Transparent Environment for Replicated Ravenscar Applications

Luís Miguel Pinho[1], Francisco Vasques[2]

[1] Department of Computer Engineering, ISEP, Polytechnic Institute of Porto,
Rua Dr. António Bernardino Almeida, 431, 4200-072 Porto, Portugal
lpinho@dei.isep.ipp.pt
[2] Department of Mechanical Engineering, FEUP, University of Porto,
Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
vasques@fe.up.pt

Abstract. This paper proposes an environment intended for the development of fault-tolerant real-time Ada 95 applications conforming to the Ravenscar profile. This environment is based on the transparent replication of application components, and it provides a set of generic task interaction objects, which are used as the basic building blocks of the supported applications. These objects provide the usual task interaction mechanisms used in hard real-time applications, and allow applications to be developed without considering replication and distribution issues.

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 pre-emptive multitasking applications introduce additional problems.

Using COTS components as the systems' building blocks provides a cost-effective solution, and at the same time allows for an easy upgrade and maintenance of the system. However, the use of COTS implies that specialised hardware can not be used to guarantee the reliability requirements. As COTS hardware and software do not usually provide the confidence level required by reliable real-time applications, reliability requirements must be guaranteed by a software-based fault-tolerance approach.

Traditionally, the development of computer control applications has been based on a non-preemptive computational model, where a scheduling table comprising the sequence of task executions is determined off-line (the cyclic-executive model). Therefore, the programming model for the development of these applications is simple, not requiring complex task interaction mechanisms, because it fixes the sequence of tasks' executions. On the other hand, by using the pre-emptive fixed priority model of computation, the system flexibility is increased, and the design effort is decreased. However, mechanisms for task interaction become more complex, since there is no fixed pattern for the sequence of tasks' executions.

Therefore, it is important to provide a generic and transparent programming model for the development of pre-emptive computer control applications. The goal is to decrease the complexity of application development, by precluding the need for the simultaneous consideration of system requirements and interaction between multitasking and replication/distribution. A set of generic mechanisms must be provided, which can be parameterised with both application-specific data and application-specific configuration (distribution and replication). These mechanisms will be the basic building blocks of distributed/replicated real-time applications, providing a higher level of abstraction to developers and maximising the capability to reuse components and mechanisms in different applications.

The Ada Ravenscar profile [1] allows multitasking pre-emptive applications to be employed for the development of fault-tolerant real-time systems, whilst providing efficient and deterministic applications. Nevertheless, it is considered that further studies are necessary for its use in replicated and distributed systems [2]. The interaction between multitasking pre-emptive software and replication introduces new problems that must be considered, particularly for the case of a transparent and generic approach.

Furthermore, the restrictions of the Ravenscar profile complicate the implementation of an efficient support for replicated or distributed programming, which may result on an increased application complexity [3]. Therefore, any environment for replication support using the Ravenscar profile must be simple to implement and use, but at the same time must provide the capabilities required by the fault-tolerant real-time applications.

This paper proposes an environment to transparently support replicated and distributed Ravenscar applications. The following section presents work related to the support to fault-tolerant real-time applications. Afterwards, Section 3 presents the system model considered for the environment. Section 4 presents the framework for replication management intended for the support to the system applications, while the proposed set of task interaction objects is presented in Section 5.

2 Related Work

Fundamental for a COTS-based fault-tolerant architecture is the issue of software-based fault tolerance. Since there is no specialised hardware with self-checking properties, it is up to the software to manage replication and fault tolerance. Three main replication approaches are addressed in the literature: active replication, primary-backup (passive) replication and semi-active replication [4]. When fail-uncontrolled (arbitrary failures) components are considered, incorrect service delivery can only be detected by active replication, because it is required that all replicas output some value, in order to perform some form of voting [4]. The use of COTS components generally implies fail-uncontrolled replicas, so it becomes necessary 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 process the same inputs in

different order, providing inconsistent results if inputs are non-commutative, thus presenting a non-deterministic behaviour.

Determinism can be achieved by forbidding the applications to use non-deterministic timing mechanisms. As a consequence, the use of multitasking would not be possible, since task synchronisation and communication mechanisms inherently lead to timing non-determinism. This is the approach taken by MARS [5] and Delta-4 [4] architectures. The former by using a static time-driven scheduling that guarantees the same execution behaviour in every replica. The latter by restricting replicas to behave as state-machines [6] 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. This imposes the modification of the underlying scheduling mechanisms, leading to a huge overhead in the system since agreement decisions must be made at every dispatching point. This is the approach followed by previous systems, such as SIFT [7] or MAFT [8], both architectures for fault-tolerant real–time systems with restricted tasking models. However, the former incurred overheads up to 80% [9], while the latter was supported by dedicated replication management hardware [8].

The use of the timed messages concept [10] allows a restricted model of multitasking to be used, while at the same time minimises 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 such message is available to all replicas.

This is the approach used in the GUARDS architecture [11] in order to guarantee the deterministic behaviour of replicated real-time transactions. However, in the GUARDS approach this mechanism is explicitly used in the application design and implementation, thus forcing system developers to simultaneously deal with both system requirements and replication issues.

3 System Model

A distributed system is considered, where, to ensure the desired level of reliability to the supported hard real-time applications, specific components of these applications may be replicated. The system nodes provide a framework to support distributed reliable hard real-time applications. Since real-time guarantees must be provided, applications have guaranteed execution resources, including processing power, memory and communication infrastructure. A multitasking environment is provided to support the real-time applications, with services for task communication and synchronisation (including replication and distribution support). Applications' timing requirements are guaranteed through the use of current off-line schedulability analysis techniques (*e.g.*, the well-known Response Time Analysis [12]).

To ensure the desired level of fault tolerance to the supported real-time applications, specific components of these applications may be replicated. This replication model supports the active replication of software (Fig. 1) with dissimilar replicated task sets in each node. The goal is to tolerate faults in the COTS

components underlying the application. In order to tolerate common mode faults in the system, COTS components diversity is also considered (operating system and hardware platform).

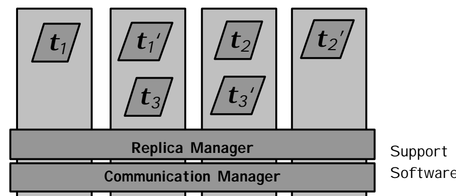

Fig. 1. Replicated hard real-time application

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 the same. 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. Using the Ada language in the replicated hard real-time applications provides the same programming model in all nodes, whilst diversity can be provided by using different compilers and runtimes [13].

In order to allow the use of the response time analysis [12], 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).

3.1 Replication Model

As there is the goal of fault tolerance 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. This component is just a configuration abstraction, which is used to structure replication units, and to allow system configuration.

The component is the unit of replication, therefore a component is one unit of fault-containment. Faults in one task may cause the failure of one 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. This means that, 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. On the other hand, a more severe fault in a component can spread to the other parts of the application in the same node, since there is 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.

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. Furthermore, error detection can be provided in an earlier stage of the processing. However, by replicating components, efficiency decreases due to the increase of the number of tasks and exchanged messages. Hence, it is possible to trade failure assumption coverage for efficiency and vice-versa. Efficiency should not be regarded as *the* goal for a fault-tolerant hard real-time system, but it can be increased by decreasing the redundancy degree.

Although the goal is to transparently manage distribution and replication, it is considered that a completely transparent use of the replication/distribution mechanisms may introduce unnecessary overheads. Therefore, during application development the use of replication or distribution is not considered (transparent approach). Later, in a configuration phase, the system developer configures the application replication and allocates the tasks in the distributed system.

The hindrance of this approach is that, as the application is not aware of the possible distribution and replication, complex applications could be built relying heavily on task interaction. This would cause a more inefficient implementation. However, the model for tasks, where task interaction is minimised, precludes such complex applications. 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 minimise task interaction, in order to improve the system's efficiency.

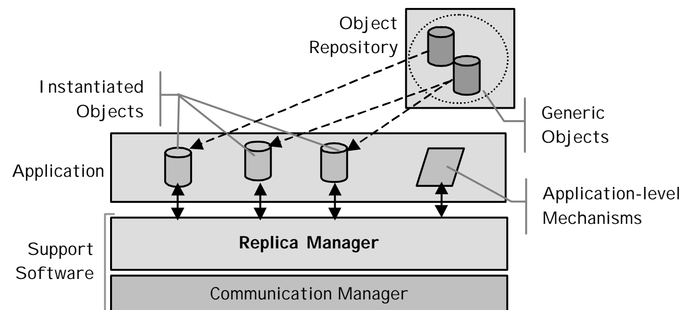

Fig. 2. Framework structure

4 Replication Management Framework

Within the presented approach, replication management is provided by means of a repository of task interaction objects (Fig. 2). These objects provide a transparent interface, by which application tasks are not aware of replication and distribution issues.

The Object Repository is used during the design and configuration phases, and provides a set of generic objects with different capabilities for the interaction between

tasks. 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 allows applications to focus on the requirements of the controlled system rather than on the distribution/replication mechanisms.

This means that during the application development there is no consideration on how the application will be replicated or distributed. These issues are only considered in a later configuration phase, by replacing the task interaction objects used in the application by their equivalent that provide the required support.

Therefore, during the application configuration phase, transparency is only considered at the mechanisms level. Basically, this means that the application code is not constrained by the low-level details of replication and distribution mechanisms. On the other hand, by performing object replacement during configuration, the full behaviour of application (considering replication and distribution) is controlled and predictable. This allows the off-line determination of the real-time and fault tolerance properties of applications, which is of paramount importance in computer control systems.

The Replica Manager layer is responsible for performing the main processing of the replication mechanisms, in order to simplify the upgrade of the Object Repository generic objects. The Communication Manager [14] layer provides the required reliable communication protocols (atomic multicast and replicated data consolidation), whilst guaranteeing the predictability of message transfers [15].

5 Object Repository

In the system, application tasks are allowed to communicate with each other using the available interaction objects, either *Shared Data* objects or *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. This task interaction model, although simple, maps the usual model found in hard real-time systems. Within the Object Repository a set of generic objects (Fig. 3) is provided, which can be instantiated by the application with the appropriate application-related data types.

Although multiple objects are available, with different capabilities and goals, during application development only three different object types are available in the repository: *Shared Data*, *Release Event* and *Release Event with Data* objects, without any distribution or replication capabilities, since 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.

```
Shared Data Object

1:  when write(data):
2:      Obj_Data := data

3:  when read:
4:      return Obj_Data

Release Event Object

5:  when wait:
6:      Task_Suspend

7:  when release:
8:      Suspended_Task_Resume

Release Event with Data Object

9:  when wait:
10:     Task_Suspend
11:     return Obj_Data

12: when release(data):
13:     Obj_Data := data
14:     Suspended_Task_Resume
```

Fig. 3. Specification of development available objects

At system configuration time, the application is distributed over the nodes of the system and some of its components are also replicated. Thus, some (or all) of the used objects must be replaced by similar ones with extra capabilities. That is, with distribution and/or replication capabilities. To support such configuration, the Object Repository makes available further objects supporting either replication, or distribution or both. Furthermore different objects are also provided if the interacting tasks are within the same component, or belong to different groups of replicated components. The interested reader may refer to [16], where the specification of all these different repository objects is presented.

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

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 must 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.

5.1 Object Implementation

In the Object Repository, all objects are implemented as generic packages. The use of generic packages allows reusing the same implementation mechanisms for objects with different data types. Even objects that do not require data instantiation (as the simple *Release Event*) are implemented as generics, since they require objects to be instantiated with extra information (for instance object identifier and ceiling priority). The use of generic packages allows parameterisation at compile time, providing significant reuse capabilities.

```
 generic
    Id    : Framework_Types.Obj_Id_Type;
    Prio  : System.Priority;
 package Object_Repository.Release_Event is

    type Release_Obj is private;
    function Request_Release_Obj return Release_Obj;

    procedure Wait    (Obj: Release_Obj);  -- potentially
                                               -- blocking
    procedure Release (Obj: Release_Obj);

private
   -- private interface
end Object_Repository.Release_Event;
```

Fig. 4. *Release_Event* object public interface

```
generic
   Id              : Framework_Types.Obj_Id_Type;
   Prio            : System.Priority;
   N_Replicas      : Framework_Types.Rep_Id_Type;
   with procedure  Decide (
       Instant_Values   : Framework_Types.Instant_Array_Type;
       Valid_Instants   : Framework_Types.Boolean_Array_Type;
       Rejected_Inst: out Framework_Types.Boolean_Array_Type;
       Release_Instant  : out Ada.Real_Time.Time;
       Release_Ok       : out Boolean);
package Object_Repository.Inter_Group.Release_Event is

   type Release_Obj is private;
   function Request_Release_Obj return Release_Obj;

   procedure Wait    (Obj: Release_Obj);   -- potentially
                                           -- blocking
   procedure Release (Obj: Release_Obj);

private
   -- private interface
end Object_Repository.Inter_Group.Release_Event;
```

Fig. 5. *Inter_Group Release_Event* object public interface

Since the goal of transparency is to allow the simple objects to be replaced by objects with distribution and replication capabilities, the public interface for similar interaction objects is the same. For instance, Fig. 4 and 5 present the public interface for the simple *Release Event* object and for its counterpart for interaction between different groups of replicated components (the *Inter Group Release Event* object).

Although the differences between the generic parameters (the *Inter Group* object requires extra parameters: the number of proposing replicas and the procedure to be used for the decision on the release instant), the interface to the application code is the same. This approach is used in all similar objects, allowing the configuration phase to modify just the objects' declaration, not changing the application tasks' code.

Internally, the *Repository* objects are implemented as Ada protected objects, in order to provide mutual exclusion for the access to the object state and, for the case of *Release* objects, to allow application tasks to be blocked by a protected entry. As an example, Fig. 6 presents the specification of the protected type used for the *Inter Group Release Event* object.

```
generic
   -- ...
package Object_Repository.Inter_Group.Release_Event is

   type Release_Obj is private;
   -- ...
private
   protected type Release_Receive_Type (
                     Prio: System.Priority;
                     Id: FT.Obj_Id_Type) is
      pragma Priority(Prio);
      entry Wait;
      procedure Release;
      function Get_Id return FT.Obj_Id_Type;
   private
      Obj_Id: FT.Obj_Id_Type := Id;
      Released: Boolean := False;
   end Release_Receive_Type;

   type Release_Obj is access all Release_Receive_Type;

end Object_Repository.Inter_Group.Release_Event;
```

Fig. 6. Example of object private implementation

5.2 Object Configuration

In the configuration phase the simple interaction objects used during design are replaced with the appropriate objects, providing replication and distribution support. In order to demonstrate the configuration of an application through the use of different interaction objects, Fig. 7 presents an example of an implementation of some *Device_Data* type.

```
package Device_Data_Package is

   type Device_Data is ...;
   type Device_Data_Array is
array (FT.Rep_Id_Type Range <>) of Device_Data;

   Device_Data_Replicas: FT.Rep_Id_Type := ...;
   Device_Obj_Id: FT.Obj_Id_Type := ...;
   Device_Obj_Prio: System.Priority := ...;

   procedure Device_Data_Decide (
        Values           : Device_Data_Array;
        Value_Instants   : FT.Instant_Array_Type;
        Valid_Values     : FT.Boolean_Array_Type;
        Rejected_Values  : out FT.Boolean_Array_Type;
        Release_Value    : out Device_Data;
        Release_Instant  : out A_RT.Time;
      Release_Ok         : out Boolean);

end Device_Data_Package;
```

Fig. 7. Data handling package

The package *Device_Data_Package* specifies the type of the data and an anonymous array type. It also provides some configuration information, such as the number of replicas of the releasing group, the identifier of the object and the objects' ceiling priority. Finally, a *Decide* procedure (to be executed by a replica consolidation protocol) is also provided.

Fig. 8 and 9 present the use of this *Device_Data_Package*, respectively before and after the configuration phase. The only difference is in the instantiation of the generic package, where the *Release Event With Data* generic package only requires the identifier, priority and data type parameters. After configuration, the *Inter-Group Release Event With Data* generic package also requires the number of replicas, the data type array and the *Decide* procedure.

```
package body Example_Application_Tasks is

     package DDP renames Device_Data_Package;

    package Device_Event_Data_P is
          new Object_Repository.Release_Event_With_Data (
                           Id           => DDP.Device_Obj_Id,
                           Prio         => DDP.Device_Obj_Prio,
                           Data_Type    => DDP.Device_Data );

    Device_Event_Obj: Device_Event_Data_P.Data_Release_Obj :=
           Device_Event_Data_P.Request_Data_Release_Obj;
    -- Other Objects and Application Tasks

end Example_Application_Tasks;
```

Fig. 8. Use of data handling package (before configuration)

```
package body Example_Application_Tasks is

    package DDP renames Device_Data_Package;

    package Device_Event_Data_P is new
       Object_Repository.Inter_Group.Release_Event_With_Data(
           Id               => DDP.Device_Obj_Id,
           Prio             => DDP.Device_Obj_Prio,
           N_Replicas       => DDP.Device_Data_Replicas,
           Data_Type        => DDP.Device_Data,
           Data_Array_Type  => DDP.Device_Data_Array,
           Decide           => DDP.Device_Data_Decide);

    Device_Event_Obj: Device_Event_Data_P.Data_Release_Obj :=
            Device_Event_Data_P.Request_Data_Release_Obj;
    -- Other Objects and Application Tasks

end Example_Application_Tasks;
```

Fig. 9. Use of data handling package (after configuration)

6 Conclusions

The Ravenscar profile allows the use of the pre-emptive fixed priority computational model in application areas where the cyclic executive model has traditionally been the preferred approach. Nevertheless, it is also considered that further studies are still necessary for its use to support replicated and distributed systems. The interaction between multitasking software and replication introduces new problems, particularly for the case of a transparent and generic approach.

This paper presents an abstraction for application replication, intended to support the transparent replication of Ravenscar applications. This approach allows applications to be configured only after being developed, thus allowing applications to be developed without considering replication and distribution issues. A transparent support to the applications is provided through a set of generic task interaction objects, which hide from applications the low level details of replication and distribution.

Acknowledgements

The authors would like to thank Andy Wellings for his valuable support in the specification of the replication framework and to the anonymous reviewers for their helpful comments and suggestions. This work was partially supported by FCT (project TERRA POSI/2001/38932).

References

1. Burns, A. (1997). Session Summary: Tasking Profiles. In *Proc. of the 8th International Real-Time Ada Workshop*, Ravenscar, England, April 1997. Ada Letters, XVII(5):5-7, ACM Press.
2. Wellings, A. (2000). Session Summary: Status and Future of the Ravenscar Profile. In Proc. of the 10th International Real-Time Ada Workshop, Avila, Spain, September 2000. Ada Letters, XXI(1):4-8, ACM Press.
3. Audsley, A. and Wellings, A. (2000). Issues with using Ravenscar and the Ada Distributed Systems Annex for High-Integrity Systems. In Proc. of the 10th International Real-Time Ada Workshop, Avila, Spain, September 2000. Ada Letters, XXI(1):33-39, ACM Press.
4. Powell, D. (Ed.). (1991). Delta-4 - A Generic Architecture for Dependable Distributed Computing. ESPRIT Research Reports. Springer Verlag.
5. 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. In IEEE Micro, 9(1):25–41.
6. Schneider, F. (1990). Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. In ACM Computing Surveys, 22(4):299-319.
7. Melliar-Smith, P. M., and Schwartz, R. L. (1982). Formal Specification and Mechanical Verification of SIFT: a Fault–Tolerance Flight Control System. In IEEE Transactions on Computers, 31(7):616-630.
8. Keickhafer, R. M., Walter, C. J., Finn, A. M., and Thambidurai, P. M. (1988). The MAFT Architecture for Distributed Fault Tolerance. In IEEE Transactions on Computers, 37 (4):398-404.
9. Pradhan, D. K. (1996). Fault–Tolerant Computer System Design. Prentice Hall.
10. Poledna, S., Burns, A., Wellings, A., and Barret, P. (2000). Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems. In IEEE Transactions on Computers, 49(2):100-111.
11. Powell, D. (Ed.). (2001). A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems. Kluwer Academic Publishers.
12. Audsley, A., Burns, A., Richardson, M., Tindell, K., and Wellings, A. (1993). Applying new scheduling theory to static priority pre-emptive scheduling. In Software Engineering Journal, 8(5):285-292.
13. Yeh, Y. (1995). Dependability of the 777 Primary Flight Control System. In Proc. Dependable Computing for Critical Applications 5, USA, pp. 1-13.
14. Pinho, L. and Vasques F. (2001a). Reliable Communication in Distributed Computer-Controlled Systems. In Proc. of Ada-Europe 2001. Leuven, Belgium, May 2001, Lecture Notes on Computer Science 2043, Springer, pp. 136-147.
15. Pinho, L. and Vasques, F. (2001b). Timing Analysis of Reliable Real-Time Communication in CAN Networks. In Proc. of the 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001, pp. 103-112.
16. Pinho, L. M. (2001). A Framework for the Transparent Replication of Real-Time Applications. PhD Thesis. School of Engineering of the University of Porto. Available at http://www.hurray.isep.ipp.pt