Replica Management in Real-Time Ada 95 Applications

Luís Miguel Pinho
Department of Computer Engineering
School of Engineering
Polytechnic Institute of Porto
Portugal
e-mail: lpinho@dei.isep.ipp.pt

Francisco Vasques
Department of Mechanical Engineering
School of Engineering
University of Porto
Portugal
e-mail: vasques@fe.up.pt

Abstract:

In this paper, we present some of the fault tolerance management mechanisms being implemented in the Multi-μ architecture, namely its support for replica non-determinism. In this architecture, fault tolerance is achieved by node active replication, with software based replica management and fault tolerance transparent algorithms. A software layer implemented between the application and the real-time kernel, the Fault Tolerance Manager (FTManager), is the responsible for the transparent incorporation of the fault tolerance mechanisms

The active replication model can be implemented either imposing replica determinism or keeping replica consistency at critical points, by means of interactive agreement mechanisms. One of the Multi-μ architecture goals is to identify such critical points, relieving the underlying system from performing the interactive agreement in every Ada dispatching point.

Keywords: *Ada 95, real-time systems, software based fault tolerance, replica determinism*

1. Introduction

From a real-time system perspective, fault tolerance can be defined as the ability to deliver the expected service in a timely manner, even in the presence of faults [6]. An important issue in real-time systems is that fault tolerance mechanisms must be time bounded, in order to achieve timing predictability.

The fault tolerance mechanisms are usually implemented either by temporal replication, redoing the calculations, or by structural replication, replicating physical (and/or logical) resources. In a real-time system, as the time resource is scarce, structural replication is usually the preferred one.

Replication management can be achieved using specialised hardware, which consequently increases the overall cost of the system. Conversely, the software-based replication allows the use of COTS (Commercial Off-The-Shelf) components, decreasing the cost of the system, and at the same time increasing its portability and upgradability.

The behaviour of COTS components can be assumed as either fail-silent or fail-uncontrolled [15]. A fail-silent component is one that only fails by omission, while fail-uncontrolled means a component that can fail in an arbitrary mode. The assumption of fail-silent components simplifies the fault tolerance mechanisms implementation, since failures can be detected just by time-out mechanisms. However, as fail-silent behaviour is only possible with the use of self-checking techniques, it demands specific COTS components. Conversely, using generic COTS components, fault tolerant mechanisms must support fail-uncontrolled behaviour.

Two main replication approaches are addressed in the literature [3]: active replication and primary-backup (passive) replication. In active replication, all replicas process the same inputs, keeping their internal state synchronised and voting all on the same outputs. In the primary-backup approach only one replica (the primary) is responsible for the inputs processing, being the replicas kept up to date by the primary, to take over in case of its failure.

Using the primary-backup approach, backup replicas can only detect the primary failure through the absence of service delivery, not being able to reason about the service correctness. This approach can be used only if we assume fail-silent replicas. Otherwise, wrong service delivery can only be detected by active replication. As a consequence, in the case of the fail-uncontrolled components, the active replication is the most adequate technique [14].

Real-time applications are based on time-dependent mechanisms, hence the different processing speed in replicated nodes can cause dissimilar 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 [13].

In the proposed approach, fault tolerance is achieved by node replication, with software based replica management and fault tolerance transparent algorithms. A software layer implemented between the application and the real-time kernel, the Fault Tolerance Manager (FTManager), is the responsible for the transparent incorporation of the fault tolerance mechanisms. The FTManager, is subdivided in the two following sub-layers:

- the Communication Manager, which implements the communication algorithms based on group abstractions;
- the Replica Manager, which provides the necessary mechanisms for the replica management, hiding its implementation from the application programmer.

The paper starts presenting issues related to replica determinism for software based fault tolerance management. Afterwards, some of Ada non-determinism problems are identified and discussed. Finally the Multi-μ approach is presented and its Fault Tolerance Manager is discussed.

2. Guaranteeing Replica Determinism

In order to guarantee replica determinism, the group abstraction can be used to implement a framework for the replica management [3]. Two problems are identified [2]: consensus, where there must be a decision despite the presence of failures, and membership, where there must be an agreement on who belongs to the group. In the proposed approach, active replication can be implemented using static groups, since there is no need to consider the case of leaving or joining a group (membership).

Assuming replica determinism the consensus between replicas can be achieved by:

- providing Byzantine Agreement on single-source data;
- providing Interactive Consistency on replicated sensor data;
- providing Consensus on output values.

To provide replica determinism, different approaches can be used:

- restricting the programmer from using mechanisms that can lead to timing non-determinism (the state-machine approach [17]);
- performing an interactive agreement on every scheduling decision;
- performing an interactive agreement only when non-deterministic execution can lead to replica inconsistency.

Replica determinism can be easily achieved restricting the programmer from using timing non-deterministic mechanisms. However, the use of multitasking which is fundamental for real-time systems would not be possible, since task synchronisation and communication mechanisms inherently lead to timing non-determinism.

Guaranteeing that replicas make the same scheduling decisions, by performing an interactive agreement in every scheduling decision, allows the use of non-deterministic mechanisms. However, it imposes the modification of the underlying scheduling mechanisms. Such approach leads to a large overhead, since scheduling decisions must be made at every dispatching point.

Delaying synchronisation until there is the possibility of divergence between replicas, due to their distinguished execution, also allows the use of non-deterministic mechanisms, but at the expense of smaller overheads, since only a subset of scheduling decisions must be synchronised.

3. Identifying Replica Non-Determinism Problems in Ada

Ada is a programming language specially suited for real-time systems programming. However, it doesn't provide direct support for fault tolerance mechanisms, apart from the exception mechanism that only is able to support forward error recovery. Furthermore, exceptions can not provide tolerance to unanticipated faults or to design faults [7]. The usual solution is to burden the application programmer with the explicit programming of fault tolerance mechanisms.

Work has (and is) being done in the integration of fault tolerance and Ada. Two approaches coexist: incorporating explicit programmer support for fault tolerance mechanisms, or providing transparent support for software replication.

To guarantee replica determinism, Ada provides mechanisms like *pragma Restrictions* that can be used to prevent multitasking [4]. Also the code can be analysed to ensure that none of the non-deterministic mechanisms are used. However, guaranteeing deterministic replicas imposes several restrictions on the application programmer, excluding Ada constructs that may cause non-deterministic replicas evolution.

Guaranteeing replica determinism by means of explicit programming mechanisms [7] [8] [19] or extending the language [11] are not appropriated in a fault tolerant transparent approach. Allowing the use of non-deterministic replication of an Ada program is not also an easy task since Ada is inherently a non-deterministic language.

In an Ada application some causes for divergence between replicas are:

- use of rendezvous;
- use of protected objects;
- use of the select construct.

When several client tasks can make a call on a server task entry, a different interleaving may cause these calls to be accepted in a different order. In the following code snapshot two different tasks of the same type (*Client*) can call task *Server* entry. It is possible that, while in one replica *Client1* executes the call (and is accepted) first, in another replica *Client2* is the first to be accepted.

```
task Server is
  entry Service;
end Server;

task body Server is
begin
  loop
     accept Service do
        --...
     end Service;
  end loop;
end Server;

task type Client;

task body Client is
begin
  --...
  Server.Service;
  --...
end Client;

Client1, Client2: Client;
```

Asynchronous communication using protected objects can cause different tasks interleaving in different replicas. In the following code both tasks *Reader* and *Writer* concurrently access protected object *Obj* and because execution order can be different, protected object *Obj* may have different state in different replicas.

```
protected Obj is
  procedure Write(D: Integer);
  function Read return Integer;
private
  Data: Integer;
end obj;

protected body Obj is

  procedure Write(D: Integer) is
  begin
     Data:= D;
  end Write;

  function Read return Integer is
  begin
     return Data;
  end Read;

end obj;

task Writer;

task body Writer is
  Data: Integer;
begin
  --...
  Obj.Write(Data);
  --...
end Writer;
task Reader;

task body Reader is
  Data: Integer;
begin
  --...
  Data:= Obj.Read;
  --...
end Reader;
```

Ada select can also provide different results depending on the different tasks interleaving or different clocks pace. Different clients (or none) can be accepted by a server in different replicas, or a client making a conditional or timed entry call can be successful in one replica and not in another. Furthermore, select is also used in the Asynchronous Transfer of Control (ATC) feature of Ada, which, as a consequence, can also provide different results in different replicas.

In the following code both *Client1* and *Client2* call an entry in *Server*. The *Server* task can accept any of these calls or even can execute the else part if no call is queued. Furthermore *Client1* executes a timed entry call, that can expire before the service is accepted. In *Client2* there is a conditional entry call, where the else part can be executed if the *Server* is not ready. Any one of these different possible execution paths may lead to inconsistent state between different replicas.

```
task Server is
  entry Service1;
  entry Service2;
end Server;

task body Server is
begin
  loop
     select
        accept Service1 do
           --...
        end Service1;
     or
        accept Service2 do
           --...
        end Service2;
     else
        --...
     end select;
     --...
  end loop;
end Server;

task Client1;

task body Client1 is
begin
  select
     Server.Service1;
  or
     delay 1.0;
  end select;
end Client1;
```

```
task Client2;

task body Client2 is
begin
  select
     Server.Service2;
  else
     --...
  end select;
end Client2;
```

If the presented services are non-commutative then, after execution, different replicas will be in different states. These presented problems are even more complicated considering additional mechanisms like requeue or guards, not due to timing non-determinism but due to their interaction with the previous mechanisms.

However, the use of these constructs is not always non-deterministic. If there is only one client that can rendezvous with the server, there is no need for synchronisation between replicas. Also, if in a protected object the reader uses an entry guarded by the writer execution, then there is no need to synchronise. As a consequence, some of these tasks precedence relations can be captured by static analysis, being synchronisation unnecessary.

Therefore, performing an interactive agreement in every Ada dispatching point introduces an unnecessary overhead. The goal of the Multi-μ architecture is to perform the agreement only when it is necessary.

4. Managing Fault Tolerance in the Multi-μ Architecture

The Multi-μ [12] architecture targets the development of fault tolerance mechanisms for systems where reliability and availability are of most importance.

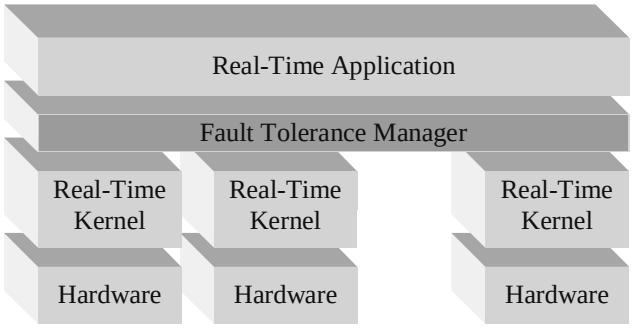

Figure 1. Multi-μ Architecture.

The architecture (figure 1) is based on replicated software components on top of replicated nodes, which are built with both COTS kernel and hardware. The fault tolerance transparent mechanisms are implemented below the application, interacting with the real-time kernel. Being a tightly-coupled system with nodes interconnected by a parallel bus, it implements a synchronous distributed system. The advantage of a synchronous system is that communication times are bounded, simplifying both the algorithms to implement fault tolerance mechanisms and the application timing analysis.

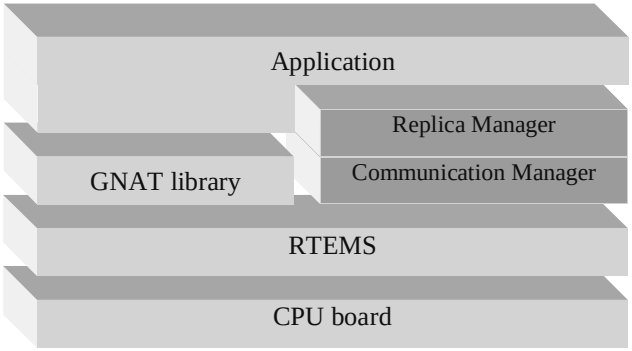

Figure 2. Multi-μ Node Architecture.

Each node (figure 2) has a real-time kernel, responsible for the multitasking environment and for inter-node communication. The application is built on top of the compiler library, to ensure abstraction from kernel implementation, and also on top of the FTManager (responsible for the replica management and for the communication algorithms implementation) to provide the fault tolerance abstraction. The selected kernel and Ada compiler are, respectively, RTEMS [16] and GNAT [18]. Both GNAT and RTEMS sources are freely available and can be adapted and extended to implement the FTManager.

The FTManager is responsible for the transparent incorporation of the fault tolerance mechanisms into the application. As replica determinism is not assumed, it allows the programmer to use all the Ada constructs. The FTManager is sub-divided in the two following layers:

- the Communication Manager, which implements the communication algorithms, based on group communication abstractions;
- the Replica Manager, which provides the necessary mechanisms for the replica management, hiding its implementation from the application programmer.

4.1 Communication Manager

To support consensus between replicas, communication mechanisms must be provided to disseminate replicas private values. The implemented fault tolerant communication algorithm is the Signed Messages (SM(m)) algorithm of Lamport, Shostak and Pease [9], to solve the byzantine agreement problem.

As it is possible to know the maximum number of exchanged messages, and giving the bounded communication times, this algorithm has bounded worst-case execution time.

To solve the interactive consistency problem we allow for the concurrent execution of several signed messages algorithms. To solve consensus we simply choose for the output value the majority of the values received in an interactive consistency exchange.

The Communication Manager layer (an extended discussion of this layer is included in [12]) is built on top of a thin Ada binding to RTEMS queues. This interface provides a mechanism to exchange messages between nodes, without knowledge of the physical distribution of the sender and receiver tasks, being a good framework to build replicated systems.

This layer provides group abstraction to the higher layer, and also logical links between nodes. The child package concept of Ada allows functionality extension without changes to the code of the parent, providing extended capabilities for the signed message algorithm. Other extensions can be provided, such as atomic broadcast or clock synchronisation.

The communication between replicas is implemented using two unidirectional queues (send and receive queues) between each pair of nodes, providing full logical connectivity. We envisage the use of queue redundancy in order to cope with queue failures, providing redundant paths between each pair of nodes.

4.2 Replica Manager

Without assuming replica determinism, there is the need to explicitly synchronise replicas, in every code point that can lead to execution divergence. The Replica Manager hides the communication algorithms from the application, and copes with the possible non-deterministic behaviour of replicas, by delaying requests until all nodes make the same request.

Two different mechanisms for synchronisation are currently being considered:

- synchronisation on rendezvous;
- synchronisation on protected objects with procedures and functions.

Synchronisation of the select construct, protected objects with entries and the use of requeue are not yet being considered, so they are not supported at the moment.

To perform the rendezvous synchronisation, the server must know each of the calling tasks group and must also be able to accept a specific one. Such synchronisation can only be made using an entry family instead of a single entry. This would however imply higher complexity code.

Another option to perform such synchronisation is doing the client-server interaction through a protected object (instead of the rendezvous) as presented in [10]. However, this approach changes the semantics of the rendezvous, as: “during rendezvous, the task accepting the entry call inherits the active priority of the caller” (D.1(22) of the Ada Reference Manual [5]).

A different approach is to use a monitor object, which performs the synchronisation of the calling clients before the entry call, maintaining the rendezvous. This approach can lead to a priority inversion, which is already present in the rendezvous due to its semantics [1]. Although an entry family to distinguish between different clients is still required, its implementation in a protected object is easier than in a task and there is no need to tailor it for each task entry.

Protected objects synchronisation is more complex than in the case of rendezvous, since there is no limit on the number and type of procedures and functions. As protected objects are passive entities, when they need to synchronise accesses a monitor task is created to perform the necessary agreement.

The approach used to the rendezvous synchronisation can not be used for protected objects, since it would introduce priority inversion problems in the use of protected objects. This solution would imply the use of Ada dynamic priorities inside the monitor to change the task base priority and resetting it again when it was released.

Protected objects are much more powerful than rendezvous, thus we can use Ada object-oriented features in order to allow the programmer to request synchronisation in any procedure or function using the same request entry of the monitor, and a private entry family where to requeue each request.

Information regarding replication (replica configuration) is only integrated in a final configuration phase. Therefore, real-time applications can be programmed disregarding distribution and still use all the Ada powerful constructs. This configuration phase scheme looks like the model of the Distributed Systems Annex of Ada, but a different goal is intended.

An automated tool can be provided to help the programmer identifying possible sources of non-determinism, hiding at the same time the configuration of the system from the application reader.

This configuration phase is made through the introduction of three *pragmas* in the application code,

- *pragma* **Replicated**, to identify the tasks and protected objects that must have replication management;
- *pragma* **Synchronise**, to identify code locations where there is the need for replica synchronisation;
- *pragma* **Agreement**, to identify where there is need for agreement on replicated (or single-source) values;

An application example shows how the referred *pragmas* may be used in order to solve some of the

identified problems in section 3. In this example, two client tasks read some device data, make requests to a single server task ensuring that it is ready for data processing, sending data to the server through a protected object. The server task then reads the data and processes it.

The protected object *Buffer* procedures *Write* and *Read* can be called from different tasks. As already stated, system replication can induce non-deterministic access to objects. The *pragma Replicated* applied to the object implies that it must be monitored. Tasks *Server*, *ClientA* and *ClientB* are replicated among the system. Thus, *pragma Replicated* is used to achieve the needed consensus. Every time that a task makes a call on a protected object, or on a server task entry, the need for synchronisation arises, so *pragma Synchronise* is used. The necessary agreement on Input and Output is provided by *pragma Agreement*, when tasks read device data.

```
--------------------------------------
-- Small Controller
-- An example of Replica Manager use
--------------------------------------

procedure Controller is

   type Some_Data is ...;

   --------------------------------------
   -- Replicated Protected Object
   --------------------------------------

   pragma Replicated;
   protected Buffer is

      procedure Write(Data: Some_Data);
      procedure Read(Data: Some_Data);

   private
      Data: Some_Data;
   end Buffer;

   protected body Buffer is separate;

   --------------------------------------
   -- Replicated Server task
   --------------------------------------

   pragma Replicated;
   task Server is
      entry Request;
   end Server;

   task body Server is
      Data: Some_Data;
   begin
      loop
         -- Synchronisation of
         -- entry call accept

         pragma Synchronise;
         accept Request do
            -- ...
         end Request;

         -- Reading from the Buffer must
         -- be synchronised

         pragma Synchronise;
         Buffer.Read(Data);

         -- ...
      end loop;
   end Server;

--------------------------------------
-- Replicated Client task
--------------------------------------

   pragma Replicated;
   task ClientA;

   task body ClientA is
      Data: Some_Data;
   begin
      loop
         -- Device1 is replicated in
         -- all nodes. Its values must
         -- be agreed upon.
         pragma Agreement;
         Request_Device1_Data(Data);

         -- Calling an entry must
         -- be synchronised
         pragma Synchronise;
         Server.Request;

         -- ...
         pragma Synchronise;
         Buffer.Write(Data);
      end loop;
   end ClientA;

--------------------------------------
-- Client task, calling Server entry,
-- write access to the protected object,
-- and providing sigle-source values
--------------------------------------

   pragma Replicated;
   task ClientB;

   task body ClientB is
      Data: Some_Data;
   begin
      loop
         -- Device2 is not replicated.
         -- Its values must be provided
         -- from Node 1 to other nodes.
         pragma Agreement (Source_Node => 1);
         Request_Device2_Data(Data);

         -- ...
         pragma Synchronise;
         Server.Request;

         -- ...
         pragma Synchronise;
         Buffer.Write(Data);
      end loop;
   end ClientB;

begin
   -- ...
end Controller;
```

5. Conclusions

The paper presented some of the fault tolerance management mechanisms being implemented in the Multi-µ architecture, which is intended to study and develop software based fault tolerance mechanisms for real-time systems, using the Ada language.

Multi-µ is implemented through the active replication of processing nodes, with fault tolerance being achieved by means of a specially proposed software layer, the Fault Tolerance Manager (FTManager).

Issues regarding fault tolerance are presented and the problems of using Ada to build fault-tolerant systems are discussed. Multi-µ architecture develops the approach of not imposing replica determinism and performing the interactive agreement only at the necessary synchronisation points. One of the Multi-µ architecture goals is to identify such critical points.

6. Acknowledgements

This work was partially supported by FLAD (project SISTER 471/97) and by DEMEGI / FEUP.

7. References

[1] Burns, A. and Wellings, A. Concurrency in Ada. 2nd Ed. Cambridge University Press, 1998.

[2] Galleni, A. and Powell, D. Consensus and Membership in Synchronous and Asynchronous Distributed Systems. LAAS Report 96104, April 1996.

[3] Guerraoui, R. and Schiper, A. Software-Based replication for Fault Tolerance. IEEE Computer, April 1997, 68-74.

[4] Heras-Quirós, P., González-Barahona, J., Centeno-González, J. Programming Distributed Fault Tolerant Systems: The ReplicAda Approach. In Proceedings of Tri-Ada'97 (St. Louis, Missouri, November 1997), ACM Press, 21-29.

[5] ISO, Information technology - Programming languages - Ada, "Ada Reference Manual", ISO/IEC 8652, 1995.

[6] Jahanian, F. Fault Tolerance in Embedded Real-Time Systems. In Hardware and Software Architectures for Fault Tolerance. Experiences and Perspectives. Banatre, M. and Lee P. A. (eds.). Lecture Notes in Computer Science 774, Springer-Verlag, 1994, 237-249.

[7] Kermarrec, Y., Nana, L. and Pautet, L. Implementing an efficient fault tolerance mechanism in Ada 9X: an early experiment with GNAT. In Proceedings of Ada Belgium Conference (Brussels, Belgium, Nov. 1994).

[8] Kermarrec, Y., Nana, L. and Pautet, L. Providing fault tolerant services to distributed Ada95 applications. In Proceedings of Tri-Ada'96 (Philadelphia, PA, Dec. 1996), ACM Press, 39-47.

[9] Lamport, L., Shostak, R. and Pease, M. The Byzantine Generals Problem. ACM Trans. on Programming Languages and Systems, 4, 3 (July 1982), 382-401.

[10] Macos, D. and Mueller, F. The Rendezvous is Dead – Long Live the Protected Object. In Proceedings of SIGAda'98 (Washington DC, USA, Nov. 1998), ACM Press, 287-293.

[11] Miranda, J., Alvarez, A., Arévalo, S. and Guerra, F. Drago: An Ada Extension to Program Fault-Tolerant Distributed Applications. In Ada-Europe'96, LNCS 1088, Springer-Verlag, 1996, 235-246.

[12] Pinho, L. and Vasques F. Multi-µ: An Ada 95 Based Architecture for Fault Tolerance Support of Real-Time Systems. In Proceedings of SIGAda'98 (Washington DC, USA, Nov. 1998), ACM Press, 52-60.

[13] Poledna, S. Replica Determinism in Distributed Real-Time Systems: A Brief Survey. Real-Time Systems, Vol. 6, 3, 1994, 289-316.

[14] Powell, D. (ed.). Delta-4: A Generic Architecture for Dependable Distributed Computing. Springer-Verlag, 1991.

[15] Powell, D. Distributed Fault Tolerance – Lessons Learnt from Delta-4. In Hardware and Software Architectures for Fault Tolerance. Experiences and Perspectives. Banatre, M. and Lee P. A. (eds.). Lecture Notes in Computer Science 774, Springer-Verlag, 1994, 199-217.

[16] RTEMS/C Applications User's Guide. On-Line Applications Research Corporation (Sep. 1997). http://www.oarcorp.com

[17] Schneider, F. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. ACM Computing Surveys, 22, 4 (Dec. 1990), 299-319

[18] Schonberg, E. and Banner, B. The GNAT project: a GNU-Ada 9X compiler. In Proceedings of Tri-Ada'94 (Baltimore, USA, Nov. 1994), ACM Press, 48-57.

[19] Wellings, A. and Burns, A. Implementing Atomic Actions in Ada 95. IEEE Transactions on Software Engineering, 23, 2 (Feb. 1997), 107-123.