way, control processes have higher priorities than other user processes.
- The address space of the process is locked in memory and hence paging is disabled.

The configuration options are implemented by calling the appropriate Linux services when the process execution starts. The facilities provided in Ada for interfacing with C functions are heavily used for this purpose. Linux system calls are invoked from the main Ada procedure, as shown in the following example:

\(^1\) http://www.gnat.org
procedure Configure_Ada_Process is

  function Fork return Integer;
  pragma Import(C, fork);

  function Getppid return Integer;
  pragma Import(C.getppid);

  function Mlockall (Mode : Integer) return Integer;
  pragma Import(C.mlockcall);

  type Sched_Param is record
    Sched_Priority : Integer;
  end record;
  pragma Convention(C,Sched_Param);
  type Access_Sched_Param is access Sched_Param;

  function Sched_Getscheduler(Pid : Integer) return Integer;
  pragma Import(C.sched_getscheduler);

  SCHED_FIFO : constant Integer := 1;
  Parameter : Access_Sched_Param := new Sched_Param;
  Error : Integer := 0;

begin
  Parameter.Sched_Priority := 99;
  Error := Mlockcall(2);
  Error := Sched_Setscheduler
    (My_C_Interface.Getppid,
     My_C_Interface.SCHED_FIFO, Parameter);
  if Error < 0 then
    Handle_Error;
  end if;
end Configure_Ada_Process

5.2.2. Drivers in Linux

The hardware platform includes a number of cards with D/A and A/D converters and digital I/O. There are additional cards for controlling the motors. Drivers have been developed in the usual Linux way. In order to ease their use within Ada, an interface package has been developed in Ada. This package calls the Linux drivers using the facilities for interfacing with C. It exports a number of functions for accessing the input/output devices with a higher abstraction level. This interface makes no reference to the underlying cards and eases the porting of the Ada code.

5.2.3. Distribution

The Distributed System Annex (DSA) of Ada defines several extensions to write distributed systems entirely in Ada. It is based on the use of remote procedure calls (RPC) and transparent remote invocation of object methods.

A distributed application is developed using the DS in much a similar way as it were an ordinary centralized application. The developer divides the application into several partitions, which are executed as different processes on the same or different computers. The packages that make up the application are assigned to different partitions with help of a configuration tool. Procedures defined in remote packages (i.e. packages in other partitions) can be called from a partition, provided they comply with some restrictions which are checked at compilation time with help of so-called categorization pragmas.

An example of categorization pragma can be seen in the code of the teleoperation interface that was given in section 5.1. The pragma Remote_Call_Interface states that the procedures declared in the package can be called remotely. The pragma Pure is used to categorize packages with type and constant definitions only. This implies that the package has no internal state and can be replicated in all the partitions.

GLADE (Pautet and Tardieu, 2001) is an implementation of the DSA programming model for GNAT. It provides means for configuring a distributed application and a full PCS (Partition Communication Subsystem), called GARLIC, which is a runtime library which is linked with applications to make different parts communicate together in a transparent way.

Gutiérrez and González (2000) have studied the use of the DSA in the development of real-time systems, coming to the conclusion is that it is not appropriate for developing hard real-time systems. However, a proper configuration of the application and the GLADE middleware makes it possible to use it in soft real-time systems.

The GOYA controller and teleoperation subsystems communicate through remote procedure calls (RPC), according to the Ada DSA model. RPC requests are automatically received and analysed by a service task. If everything is correct, it passes the request to one anonymous task in a server pool in order to execute it.

Some of the DSA real-time problems identified by Gutiérrez and González (2000), and the way they have been coped with in GOYA are:

- **Priority type.** The type System.Priority represents relative priorities that are meaningful only to one processor in the system. This can be overcome by carefully assigning priorities in all computers in a consistent way. The only problem with this approach is that it is not portable.

- **Priority of the service task.** This is the task that initially handles the RPC request and allocates another task from the server pool to execute it. The priority of this task is set to Priority’ Last in order to start the execution of the RPC as soon as possible, and also to bound the effect of priority inversion.

- **Priority of the anonymous server tasks.** The priority of tasks in the server pool is initially set to a default value. When one of these tasks is signalled to handle a RPC request, the task inherits the priority of the caller task. This priority is indeed the appropriate one for executing the request, as priorities have been set with this purpose on both sides. However, if the initial priority of the tasks in the server pool is too low,
there is a potential danger of priority inversion. In fact, this is the case with GLADE. In order to overcome this problem, the code of GARLIC has been slightly modified to set a higher initial priority to the tasks in the pool.

- **Number of tasks in the server pool.** If the number of tasks in the pool is less than the maximum number of simultaneous requests, it may happen that some of the calling tasks are blocked. This situation may compromise the system real-time behaviour if the blocked requests have a high priority. An easy way to prevent this to happen is to ensure that there are enough tasks in the pool for dealing with the maximum number of simultaneous requests. In the GOYA controller subsystem there may be up to three simultaneous requests:
  - One request to get the robot state. This kind request is periodically issued by the system.
  - One request to execute a robot command. This kind of request is sporadically issued by the operator. The system is designed so that it is not possible to issue a new command until the previous one has been received and initiated. In this case, the caller does not wait until the end of the command execution, because of its possibly long duration. The command is sent to the robot controller and the operator checks the status of the operation, looking at the robot state which is displayed on the teleoperation monitor.
  - One request to execute a robot stop command.

Hence the system is configured to have a pool of three tasks in order to handle these requests.

### 5.3 Testing the approach

A number of tests have been performed in order to check the suitability of this approach. A common Linux configuration, with all unused services (including the X windows system) removed, has been used. The test results show that it is indeed possible to make real-time application tasks run with a high priority, suffering only a small amount of interference from system activities.

Additional tests have been performed in order to test the behaviour of the GLADE middleware. The tests confirmed that priority inheritance mechanism described in the documentation really works. The pool tasks execute the RPCs with the priority of the caller.

### 6. CONCLUSIONS

The experiments performed let us conclude that it is feasible to use Linux and Ada in the development of soft real-time systems with time requirements longer than 50 ms. A number of guidelines on how to configure the Ada program and GARLIC in order to get an appropriate real-time application behaviour have been presented. The use of GLADE for distribution has been found especially interesting and positive.

### 7. REFERENCES


