Application-Level Fault Tolerance in Real-Time Embedded Systems

Francisco Afonso Carlos Silva Adriano Tavares
Department of Industrial Electronics
University of Minho
Guimarães, Portugal
{fafonso, csilva, atavares}@dei.uminho.pt

Sergio Montenegro
Compact Satellite Project
German Space Agency (DRL)
Bremen, Germany
sergio.montenegro@dlr.de

***Abstract*— Critical real-time embedded systems need to make use of fault-tolerant mechanisms to cope with operation time errors, either in hardware or software. Fault tolerance is usually applied by means of redundancy and diversity. Redundant hardware implies in the establishment of a distributed system executing a set of fault-tolerant strategies by software, and may also employ some form of diversity, by using different variants or versions for the same processing.**

This work proposes and evaluates a fault-tolerant framework for supporting the development of dependable applications. This framework is build upon basic operating system services and middleware communications and brings flexible and transparent support for application threads. A case study involving radar filtering is described and the framework advantages and drawbacks are discussed.

Keywords: Real-time systems; Fault tolerance.

I. INTRODUCTION

Real-time embedded systems are applied in several safety-critical domains as aerospace, automotive and industrial. In these applications, high dependability [1] must be a goal in the system design. Dependability involves several attributes like reliability, availability and safety, and may be achieved with fault prevention and removal at design and implementation phases. However, hardware faults, either permanent or transient ones, and residual software faults, may happen during system operation. Therefore, fault tolerance must be considered in system design, to prevent faults from becoming system failures.

Fault tolerance is usually applied by means of redundancy and diversity. Redundant hardware involves extra software coordination, which makes the software system more complex and prone to errors. Software fault tolerance may be implemented by software re-execution or multiple versions techniques, which also requires the application of additional control mechanisms.

Generally a real-time system executes a series of tasks subjected to deadlines and jitter constraints. For many of these applications, there are serious limitations in physical size and energy consumption, which imply reduced processing power and memory size.

The contribution of this work is the proposal and evaluation of a framework for developing real-time embedded fault-tolerant software. In contrast with previous works, we target the application thread level, based on a thread model commonly used for embedded systems software development.

Our approach is to provide support for the implementation of a wide variety of fault tolerant strategies at the application level with maximum transparency. For this reason, an FT framework was developed and integrated to a real-time embedded operating system (BOSS). The proposed framework was evaluated in terms of flexibility and performance in a case study involving radar filtering.

This paper is organized as follows. Section II introduces the fault-tolerant mechanisms applied in this work. Section III makes a brief description of the BOSS operating system. Section IV presents our thread model and describes the FT framework application. Section V describes the implementation of the framework. Section VI presents the case study and its results. Section VII evaluates the framework. Section VIII contains the related work, and Section IX concludes this paper.

II. FAULT TOLERANCE CONCEPTS

Fault tolerance is a means of achieving a continuous system service in the presence of activated faults [1]. Several FT strategies have been proposed and applied in the last 30 years. Some strategies are based on single version software, and can only be effective with hardware faults and transient software faults. One example is Rollback/Retry, also called "checkpoint and restart" [2]. In this strategy the detection of an error causes system to rollback at a previously saved state and to retry the same processing. This kind of technique is based on backward error recovery and needs an efficient error detection mechanism. Other strategies apply hardware redundancy to detect and mask errors, as Triple Modular Redundancy (TMR)[3], where error detection is performed by comparison of the results of multiple hardware/software units.

In order to deal with permanent software faults, multiple version software (software diversity) is needed. Several strategies have been proposed as Recovery Blocks (RB) [4], Distributed Recovery Blocks (DRB) [5] and N-Version Programming (NVP) [6].

RB and DRB perform backward error recovery like Roll-back/Retry, but use different software versions, or variants, in each execution block. The main difference between RB and DRB is the distributed nature of the latter, which allows concurrent execution of variants in two distinct nodes and coordination between them to define what node will send the final output.

NVP is a FT strategy that users forward error recovery in which multiple variants (at least 3) run sequentially or concurrently. A decision mechanism selects the correct response usually by majority voting. In a multi-computer system, each variant runs in a different node and the decision mechanism (voter) may be replicated too.

In this work, RB, DRB and NVP strategies are supported, as long as single version techniques related to them, as Roll-back/Retry, Pair of Self-Checking Processors (PSP) [7] and TMR.

III. BOSS Operating System

BOSS is a real-time operating system developed by FHG-FIRST. The BIRD (Bi-Spectral Infrared Detection) satellite [8], designed for early detection of fires, uses BOSS as its multiple-computer control operating system. BOSS has also been applied in several other projects, and future utilizations include CubeSat satellites [9] and robotics in space [10].

BOSS design has been driven by reducing software complexity as a means to achieving dependability, as complexity is the cause of most development faults. The system had several parts validated by formal verification. It was developed using object-oriented programming with C++ and it has been ported to several platforms as PowerPC, x86 and Atmel AVR. There is also available an on-top-of Linux porting, primarily used for early testing.

BOSS supports fault tolerance in hardware redundant systems, by including a middleware layer which carries out transparent communications between nodes, using the publisher-subscriber protocol. A message object can be sent locally or to the network, using a string as message subject. Receiving messages must specify which subject they are expected to receive from. Threads are usually consumers of receiving messages, by attaching to mail box objects. The middleware also supports message marshaling and the elimination of duplicate messages, based on a message identification number.

IV. Fault Tolerant Framework

This Section describes our thread model, the basic features of the tolerance framework which was integrated to the BOSS operating system to support application level fault tolerance, and how this framework is applied.

Fault tolerance can be applied to several layers of software, as at the operating system level, function/method level, object level or process level. Our work applies FT techniques to the thread level, but targeting only application threads, as operating system threads are supposed to be more robust.

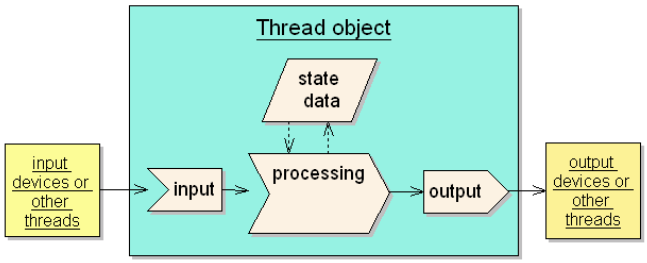

Figure 1. Thread model.

Our purpose is presenting a general description of the frame-work fault tolerance capacities and how they are employed by the application programs.

A. Thread Model

Figure 1 shows the thread model required for fault-tolerant threads. The thread to be made fault-tolerant runs in an infinite loop, reading from input devices or receiving input messages from other threads. After processing the inputs, an output is generated either by writing to an output device or sending a result message to other threads. The model supports both state threads and stateless threads. For state threads, the output result will depend both on the input data and on the previous state data.

An example of a candidate thread for fault tolerance implementation is presented in Figure 2.

```
class ExampleThread : public Thread {

  Msg* recMsg;
  Msg outMsg;
  IncommingMessageAdministrator<Msg, 20>
                              incommingMessages;
public:
  ExampleThread(){ ... // init code}

  void run () {
    while(1) {
      recMsg = incommingMessages.receive();
      process();
      output();
    }
  }

  void process(){
    ... // uses msg data and state data
  }

  void output(){
    ... // prepares output message
    outMsg.send("exampleResult");
  }
};

ExampleThread myThread;
```

Figure 2. Example of application thread.

In BOSS, all application threads must inherit from the *Thread* class and implement the *run* virtual function, which defines the thread run-time behavior. In this example,

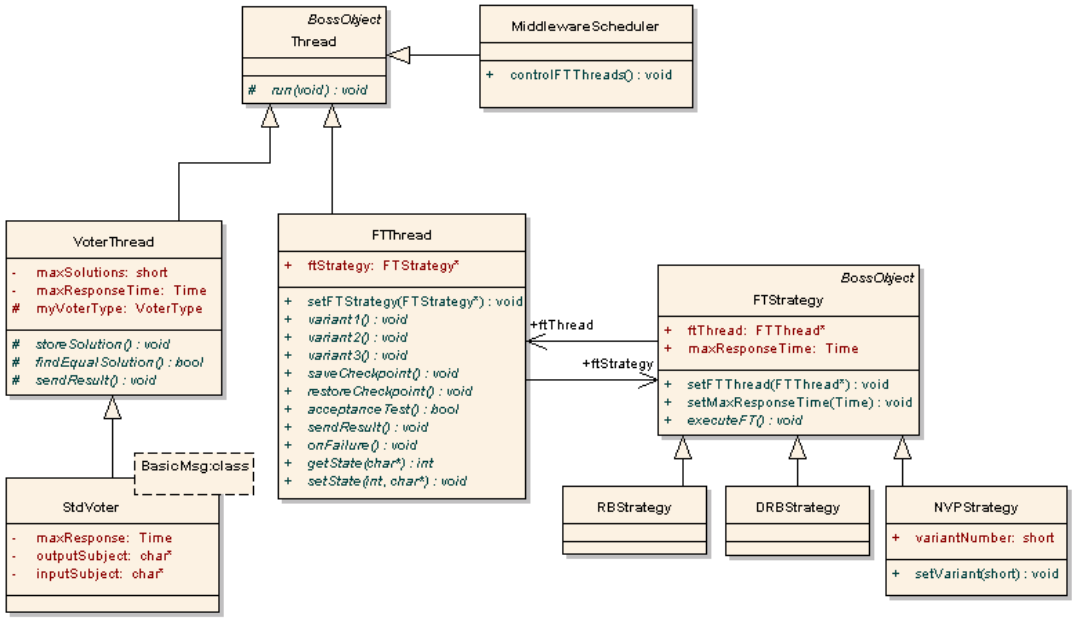

Figure 3. FT framework class diagram.

ExampleThread runs cyclically, reading messages from an *IncommingMessageAdministrator* object, which consists of a mailbox for messages of the *Msg* class. The *process* method is executed next, and implements some computing algorithm using data from the incoming message and possibly from an internal state (attributes not shown). Finally the output method prepares the output message and sends it locally and over the network, using the string *exampleResult* as subject. The instantiation of thread objects is normally static, as shown in the last line of Figure 2. Dynamic memory allocation is normally avoided for performance reasons.

B. Framework description

A simplified FT framework class diagram is shown in Figure 3. A fault-tolerant thread must inherit from the *FTThread* class and must define an *FTStrategy* object that will implement the fault tolerant functionality. Presently, three FT strategies have been implemented: RB, DRB and NVP, but others can be developed and integrated to the framework. The *FTThread* class declares several virtual functions which must be defined by the FT application thread, depending on the selected FT strategy.

VoterThread is a class designed to support the development of voters, as required in the NVP strategy. A voter thread must inherit from *VoterThread* and define some virtual functions, like *findEqualSolution*. A standard voter class (*StdVoter*) is supplied. This class provides exact voting when both inputs and outputs depend on message passing.

The *MiddlewareScheduler* (MS) class controls all FT and voter threads. This thread periodically searches for active FT/voter threads and executes part of the required control logic. Besides, this thread triggers periodic middleware messages for executing role definitions and thread state synchronization.

The modifications required to make an application thread fault-tolerant include:

- Instantiation and registration of an *FTStrategy* object that will implement the desired fault tolerance strategy, as RB, DRB and NVP.
- Execution of the *executeFT* method of the *FTStrategy* object after the thread activation.
- Implementation of application specific methods related to the selected fault-tolerant strategy (as the acceptance test in RB and DRB). Some of them consist in new functionality but others will contain the code originally defined in the processing and output methods.

Figure 4 shows an example of fault tolerant implementation for *ExampleThread*, using the DRB strategy. The main differences between this version and the original code in Figure 2 are highlighted. The thread itself now inherits from the *FTThread* class, instead of the *Thread* class. A concrete *FTStrategy* is instantiated as a *DRBStrategy* (*myDRB*). In the class constructor, the maximum response time for execution is set to 20,000 microseconds and the *setFTStrategy* method is called, assigning the address of the *DRBSstrategy* to the *ftStrategy* pointer. In the run method, the original process and output methods are replaced by a call to the *executeFT* method of the *FTStrategy* class. This method is responsible for executing the particular strategy and to activate the application specific methods defined in the application thread, as for example, *variant1* (primary block) and *acceptanceTest*. Some of these methods correspond to original implementations, but others, like *variant2* (recovery block) and *saveCheckpoint* should be defined to allow the DRB strategy operation.

```
class FTExampleThread : public FTThread {

  DRBStrategy myDRB;
  Msg* recMsg;
  Msg outMsg;
  IncommingMessageAdministrator<Msg, 20>
                              incommingMessages;
public:

  FTExampleThread(){
    ... // init code
    myDRB.setMaxResponseTime(20000);
    setFTStrategy(&myDRB);
  }

  void run () {
    while(1) {
      recMsg = incommingMessages.receive();
      ftStrategy->executeFT();
    }
  }

  void variant1(){
    ... // same code of original process method
  }

  void sendResult(){
    ... // same code of original output method
  }
  // to be defined
  void variant2(){ ...}
  void saveCheckpoint(){ ... }
  void restoreCheckpoint(){...}
  bool acceptanceTest(){...}
};
```

Figure 4. Example of FT application thread.

In this example, *ExampleThread* is stateless; otherwise *FTExampleThread* should also implement the methods *getState* and *setState*, to provide state initialization between the primary and the shadow nodes in DRB. None of these methods are necessary in the original version, as only one *ExampleThrea*d instance runs in a single node.

C. Application Specific Entities

Each FT strategy instantiation and usage demands the definition of strategy attributes and application-specific behavior. These requirements are summarized in Tables I, II and III. Table I represents requirements for multiple version software, Table II for single version software and Table III for voters.

The fault tolerance strategies in Table II use the same FTStrategy objects of RB, DRB and NVP, but do not implement their full functionality, as several methods don't need to be defined. In these cases they inherit a default implementation. For example, the default implementation for *save/restoreCheckpoint* is empty and for *acceptanceTest* is to return true (success).

The simplest FT strategy in Table II is the *Restart* strategy. In this technique only one variant is defined, and the acceptance test is not implemented. Therefore, the only possible error detection is deadline expiration, which is set by the *Response Time* parameter. *Rollback/Retry* can be implemented as single version simplification of the RB strategy. In this case, only one real variant is defined, and the body of *variant2* should contain a call to the variant1 method. In a similar way, PSP is implemented with the DRB strategy and TMR with the NVP strategy.

TABLE I. MULTIPLE VERSION STRATEGIES REQUIREMENTS.

Definition Requirements		**RB**	**DRB**	**NVP**
Entity	**Type**			
FT Strategy	object	RBStrategy	DRBStrategy	NVPStrategy
Response time	parameter	Yes	Yes	Yes
variant 1	method	Yes	Yes	Yes
variant 2	method	Yes	Yes	Yes
variant 3	method	-	-	Yes
saveCheckpoint	method	Yes	Yes	-
restoreCheckpoint	method	Yes	Yes	-
acceptanceTest	method	Yes	Yes	-
sendResult	method	Yes	Yes	Yes
onFailure	method	Optional	Optional	Optional
Voter Thread	object	-	-	Yes
getState	method	-	state threads only	state threads only
setState	method	-	state threads only	state threads only

TABLE II. SINGLE VERSION STRATEGIES REQUIREMENTS.

Definition Requirements		**Restart**	**Rollback/ Retry**	**PSP**	**TMR**
Entity	**Type**				
FT Strategy	object	RBStrategy	RBStrategy	DRBStrategy	NVPStrategy
Response time	param.	Yes	Yes	Yes	Yes
variant 1	method	Yes	Yes	Yes	Yes
variant 2	method	-	-	-	-
variant 3	method	-	-	-	-
saveCheckpoint	method	-	Yes	Yes	-
restoreCheckpoint	method	-	Yes	Yes	-
acceptanceTest	method	-	Yes	Yes	-
sendResult	method	Yes	Yes	Yes	Yes
onFailure	method	Optional	Optional	Optional	Optional
Voter Thread	object	-	-	-	Yes
getState	method	-	-	state threads only	state threads only
setState	method	-	-	state threads only	state threads only

TABLE III. VOTER REQUIREMENTS

Definition Requirements		**Application Dependent Voter**	**Standard Voter**
Entity	**Type**		
Response Time	parameter	Yes	Yes
Coordination method	parameter	Yes	Yes
Input subject	parameter	-	Yes
Output subject	parameter	-	Yes
storeSolution	method	Yes	-
findEqualSolution	method	Yes	-
sendResult	method	Yes	-

The *onFailure* method in Tables I and II is always optional. It can be used to define application dependent fault handling mechanisms when a failure in the strategy execution occurs. After running the code defined in the *onFailure* method, the thread will be restarted by the operating system.

Table III displays the requirements for voting threads. These threads are only needed when using TMR or NVP. In the general case, a voter is application specific and this thread must implement the *VoterThread* methods shown in Table III. However, for exact majority voting using messages, a *StdVoter* class which compares results byte by byte may be used. Using this standard voter, some other parameters must be defined, as

the subject of the input and output messages. The coordination method parameter defines if all replica voters will execute the *sendResult* method or if only a master voter will do it. The definition of the master voter in a coordinated voting is performed by the FT framework.

V. IMPLEMENTATION

The *MiddlewareScheduler* (MS) thread runs at the beginning of every clock tick interval (e.g. 1ms) and controls the behavior and execution of each FT thread and voter. Besides, this thread is also responsible for activating other middleware threads, as the one that delivers the new external incoming messages.

Figure 5 shows an example of the execution of a RB thread. The MS thread runs periodically and releases message reception each two activation periods. The message reception is not executed in every cycle in order to reduce CPU utilization and to provide at least one period in two for FT threads free execution. In the first cycle, the RB thread receives a message and starts the FT execution. This example shows a failure in the primary block and a success in the recovery block.

Figure 6 contains a activity diagram showing the interaction between the FT Thread and the *MiddlewareScheduler* thread in the execution of the RB strategy. After setting up a deadline for execution, based on the actual time and the maximum allowed response time, the thread suspends. In subsequent MS activations, this thread verifies if the deadline has expired and, in that case, restarts the RB thread. This represents a failure in delivering the correct response on time, but after restarting, the RB thread is ready again for receiving the next request or activation. If the deadline has not expired, the MS thread commands the next actions to be performed by the RB thread and schedules it for execution. After executing the right operations (save/restore state, run primary/recovery block, run acceptance test) the RB thread suspends again and the MS thread checks the acceptance test (AT) result. If the RB thread succeeds in AT, the MS thread allows it to send the results and the interaction finishes. If the RB thread fails in both blocks it is restarted by the MS thread.

As seen in Figure 6, the control algorithm of an FT Strategy is executed in two separate threads: the FT thread and the MiddlewareScheduler thread. However, all this code is defined in the FTStrategy concrete class. As seen in the class diagram of Figure 7, every FTStrategy must implement the *executeFT* method, which corresponds to the FT thread execution code, and the *executeMSControl* method, which is called by the MS.

Using this approach, the *MiddlewareScheduler* does not

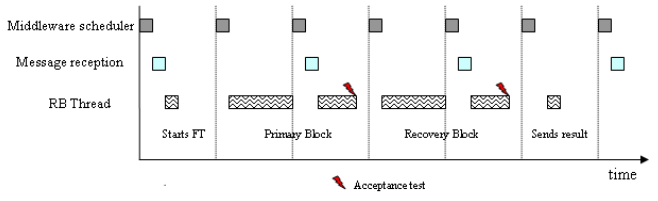

Figure 5. RB execution timing example.

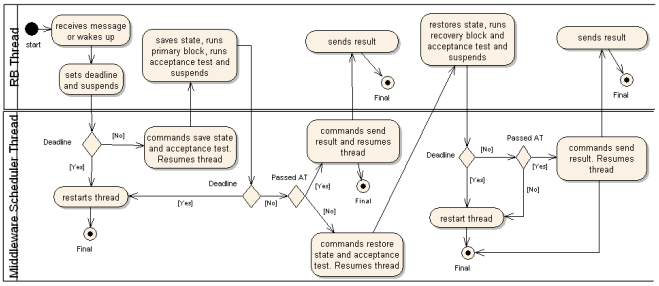

Figure 6. RB execution activity diagram.

depend on any FT Strategy implementation, and FT strategies can be added to the framework transparently.

In contrast with the RB strategy presented so far, other FT strategies involve the utilization of multiples instances of the FT thread, running in different nodes. These FT threads have to communicate in order to establish roles and initialize states. In this framework, the required communication between FT threads is executed by message passing between the *MiddlewareScheduler* threads in each node. If an FT thread needs to send a message it calls the *sendMiddlewareMessage* method of MS. Then, the message is broadcasted to all other nodes and their MS threads will distribute it to the related FT threads in their nodes, if any, by calling the *processMiddlewareMessage* of corresponding *FTStrategy*.

Another feature performed by *MiddlewareScheduler* is the activation of *FTStrategies* periodically (e.g. 300ms) in order to trigger the execution of periodic tasks as, for instance, role conflicts detection in the DRB strategy. In that case, the *startPeriodicMsg* of *FTStrategy* is called.

Finally, the *MiddlewareScheduler* thread is responsible for changing the FT threads priorities according to the Earliest Deadline First (EDF) scheduling. Therefore, in each MS activation the FT thread with earliest deadline is found and its priority is raised to a maximum among application threads. This feature can be enabled or disabled in the framework.

The MS thread also controls *VoterThreads* execution, in a similar way of the control of FT Threads. However, the MS control is simpler, as only detects if the voting deadline has elapsed. The algorithm implemented in VoterThread uses single match voting. Upon receiving a solution message, the voter thread compares the solution with the previous ones just received and if a match is found (*findEqualSolution* method) it is considered as correct and the output is immediately sent. In

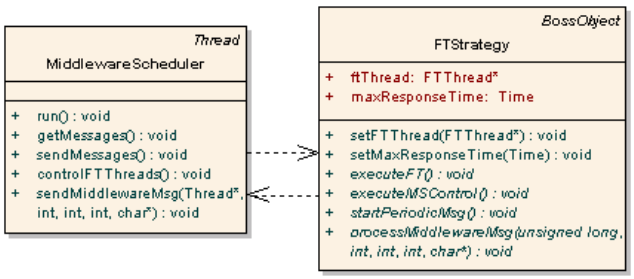

Figure 7. FTStrategy execution methods.

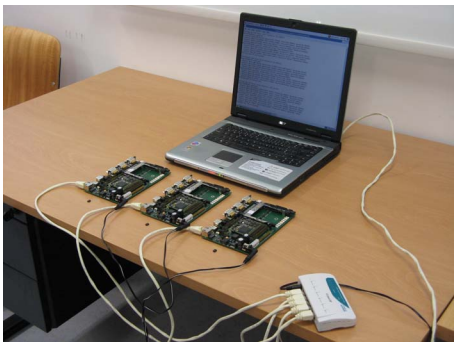

Figure 8. Case study equipment.

this case, further messages are discarded. If only one solution message arrives and the deadline occurs, this solution is also considered correct and it is sent as the output. For the implementation of voting, it is required sequential message identification, already supported by the middleware. Voter threads can also send and receive middleware messages for establishing the role in coordinated voting (Master or Slave).

VI. CASE STUDY: RADAR FILTERING SYSTEM

We applied this FT framework in the development of a radar filtering system. This setup is composed of a portable PC and three PowerPC 823 boards connected by an Ethernet network, as shown in Figure 8. The portable PC runs an on-top-of Linux implementation of BOSS and simulates a radar system, generating detection data of several planes periodically. The data generation includes simulated errors in bearing and distance, typical of this kind of equipment. This data is received by the PowerPC 823 boards running an application that filters the planes' position, using an alpha-beta filter, and also calculates the planes' course and speed. The results are sent back to the portable PC, where they are displayed by a Java GUI program.

Three basic configurations are applied, as shown by the UML deployment diagrams of Figures 9, 10 and 11. The first configuration uses a single node version of the filtering application, without any fault tolerance mechanism. The other configurations implement the PSP and TMR strategies. In these figures, broadcast messages are represented by buses with the message subject on top. The *Radar* thread in the PC always sends its data using *radar_data* as subject. Similarly, the final result data from the PowerPC boards are sent to the Display

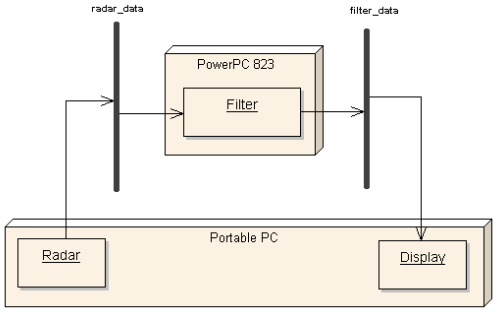

Figure 9. Non-FT configuration

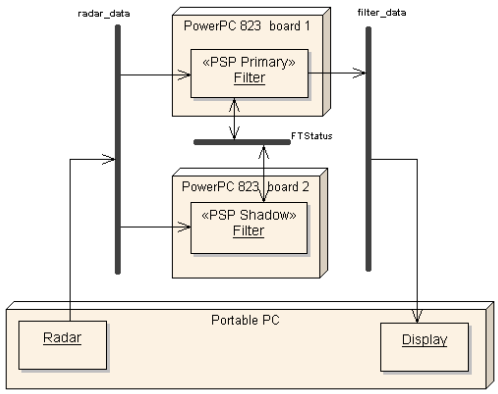

Figure 10. PSP configuration

with *filter_data* as subject. For the PSP configuration in Figure 10, both Filter threads receive the radar data and execute the computation, but only the primary thread sends its results. In the TMR configuration of Figure 11, all Filter threads send their results with *unvoted_data* as subject, which are received by the voter threads. In this particular configuration, coordinated voting is used and so only the master voter thread sends the final results to the Display thread. Messages between FT threads and voters are sent with *FTStatus* as subject. Figure 12 presents an example of display output for the TMR configuration.

In the FT configurations, hardware faults were simulated by turning PowerPC boards off and software faults were simulated by introducing value errors in the filter calculation. In the PSP configuration, a hardware fault in a board running as primary causes a switch to primary in the other node. A software fault is detected by the acceptance test, and a rollback and retry is performed with the same algorithm. If the simulated fault is still present, the PSP thread will restart. For the TMR configuration, a hardware fault in the board with the master voter will imply in a new master voter board after the next master election. A software fault in one of the boards will be masked by the voter mechanism.

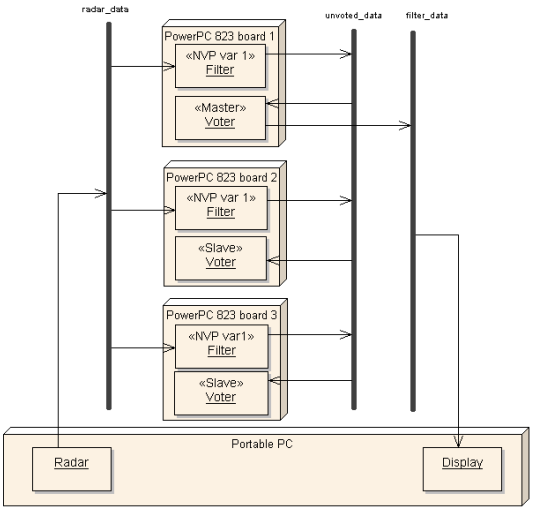

Figure 11 TMR configuration.

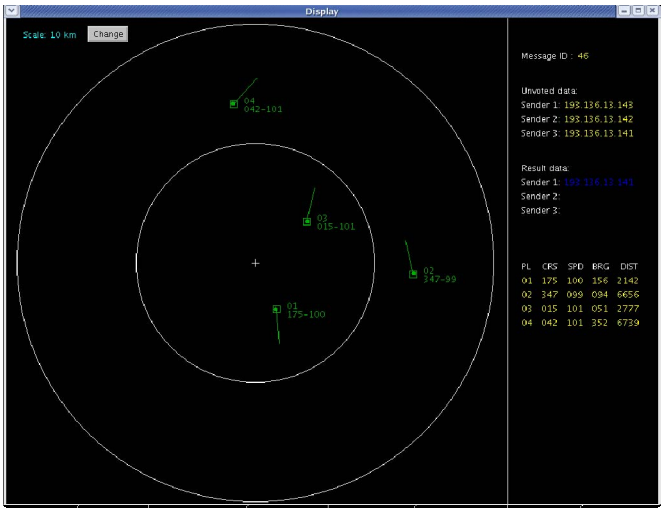

Figure 12. Display example.

If a board is initialized or if a FT thread is restarted, distributed filter threads must initialize its state, as the filter output depends on the planes' last position and alpha-beta parameters. This initialization algorithm is performed by the corresponding *FTStrategy* object, transparently to the application program, which has only to define the *getState* and *setState* methods.

A. Performance

The radar simulation periodically sends planes´ data every 2 seconds. This corresponds to the rotation period of the radar antenna. To test the system under more severe timing conditions and compare the performance of the FT configurations we varied the radar simulation period, reducing it by factors of two. Figure 13 shows performance results in terms of CPU utilization for several configurations and simulation frequencies varying form 0.5 Hz (2 seconds) to 32 Hz (31.25 ms).

The curves labeled "Non-FT" are related to the non-fault

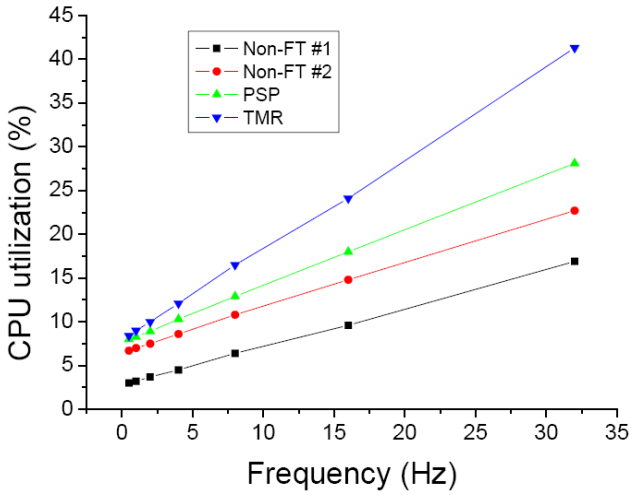

Figure 13. Performance results.

tolerant single node version shown in Figure 9. The "Non-FT #1" setup employed an operating system version with no FT framework, while in #2 the FT framework was integrated. We can notice that the utilization of the FT frameworks implies in a performance cost of about 3%. This difference is due to the periodic activation of the *MiddllewareScheduler* thread in addition to the operating system scheduler.

The PSP and TMR configurations present poorer performance than Non-FT configurations as expected. The reason is the extra processing time associated with the FT control, application specific procedures and message communication. The TMR configuration achieved the worst results as it demands more threads for voting and more message exchanges.

We conclude that the performance cost of fault tolerance implementation is still acceptable, considering the benefits in system reliability. However, for systems already demanding high CPU utilization, the introduction of fault tolerance might be a problem, and special care must be taken, including in selection of the FT strategy.

VII. Evaluation

The utilization of a FT framework in the development of embedded fault-tolerant systems has several benefits:

- Simplifies the application level programming, as programs don't have to implement fault-tolerant mechanisms, and only have to provide application specific parameters and procedures.
- The application program follows a standard structure in which changing the FT strategy becomes easy and straightforward. This reduces efforts in strategy selection, configuration and testing.
- Facilitates the creation and integration of new fault tolerance strategies. The proposed framework is easily extendable by adding new *FTStrategy* and *VoterThread* derived classes.
- Provides a means of implementing adaptive fault tolerance [11], as changing the FT strategy can be performed at run-time by simply calling the *setStrategy* method. The strategy can be modified based on the reliability requirements of each the mission phase, or even for other factors as resource availability and power consumption.

An addition benefit of this framework is that it makes easy to modularize all fault tolerant code using Aspect Oriented Programming (AOP) [12]. This framework has been applied in [13] to provide a full separation between the application functionality and the fault tolerance concern.

The drawbacks of the FT framework are not related with the framework itself but with the fault tolerance domain. This includes the usage of more physical resources and, for each system node, more run-time overhead and memory consumption.

VIII. Related Work

Several patterns and frameworks for fault tolerance design using object-oriented approaches have been proposed in the last ten years [15][16][17][18]. In all them, concepts as checkpointing, try-blocks, acceptance tests, versions and voters are represented by classes. Each proposal has its own class structure, using abstract classes to represent more general constructs as variants and abjudicators. Some common patterns are used in these frameworks, like the composite pattern [14] as in [17] and [18]. In general, these proposals do not address thread models and distributed architectures.

Few implementations of fault tolerance support by the operating system or by a middleware were found.

FT-RT-Mach, an academic general purpose operating systems, and the DEOS operating system, a certified operating system for critical avionics applications, use re-execution of tasks as the primary method for achieving fault tolerance [19]. In these systems, an error can be detected either by an acceptance test or any other exception, and the operating system scheduler tries to guarantee the rescheduling of the thread before its deadline.

ROAFTS (Real-Time Object-Oriented Adaptive Fault Tolerant Support) is a middleware architecture that can support several strategies of fault tolerance, like RB and DRB, and dynamically switches the units operating mode in response to changes in the resource and application modes [20]. This middleware is applied as a component of the TMO (Time-triggered Message-triggered Object structuring scheme) model of computation [21] where the basic units of computation are time-triggered and service methods of real-time distributed objects.

Despite having the same goal of this work, the systems described target large-scale critical systems, and do not fit into embedded systems applications because of its intense resource utilization and complexity.

IX. Conclusion

We have presented a framework for supporting the development of application-level fault tolerance for real-time embedded systems. The framework already implements a wide set of FT mechanisms and extending the framework is considered very simple. Other advantages include easiness of configuration and high flexibility both at compile and run-time.

Additionally, a case study using radar filtering was presented and the performance of several configurations were measured and compared. The extra costs involved with the introduction of FT mechanisms are considered acceptable for systems demanding high dependability.

Acknowledgment

This work has been supported by the Portuguese Foundation for Science and Technology (FCT).

References

[1] A. Avizienis, J.-C. Laprie and B. Randell, “Fundamental Concepts of Dependability,” *in Technical Report 739*, Department of Computing Science, University of Newcastle upon Tyne, 2001.

[2] D.K. Pradhan, Fault-Tolerant Computer System Design, Prentice-Hall, Inc., 1996.

[3] B. Randell, P. Lee and P.C. Treleaven, “Reliability Issues in Computing System Design,” *in ACM computing Surveys*, vol. 10, issue 2, pp. 123-165, 1978.

[4] B. Randell, “System Structure for Software Fault Tolerance,” *in IEEE Trans. Software Engineering*, vol. 1, no.2, pp. 220-232, June 1975.

[5] K. Kim and O. Welch, “ Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications,” *in IEEE Transactions on Computers*, vol. 38, Nº 5, pp. 626-636, 1989.

[6] L. Chen and A. Avizienis, “ N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation,” *in Proceedings of FTCS-8*, pp. 3-9, Toulouse, France, 1978.

[7] K. Kim, “Toward Integration of Major Design Techniques for Real-Time Fault-Tolerant Computer Systems,”, *in Journal of Integrated Design and Process Science*, vol. 6, issue 1, pp. 83-101, 2002.

[8] S. Montenegro and F. Zolzky, “BOSS /EVERCONTROL OS/Middleware Target Ultra High Dependability,” *in Proceedings of Data Systems on Aerospace -DASIA*, Edinburgh, Scotland, 2005.

[9] S. Montenegro, K. Briess and H. Kayal, “Dependable Software (BOSS) for the BEESat Pico Satellite,” in Proceedings of Data Systems on Aerospace - DASIA, Berlin, Germany, 2006.

[10] P. Massa, et al.,”HiPeRCAR: the High Performance Resilient Computer for Autonomous Robotics,” *in Proceedings of Data Systems on Aerospace - DASIA*, Berlin, Germany, 2006.

[11] M. Hecht, H. Hecht and H. Shokri, "Adaptive fault tolerance for spacecraft," *in Aerospace Conference Proceedings*, IEEE 2000 , vol.5, pp.521-533 vol.5, 2000.

[12] G. Kiczales et al., “Aspect Oriented Programming*,” in Proceedings European Conference on Object-Oriented Programming - ECOOP´97*, LNCS 1241, pp. 220-242, 1997.

[13] F. Afonso, C. Silva, N. Brito, S. Montenegro and A. Tavares, “Aspect-Oriented Fault Tolerance for Real-Time Embedded Systems,” unpublished.

[14] E. Gamma, R Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[15] K. Tso, E. Shokri, A. Tai and R. Dziegiel, “A Reuse Framework for Software Fault Tolerance,” *in Proceedings of AIAA 10th Computers in Aerospace Conference*, San Antonio, March, 1995.

[16] F.. Daniels, K. Kim, and M.A. Vouk, “The Reliable Hybrid Pattern - A Generalized Software Fault Tolerant Design Pattern,”, *in Proceedings of Pattern Language of Programming Conference - PLOP'97*, 1997.

[17] R. Duncan and L. Pullum, “Object-Oriented Executives and Components for Fault Tolerance,” *in IEEE Proceedings of Aerospace Conference*, vol. 6, pp. 2849-2855, 2001.

[18] J. Xu, B. Randell and A. Romanovsky, “A Generic Approach to Structuring and Implementing Complex Fault-Tolerant Software,” *in Proceedings of the 5th International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 207-214, 2002.

[19] L. Dong et al., “Implementation of a Transient-Fault-Tolerance Scheme on DEOS,” *in Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*, pp. 56-65, 1999.

[20] K. Kim, “ROAFTS: A Middleware Architecture for Real-Time Object-oriented Adaptive Fault Tolerance Support,” *in Proceedings of the 3rd IEEE International High-Assurance Systems Engineering Symposium*, pp. 50-57, Washington, D.C., 1998.

[21] K. Kim, M. Ishida and J. liu, “An Efficient Middleware Architecture Supporting Time-Triggered, Message-Triggered Objects and an NT-based Implementation,” *in Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 54-63, 1999.