Real-Time Systems, 26, 261–296, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Replication Management in Reliable Real-Time Systems

LUÍS MIGUEL PINHO lpinho@dei.isep.ipp.pt
Department of Computer Engineering, School of Engineering, Polytechnic Institute of Porto, Portugal

FRANCISCO VASQUES vasques@fe.up.pt
Department of Mechanical Engineering, University of Porto, Portugal

ANDY WELLINGS andy@cs.york.ac.uk
Department of Computer Science, University of York, York, UK

Abstract. Building reliable real-time applications on top of commercial off-the-shelf (COTS) components is not a straightforward task. Thus, it is essential to provide a simple and transparent programming model, in order to abstract programmers from the low-level implementation details of distribution and replication. However, the recent trend for incorporating pre-emptive multitasking applications in reliable real-time systems inherently increases its complexity. It is therefore important to provide a transparent programming model, enabling pre-emptive multitasking applications to be implemented without resorting to simultaneously dealing with both system requirements and distribution and replication issues. The distributed embedded architecture using COTS components (DEAR-COTS) architecture has been previously proposed as an architecture to support real-time and reliable distributed computer-controlled systems (DCCS) using COTS components. Within the DEAR-COTS architecture, the hard real-time subsystem provides a framework for the development of reliable real-time applications, which are the core of DCCS applications. This paper presents the proposed framework, and demonstrates how it can be used to support the transparent replication of software components.

Keywords: real-time systems, software-based fault tolerance, distributed computer-controlled systems, transparent replication

1. Introduction

The problem of supporting reliable real-time distributed computer-controlled systems (DCCS) is reasonably well understood, when considering synchronous and predictable environments. However, the use of commercial off-the-shelf (COTS) components and the integration, in the same environment, of hard and soft real-time applications introduces additional problems. The integration of COTS components in DCCS needs to be combined and complemented with real-time and fault tolerance attributes, in order to provide the required timeliness and reliability guarantees. The guarantee of these properties cannot be achieved by an improvized solution, as it has been shown by structured approaches to the problem of designing dependable real-time systems (e.g., DELTA-4, Powell, 1991; MARS, Kopetz et al., 1989; or GUARDS, Powell, 2001).

To ease the task of building reliable real-time applications on top of COTS components, it is essential to provide a simple and transparent programming model, in order to abstract programmers from the low-level implementation details of distribution and replication. However, replication of pre-emptive applications usually requires the explicit programming of such low-level mechanisms. It is therefore important to devise a transparent and generic framework to implement pre-emptive multitasking applications without resorting to simultaneously dealing with both system requirements and distribution and replication issues.

The distributed embedded architecture using COTS components (DEAR-COTS) architecture (Veríssimo et al., 2000b) has been proposed to support hard real-time and reliable DCCS using COTS components, allowing the easy integration of soft and non-real-time applications. In order to cope with such integration requirements, DEAR-COTS relies on the separation of a hard real-time subsystem (HRTS) from a soft real-time subsystem (SRTS), which communicate through a well-defined interface. The timely computing base (TCB) model (Veríssimo et al., 2000a) is used to deal with the heterogeneity both of system components and of the environment, with respect to timing properties. It is within the HRTS that distributed reliable hard real-time applications are supported. This paper proposes a framework intended for the support of replicated software components, within the HRTS of the DEAR-COTS architecture. This framework hides from the supported applications the details of the mechanisms for replication and distribution. Such mechanisms are only considered in a later configuration phase, where the system is configured and distributed over the DEAR-COTS nodes.

The paper is structured as follows. The following section presents relevant work related to the support of reliable real-time systems. Section 3 presents the DEAR-COTS architecture, while Section 4 presents the proposed framework for supporting replicated applications. Sections 5 and 6 detail the framework's repository of generic objects for task interaction, and the replica manager layer which is intended to support the application objects. Finally, Section 7 overviews the prototype implementation, and Section 8 provides some conclusions.

2. Related Work

A considerable research effort has been devoted to the design and validation of dependable real-time systems architectures. The most significant examples are DELTA-4 (Powell, 1991), MARS (Kopetz et al., 1989) and GUARDS (Powell, 2001), as these architectures intend to provide non-application specific frameworks to the design of dependable real-time applications.

The DELTA-4 (Powell, 1991) project aimed to develop an open, dependable architecture for large distributed real-time systems. In DELTA-4, nodes are split into two different subsystems: the host, which is a COTS component, and the network attachment controller (NAC), which is a fail-silent component making use of specialized self-checking hardware. The need to target systems with more stringent timing requirements, lead to the specification of the extended performance architecture (XPA). XPA systems are constituted by a set of distributed homogeneous nodes, connected by a fiber distributed data interface (FDDI) network, where the host is also considered to be fail-silent. However, contrarily to the NAC, the fail-silent behavior of hosts is achieved through the use of soft fail-silent nodes, where fail-silence behavior is achieved through software management of replicated processors.

MARS (Kopetz et al., 1989) is a fault-tolerant distributed real-time system intended to support process control applications. The architecture consists of one or more clusters, which are distributed systems composed of single board computers, called components, connected by a time division multiple access (TDMA) network. All the components maintain a global time base, which allows them to synchronize their actions and to use a time-triggered approach. In MARS, both node and network schedules are determined off-line and stored in a static schedule table. Components are guaranteed to be fail-silent, through the use of self-checking hardware, running in dual active redundancy, and the use of two redundant real-time networks where messages are sent in duplicate.

The GUARDS (Powell, 2001) project intended to develop a generic architecture, substantially based on the use of COTS components, in order to minimize the development time and costs associated with critical real-time applications. The architecture resorts to software-based fault tolerance mechanisms, in order to cope with the unreliability in the underlying COTS components. The difficulty of providing fault tolerance based on COTS components, led to the development of a two-level replication approach. The architecture is constituted by a set of channels, each one containing replicated hosts interconnected by a shared memory scheme. These channels are interconnected at a second level by the interchannel communication network (ICN), which is based on unidirectional serial links scheduled in a static off-line table-driven approach. As the channel replication is motivated by fault tolerance, it is not foreseen the need for systems with more than three or four channels. This architecture is targeted to safety- or mission-critical systems (in the domains of railway, nuclear and space applications) requiring a greater level of dependability and a restrictive set of failure assumptions (Laprie, 1992).

A software-based fault tolerance approach is a fundamental issue for COTS-based reliable architectures. As there is no specialized hardware with self-checking properties, it is the software that must manage replication and fault tolerance. In this approach, a fault-tolerant service is implemented by coordinating a group of software components replicated on different nodes. The idea is to manage the group of software components in order to mask failures in some of its members. Inter-replica co-ordination gives the illusion to other software components that the group is a single (fault-free) software component (Powell, 1991).

Three main replication approaches are addressed in the literature: active replication, primary-backup (passive) replication and semi-active replication (Powell, 1991). In active replication, all replicas process the same inputs, keeping their internal state synchronized and voting all on the same outputs. In the primary-backup approach only one replica (the primary) is responsible for the inputs processing. In the semi-active replication one of the replicas (the leader) coordinates the non-deterministic decisions. If fail-silent replicas are assumed, then the three approaches can be used. Otherwise, in the absence of the fail-silent assumption, wrong service delivery can only be detected by active replication, since it is required that all replicas output some value, in order to perform some form of voting. Therefore, active replication is the most adequate technique (Powell, 1991). The use of COTS components implies generally fail-uncontrolled replicas, as these components usually do not have the required self-checking mechanisms that could be implemented in custom-built components. It is thus mandatory to use active replication techniques.

As real-time applications are based on time-dependent mechanisms, the different processing speed in replicated nodes can cause different task interleaving. Consequently, different replicas (even if correct) can respond to the same inputs in different orders, providing inconsistent results if inputs are non-commutative. That is the problem of replica determinism in distributed real-time systems (Poledna, 1994).

Determinism can be achieved by restricting the application from using timing non-deterministic mechanisms. However, the use of multitasking would not be possible, since task synchronization and communication mechanisms inherently lead to timing non-determinism. This is the approach taken by both MARS and DELTA-4. The former by using a static time-driven scheduling that guarantees the same execution behavior in every replica. The latter by restricting replicas to behave as state-machines (Schneider, 1990), when active replication is used.

Guaranteeing that replicas take the same scheduling decisions, by performing an agreement in every scheduling decision, allows for the use of non-deterministic mechanisms. However, it imposes the modification of the underlying scheduling mechanisms and leads to a huge system overhead, since agreement must be made at every dispatching point. This is the approach followed by previous systems, such as SIFT (Melliar-Smith and Schwartz, 1982) or MAFT (Keickhafer et al., 1988), both architectures for reliable hard real-time systems with restricted tasking models. However, the former incurred overheads up to 80% (Pradhan, 1996), while the latter was supported by dedicated replication management hardware.

The use of the timed messages concept, independently developed by Barrett et al. (1995) and by Poledna (1998) and then integrated by Poledna et al. (2000), allows a restricted model of multitasking to be used, whilst at the same time minimizing the need for agreement mechanisms. This approach is based on preventing replicated tasks from using different inputs, by delaying the use of a message until it can be proven (using global knowledge) that said message is available to all replicas.

Timed messages are based in the global knowledge of the release time (the instant where the task becomes ready for execution) and worst-case response times of tasks. Using this global knowledge, it is possible for tasks to read the latest version of a value that it is known to be available in all replicas. In this approach, messages are associated with a validity time, defined as the instant where the message value becomes valid,[1] that is, the instant when it is known that all replicas of the writer task have already written the value. In order to guarantee that replicated tasks read the same value, it is necessary to store several versions of the same message. Reader tasks read the version that has the maximum validity time older than their release time (the release time of replicated tasks is known globally in the system).

This is the approach used in the GUARDS architecture (Wellings et al., 1998) in order to guarantee the deterministic behavior of replicated real-time transactions. However, in the GUARDS approach, this mechanism is explicitly used in the application design and implementation, thus forcing designers and programmers to simultaneously deal with both system requirements and replication issues.

The DEAR-COTS HRTS also uses the timed messages mechanism to achieve deterministic behavior of replicated components. However, it is considered that a transparent approach to replication and distribution provides a simpler programming model, so that programmers abstract from the details associated to distribution and replication issues. Therefore, in DEAR-COTS the timed messages concept is transparent to applications.

3. The DEAR-COTS Architecture

The DEAR-COTS architecture (Veríssimo et al., 2000b) provides a COTS-based framework to execute reliable hard real-time applications, allowing the integration of soft real-time and non-real-time applications without interfering with the guarantees provided to hard real-time applications. In order to cope with the integration requirements, DEAR-COTS relies on the separation between a HRTS and a SRTS, which communicate through a well-defined interface. In the DEAR-COTS architecture, the TCB model (Veríssimo et al., 2000a) is used as a reference model to deal with the heterogeneity both of system components and of the environment, with respect to timeliness. The TCB model requires systems to be constructed with a small control part, the TCB module, to protect resources with respect to timeliness and to provide basic time related services to applications. Soft real-time applications can use the TCB to achieve different levels of timing guarantees, even in a soft real-time environment as the SRTS. Additionally, hard real-time applications can use the TCB services to be aware of their timeliness and, therefore, to preserve the reliability of the system. The reasoning is that the TCB module is built as a small control module, therefore it can be built with greater coverage of failure assumptions.

A DEAR-COTS system is built using distributed processing nodes, where distributed hard real-time and soft real-time applications may coexist. To ensure the desired level of reliability to the supported hard real-time applications, specific components of these applications may be replicated. Each DEAR-COTS node can be constituted by several different subsystems, within which applications with different requirements will be executed. A DEAR-COTS node is characterized by the subsystems it is composed of. There are essentially three basic node types: hard real-time nodes (H), soft real-time nodes (S) and gateway nodes (H/S).

Hard real-time nodes are those where only a HRTS exists. Therefore, they will exclusively be used to support reliable hard real-time applications, which are the core of the DCCS application. Soft real-time nodes only include a SRTS, providing the execution environment for the remote supervision and remote management of DCCS applications. A gateway node integrates both subsystems, with two distinct and well-defined execution environments. The idea is to allow hard real-time components, executing in the HRTS, to interact in a controlled manner with soft real-time components, executing in the SRTS.

3.1. The HRTS

The set of H and H/S nodes provides a framework to support distributed reliable hard real-time applications. It supports the active replication of software (Figure 1) with

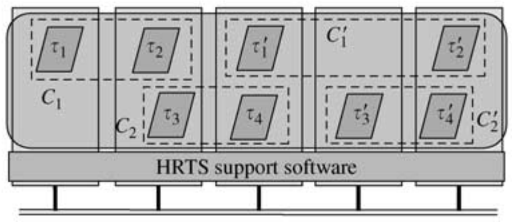

Figure 1. Replicated hard real-time application.

dissimilar replicated task sets in each node. The HRTS support software is a run-time middleware layer between the application and the COTS components, providing the replication and distribution support to hard real-time applications.

The goal of the framework is to tolerate faults in the COTS components underlying the application. In order to tolerate common mode faults in these components, COTS diversity is also considered (operating system and hardware platform). However, using diverse operating systems has to be carefully considered, since in order to guarantee a transparent approach, the programming environment in each node must be similar. This can be achieved by using operating systems with a standard programming interface or by using a programming language that abstracts from the operating system details. DEAR-COTS considers the use of the Ada 95 (ISO/IEC, 1995) language in the replicated hard real-time applications. This solution provides the same programming model in all nodes, while diversity can be provided by using different compilers and runtimes (Yeh, 1995).

The DEAR-COTS architecture does not address tolerance to application design faults. Nevertheless, by providing different execution environments in each node, the tolerance to temporary design faults is increased. Temporary design faults can be tolerated due to the differences in the replicas' execution environment (Powell, 1994), since nodes are considered independent from the point of view of failures. Moreover, dissimilar replicated task sets in each node also increase the system flexibility, as nodes are not just copies of each other, allowing for a more flexible design of real-time applications.

As there is the target of reliability through replication, it is important to define the replication unit. Therefore, the notion of component is introduced. Applications are divided in components, each one being a set of tasks and resources that interact to perform a common job. The component can include tasks and resources from several nodes, or it can be located in just one node. In each node, several components may coexist.

As an example, Figure 1 shows a real-time application with four tasks (τ_1, τ_2, τ_3 and τ_4). The application is divided in two different components (C_1 and C_2), which are replicated (C'_1 and C'_2). Note that, in order to tolerate fail-uncontrolled behavior of the replicas it would be necessary to use $2 * f + 1$ replicas to tolerate f faults (thus three-replicated components to tolerate a single fault).

A similar concept is the ''capsule'' defined in the DELTA-4 architecture (Powell, 1991). As with the DEAR-COTS ''component'', a Delta-4 ''capsule'' is the unit of replication, embodying a set of tasks (referred as threads) and objects. However, a ''capsule'' has its own thread scheduling and separated memory space, and is also the unit of distribution. Thus, the Delta-4 concept of ''capsule'' is more related to Unix processes, whilst the DEAR-COTS component is a more lightweight concept without an implementation counterpart. It is just a configuration abstraction that is used to structure replication units and to allow system configuration.

By creating components, it is possible to define the replication degree of specific parts of the real-time application, according to the reliability of its components. However, by replicating components, efficiency decreases due to the increase of the number of tasks and exchanged messages. Hence, it is possible to trade reliability for efficiency and *vice versa*. Although efficiency should not be regarded as the goal for a reliable hard real-time system, it can be increased by means of decreasing the redundancy degree.

The component concept does not impose restrictions on how applications are structured. Although it is considered that tasks joined in the same component are somehow related, it is the system's engineer role to decide the component structure of the system.

The component is the unit of replication, therefore a component is a unit of fault-containment. Faults in one task may produce the failure of the component. However, if a component fails, by producing an incorrect value (or not producing any value), the group of replicated components will not fail since the output consolidation will mask the failed component. Therefore, in the model of replication, the internal outputs of tasks (task interaction within a component) do not need to be agreed. The output consolidation is only needed when the result is made available to other components or to the controlled system.

However, a more severe fault in a component can spread to the other parts of the application in the same node, since there are no separate memory spaces inside the application. In such case, other application components in the node may also fail, but component replication will mask such failure.

A separate memory space for applications is not mandatory, depending on the operating system support. It is, however, advocated that in order to provide the required reliability level, an operating system allowing the use of separate memory spaces should be used. In order to increase the effectiveness of the fault tolerance mechanisms, the DEAR-COTS support software should also reside in a separate memory space.

The correct implementation of the replication/distribution mechanisms is essential to the correct behavior of the system. If a software fault in the replication/distribution mechanisms occurs in just one node, it can be masked due to the node replication, or the node can be made silent by using the TCB (Veríssimo et al., 2000a). However, if a greater reliability in the development of such mechanisms is considered necessary, approaches considering software diversity should be used.

As active replication is used, there is the need to guarantee replica determinism, that is, that replicated tasks execute with the same data and timing-related decisions are the same in each replica. The use of timed messages (Poledna et al., 2000) allows a restricted model of multitasking to be used and eliminates the need for agreement between the internal tasks of each component. With timed messages, agreement is only needed to guarantee that all replicated components work with the same input values and that they all vote on the final output. The use of timed messages implies the use of appropriate clock synchronization algorithms, since clock deviations must be upper bounded.

Throughout the system lifetime, it is often required to change the applications operational mode, in order to cope with a different set of requirements. Therefore, the HRTS provides support for application-level mode change protocols, through the use of components' shutdown and restart mechanisms. This mode change is restricted to the component level, since it is considered that tasks joined in the same component are somehow related, thus any needed reconfiguration is performed at the component level.

Therefore, to change the operational mode of an application, some of the application components can be shutdown (by preventing its tasks from being released) or silenced (by preventing its output from being disseminated). Components can also be activated, being possible to specify the release time of the periodic tasks of the component.

3.2. The TCB in the HRTS

In the generic model of DEAR-COTS nodes, the TCB can be used to guarantee the timing requirements of soft real-time applications, or to increase coverage of timing faults in the HRTS. Figure 2 presents three different approaches for the use of the TCB within the HRTS of DEAR-COTS. These different approaches can be combined in each particular instantiation of the architecture.

In the first approach (Figure 2(a)), the TCB is used as an additional hard real-time application, to deal with the timing requirements of soft real-time applications executing in the SRTS. In this approach, the replication framework sees the TCB as a hard real-time application.

The second approach (Figure 2(b)) is to use the TCB as a timing error detector at the hard real-time applications level. That is, tasks in the HRTS may use the services of a TCB to detect timing errors, thus increasing the failure assumption coverage of the application. In this approach, the TCB is used as a second independent level of fault tolerance. However, to the framework, it appears as a hard real-time application.

The third approach (Figure 2(c)) is to use the TCB to increase the reliability of the HRTS support software itself. In this approach, support software tasks use the TCB to detect their own timing errors. This solution increases the system reliability, since it is possible to detect both support software tasks' overruns and incorrect communication requests.

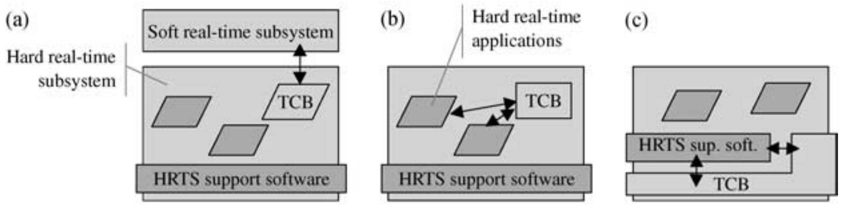

Figure 2. The TCB in the HRTS.

3.3. Error Detection and Recovering

On the occurrence of faults, they will be masked by the component replication. However, in the case of permanent (or intermittent) faults, the system can no longer provide the same reliability level as it was designed to. Therefore, recovery actions must be executed.

Three different approaches are considered in instantiated DEAR-COTS systems. First, the occurrence of faults may require just the notification of the higher levels of the DCCS, and thus no further action is taken. Second, it is possible to design applications in order to provide a fail-safe system, thus to correctly shutdown when the required reliability level cannot be provided. Finally, applications may attempt to maintain a functioning system in a degraded mode, until the faulty components can be repaired (or replaced) and the required system capabilities are restored.

However, integrating new components in active systems is not an easy task, as these newly created components are in an ''amnesia'' state (Powell, 1994), that is, they have no knowledge of the system state. This implies that, before its activation, its state must be made consistent with the replicas that continued execution. However, this state is very application specific and therefore this transfer cannot be easily performed in a generic or transparent approach (Powell, 1991; Rushby, 1996; Bondavalli et al., 1998). Moreover, the efficient transfer of this internal state is highly dependent on the properties of applications, namely data-flow dependencies among tasks, and also on how internal task state data is replaced by external data (Rushby, 1996).

Error recovery and state restoration is still an open issue in the DEAR-COTS architecture. It is considered that an efficient and reliable mechanism can only be obtained with some knowledge of the semantics and data-flow properties of applications, therefore it cannot be provided by the support software itself and application level mechanisms must be used (Figure 3). Nevertheless, it is also considered that the replication management framework must support error detection and recovery actions. Therefore, the support software of the HRTS provides mechanisms to notify the applications of error detection (thus, to take actions to recover from them) and component's shutdown (or silence) mechanisms can be used to prevent components that are performing incorrectly from contaminating the application.

Error detection in the HRTS may take two complementary forms: by the detection of errors in the communication algorithms and the consequent consolidation of replicated values, or by the use of the TCB. The support software may also be used to notify

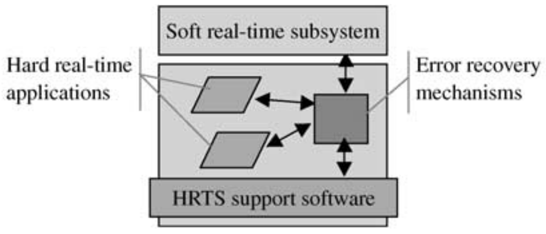

Figure 3. Error recovery mechanisms.

applications and/or to disseminate the error detection through the real-time network. The TCB can also be used to detect the occurrence of timing errors in the HRTS (using approach (c) of Figure 2), in order to, for instance, silence the node so that the network is not contaminated.

4. Framework for Reliable Hard Real-Time Applications

In the HRTS, each application is composed of a set of related tasks $(\tau_1, \ldots, \tau_n)$, each task being a single processing unit. Tasks from the same application can be allocated to different nodes. In order to allow the use of current off-line schedulability analysis (Audsley et al., 1993), each task is released only by one invocation event, but can be released an unbounded number of times. A periodic task is released by the runtime (temporal invocation), while a sporadic task can be released either by another task or by the environment. After being released, a task cannot suspend itself or be blocked while accessing remote data (remote blocking). Tasks are designed as small processing units, which, in each invocation, read inputs, carry out the processing, and output the results. The goal is to minimize task interaction, in order to improve the system's efficiency.

Tasks are allowed to communicate with each other either through Shared Data objects or by Release Event objects (which can also carry data). Shared Data objects are used for asynchronous data communication between tasks, while Release Event objects are used for the release of sporadic tasks.

Although the goal is to transparently manage distribution and replication, it is considered that a completely transparent use of these mechanisms may introduce unnecessary overheads. Therefore, the application programmer (transparent approach) does not consider the use of components or distribution at the design phase. Later, in a configuration phase, the system engineer configures the components and their replication level, allocating them to the adequate nodes in the distributed system.

The hindrance of this approach is that, as the programmer is not aware of distribution and replication issues, complex applications could be built relying heavily in task interaction. This would cause a more inefficient implementation. However, the model for tasks, where task interaction is minimized, precludes such complex applications.

From the application programmer perspective, simple resources (objects) are available for sharing data between tasks and for releasing tasks, which do not implement any distribution or replication mechanisms. The appropriated distributed/replicated resources replace these simple resources in the configuration phase. Also, when replication is considered, resources that provide deterministic execution are needed for communication between tasks without schedule-captured precedence relations.

Note that, in the HRTS scheduling model, remote blocking is avoided by preventing tasks from reading remote data. Hence, when sharing data between tasks configured to reside in different nodes, the Shared Data object must be replicated in these nodes. It is important to guarantee that tasks in different nodes have the same consistent view of the data. This is accomplished by multicasting all data changes to all replicas. This multicasting must guarantee that all replicas receive the same set of data change requests in the same order, thus atomic multicasts must be used.

4.1. Framework Structure

The support to the applications is provided in a two-fold way (Figure 4). First, an underlying software layer (the communication manager; Pinho and Vasques, 2001) provides the appropriate communication algorithms. This layer provides a group communication interface, including the needed communication mechanisms for replication and distribution. It provides atomic multicasts and appropriate algorithms for replicated data consolidation. Nevertheless, group communication technology is by itself very simplistic to support the development of reliable applications, and many extra functionalities must be added to the applications in order to guarantee its reliability requirements (Johnson et al., 2000). Thus, there is the need for an extra abstraction between the application and the group communication mechanisms.

The DEAR-COTS architecture provides such extra abstraction by means of a repository of task interaction objects. These objects provide a transparent interface, by which application tasks are not aware of replication and distribution issues. Their run-time execution is supported by the replica manager, which is a software layer underlying the application. This layer is also responsible for the interface to the communication mechanisms provided by the communication manager. Together, these two software layers constitute the HRTS support software, a run-time middleware layer between the application and the COTS components.

The object repository is used during the design and configuration phase, providing a set of generic objects with different capabilities. These generic objects are instantiated with the appropriate data types and incorporated into the application. They are responsible for hiding from the application the details of the lower-level calls to the support software. This development approach allows application programmers to focus on the requirements of the controlled system rather than on the distribution/replication mechanisms.

The object repository does not use an object-oriented approach, as there is no provision for run-time polymorphism and no code inheritance between objects. However, as an object-based approach, it possesses several advantages:

- It allows objects with the same interface to be replaced in the pre-run-time configuration phase.

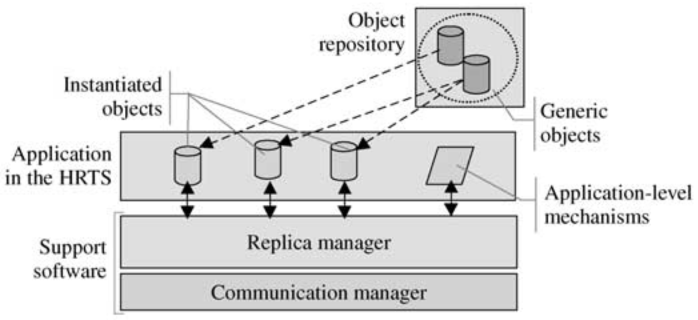

Figure 4. Framework structure.

- It allows the transparent wrapping of replication and distribution mechanisms.
- It eases the modularity and decomposition of the system.

This approach also allows the addition and reuse of new objects. If other generic task interaction objects are later realized to be important, they can be incorporated in the object repository (and thus made available to new applications) as long as they follow the same approach as those currently available. In order to simplify the upgrade of the object repository, the replica manager layer is responsible for the replication mechanisms, while the generic objects in the repository just provide the transparency and the object model to the applications.

The replica manager is responsible for the consistency of the replicated components. It provides the required mechanisms for supporting both the task interaction objects and the applications (recording periodic tasks' release times and allowing applications to change the state of applications components). The replica manager also shields the object repository generic objects from changes in the communication manager structure, either due to network changes, or due to the use of a different group communication middleware.

4.2. Guaranteeing Replica Determinism

As active replication is used, it is essential to guarantee the determinism of the replicated components. The framework uses the timed messages concept (Poledna et al., 2000), with mechanisms implemented both in the generic objects of the object repository and in the replica manager layer.

When there is a synchronous interaction between the releasing and released tasks, timed messages are not required, as it is known that all the replicas of the released task will also be released by the equivalent event in the replicated component. The same happens with asynchronous interactions where tasks have a precedence relation, for example, through offsets.

However, for the case of tasks sharing data with no precedence relation (using a Shared Data object) the timed messages concept must be used, by keeping several versions of the written value, each one associated with the corresponding validity time. When a task reads the value, it must read the most recent value that has a validity time older than the release time of the task. Therefore it is necessary to maintain a record of tasks' release times. For periodic tasks, it is simply necessary to store the task release time, each time it is released. The release time of sporadic tasks released by other tasks is determined by adding the release time of the releasing task with its best-case response time (sporadic tasks released by external events are considered in Sections 5.4 and 5.5).

When distributed computations are involved, it is not possible in the destination node to determine sporadic tasks' release times, or for objects to determine messages' validity times, since these are based on the release time of the source task. Therefore, these must be determined at the source and transmitted together with the actual release event or data message.

5. Object Repository

The object repository provides a set of generic objects (Figure 5), which are instantiated by the application with the appropriated application-related data types. Although multiple objects are available, with different capabilities and goals, the application programmer has only three available object types: Shared Data, Release Event and Release Event with Data objects, without any distribution or replication capabilities (at this stage the system is not yet configured).

These objects have a well-defined interface. Tasks may write and read a Shared Data object and wait in or release a Release Event object. Note that Release Event objects are to be used in a one-way communication, thus a task can only have one of two different roles (wait or release). Release Event and Release Event with Data objects have a similar interface; the only difference is that with the latter it is also possible to transfer data values.

In the system configuration phase, the application will be distributed over the system and some of its components will be replicated. Thus, some (or all) of the used objects be replaced by similar objects with distribution and/or replication capabilities.

5.1. Simple Program Example

In order to exemplify how the task interaction mechanisms can be used, a simple example is presented. Later on, in Section 5.6, the same example is used to demonstrate how applications can be replicated and distributed and how they must be updated at the configuration phase.

```
Shared Data Object

when write (data):
      Obj_Data := data

when read:
      return Obj_Data

Release Event Object

when wait:
      Task_Suspend

when release:
      Suspended_Task_Resume

Release Event with Data Object

when wait:
      Task_Suspend
      return Obj_Data

when release(data):
      Obj_Data := data
      Suspended_Task_Resume
```

Figure 5. Specification of programmer available objects.

Figure 6 presents the structure of the application example. The application is constituted by four tasks, which provide a simple control loop between a sensor and an actuator. The Sensor task is a periodic task, responsible for reading the value of the sensor and passing it to the Controller task, which performs the control algorithm. A periodic Actuator task is then responsible for the actual writing of the output. The Alarm task is responsible for some type of notification if the Controller task signals an abnormal condition. Note that the purpose is to exemplify how the interaction objects can be used and not to indicate how applications should be structured. Therefore, the structuring of tasks as periodic or sporadic is provided for demonstration purposes only.

The Sensor task interacts with the Controller task through a Release Event with Data object, in order to simultaneously release the Controller task and forward it to the device data. The Controller task performs the control algorithm, and then uses a Shared Data object to make the control data available to the Actuator task. If some abnormal situation is detected, the Controller task releases the Alarm task, through the use of a Release Event object.

Figure 7 presents an example of the application program using the Ada 95 language. Lines 1–5 present the declaration of the required repository objects. Note that, at this phase, replication and distribution issues are still not considered. However, due to the necessity of storing the release time of periodic tasks, these must use an available interface to the replica manager (lines 13 and 40) to request its periodic execution (this situation is further detailed in Section 6.3).

For the interaction between the Sensor and Controller tasks, a Release Event with Data object is created (lines 1 and 2) by instantiating a generic package with the appropriated data type and by declaring an instance of the object. The Sensor task uses this object (line 15) to release the Controller task (line 25).

For the interaction between Controller and Actuator tasks a similar approach is used, but with a Shared Data object (created in lines 3 and 4). The Controller task writes in the object (line 27), while the Actuator task performs the related read (line 41).

Release Event objects (without data) are not in generic packages, since they do not carry any data. Therefore, line 7 merely declares the object for the interaction between Controller (release in line 29) and Alarm (wait in line 50) tasks.

Application tasks use these objects, through their well-defined interfaces. The goal is to just change the employed objects at the configuration phase, avoiding any modifications of the tasks.

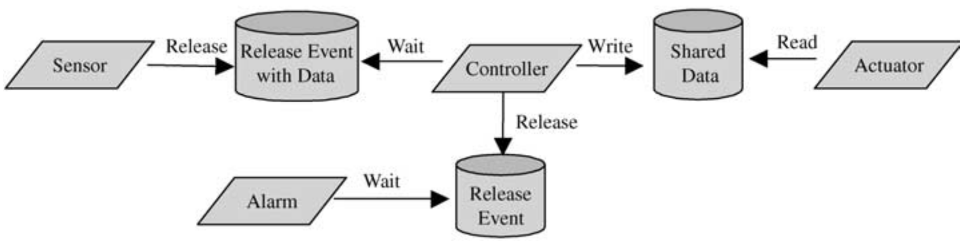

Figure 6. Application example.

```
-- Declaration of object repository objects
package Device_Event is new Object_Repository.Release_Event_With_Data(Device_Data);
Device_Event_Obj: Device_Event.Release_Event_With_Data_Obj;
package Control_Shared_Data is new Object_Repository.Shared_Data(Control_Data);
Control_Data_Obj: Control_Shared_Data.Shared_Data_Obj;
Alarm_Obj: Object_Repository.Release_Event.Release_Event_Obj;

-- Task that reads the sensor
task Sensor;
task body Sensor is
   Start: Ada.Real_Time.Time := ...;
   Period: Ada.Real_Time.Time_Span := ...;
   Dev_Data: Device_Data;
begin
   loop
      Replica_Manager.Request_Periodic(Start);
      Read_Some_Device(Dev_Data);
      Device_Event_Obj.Release(Dev_Data);
      Start := Start + Period;
   end loop;
end Sensor;

-- Task responsible for controlling the system
task Controller;
task body Controller is
   Dev_Data: Device_Data;
   Ctrl_Data: Control_Data;
begin
   loop
      Device_Event_Obj.Wait(Dev_Data);
      Ctrl_Data := Do_Some_Processing(Dev_Data);
      Control_Data_Obj.Write(Ctrl_Data);
      if Some_Test(D) then
         Alarm_Obj.Release;
      end if;

   end loop;
end Controller;

-- Task that performs the output
task Actuator;
task body Actuator is
   Start: Ada.Real_Time.Time := ...;
   Period: Ada.Real_Time.Time_Span := ...;
   Ctrl_Data: Control_Data;
begin
   loop
      Replica_Manager.Request_Periodic(Start);
      Ctrl_Data := Control_Data_Obj.Read;
      Actuate(Ctrl_Data);
      Start := Start + Period;
   end loop;
end Actuator;

-- Task responsible for notification of alarms
task Alarm;
task body Alarm is
begin
   loop
      Alarm_Obj.Wait;
      Some_Notification;
   end loop;
end Alarm;
```

Figure 7. Application program.

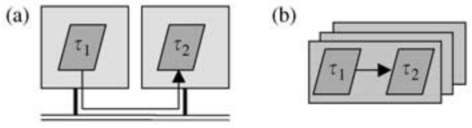

Figure 8. Internal interaction in a (a) distributed component or (b) replicated component.

5.2. Interaction Internal to a Component

The interaction between tasks belonging to the same component (Figure 8) does not require any consolidation between replicas. However, it may require the use of distributed mechanisms (if the component is spread through several nodes) or the use of timed messages (if the component is replicated).

In the case of Release Events objects, there is no need for replica determinism mechanisms (synchronous one-way interaction). However, it is necessary to change the specification for the Release Event objects (Figure 5), as the replica manager must store the release time of the sporadic task in the case of replicated components. Therefore, in the Deterministic Release Event object (Figure 9), the release interface requests the release to the replica manager, and a private_release interface is used by the replica manager to release the task. Note that for the case of a non-replicated component it is not necessary to use this object, since storing the release times is only required for replicated components.

If releasing and released tasks are allocated to different nodes, the simple Release Event object must be replaced by two objects (Figure 10) that co-operate to perform the desired action (proxy model). In the releasing task side, the Release Event Proxy object is responsible for forwarding the event to the remote Release Event Receive object. The

```
Release Event Object

when wait:
     Task_Suspend

when release:
     Replica__Manager.Request_Release_Event

when private_release:
     Suspended_Task_Resume

Release Event with Data Object

when wait:
     Task_Suspend
     return Obj_Data

when release(data):
     Obj_Data := data
     Replica_Manager.Request_Release_Event

when private_release:
     Suspended_Task_Resume
```

Figure 9. Deterministic Release Event objects.

```
Release Event Proxy Object

when release:
     Replica_Manager.Forward_Release_Event

Release Event Receive Object

when wait:
     Task_Suspend

when private_release:
     Suspended_Task_Resume

Release Event with Data Proxy Object

when release(data):
     Replica_Manager.Forward_Release_Event(data)

Release Event with Data Receive Object

when wait:
     Task_Suspend
     return Obj_Data

when private_release(data):
     Obj_Data := data
     Suspended_Task_Resume
```

Figure 10. Distributed Release Event objects.

equivalent pair of objects is also available for the case of the Release Event with Data object. The Proxy object has only the release interface, which requests the replica manager to forward the request to the other node. The Receive object has the corresponding wait interface, and it also provides a private_release to be used by the replica manager when a request arrives. The release time of the task being released is determined at the source by the replica manager (using the available information about the releasing task) and is forwarded to the destination node.

For the case of the Shared Data object, three situations are identified, concerning the need to support replica determinism, to support distribution (when the object is used by tasks in different nodes) or both.

To support replica determinism, the repository provides a generic object, the Deterministic Shared Data object (Figure 11), with the same interface of the Shared Data object, but with extra functionalities related to the support of replica determinism.

This object no longer holds a single data element, but a buffer of elements. Associated with each element, the object also records the related validity time. The write interface requests its validity time from the replica manager layer. The read interface layer requests from the replica manager the release time of the reading task, and chooses from the buffer the newest value that is older than this release time.

To support distribution (when a Shared Data object is used by tasks allocated to different nodes), the object must be replicated in every requesting node (in order to prevent remote reading). Therefore, every write to the object must be atomically multicasted to all image objects. Note that this is a second level replication, which is independent of component replication.

```
when write (data):
     $t_{val}$ := Replica_Manager.Request_Validity_Time(Writing_Task)
     DataBuffer := DataBuffer $\cup$ (data, $t_{val}$)

when read:
     newest_data := null
     $t_{val}$ :=
     $t_{rel}$ := Replica_Manager.Request_Release_Time(Reading_Task)
     for all i in Data_Buffer loop
          if $t_{val}(i) < t_{rel}$ and $t_{val}(i) > t_{val}$ then
               newest_data := Data(i)
          end if
     end loop
     return newest_data
```

Figure 11. Deterministic Shared Data object.

As a consequence, the Distributed Shared Data object (Figure 12) provides a single data element, and the read interface is the same as in the simplest form of the object. However, the write interface is different, as it no longer updates the value in the object. It simply requests the replica manager to disseminate that value, according to the required level of failure assumptions (service requested to the communication manager layer). Additionally, there is a third (private_write) interface, which is used by the replica manager to update the value when it is delivered.

To simultaneously support replication and distribution (when an object requiring deterministic execution is simultaneously used by tasks in different nodes), a Deterministic Distributed Shared Data object (Figure 13) must be used. This object has the buffer and validity times of the Deterministic object, and its write interface must also request the validity time of the value. However, as the Distributed object, it does not add the value to the buffer, but request its dissemination by the replica manager. A third interface (private_write) is also available, enabling the replica manager to update the value when it is delivered by the atomic multicast mechanism. The read interface is similar to the one in the Deterministic object.

5.3. Inter-Group Interaction

When tasks belonging to different components interact (Figure 14), there is the need to consolidate values or events between the component replicas. As a set of replicated

```
when write (data):
     Replica_Manager.Request_Dessimination(Data)

when read:
     return Obj_Data

when private_write (data):
     Obj_Data := data
```

Figure 12. Distributed Shared Data object.

```
when write (data):
    $t_{val}$ := Replica_Manager.Request_Validity_Time(Writing_Task)
    Replica_Manager.Request_Dessimination(Data, $t_{val}$)

when read:
    newest_data := null
    $t_{val}$ := 0
    $t_{rel}$ := Replica_Manager.Request_Release_Time(Reading_Task)
    for all i in Data_Buffer loop
        if $t_{val}(i) < t_{rel}$ and $t_{val}(i) > t_{val}$ then
            newest_data := Data(i)
        end if
    end loop
    return newest_data

when private_write (data, $t_{val}$):
    DataBuffer := DataBuffer $\cup$ (data, $t_{val}$)
```

Figure 13. Deterministic Distributed Shared Data object.

components is defined as a group, these types of interactions are referred as inter-group interactions.

When a group is releasing a task in another group (Figure 15), the replica manager must consolidate both the release proposals and the data values (if any) from the replicas. A similar approach to the Distributed Release Event is used. When a task requests a release in another group, the Inter-Group Release Event Proxy object forwards this request to the replica manager. In the receiving side, the related tasks wait in the corresponding Receive object, which is released by the replica manager.

The replica manager is also responsible for determining the release time of the sporadic task being released. However, as the communication manager guarantees a common delivery time of consolidated values in every node (Pinho and Vasques, 2001), this common time can be considered as the release time of the sporadic task.

However, a special case needs to be considered. If the releasing task is in a non-replicated component, it is not necessary to propose the release, but only to request it. The replica manager is responsible for detecting such cases and for bypassing the regular behavior, in order to optimize the system.

For the case of the Shared Data objects, the Inter-Group Shared Data object (Figure 16) is responsible for transparently managing the consolidation of the value being written, and at the same time, for determining its validity time. This approach is similar to the one used for the Deterministic Distributed object, where the replica manager is just requested to propose a value (and not to disseminate it). Also, as in the case of the Inter-Group Release Event, the common delivery time of the communication manager can be used for the validity time of the data being written.

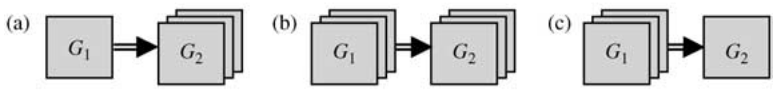

Figure 14. Inter-Group interaction: (a) one-to-many, (b) many-to-many or (c) many-to-one.

```
Inter-Group Release Event Proxy Object

when release:
     Replica_Manager.Propose_Release_Event

Inter-Group Release Event Receive Object

when wait:
     Task_Suspend

when private_release:
     Suspended_Task_Resume

Inter-Group Release Event with Data Proxy Object

when release(data):
     Replica_Manager.Propose_Release_Event(data)

Inter-Group Release Event with Data Receive Object

when wait:
     Task_Suspend
     return Obj_Data

when private_release(data):
     Obj_Data := data
     Suspended_Task_Resume
```

Figure 15. Inter-Group Release Event objects.

Two special cases must be considered for the Inter-Group Shared Data object. First, when the writer task is in a non-replicated component and thus it is not necessary to propose a value, but only to disseminate it. Second, when the reader task is in a non-replicated component and thus it is not necessary to determine a data validity time. Once more, the replica manager is responsible for detecting such cases and for bypassing the regular behavior.

```
Inter-Group Shared Data Object

when write (data):
     Replica_Manager.Propose_Value(Data)

when read:
     newest_data := null
     $t_{val}$ := 0
     $t_{rel}$ := Replica_Manager.Request_Release_Time(Reading_Task)
     for all i in Data_Buffer loop
          if $t_{val}(i) < t_{rel}$ and $t_{val}(i) > t_{val}$ then
               newest_data := Data(i)
          end if
     end loop
     return newest_data

when private_write (data, $t_{val}$):
     DataBuffer := DataBuffer $\cup$ (data, $t_{val}$)
```

Figure 16. Inter-Group Shared Data object.

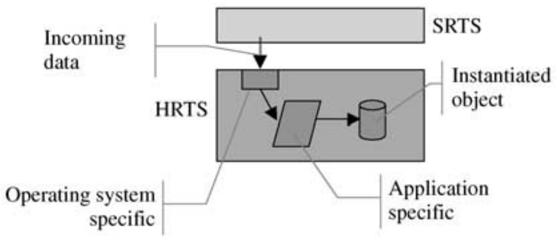

Figure 17. Model for the interconnection with the SRTS.

5.4. *Interaction with the SRTS*

The interconnection between the HRTS and the SRTS must support the information transfer between both subsystems. Data being transferred from the HRTS to the SRTS does not present any major problem, since it is assumed that such information has a higher reliability level, as it is considered that hard real-time applications have been design to achieve higher reliability levels.

Conversely, the reliability of the data arriving from the SRTS may not be high enough to be directly used in the HRTS. If possibly erroneous values are expected, the arriving data must be filtered. As the definition of what it is erroneous is very application specific (since it depends on the semantics and not just on the syntax), a generic mechanism cannot be used. Additionally, if the data is to be provided to replicated components, it must be disseminated using the appropriated mechanisms.

As the communication mechanisms between both subsystems are expected to be dependent from the actual platform (mainly from the operating system), just a generic model for such interconnection is provided. Figure 17 presents such generic model, for the case of data arriving from the SRTS (data going to the SRTS is similar, just flows in the opposite direction). The HRTS application must provide a specific interconnection task, which reads the value from the HRTS/SRTS interface, performs the necessary filtering and interacts with the rest of the system through one of the available objects (it can write to a shared data object, or release a sporadic task).

Note that, if a higher reliability level is to be achieved, the interconnection with the SRTS must also be replicated. Therefore, in order to achieve deterministic execution, every external interaction with the system must have a common time reference. As a consequence, the interface task needs to be a component by itself, in order to interact with the remaining system with consolidated values and times (data validity time and/or sporadic task release time).

5.5. *Interaction with the Controlled System*

The interconnection with the controlled system (performed through the use of sensors and actuators) is application and platform specific. Therefore, its model is similar to the interconnection with the SRTS, where application tasks are responsible for such interconnection and for inserting their values (or events) in the system.

Note that output actuator agreement may be made either in the computational system or by mechanical or electronic voting on the result. In the first case, it means that there is a single task responsible for the interconnection with a single actuator. Thus, the system relies on the reliability of both the task (and the node where it is allocated) and the actuator. Although the architecture itself does not forbid such configuration, it is however considered that it implies assumptions not provided by COTS components. Therefore, it is advocated that voting outside the computational system provides much better coverage of the COTS failure assumptions (fail-uncontrolled). The way that such agreement is made outside the computational system is, however, outside of the scope of the generic architecture.

5.6. *Configured Application Example*

Section 5.1 presented a simple application to exemplify how the object repository could be used during the programming phase. In this subsection the same example is re-visited, in order to exemplify how distributed/replicated applications are modified during the configuration phase.

Figure 18 presents such configuration phase. As noted in subsection 5.5, as tasks Sensor and Actuator interact with the controlled system, they must be configured as components. Therefore, component C_1 is constituted by task Sensor (τ_1) and component C_3 by task Actuator (τ_4). Component C_2 encompasses tasks Controller (τ_2) and Alarm (τ_3). For the purpose of this example, a simplified replication model is considered. Obviously, in order to tolerate the fail-uncontrolled behavior of the replicas, it would be necessary to use $2 * f + 1$ replicas to tolerate f faults.

Figure 19 presents the configuration of the application over the HRTS nodes. Node 1 is configured with components C_1 and C_2, while node 2 is configured with a replica of

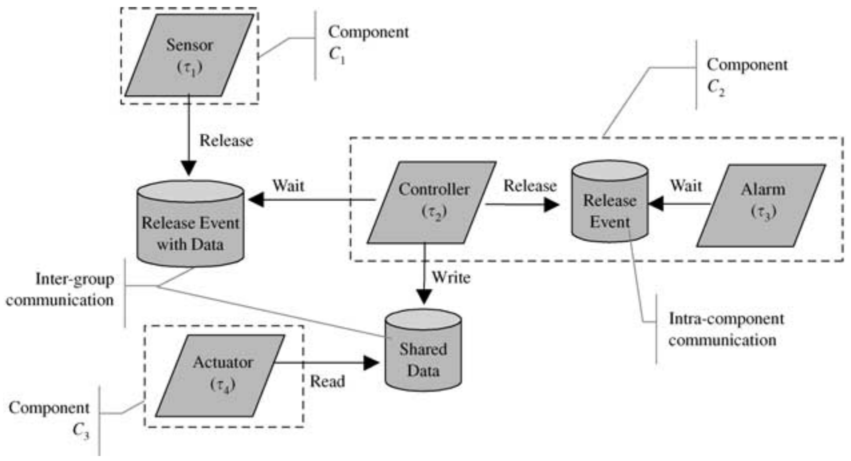

Figure 18. Application configuration.

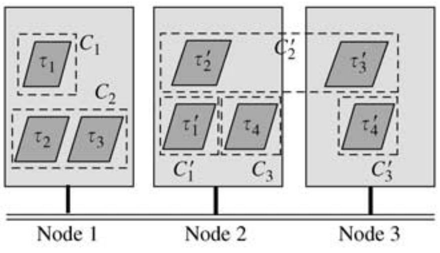

Figure 19. Node configuration.

component C_1, with component C_3 and with a task of a replica of component C_2. Finally, node 3 is configured with the other task of the replica of component C_2, and with a replica of component C_3.

Figure 20 presents the program configured to execute in node 1. The Release Event with Data object of Figure 7 (Section 5.1) is replaced by its inter-group equivalent (lines 1 and 2), since tasks Sensor and Controller are in different components. The same occurs with the Shared Data object between Controller and Actuator tasks. The Release Event object used for the interaction between Controller and Alarm tasks is replaced by a Deterministic Release Event object, since it is related to an intra-component interaction and, as it is replicated, the release time of the Alarm task must also be recorded. Note that application tasks are not changed since the new objects have the same interface as the ones in Section 5.1. The program in this node does not provide any actuator task, since no replica of component C_3 is allocated to the node.

Figure 21 presents the program for node 2. In this node, there is no Alarm task, and the objects used by the application are the same as those in Node 1. In Figure 22 (program in node 3), there is only the Actuator and Alarm tasks, therefore it is not necessary to create any object responsible for the interaction between Sensor and Controller tasks. Note that, as the replica of component C_2 is spread between nodes 2 and 3, in the Controller task side a Distributed Release Event Proxy object is used, while its counterpart Receive object is used in the Alarm task side.

```
    -- Declaration of object repository objects
1:  package Device_Event is new Object_Repository.Inter_Group.Release_Event_With_Data(
                                                  Device_Data);
2:  Device_Event_Obj: Device_Event.Release_Event_With_Data_Obj;
3:  package Control_Shared_Data is new Object_Repository.Inter_Group.Shared_Data(
                                                  Control_Data);
4:  Control_Data_Obj: Control_Shared_Data.Shared_Data_Obj;
5:  Alarm_Obj: Object_Repository.Deterministic_Release_Event.Release_Event_Obj;

6:  task Sensor;        -- no changes

19: task Controller;    -- no changes

    -- no Task Actuator

46: task Alarm;         -- no changes
```

Figure 20. Node 1 program.

```
     -- Declaration of object repository objects
1:   package Device_Event is new Object_Repository.Inter_Group.Release_Event_With_Data(
                                                      Device_Data);
2:   Device_Event_Obj: Device_Event.Release_Event_With_Data_Obj;
3:   package Control_Shared_Data is new Object_Repository.Inter_Group.Shared_Data(
                                                      Control_Data);
4:   Control_Data_Obj: Control_Shared_Data.Shared_Data_Obj;
5:   Alarm_Obj: Object_Repository.Distributed_Release_Event.Proxy_Release_Event_Obj;

6:   task Sensor;        -- no changes

19:  task Controller;    -- no changes

35:  task Actuator;      -- no changes

     -- no Task Alarm
```

Figure 21. Node 2 program.

```
     -- Declaration of object repository objects
1:   package Control_Shared_Data is new Object_Repository.Inter_Group.Shared_Data(
                                                      Control_Data);
2:   Control_Data_Obj: Control_Shared_Data.Shared_Data_Obj;
3:   Alarm_Obj: Object_Repository.Distributed_Release_Event.Receive_Release_Event_Obj;

     -- no Task Sensor
     -- no Task Controller

35:  task Actuator;      -- no changes

46:  task Alarm;         -- no changes
```

Figure 22. Node 3 program.

6. HRTS Replica Manager

The replica manager layer (Figure 23) is intended to support the proposed resources and to guarantee their correct behavior.

The property recorder module is the database of the replica manager. It records both the structure and information of tasks and components. Application objects may use this

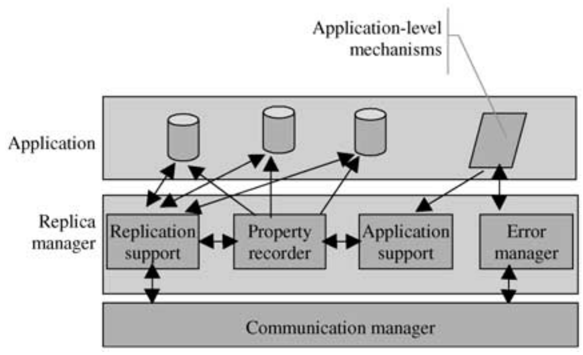

Figure 23. Replica manager structure.

module to query and/or change the release times of tasks, when a deterministic behavior is required. This module is also used both by the replication and application support modules, to query and record tasks' release times and application configuration information.

The replication support module provides the required mechanisms for supporting replicated task interaction. It also interfaces the repository objects with the communication subsystem, thus it also receives or sends data values and events both from the objects and from the communication manager.

The application support module takes care of the recording of periodic tasks' release times, allowing components from applications executing in the HRTS to be shutdown, silenced or activated.

The error manager module keeps a record of detected node errors (and also of system errors, if configured to disseminate error detection). This module also provides mechanisms for notification of errors, which can be used by the error recovery procedures.

6.1. Property Recorder Module

The property recorder module is responsible for storing all the information (Table 1) required to guarantee the correct behavior of the replication/distribution framework. Two different categories of information are stored. The application configuration information, which is off-line knowledge, and is constituted by the component structure, task information and network information.

The application execution information cannot be off-line defined, since it is constituted by the tasks' release times and the state of the applications components (active, silenced, shutdown). The property recorder at each node only stores the information related to the components and tasks that execute in the node. The replication support module is responsible for, when necessary, forwarding such local information to the related nodes.

Table 2 presents the interface of the property recorder. This interface is used both by the repository objects (Section 5) and by the replication and application support modules (modules of the replica manager). Note that there are two different interfaces to record the release time of a task. The first one is to be used when both released and releasing tasks are in the same node, thus the property recorder has the necessary information to

Table 1. Application information.

Application configuration	
Component structure:	Task and object identifiers of each component
Task information:	Worst- and best-case execution time (may include internal computations)
Network information:	Message streams worst-case delivery time
Application execution	
Component structure:	Component state
Task information:	Release time of tasks

Table 2. Property recorder interface.

Application configuration
Query_Released_Task(Obj_Id)
Query_Component_Replication_Degree(Comp_Id)
Query_Component_Id (Task_Id)
Query_Component_Id (Obj_Id)
Query_Group(Obj_Id)
Query_Source_Group(Obj_Id)
Query_Dest_Groups(Obj_Id)
Application execution
Query_Message_Validity_Time(Task,Msg)
Record_Task_Release_Time(Task_Id)
Record_Task_Release_Time(Task_Id, Time)
Query_Task_Release_Time(Task_Id)
Query_Component_State(Comp_Id)
Record_Component_State(Comp_Id, State)

determine the release time (release time and best-case execution time of the releasing task). However, when distributed release events are used, this information is only available at the source node, in spite of the release time being required at the destination node. Therefore, Query_Task_Release_Time is used at the source node to determine the release time of the task. Such release time is then sent through the network, in order to be stored at the destination node, using the second Record_Task_Release_Time interface. Moreover, for the case of activating a component, the application support module needs also to record the release time of the periodic tasks, thus it also uses the second interface.

6.2. Replication Support Module

The replication support module is the core of the replica manager layer. This module is responsible for all the used replication mechanisms, and also for the interconnection with the communication subsystem through the interface provided by the communication manager. This latter interface allows the atomic multicast of messages to destination groups, the notification of message reception, and also the atomic multicasting of messages within the same group. Using a group communication interface to the communication subsystem, allows an easier management of component replication, and also simplifies the necessary adaptations, if the communication subsystem is changed.

This group communication interface is specified in terms of Groups and Group Objects. Several different calls can be made to the same group, referring to different interaction objects. Therefore, this interface specifies which group is being addressed and, inside the group, which object is being addressed. Table 3 presents the syntax of the calls that can be made to the interface, and also the syntax of the necessary handler to receive messages.

Table 3. Communication manager interface.

To the communication manager
Multicast(Message, Sender_Group_Id, Receiver_Group_Array, Receiver_Obj_Id)
Consolidated_Multicast(Message,Sender_Group_Id,Receiver_Group_Array,Receiver_Obj_Id)
From the communication manager
Receive(Message, From_Group, To_Group, To_Obj, Deliver_Time)

The Multicast call allows the atomic multicasting of a message to a set of groups, without requiring any type of consolidation from the communication system. The Consolidated_Multicast call is to be used when consolidation is required between the replicas of the sending component. Messages between distributed elements within the same replica can use the Multicast call, using the same group for Sender and Receiver groups. Note that using a Multicast call the same message can be sent or proposed to a set of groups. This is necessary, since the same interaction object can be used by a set of components, thus, when replication is considered, by a set of groups. It is expected that most of the time the writing group will also be one of the receiving groups, if the same object is used for reading and writing in a component. If it is to be sent or proposed to a single group, then a Receiver_Group_Array with a cardinality of one can be used.

The replication support module is responsible for performing the main processing of the replication mechanisms provided to the objects in the repository. When a release event is requested (Figure 24), it is necessary to record the release time of the released task. When a release event is to be forwarded (Figure 24), it is necessary to obtain from the property recorder the release time of the sporadic task being released, as such value must be sent together with the event message.

When data is to be disseminated inside a component (Figure 25), its validity time must also be disseminated if the source component is replicated (Figure 25, lines 5–8). To propose release events or data values (Figures 26 and 27), it is necessary to query the property recorder about the source and destination group identifiers. Moreover, if the

```
when Request_Release_Event
     Task_Id := Property_Recorder.Query_Released_Task(Obj_Id)
     Property_Recorder.Record_Task_Release_Time(Task_Id)
     Object(Obj_Id).private_release

when Forward_Release_Event
     Task_Id := Property_Recorder.Query_Released_Task(Obj_Id)
     Message := Msg_Type(Event) $\cup$
                Property_Recorder.Query_Task_Release_Time(Task_Id)
     This_Group := Property_Recorder.Query_Group(Obj_Id)
     Communication_Manager.Multicast(Message,This_Group,This_Group,Obj_Id)

when Forward_Release_Event(data)
     Task_Id := Property_Recorder.Query_Released_Task(Obj_Id)
     Message := Msg_Type(Event) $\cup$ data $\cup$
                Property_Recorder.Query_Task_Release_Time(Task_Id)
     This_Group := Property_Recorder.Query_Group(Obj_Id)
     Communication_Manager. Multicast(Message,This_Group,This_Group,Obj_Id)
```

Figure 24. Support for Intra-Component Release Events.

```
when Request_Dessimination(Data)
    Message := Msg_Type(Data) ∪ Data
    This_Group := Property_Recorder.Query_Group(Obj_Id)
    Communication_Manager.Multicast(Message,This_Group,This_Group,Obj_Id)

when Request_Dessimination(Data, $t_{val}$)
    Message := Msg_Type(Data_and_Validity) ∪ Data ∪ $t_{val}$
    This_Group := Property_Recorder.Query_Group(Obj_Id)
    Communication_Manager.Multicast(Message,This_Group,This_Group,Obj_Id)
```

Figure 25. Support for Intra-Component Data Dissemination.

source component is replicated, it is necessary to use the Consolidated_Multicast interface to consolidate the outputs from the different replicas.

Note that both the support for inter-group release events and data dissemination query the property recorder module, in order to avoid communication if the component is silenced.

The replica manager, in order to be notified of a message reception and its associated delivery (or consolidation) time, must use the Receive handler (Figure 28). The communication manager algorithms guarantee that this timestamp is the same in all receiving nodes (Pinho and Vasques, 2001). Note that, although the message may have been sent (or proposed) to a set of groups, the Receive handler receives messages to a

```
when Propose_Release_Event
    comp := Property_Recorder.Query_Component_Id(Obj_Id)
    if Property_Recorder.Query_Component_State(comp) not Silenced then
        Message := Msg_Type(Event)
        Source_Group := Property_Recorder.Query_Source_Group(Obj_Id)
        Dest_Groups := Property_Recorder.Query_Dest_Groups(Obj_Id)
        degree :=
            Property_Recorder.Query_Component_Replication_Degree(comp)
        if degree = 1 then
            Communication_Manager.Multicast(Message, Source_Group,
                                            Dest_Groups, Obj_Id)
        else
            Communication_Manager.Consolidated_Multicast(Message,
                                 Source_Group, Dest_Groups, Obj_Id)
        end if
    end if

when Propose_Release_Event(data)
    comp := Property_Recorder.Query_Component_Id(Obj_Id)
    if Property_Recorder.Query_Component_State(comp) not Silenced then
        Message := Msg_Type(Event_Data) ∪ data
        Source_Group := Property_Recorder.Query_Source_Group(Obj_Id)
        Dest_Groups := Property_Recorder.Query_Dest_Groups(Obj_Id)
        degree :=
            Property_Recorder.Query_Component_Replication_Degree(comp)
        if degree = 1 then
            Communication_Manager.Multicast(Message, Source_Group,
                                            Dest_Groups, Obj_Id)
        else
            Communication_Manager.Consolidated_Multicast(Message,
                                 Source_Group, Dest_Groups, Obj_Id)
        end if
    end if
```

Figure 26. Support for Inter-Group Release Events.

```
when Propose_Value(data)
    comp := Property_Recorder.Query_Component_Id(Obj_Id)
    if Property_Recorder.Query_Component_State(comp) not Silenced then
        Message := Msg_Type(Data) ∪ data
        Source_Group := Property_Recorder.Query_Source_Group(Obj_Id)
        Dest_Groups := Property_Recorder.Query_Dest_Groups(Obj_Id)
        degree :=
            Property_Recorder.Query_Component_Replication_Degree(comp)
        if degree = 1 then
            Communication_Manager.Multicast(Message, Source_Group,
                                            Dest_Groups, Obj_Id)
        else
            Communication_Manager.Consolidated_Multicast(Message,
                                    Source_Group, Dest_Groups, Obj_Id)
        end if
    end if
```

Figure 27. Support for Inter-Group Data Dissemination.

single group. Messages sent to a set of groups have already been consolidated (if needed) and are individually delivered to each group by the communication manager.

This Receive handler is also used to prevent sporadic tasks from being released, if the related component has been shutdown. Note that it is not necessary to prevent the release of sporadic tasks internal to a component, since, by preventing the release of both sporadic tasks released by other components and periodic tasks within the component, internal sporadic tasks will not be released.

6.3. Application Support Module

The application support module provides support to applications executing in the HRTS. First, as periodic tasks are not released by the support software, but by the operating system, there is the need for an appropriate interface. Also, the support software needs both to record the release time of periodic tasks and to prevent periodic tasks from being released if the related component has been shutdown. A solution to this problem is to restrict the programmer to use a support software interface to the operating system, in order to request its next release (Figure 29).

This interface records the release time of the task, being the task scheduled for execution only if the component has not been shutdown. If the component has been shutdown, the task is suspended and will not be released again. Only after restarting the component, the task will be re-activated, the property recorder will be queried about its new release time, and the task is re-scheduled. Note that the programmer must be aware that next_release_time may have changed since the last execution of the task.

Finally, there is the possibility to reconfigure the system, either due to mode changes, or due to error recovery situations (Figure 30). The reconfiguration support allows components to be shutdown, silenced and/or activated. In this last case, it is necessary to provide a new release time for the component tasks, which is stored in the property recorder before restoring the suspended tasks. The replication support module uses this configuration information to prevent the release of the components' tasks or the interaction with other components.

```
when Receive(Message, From_Group, To_Group, Obj_Id, $t_{deliver}$)
    if From_Group = To_Group then -- internal to a group
        case Msg_Type(Message) is
            Event =>
                Task_Id := Property_Recorder.Query_Released_Task(Obj_Id)
                Property_Recorder.Record_Task_Release_Time(Task_Id,
                                           Released_Time(Message))
                Object(Obj_Id).private_release
            Event_Data =>
                Task_Id := Property_Recorder.Query_Released_Task(Obj_Id)
                Property_Recorder.Record_Task_Release_Time(Task_Id,
                                           Released_Time(Message))
                Object(Obj_Id).private_release(Data(Message))
            Data =>
                Object(Obj_Id).private_write(Data(Message))
            Data_and_Validity =>
                    Object(Obj_Id).private_write(Data(Message),
                                           Validity_Time(Message))
        end case
    else
        case Msg_Type(Message) is
            Event =>
                Task_Id := Property_Recorder.Query_Released_Task(Obj_Id)
                Property_Recorder.Record_Task_Release_Time(Task_Id,
                                                             $t_{deliver}$)
                comp := Property_Recorder.Query_Component_Id(Released_Task)
                if Property_Recorder.Query_Component_State(comp) not Shutdown then
                    Object(Obj_Id).private_release
                end if
            Event_Data =>
                Task_Id := Property_Recorder.Query_Released_Task(Obj_Id)
                Property_Recorder.Record_Task_Release_Time(Task_Id,
                                                             $t_{deliver}$)
                comp := Property_Recorder.Query_Component_Id(Released_Task)
                if Property_Recorder.Query_Component_State(comp) not Shutdown then
                    Object(Obj_Id).private_release(Data(Message))
                end if
            Data =>
                comp :=
                    Property_Recorder.Query_Component (Obj_Id)
                degree :=
                    Property_Recorder.Query_Component_Replication_Degree(comp)
                if degree = 1 then
                    Object(Obj_Id).private_write(Data(Message)
                else
                    Object(Obj_Id).private_write(Data(Message),
                                 Validity_Time(Message))
                end if
        end case
    end if
```

Figure 28. Receive handler specification.

6.4. Error Manager Module

The goal of the error manager module is to record (and possibly notify) errors occurring in the communication and consolidation of values. It also provides mechanisms by which error recovery procedures can be notified, in order to take actions to shutdown or silence components. Table 4 presents the interfaces provided by such module.

As the communication manager is responsible for reliable and atomic multicasts and also for the consolidation of replicated components' outputs, it is up to this layer to detect

```
when Request_Periodic(next_release_time)
    Property_Recorder.Record_Task_Release_Time(This_Task,
                                           next_release_time)
    comp := Property_Recorder.Query_Component_Id(This_Task)
    if Property_Recorder.Query_Component_State(comp) not Shutdown then
        OS_Specific_Release(This_Task,next_release_time)
    else
        Task_Suspend
        next_release_time :=
                Property_Recorder.Query_Task_Release_Time(This_Task)
        OS_Specific_Release(This_Task,next_release_time)
    end if
```

Figure 29. Periodic task support.

```
when Shutdown_Component(Comp_Id)
    Property_Recorder.Record_Component_State(Comp_Id, Shutdown)

when Silence_Component(Comp_Id)
    Property_Recorder.Record_Component_State(Comp_Id, Silenced)

when Start_Component(Comp_Id, Tasks_Release_Times[Task_Id, Release_time])
    for all Task_Id in Tasks_Release_Times loop
        Property_Recorder.Record_Task_Release_Time(Task_Id, Release_Time)
    end loop
    Property_Recorder.Record_Component_State(Comp_Id, Active)
```

Figure 30. Reconfiguration support.

possible errors in the system. It notifies the error manager module, which, in addition to storing the error information, may execute two (non-exclusive) actions: it can notify an application-level error recovery module; or it can disseminate this error detection throughout the system. This error information is related to the detectable errors in the communication system and in the consolidation of replicated outputs.

In the consolidation phase, component errors may be detected either if the component has not proposed a value (omission fault), or the proposed value was rejected by the underlying decide function (application-specific function, which is part of the communication manager Consolidate (Pinho and Vasques, 2001) module). Consolidation-related errors may occur when the Consolidate module is unable to consolidate values, due to the violation of the failure assumptions.

Communication-related errors occur when an error is detected in the network. Currently, as the communication manager is targeted to the controller area network (CAN) (ISO, 1993), the possible errors in the network are inconsistent message duplicates

Table 4. Error manager interface.

From the communication manager
Record_Error(Error_Information)
To the error recovery module
Query_Error_Information
Error_Notification

and omissions (Rufino et al., 1998). Although these are tolerated by the provided atomic multicast mechanisms (Pinho and Vasques, 2001), it indicates the existence of network problems that may bring down the system. Moreover, the occurrence of inconsistent message omissions indicates that there was, at least, one node crash.

In addition to recording errors, the module can disseminate the error detection throughout the system, requesting from the communication manager a specific message to be sent to the other error managers in the system. The module may also interact with an application or system error recovery module to notify the error detection. This can be performed either by periodically querying the error manager module, or by requesting the notification of a sporadic task.

7. Implementation Considerations

A prototype of the replication management framework has been implemented (Pinho, 2001) to assess the applicability of the Ada 95 programming language, and particularly of its Ravenscar profile (Burns, 1997), for the development of the proposed replication/distribution mechanisms.

The Ravenscar profile defines a subset of the Ada 95 language's multitasking mechanisms, which is considered suitable for the development of efficient and deterministic real-time applications. It allows the consideration of multitasking pre-emptive applications for the development of certifiable real-time systems. Nevertheless, this profile presents some particular restrictions, which may induce a greater complexity and resource usage in Ravenscar compliant applications.

The genericity and transparency requirements of the replication management framework were addressed through the use of Ada generic packages. The use of generics allows reusing the same implementation mechanisms for objects with different data types, being object parameterization performed at compile time. Furthermore, these mechanisms are hidden from the application programmer, which only accesses the objects through their clearly defined public interfaces. Since the goal of transparency is to allow interaction objects to be replaced in the configuration phase by the objects providing replication and distribution support, the public interface for similar interaction objects is the same.

From the prototype implementation it is concluded that, although the increase of complexity and resource usage imposed by some of the Ravenscar restrictions, in general this profile provides a suitable approach for the development of reliable real-time applications.

It is also clear that the proposed transparent and generic approach introduces an additional level of complexity and resource usage in the development of the replication management framework. Nevertheless, a positive trade-off is obtained, since a clearer and simpler programming model is provided to the application development. The framework implementation is not application-specific, thus itself can be used as an off-the-shelf component to support a wide range of DCCS. By not being application-specific, it is then possible to focus on the framework implementation, considering the use of software-based fault tolerance approaches (such as diversity or validation) in order to increase the reliability level of the system.

8. Conclusions

The DEAR-COTS architecture has been previously proposed as a COTS-based framework for the development of DCCS. It supports the execution of reliable hard real-time applications, allowing, at the same time, soft real-time or non-real-time applications to be executed, without interfering with the guarantees provided to hard real-time applications.

This paper presents the replication management framework provided by the DEAR-COTS architecture, for the support of replicated software components. This framework relies on a repository of generic task interaction objects that are used during the design and configuration phases of the development of reliable hard real-time applications. This repository provides a set of generic objects, which are responsible for hiding from the application the low-level mechanisms for replication and distribution support. This allows applications to focus on the requirements of the controlled system rather than on the distribution/replication details. The provided set of generic objects is supported by a middleware layer, which both shields the repository from changes in the lower-level communication mechanisms, and minimizes the effort necessary for the creation of new task interaction generic objects.

Acknowledgments

This work was partially supported by FCT (projects DEAR-COTS 14187/98 and TERRA 38932/01) and by IDMEC.

Notes

1. Note that this is not the usual meaning of validity. In this case it is a ''not to use before'' rather than a ''not to use after''.

References

Audsley, A. N., Burns, A., Richardson, M., Tindell, K., and Wellings, A. 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal* 8(5): 285–292.

Barrett, P. A., Burns, A., and Wellings, A. J. 1995. Models of replication for safety critical hard real-time systems. In *Proceedings of the 20th IFAC/IFIP Workshop on Real-Time Programming*. USA, pp. 181–188.

Bondavalli, A., Giandomenico, F. D., Grandoni, F., Powell, D., and Rabejac, C. 1998. State restoration in a COTS-based N-modular architecture. In *Proceedings of the First International Symposium in Object-Oriented Real-Time Distributed Computing*. Japan, pp. 174–183.

Burns, A. 1997. Session summary: Tasking profiles. In *Proceedings of the 8th International Real-Time Ada Workshop*, *Ravenscar, England*. Ada Letters, XVII(5): 5–7. ACM Press.

ISO 11898. 1993. Road Vehicle—Interchange of Digital Information—Controller Area Network (CAN) for High-Speed Communication, ISO.

ISO/IEC 8652. 1995. Information technology—Programming Languages—Ada. *Ada Reference Manual*, ISO/IEC.

Johnson, S., Jahanian, F., Ghosh, S., VanVoorst, B., and Weininger, N. 2000. Experiences with group communication middleware. In *Proceedings of the International Conference on Dependable Systems and Networks*. New York City, USA, pp. 37–42.

Keickhafer, R. M., Walter, C. J., Finn, A. M., and Thambidurai, P. M. 1988. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers* 37(4): 398–404.

Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., and Zainlinger, R. 1989. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro* 9(1): 25–41.

Laprie, J. L. 1992. *Dependability: Basic Concepts and Terminology. Dependable Computing and Fault-Tolerant Systems*, Vol. 5. Berlin: Springer Verlag.

Melliar-Smith, P. M., and Schwartz, R. L. 1982. Formal specification and mechanical verification of sift: A fault–tolerance flight control system. *IEEE Transactions on Computers* 31(7): 616–630.

Pinho, L. 2001. A framework for the transparent replication of real-time applications. Ph.D. thesis. School of Engineering of the University of Porto, Portugal. Available at http://www.hurray.isep.ipp.pt

Pinho, L., and Vasques, F. 2001. Timing analysis of reliable real-time communication in CAN networks. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*. Delft, The Netherlands, pp. 103–112.

Poledna, S. 1994. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems* 6(3): 289–316.

Poledna, S. 1998. Deterministic operation of dissimilar replicated task sets in fault-tolerant distributed real-time systems. In *Proceedings of the dependable computing for critical applications 6*. Grainau, Germany, pp. 103–119.

Poledna, S., Burns, A., Wellings, A., and Barret, P. 2000. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Transactions on Computers* 49(2): 100–111.

Powell, D. (ed.). 1991. *Delta-4—A Generic Architecture for Dependable Distributed Computing*. ESPRIT Research Reports. Berlin: Springer Verlag.

Powell, D. 1994. Distributed fault tolerance—lessons learnt from Delta-4. Hardware and software architectures for fault tolerance. In Banatre, M., and Lee P. A. (eds.), *Experiences and Perspectives*. Lecture Notes in Computer Science 774. Berlin: Springer Verlag, pp. 199–217.

Powell, D. (ed.) 2001. *A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems*. Dordrecht: Kluwer Academic Publishers.

Pradhan, D. K. 1996. *fault-tolerant Computer System Design*. Prentice Hall.

Rufino, J., Veríssimo, P., Arroz, G., Almeida, C., and Rodrigues, L. 1998. Fault-tolerant broadcasts in CAN. In *Proceedings of the Symposium on Fault-Tolerant Computing*. Munich, Germany, pp. 150–159.

Rushby, J. 1996. Reconfiguration and transient recovery in state machines architectures. In *Proceedings of the 26th Symposium on Fault-Tolerant Computing*. Sendai, Japan, pp. 6–15.

Schneider, F. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4): 299–319.

Veríssimo, P., Casimiro, A., and Fetzer, C. 2000. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*. New York City, USA, pp. 533–542.

Veríssimo, P., Casimiro, A., Pinho, L. M., Vasques, F., Rodrigues, L., and Tovar, E. 2000. Distributed computer-controlled systems: The DEAR-COTS approach. In *Proceedings of the 16th IFAC Workshop on Distributed Computer Control Systems*. Sydney, Australia, pp. 128–135.

Wellings, A. J., Beus-Dukic, Lj., and Powell, D. 1998. Real-time scheduling in a generic fault-tolerant architecture. In *Proceedings of the IEEE Real-Time Systems Symposium*. Madrid, Spain, pp. 390–398.

Yeh, Y. 1995. Dependability of the 777 primary flight control system. In *Proceedings of the Dependable Computing for Critical Applications 5*. Urbana-Champaign, USA, pp. 1–13.

Luís Miguel Pinho received his B.Sc., M.Sc. and Ph.D. degrees in Electrical and Computer Engineering from the University of Porto, Portugal, in 1994, 1997 and 2001, respectively. Since 1996 he has been a teaching assistant in the Department of Computer Engineering, School of Engineering of the Polytechnic Institute of Porto. His main research interests include fault tolerant real-time systems, real-time system architectures and real-time programming languages. He is particularly interested in the analysis, design and implementation of programming mechanisms for the implementation of fault-tolerant real-time distributed systems.

Francisco Vasques received his B.Sc. degree in Electrical Engineering from the University of Porto, Portugal, in 1987 and M.Sc. and Ph.D. degrees in Computer Science from LAAS-CNRS, Toulouse, France, in 1992 and 1996. Currently, he is Assistant Professor in the Department of Mechanical Engineering at the University of Porto. He is also responsible for the Real-Time Systems course within the Electrical and Computer Engineering M.Sc. degree. His research interests include real-time communication systems, real-time factory communications, fault tolerant systems and real-time system architectures. Since 1991 he has authored and co-authored more than 40 technical papers in the area of real-time systems.

Andy Wellings is Professor of Real-Time Systems at the University of York, UK, in the Computer Science Department. He is interested in most aspects of the design and implementation of real-time dependable computer systems and, in particular, real-time programming languages and operating systems. He is European Editor-in-Chief for the Computer Science journal *Software-Practice and Experience* and a member of the International Expert Group

currently developing extensions to the Java Language for distributed real-time programming. He has authored/co-authored over 150 papers/reports. He is also the author of several books including: *Hard Real-Time HOOD: A Structured Design Method for Hard Real-Time Ada Systems*, *Concurrency in Ada* (2nd edn) and *Real-Time Systems and Programming Languages* (3rd edn).