Multi-µ: an Ada 95 Based Architecture for Fault Tolerance Support of Real-Time Systems

Luís Miguel Pinho
Department of Computer Engineering
School of Engineering
Polytechnic Institute of Porto
Rua de São Tomé, 4200 Porto, Portugal
Tel.: +351.2.8340500 Fax.: +351.2.821159
e-mail: lpinho@dei.isep.ipp.pt

Francisco Vasques
Department of Mechanical Engineering
School of Engineering
University of Porto,
Rua dos Bragas, 4099 Porto CODEX, Portugal
Tel: +351.2.2041774
e-mail: vasques@fe.up.pt

1. ABSTRACT

This paper presents an architecture (Multi-µ) being implemented to study and develop software based fault tolerant mechanisms for Real-Time Systems, using the Ada language (Ada 95) and Commercial Off-The-Shelf (COTS) components. Several issues regarding fault tolerance are presented and mechanisms to achieve fault tolerance by software active replication in Ada 95 are discussed. The Multi-µ architecture, based on a specifically proposed Fault Tolerance Manager (FTManager), is then described. Finally, some considerations are made about the work being done and essential future developments.

1.1 Keywords

Ada 95, Real-Time Systems, Software Based Fault Tolerance, Off-The-Shelf Components.

2. INTRODUCTION

Dependability is an important topic in real-time systems supporting industrial applications, since reliability and availability are important issues for these applications. One of the means to achieve dependability is fault tolerance, *i. e.*, how to provide a service complying with the specification even in the presence of faults.

In real-time applications, unexpected failures of the system are not acceptable, since value or timing requirements would not be met, consequently leading to money and/or human losses. It is clear that a real-time system must provide mechanisms to tolerate faults, in order to respect its requirements. Optionally it can gracefully degrade its behaviour into a safe-state, guaranteeing that a task subset still respects its requirements.

A usual approach to achieve fault tolerance is by distributing the system elements, *i. e.* by means of structural redundancy. However there is a difference between distribution motivated fault tolerance (implementing fault tolerance in a distributed environment) and fault tolerance motivated distribution (implementing distribution to achieve fault tolerance) [11]. The latter has different requirements and the system specification must be made considering fault tolerance assumptions.

The proposed architecture (Multi-µ) targets applications where fault tolerance is an important issue, mainly due to availability and reliability requirements and not due to safety requirements. It is implemented through the active replication of processing nodes, based on the use of commercial off-the-shelf (COTS) components. Fault tolerance is achieved by means of a specially proposed software architecture, based on a layer between the application and the real-time kernel, responsible for the fault tolerance management.

This paper starts discussing some generic issues related to software based architectures for fault tolerant systems, namely concerning the active replication requirements. Afterwards, current work on Ada (from now on Ada will be used to refer the 95 standard) support for fault tolerant systems is presented and discussed. Finally, the Multi-µ architecture is presented and a Fault Tolerance Manager, which implements the fault tolerance mechanisms in Ada, is discussed. Then, some conclusions are drawn, and present and future work is described.

3. ACHIEVING FAULT TOLERANCE BY SOFTWARE ACTIVE REPLICATION

Fault tolerance techniques must be used accordingly with failure mode assumptions, so identification of which are the faults that must be tolerated by the system is of most importance for the system specification. From the Laprie [9] classification of faults, we can identify those that must be tolerated by the system:

- internal physical faults (permanent and temporary);
- temporary external physical faults;
- design faults (permanent and temporary).

Temporary external physical faults must be addressed but these can be avoided with appropriate filtering and shielding of the system. They will not be considered in the development of the software architecture.

Internal physical faults are addressed through the component replication. Permanent design faults must be tolerated and this means that there must be a way to allow for design diversity. Temporary design faults are a strange concept, but they can be tolerated because of the differences in the replicas execution environment [11], as they were temporary internal physical faults.

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

Replication management can be achieved using specialised hardware, which consequently increases the overall cost of the system. Another problem with hardware replication management is that as hardware evolves, specialised hardware must be re-designed. Conversely, the software-based replication allows the use of off-the-shelf hardware, decreasing the cost of the system, and at the same time increasing its portability and upgradability.

COTS components can be assumed to be fail-silent or fail-uncontrolled [11]. A fail-silent component is one that only fails by crashing 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 using COTS components, achieving fail-silent behaviour is only possible with the use of self-checking techniques, increasing the system cost and complexity. So, fault tolerant mechanisms must also address the components with 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, in the absence of the fail-silent assumption, wrong service delivery can only be detected by active replication. As a consequence, in the absence of the fail-silent approach active replication is the most adequate technique [10].

The use of COTS implies generally fail-uncontrolled replicas, so it becomes necessary the use of active replication techniques. This approach implies, usually, the need of replica determinism, otherwise the overhead due to replica synchronisation may largely increase.

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.

Active replication can be implemented using static groups, simplifying the necessary techniques. There is no need to consider the case of leaving or joining the group (membership), but there is still the need of agreement between the processes in the group (consensus).

Even assuming replica determinism (replicas being state machines [13]) there are some mechanisms that must be implemented to support consensus between replicas. They must guarantee that all replicas work with the same input values and that they all vote on the final output. Three problems are identified:

- achieving interactive consistency on replicated sensor data;
- achieving Byzantine agreement on single-source data;
- achieving consensus on output values.

With replica non-determinism there is still a remaining problem:

- the need of replica synchronisation in every point that can lead to execution divergence.

From a real-time system perspective, fault tolerance can be defined as the ability of a system to deliver the expected service in a timely manner, even in the presence of faults [5]. An important issue in real-time systems is that such fault tolerance mechanisms must be time bounded, in order to achieve timing predictability.

4. ADA SUPPORT FOR FAULT TOLERANT SYSTEMS

The Ada language doesn't provide direct support for fault tolerance mechanisms, apart from the exception mechanism, which can provide forward error recovery. However, exceptions can't provide tolerance to anticipated faults, or to design faults [6]. The solution is to burden the application programmer by explicit programming 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.

Kermarrec *et al* present their implementation of recovery blocks (providing backward error recovery) in GNAT [6] and recovery blocks for distributed fault tolerance using their implementation of the Ada Distributed Systems Annex, GLADE [7]. Their approach is to provide compiler extensions through a *pragma*, so as the programmer can explicitly use recovery blocks in the application to build a fault tolerant system.

Wellings and Burns [15] evaluate Ada capabilities to support fault tolerant applications. They use those capabilities to program Atomic Actions, which can be used to build fault tolerant applications. Later [1], they discuss replication, either active or passive, on the Distributed Systems Annex of Ada, providing replication mechanisms, which must be explicitly used by the application programmer.

Wolf [16] presents some issues regarding replica implementation within the partition model of Ada Distributed Systems Annex. He initially assumes replica determinism, but then extends the discussion to non-deterministic replicas. This approach is based on extending the run-time support to implement transparent replication of partitions. However, this approach is intended for fail-silent components, not being appropriate for COTS components (as discussed in the previous section).

ReplicAda [4] presents another fault tolerant implementation using Ada Distributed Systems Annex. It is based on a layer under the Partition Communication Subsystem that presents a transparent view to the programmer, hiding all the replication issues. This approach assumes replica determinism, mainly through programmer supported use of Ada mechanisms (like *pragma* Restrictions).

The replication work being done in Ada is for fault tolerance in distributed systems (distribution motivated fault tolerance), where the goal is to replicate application partitions. Conversely, the architecture here presented doesn't pretend to replicate application partitions, but the application as a whole. It is not intended for distributed systems, but it distributes the system to achieve fault tolerance.

The active replication model to achieve fault tolerance can be implemented in Ada either imposing replica determinism or keeping replica consistency at critical points, by means of consensus mechanisms.

However, guaranteeing deterministic replicas imposes several restrictions on the application programmer, excluding constructs that may cause non-deterministic replicas evolution. As each node has different execution environments, interleaving tasks, calls to protected objects or time dependencies may cause divergence between replicas.

A transparent approach to fault tolerance must provide consensus in every replica's point that can lead to non-deterministic behaviour. However, it simplifies the application programmer burden, if the fault tolerance manager can transparently implement such points. Mechanisms like recovery blocks [6] [7] and atomic actions [15], which must be explicitly supported by the application, are not appropriated in a fault tolerant transparent approach.

Considering a tightly-coupled architecture, based on hardware buses like VME or PCI, the communication overheads are small and time bounded. The necessary consensus algorithms can be efficiently implemented, and their use can be made transparent to the application programmer. This approach hides the fault tolerance mechanisms from the application programmer, not imposing replica determinism.

5. MULTI-μ ARCHITECTURE

The Multi-μ architecture targets the development of fault tolerance mechanisms for systems where reliability and availability are of most importance, and where safety is not addressed (non-critical systems). It is intended to implement a fault tolerant architecture capable of being expanded to cope with an increasing number of faults. It implements this approach by node replication with software based replica management, and fault tolerance transparent algorithms.

Its 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 mechanisms are implemented below the application, interacting with the real-time kernel. Being a tightly-coupled system it can be implemented using hardware buses like VME or PCI, and thus implementing a synchronous distributed system. The advantage of a synchronous system is that communication times are bounded, simplifying algorithms to implement fault tolerance mechanisms.

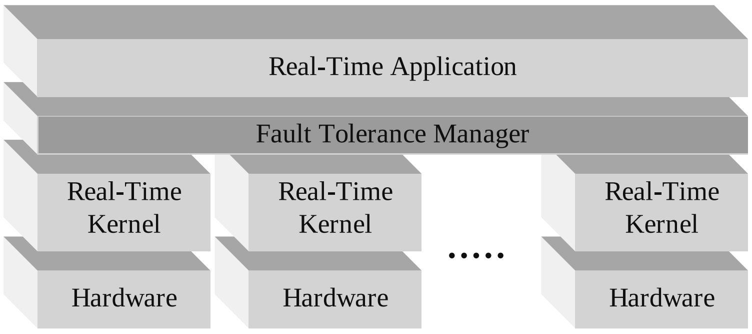

Fig. 1 - Multi-μ Architecture

Being based in a COTS real-time kernel, we keep the portability and upgradability of the system, allowing the software to manage replication without too many layers, and thus increasing the overall system performance. The Fault Tolerance Manager (FTManager) is responsible for the replica management and for the communication algorithms implementation.

Each node (figure 2) has a real-time kernel, responsible for the multitasking environment and for the communication with other nodes. The application is built on top of the compiler library, to ensure abstraction from kernel implementation, and also on top of the FTManager, providing the fault tolerance abstraction.

The selected kernel is the Real-Time Executive for Multiprocessor Systems (RTEMS) [12], and the selected Ada compiler is GNAT [14]. RTEMS is a real-time kernel suitable for real-time applications as it implements the needed features (multitasking, multiprocessing, preemptive scheduling, intertask communication, priority inheritance, etc). It has a modular architecture, and so it is possible for non-used features not to be integrated in the application code. The RTEMS tasking system will be used to support the GNAT run-time system, which is a work currently being done by RTEMS and GNAT people.

The RTEMS kernel provides communication links between nodes making use of queues. This mechanism can be used without knowledge of the physical distribution of the sender and receiver tasks, being a good framework for building replicated systems.

Both GNAT and RTEMS sources are freely available and can be adapted and extended to implement the FTManager.

6. FAULT TOLERANCE MANAGER

The FTManager is responsible for the transparent incorporation of the fault tolerance mechanisms into the application. We don't assume replica determinism, allowing the programmer to use all the Ada constructs. The FTManager has two layers:

- the Communication Manager, which is responsible for the implementation of the communication algorithms;
- the Replica Manager, which provides the necessary mechanisms for replica management, hiding its implementation from the application programmer.

Information regarding replication (replica configuration) is considered only at a final configuration phase. In such way, 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 as already referred, a different goal is intended.

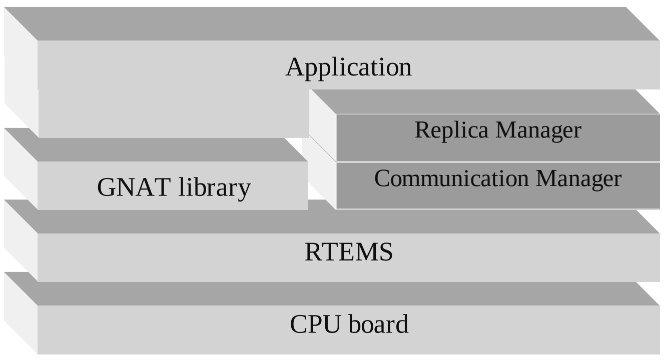

Fig. 2 – Multi-μ Node Architecture

To support consensus between replicas, communication mechanisms must be provided to support the dissemination of replica private values.

The implemented fault tolerant communication algorithm is the Signed Messages (**SM(m)**) algorithm of Lamport, Shostak and Pease [8]. A node pretending to disseminate its private value signs and broadcasts it to all other nodes. Each node, when receiving another node's value, co-signs it and sends it to all other nodes. When receiving a value that is already co-signed by a set of nodes it also co-signs it and sends the value to the nodes not yet present in the set. When a node knows that it will not receive any more messages it finally selects and delivers the correct value. As the system does not assume malicious replicas, checksums can be used to provide the required authentication.

With this algorithm each node may know another node's private value, even in the presence of, at the most, *m* faults (considering that at least *m*+2 nodes are being used) [8].

6.1 Communication Manager

The Communication Manager supports the needed communication abstractions in order to solve the earlier presented problems. As the system is tightly-coupled, communication algorithms can be made very efficient, thus simplifying the replica management mechanisms.

6.1.1 ***Rtems_Interface.Queues*** *package*

The Communication Manager is built on top of a thin Ada binding to RTEMS queues. This interface (package **Rtems_Interface.Queues**) provides a mechanism to exchange messages between nodes, without any knowledge of its content.

```
-----------------------------------------
-- Spec of package Rtems_Interface.Queues
-----------------------------------------

package Rtems_Interface.Queues is

-----------------------------------------
-- Definition of package types
-----------------------------------------

   -- Rtems queues exchange bytes,
   -- encapsulated in a vector of C long
   -- This Interface translates
   -- them to and from bytes
   type Serial_Byte is mod 2**8;
   for Serial_Byte'Size use 8;
   type Serial_Message is
      array(Positive range<>) of Serial_Byte;

   type Queue_Name is new String(1..4);
   type Queue is limited private;
-----------------------------------------
-- Public Subprograms
-----------------------------------------

   procedure Send(
         Q: Queue;
         Msg: Serial_Message;
         Size: Positive);
   procedure Receive(
         Q: Queue;
         Msg: out Serial_Message;
         Size: out Positive;
         Time: Integer;
         Timeout: out Boolean);
   procedure Blocking_Receive(
         Q: Queue;
         Msg: out Serial_Message;
         Size: out Positive);
   procedure Get_Queue(
         Q: out Queue;
         Name: Queue_Name);

private

-----------------------------------------
-- Queue Data
-----------------------------------------

   type Queue is
   record
      Name: Queue_Name;
      -- type unsigned32 is in
      -- package Rtems_Interface
      Id: unsigned32;
   end record;
end Rtems_Interface.Queues;

-----------------------------------------
-- End of Rtems_Interface.Queues Spec
-----------------------------------------
```

6.1.2 ***Group_Communication*** *package*

The **Group_Communication** package provides group abstraction to the higher layers, and also logical links between nodes, on top of the **Rtems_Interface.Queues** package. It is a generic package that can be instantiated within each particular application. Its parameters are the number of nodes in the system, the number of groups in each node, the maximum number of messages that can be received by a group, and the maximum size of data that can be exchanged.

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.

The **Send_Queue** is implemented using a protected type, giving exclusive access to the **Send** procedure. The **Receive_Queue** is implemented using a task type, which is blocked on the queue, waiting to receive, and re-routing the message to the appropriated group.

Groups are implemented using a protected type (**Send**, **Received** and **Broadcast** procedures), with only one entry (**Receive**) where a task waiting on a message can be blocked.

```
-----------------------------------------
-- Spec of generic package
-- Group_Communication
-----------------------------------------

with Rtems_Interface.Queues;
use Rtems_Interface;

generic
   No_Nodes,
   No_Groups,
   Max_Data_Size,
   Max_Messages: Positive;
package Group_Communication is

-----------------------------------------
-- Definition of package types
-----------------------------------------

   type Group_Id is new
      Integer range 1 .. No_Groups;
   type Message_Type is
      (Direct, Forward);
   type Node_Id is new
      Integer range 1 .. No_Nodes;

   type Message is limited private;
   type Message_Index is new
      Integer range 1 .. Max_Messages;
   type Message_Buffer is
      array(Message_Index) of Message;

-----------------------------------------
-- Protected type Group Spec
-----------------------------------------

   protected type Group(Ident: Group_Id) is

      procedure Send(
            Msg: Message;
            Node: Node_Id);
      procedure Broadcast(Msg: Message);
      procedure Received(Msg: Message);
      -- procedure Received is to
      -- be called by task Receive_Queue
      entry Receive(Msg: out Message);
      -- tasks waiting for a message
      -- will call Receive entry

   private

      Id: Group_Id := Ident;
      Message_Has_Arrived:
            Boolean := False;
      Buffer: Message_Buffer;
      Top, Bottom: Message_Index:=1;
   end Group;

   type Access_Group is access Group;

-----------------------------------------
-- Public Subprograms
-----------------------------------------

   procedure Initialize(Node: Node_Id);

   procedure Join_Group(
            Id: Group_Id;
            Grp: out Access_Group);

private

-----------------------------------------
-- Private type definitions
-----------------------------------------

   subtype Data_Type is
      Queues.Serial_Message(1..Max_Data_Size);

   type Message is
   record
      Source: Node_Id;
      Msg_Type: Message_Type;
      Data_Size: Positive;
      Data: Data_Type;
   end record;

   protected type Send_Queue is
      procedure Attach_Queue(
            Name: Queues.Queue_Name);
      procedure Send(Msg: Message);
   private
      My_Queue: Queues.Queue;
   end Send_Queue;

   task type Receive_Queue is
      entry Attach_Queue(
            Name: Queues.Queue_Name);
   end Receive_Queue;

-----------------------------------------
-- Private Information
-----------------------------------------

   This_Node: Node_Id;
   Groups: array (Group_Id) of
      Access_Group;

   Send_Queues: array(1..No_Nodes-1)
      of Send_Queue;
   Receive_Queues: array(1..No_Nodes-1)
      of Receive_Queue;

end Group_Communication;

-----------------------------------------
-- End of Group_Communication Spec
-----------------------------------------
```

*6.1.3 **SM_Algorithm** package*

The Package **SM_Algorithm,** a child of the **Group_Communication** package**,** implements the Signed Messages (SM(m)) algorithm [8]. It is a generic package, which can be instantiated for each data type that must be exchanged between the replicas. The only restriction is that the data type must have a defined assignment and equality, and it must be a definite type so that uninitialised objects can be declared.

Two procedures are implemented. Procedure **Provide** sends data to every other node on the system. Procedure **Agree_On** is used to agree on data replicated across the system, provided by any set of nodes.

```
-----------------------------------------
-- Spec of package SM_Algorithm
-- Child of Group_Communication
-----------------------------------------

with Ada.Real_Time;
use Ada.Real_Time;

generic
   type Value_Type is private;
   -- Assignement and Equality
   -- must be defined for Value_Type
   -- and it must be definite
package Group_Communication.SM_Algorithm is

-----------------------------------------
-- Definition of package types
-----------------------------------------
   type Source_Nodes is
      array(Node_Id range <>) of Node_Id;

-----------------------------------------
-- Public Subprograms
-----------------------------------------

   procedure Provide(
            Grp: Access_Group;
            Data: Value_Type);

   procedure Agree_On(
            Data: in out Data_Type;
            Grp: Access_Group;
            Nodes: Source_Nodes;
            Timeout: Time);

end Group_Communication.SM_Algorithm;

-----------------------------------------
-- End of SM_Algorithm Spec
-----------------------------------------
```

6.2 Replica Manager

Without assuming replica determinism, there is the need to explicitly synchronise the different replicas. The Replica Manager, besides hiding the communication algorithms from the application, must also cope with the non-deterministic behaviour of the replicas.

In an Ada application some problems can be identified that may cause divergence between replicas:

- Synchronous communication between tasks (Rendezvous). When there are several client tasks that can make a call on a server task entry, different interleaving may cause divergence;
- Asynchronous communication. When two tasks communicate asynchronously using protected objects, their different interleaving can cause different replica behaviour, such as, in one replica the reader executing after the writer, while in some other these operations may be executed in the opposite order;
- Use of the Select construct can have different results depending on the different interleaving of tasks (it is not surprisingly that Ada Select construct is often referred when there is the need for an example of any non-deterministic language construct).

To prevent these problems, the Replica Manager must synchronise the replicas behaviour. It does so by delaying a request until all nodes make the same request.

As protected objects are passive entities, when they need to synchronise accesses, a monitor task is created to receive the synchronisation requests in the same way as for replicated tasks.

The configuration of the application is made through the introduction of *pragmas* in the application code. Tools can be used to automate that job. As not all tasks need to be synchronised, off-line scheduling analysis can be used to detect precedence constraints between them, which can be captured by the scheduler.

In order to configure the application, three *pragmas* will be used:

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

The Replica Manager is, at the moment, under development.

7. APPLICATION EXAMPLE

To introduce some of Ada non-determinism problems, and to show how referred *pragmas* may be used, an application example is used. 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 there must be a monitor task to prevent it. Tasks Server, ClientA and ClientB are replicated among the system. The p*ragma* **Replicated** is used to achieve the needed consensus.

Task Server uses a Select statement to accept calls on two different entries. The *pragma* **Synchronise** is used to state the need of synchronisation between that task group, in which call is accepted. Every time that a task makes a call on a protected object, or on a server task entry, the need for synchronisation arises, so the *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
   -- By stating that Object Buffer
   -- is replicated we provide it with a
   -- monitor task guaranteeing that all
   -- requests are synchronised
   --------------------------------------
   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;

-----------------------------------------
-- Server task, with two entries,
-- use of select, and read access to the
-- protected object
-- By stating that the Task is replicated
-- it will implement a group so to
-- achieve consensous between replicas
-----------------------------------------
   pragma Replicated;
   task Server is
      entry RequestA;
      entry RequestB;
   end Server;

   task body Server is
      Data: Some_Data;
   begin
      loop
         -- Ada select causes problems,
         -- because it will choose the
         -- first entry that is ready. So
         -- it must have a synchronise
         pragma Synchronise;
         select
            accept RequestA do
               ...
            end RequestA;
         or
            accept RequestB do
               ...
            end RequestB;
         end select;
         -- Reading from the Buffer must
         -- be synchronised
         pragma Synchronise;
         Buffer.Read(Data);
         ...
      end loop;
   end Server;

-----------------------------------------
-- Client task, calling two entries,
-- write access to the protected object,
-- and agreement on replicated values
-----------------------------------------
   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);
         ...
         -- Requesting an entry must
         -- be synchronised
         pragma Synchronise;
         Server.RequestA;
         ...
         pragma Synchronise;
         Buffer.Write(Data);
      end loop;
   end ClientA;

-----------------------------------------
-- Client task, calling two entries,
-- 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.RequestB;
         ...
         pragma Synchronise;
         Server.RequestA;
         ...
         pragma Synchronise;
         Buffer.Write(Data);
      end loop;
   end ClientB;

begin
   ...
end Controller;
```

8. CONCLUSIONS

This paper proposes an architecture to study and develop software based fault tolerant mechanisms for Real-Time Systems, using the Ada language. Issues regarding fault tolerance were discussed and mechanisms to achieve fault tolerance by software active replication in Ada were presented.

The proposed architecture targets applications where fault tolerance is an important issue, mainly due to availability and reliability requirements and not due to safety requirements. It 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).

The proposed FTManager is based on a Communication Manager, which is responsible for the implementation of a Group Communication framework, and a Replica Manager, which provides the necessary mechanisms for replica management, hiding its implementation from the application programmer.

9. ACKNOWLEDGMENTS

The Authors would like to thank the anonymous reviewers for their helpful comments and suggestions.

10. 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] 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.

[6] 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).

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

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

[9] Laprie, J. L. (ed.). Dependability: Basic Concepts and Terminology. Dependable Computing and Fault-Tolerant Systems, Vol. 5, Springer-Verlag, 1992.

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

[11] 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.

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

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

[14] 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.

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

[16] Wolf, T. Fault Tolerance in Distributed Ada 95. In Proceedings of IRTAW8, Ada Letters, Vol. XVII, 5 (Sep/Oct 1997), 106-110.