Using Ravenscar to Support Fault-Tolerant Real-Time Applications

Luís Miguel Pinho
Department of Computer Engineering,
ISEP, Polytechnic Institute of Porto
Rua Dr. António Bernardino de Almeida, 431
4200-072 Porto
Portugal
E-mail: lpinho@dei.isep.ipp.pt

Francisco Vasques
Department of Mechanical Engineering
FEUP, University of Porto
Rua dos Bragas
4050-123 Porto
Portugal
E-mail: vasques@fe.up.pt

Abstract

Recently, a framework was proposed intended to support the development of replicated and distributed Ravenscar applications. This framework provides a transparent abstraction for application replication, allowing applications to be developed without considering replication and distribution issues. However, it is also necessary to assess if the Ravenscar profile is expressive enough for the implementation of this complex middleware. Therefore, this paper presents some conclusions that were drawn on the implementation of a prototype of the framework.

1. Introduction

The Ravenscar profile defines a subset of the language's multitasking mechanisms, considered suitable for the development of efficient and deterministic real-time applications. It allows multitasking pre-emptive applications to be considered 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 [1]. 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.

Recently, a framework intended to support the development of replicated and distributed Ravenscar applications [2] was proposed. This framework provides an abstraction for application replication, allowing 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, allowing Ravenscar applications with different structures and configurations to be developed.

However, it is also necessary to assess if the Ravenscar profile is expressive enough for the implementation of the complex middleware intended for the support to replicated/distributed applications. The restrictions of the Ravenscar profile make difficult the implementation of an efficient support for replicated or distributed programming, which may result on an increased application complexity [3].

For that purpose, a prototype implementation of the proposed framework was developed. Although quantitative measures were not obtained, there are however some conclusions that can be drawn from this implementation.

This paper is structured as follows. Section 2 presents the considered replication model, while Section 3 briefly presents the replication management framework. Afterwards, Section 4 presents the conclusions that were drawn from the prototype implementation.

2. Replication Model

The replication model considers a set of distributed nodes, where a framework is provided to support distributed reliable hard real-time applications. The framework supports the active replication of software (Figure 1) with dissimilar replicated task sets in each node.

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.

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.

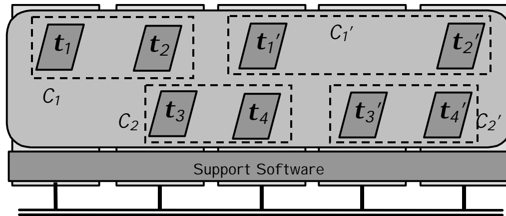

Fig. 1 – Replicated Hard Real-Time Application

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.

As active replication is used, there is the need to guarantee replica determinism, *i.e.*, that replicated tasks execute with the same data and timing-related decisions are the same in each replica. The use of timed messages [4] 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 synchronisation algorithms, since clock deviations must be upper bounded.

3. Replication Framework

Within the replication model, 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 its replication level, and allocates the different tasks in the distributed system.

The hindrance of this approach is that, as the programmer is not aware of the possible distribution and replication, 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 minimised, 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 management. 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 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 on 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.

3.1 Framework Structure

The support to the applications is provided in a twofold way. Firstly, an underlying software layer (the Communication Manager) provides the needed communication mechanisms for replication and distribution. It provides atomic multicasts and appropriate algorithms for replicated data consolidation. Nevertheless, group communication technology by itself is too low-level to allow complex fault-tolerant applications, and a lot of extra functionality has to be added to applications to guarantee its fault-tolerance and performance requirements [5]. Thus, there is the need for an extra abstraction between the application and the group communication mechanisms.

Such abstraction is provided by means of a repository of task interaction objects (Figure 2). 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 Support Software, a run-time middleware layer between the application and the COTS components.

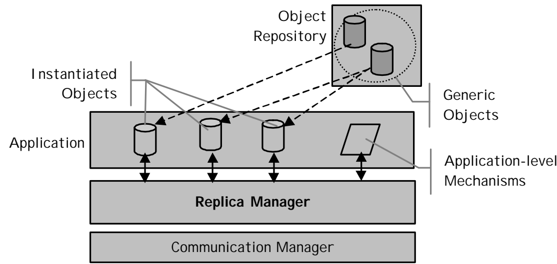

Fig. 2 – Framework Structure

The Object Repository is used during the design and configuration phase, and provides 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 allows applications to focus on the requirements of the controlled system rather than on the distribution and replication mechanisms.

This approach also allows the addition and reuse of new objects. If other generic task interaction objects are later realised to be important, they can be incorporated in the Object Repository, thus available for 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 the responsible for performing the main processing of the provided mechanisms, and the generic objects in the repository just provide the transparency and object model to the applications.

The Replica Manager is the responsible for the consistency of the replicated components. It provides the required mechanisms for supporting the task interaction objects, and a restricted direct support to applications. 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.

3.2 Programming Model

Application tasks communicate with each other by using *Shared Data* objects and the release of *Release Event* objects. The Object Repository provides a set of generic objects, 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 different object types 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 values.

However, in system configuration, 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. Three different types of interactions are identified: interaction internal to a component, between groups of replicated components and with the environment.

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 (if the component is replicated).

When tasks belonging to different components interact, there is the need to consolidate values or events between the component replicas. Therefore objects are provided that perform a transparent consolidation of values between the replicas of the writing component. This consolidation is supported by the appropriate communication protocols, available in the Communication Manager.

The interconnection with the environment must provide mechanisms to support the transfer of information. However, this interconnection is highly dependent of the used platform thus a generic mechanism can not be provided. Instead, the application supplies a specific interconnection task, which interacts with the environment, performs the necessary filtering, and interacts with the rest of the system through one of the available objects.

4. Protoype Considerations

From the implementation of the framework prototype, some conclusions about the use of the Ravenscar profile for the implementation of this complex middleware can be drawn. It is possible to assess the increase of complexity in the framework code, and also the increase in the used resources, particularly when considering the interaction with the communication-related mechanisms.

Although Ravenscar presents an extensive number of restrictions, only four of them were considered to have a relevant impact in the implementation:

- The restrictions imposed to protected types, namely only one entry and no more than one task simultaneously calling a protected entry;
- Not allowing the dynamic allocation of tasks and protected objects;
- Not allowing dynamic priorities;
- Not allowing timed entry calls (no select statement).

```
protected Timer_Management is
   procedure Request_Timer (
            Id: out FT.Timeout_Id_Type;
            Ok: out Boolean);
   procedure End_Timer (
            Id: FT.Timeout_Id_Type);
private
   Available: Used_Timeout_Type :=
                    (others => True);
   pragma Priority ( Any_Priority'Last-1);
end Timer_Management;

protected type Timer_Launch is
   entry Wait(
            T: out A_RT.Time;
            Call: out Timeout_Callback);
   procedure Release(
            T: A_RT.Time;
            Call: Timeout_Callback);
   procedure Cancel;
   function Is_Cancelled return Boolean;
private
   Time: A_RT.Time;
   Callback: Timeout_Callback;
   Cancelled: Boolean := False;
   Released: Boolean := False;
   pragma Priority ( Any_Priority'Last-1);
end Timer_Launch;
```

Fig. 3 – Timer management

4.1 Protected Entries

In the Ravenscar profile, protected objects are restricted to have at most one protected entry and only one task simultaneously queued. It is clear that these restrictions will impose an increased complexity of the middleware code. The reason is that the mechanisms that could be performed by a single object must now be jointly performed by a group of objects. For instance, Figure 3 presents a timer management and release mechanism provided in the framework, that must be implemented by two different protected types (there must be one *Timer_Launch* object for each used timer, since only one task can be queued in a protected object entry).

However, it is considered that this complexity increase is not sufficient to produce significant difficulties to the implementation of these mechanisms. There will be, however, an increase of the resource usage (higher number of protected objects). Nevertheless, this increase may be counterbalanced by the decrease of the complexity related to the management of the protected objects in the runtime.

4.2 Dynamic Allocation of Tasks and Protected Objects

Another restriction in the Ravenscar profile (which was not present in the initial specification) is that the dynamic creation of tasks and protected objects is not allowed, even during the program elaboration. Thus, the initialisation of the middleware becomes more complex, since it is not possible to create just the resources required by the implementation. This can be seen in Figure 4, where an array of objects for managing received messages must be created (*Received_Msgs*), one for each possible message identifier (the *Used* array is to, during initialisation, record the identifiers that will use the protocol).

```
protected body Received_Msg_Obj is
   -- ...
end Received_Msg_Obj;

Used: array (FT.Message_Identifier)
         of Boolean := (others => False);

Received_Msgs: array (FT.Message_Identifier)
         of Received_Msg_Obj;
```

Fig. 4 – Example of timer tasks use

If dynamic creation of objects were possible, only the message identifiers using the protocol would have a related object. Although not increasing the code complexity, this restriction increases the amount of resources required for the implementation.

4.3 Dynamic Priorities

A restriction with a significant impact in the implementation is the impossibility of dynamically changing the priority of a task. The main problem arises during the interaction of protected objects with communications. Tasks processing the messages' reception have to update values or to release entries in the protected objects of the *Repository*. However, it must be

guaranteed that the priority of the task is not greater than the ceiling priority of the called object.
If dynamic task priorities were allowed, it would be easy to change the task priority to the ceiling priority of the object, before making the call. Since this is not possible, each object requiring the reception of communication streams has to declare its own handler task, which consequently increases the code complexity and the resource usage by the framework.

4.4 Timed Entry Calls

Another restriction with a relevant impact in the complexity and resource usage of the implementation is the impossibility of using the timed entry call mechanism of Ada. This mechanism allows a task to abort a protected entry call where it is suspended, when a specific time is reached. As this mechanism provides an efficient support for managing timeouts, it would not be necessary to use the complex and inefficient mechanism presented in Figure 3.
Nevertheless, it is considered that due to the nature of the delays used in the communication protocols [6], the most efficient approach would be to use a high resolution hardware timer. This timer could be reprogrammed each time a new timeout was required (as is usually done for task management), without the overhead related to the timed entry call mechanism.

4.5 Final Considerations

From this evaluation, it is possible to conclude that the Ravenscar profile, by decreasing the complexity and overhead of the Ada runtime, introduces a greater complexity and overhead in applications requiring distribution and replication support. However, within the restrictions that were found to have some impact, only dynamic task priorities and dynamic allocation of tasks and protected objects are considered to be useful in the implementation of the prototype. The availability of these mechanisms would reduce the overhead and resource usage of the framework.
The restrictions on protected entries do not present a relevant impact, since more complex mechanisms can be built with this basic block when needed. The lack of the timed entry call mechanism, although decreasing the related overhead, would not present a significant improvement, considering the nature of the required timeouts for the communication protocols.
The introduction of a transparent and generic approach allows applications to abstract from the requirements of replication and distribution, thus becoming simpler to develop. Nevertheless, this approach increases the complexity and resource usage of the framework, since the mechanisms for managing generics together with replication and distribution become more complex.
Nonetheless, the same reasoning that is applied to the Ada tasking model can also be applied to the framework. As the framework is not application-specific, it does not have to be designed for each application, thus much more attention can be given to its implementation. Note that, in the absence of the framework, the mechanisms of replication and distribution management would have to be implemented in the application. Although such approach could be more efficient, the application itself would be more complex, thus more difficult to develop and maintain.

5. Summary

This paper presented the main issues related to the prototype implementation of a proposed framework for the development of fault-tolerant real-time applications conforming to the Ravenscar profile. The main goal of the implementation was to assess the expressiveness of the Ravenscar profile for the development of the mechanisms required by the framework.
From the implementation, it is considered that some of the Ravenscar restrictions present some impact, but that in general, the profile presents a suitable model for the development of these mechanisms. It is also considered that by producing a more efficient runtime, the profile may compensate the greater complexity it causes in applications.
Since quantitative measures were not obtained, it would be worthwhile to further investigate how much is gained from applying certain restrictions, at the cost of an increased complexity. To achieve this, a suitable platform (compiler and runtime) is required, in order to obtain relevant measures of performance.

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] Wellings, A. , ”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.

[2] Pinho, L. M., ”A Framework for the Transparent Replication of Real-Time Applications”. PhD Thesis. School of Engineering of the University of Porto, September 2001. Available at http://www.hurray.isep.ipp.pt

[3] Audsley, A. and Wellings, A., ”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] Poledna, S., Burns, A., Wellings, A., and Barret, P., “Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems”. IEEE Transactions on Computers, 49(2):100-111, 2000.

[5] Johnson, S., Jahanian, F., Ghosh, S., VanVoorst, B., and Weininger, N., “Experiences with Group Communication Middleware”. Proc. International Conference on Dependable Systems and Networks, New York City, USA, pp. 37-42, 2000.

[6] Pinho, L. and Vasques, F., “Timing Analysis of Reliable Real-Time Communication in CAN Networks”. Proc. 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, 2000.