Implementation of middleware fault tolerance support for real-time embedded applications

F. Afonso[1], C. Silva[1], S. Montenegro[2], A. Tavares[1]
[1]*Department of Industrial Electronics, University of Minho, Portugal*
[2]*Fraunhofer Institute for Computer Architecture and Software Technology, Germany*
[1]*{fafonso, csilva, atavares}@dei.uminho.pt,* [2]*sergio@first.fhg.de*

Abstract

Critical real-time embedded systems need to apply fault tolerance strategies to deal with operation time errors, either in hardware or software. In this paper we present the ongoing work to provide application fault tolerance by means of implementing middleware transparent support over the BOSS embedded operating system. The middleware uses a publisher-subscriber protocol and enables the execution of several fault tolerance strategies with minimum burden to the application level software[*].

1. 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. However, despite all efforts to prevent and remove faults during system development, some sort of fault tolerance is required to deal with residual software faults and hardware faults at run-time.

Fault tolerance is usually achieved by redundancy and diversity. Hardware redundancy and software diversity are the most common techniques for increasing system reliability, but several other techniques may be applied, as time redundancy (task re-execution), information redundancy (correction codes) and data diversity (data re-expression).

The purpose of this work is to support fault-tolerant (FT) strategies in applications developed using the BOSS embedded operating system. As critical applications are usually implemented with multi-computer systems connected by one or more networks, the middleware was the selected layer of software to deliver transparent fault tolerance support to applications. However, the operating system kernel had to be modified, because FT was expected to work at the thread level.

2. BOSS operating system

BOSS is a real-time embedded operating system designed for applications demanding high dependability [2]. Simplicity is the main strategy for achieving dependability in BOSS, as complexity is the cause of most development faults. The system was developed in C++, using an object-oriented framework simple enough to be understood and applied in several application domains.

The BIRD Satellite, designed for early detection of fires, uses BOSS as the multi-computer control operating system. BOSS has been ported to different projects and platforms as PowerPC, x86 and Atmel AVR. It also runs on top of Linux, mainly for developing and testing purposes

BOSS was designed to support fault tolerance in applications with hardware redundancy by including a middleware which carries out transparent communications between nodes. The messages exchange is asynchronous, using the publisher-subscriber protocol. Using this approach, no fix communication paths are established and the system can be reconfigured at run-time easily.

3. Fault tolerance strategies

Several FT strategies have been proposed and applied in the last 30 years. The simplest strategy is Rollback/Retry, also called “checkpoint and restart” [3], which uses time redundancy. This strategy is effective only against transient faults, like hardware transient faults caused by electromagnetic radiation,

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

and even some software transient faults like as race conditions.

In order to deal with permanent software faults, other 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 Rollback/Retry, but use different software versions, or variants, in each execution block. In these techniques there are at least two software versions in RB and two software versions in DRB, which must deliver similar correct results. The main difference between RB and DRB is the distributed nature of the later, allowing concurrent running 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 and 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, Rollback/Retry, RB, DRB and NVP strategies are supported. In fact, the Rollback/Retry strategy can be implemented as an RB strategy, if we define the primary and the recovery block as the same. For NVP, this work will support the decision mechanism implementation only, as the variants can be implemented by normal application threads running in different nodes and sending the results to the voting thread.

4. Design

The main design goal is to provide the fault tolerance strategies just presented with minimal burden to the application level.

A new middleware periodic thread was created to control and schedule all FT threads. This thread, called MiddlewareScheduler (MS), runs at the beginning of every clock tick interval and defines the behavior of each FT thread.

4.1. RB design

For the execution of a RB fault-tolerant strategy with two variants, the application thread has to define the implementation of the following procedures: primary block, recovery block, save state, restore state, acceptance test and send result.

Figure 1 presents an example of a RB run when the primary block fails and the recovery block succeeds.

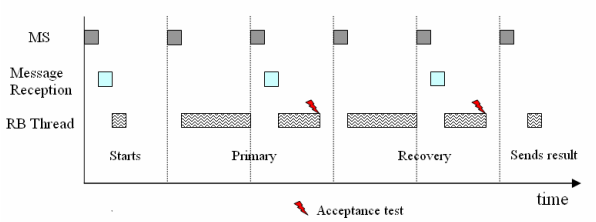

Fig 1. RB execution example

The operation is started by the application thread upon receiving an input message or waking up at a specific time. After setting up a deadline for execution, based on the actual time and the maximum allowed response time, the thread suspends. In subsequent MS thread activations, this thread verifies if the RB thread deadline has expired and, in that case, restarts the 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 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.

4.2. DRB design

For applying the DRB strategy, the DRB thread should define the same procedures of the RB strategy but the DRB execution involves the coordination between two nodes for delivering a unique result to the system. This coordination is performed by message exchanges between the middleware of both nodes, without any intervention from the DRB threads.

Figure 2 presents a general representation of DRB message exchanges. The “AT OK” message is sent by the MS thread if the primary node has succeeded in one of the two blocks. After sending this message, the MS thread releases the primary DRB thread to send its results, which could imply in sending an “output message” to another node. If the primary node fails in both AT or is unable to terminate before its deadline, no message is sent to the shadow node. In that case, the DRB primary node will be restarted and it will change its role to shadow, while the shadow node will send its results just after its deadline, and it will assume as the primary node. In case of failure of both primary and shadow nodes, no output will be released and both threads will be restarted as shadow nodes. In

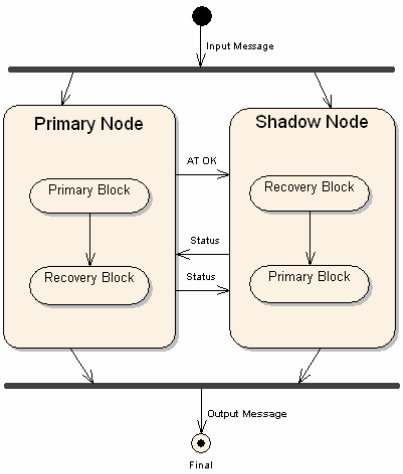

Fig. 2. DRB message exchange

order to avoid this condition, an agreement protocol had to be established to detect role conflicts and set up alternate roles. This involves periodic status message exchanges between MS threads when the DRB execution is not active, and a conflict solution procedure based on the order of the node identifications. These messages are represented in Figure 2 as "Status" messages.

4.3. NVP design

For voting support in the NVP strategy, an application voter thread will have to implement the procedures for storing a received solution, comparing solutions and sending the correct results.

The proposed algorithm uses single match voting. Upon receiving a solution message, the voter thread compares the solution with the previous ones just received and if one "equal" solution is found it is considered as correct and the output is immediately sent. In this case, further messages are discarded. If only one solution message arrives and a deadline occurs, this solution is also considered correct and it is sent as the output. For the implementation of voting sequential message identification is required.

Two types of voter threads were provided: a free voter and a coordinated voter. The free voter is used when multiple replicas of the voter thread can send its results disregarding the presence of other voters. The coordinated voter is used when the voting output must be unique among the replicas, like in the configuration of Figure 3, and involves the establishment of a master voter thread. The voter role definition, master or slave, is carried out by the middleware, by exchanging periodic status messages between nodes with coordinated voting threads.

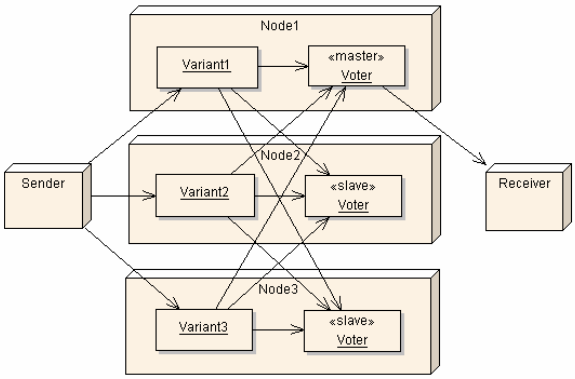

Fig. 3. Coordinated voters

In all strategies presented, the scheduling of FT threads is performed by the MS thread, which selects the active FT thread with the earliest deadline and increases its priority to a value greater than all priorities in the system, with exception to middleware threads. After finishing the FT execution, FT threads priorities are changed to its initial priority, and will remain with this priority until the next FT activation.

5. Implementation

Two implementations were provided. The first was based on object-oriented inheritance and application threads are defined by single and direct inheritance of classes Thread (for common threads), RBThread, DRBThread or VoterThread. These FT base classes define application specific procedures as virtual functions and provide an empty (stub) implementation for them. Therefore, FT application threads must overwrite these methods. The non-virtual functions of these base classes define the operation of the fault-tolerant strategy in coordination with the MiddlewareScheduler thread using the data members of these classes to exchange information.

The second implementation uses callback functions instead of virtual functions. In this implementation, FT application threads inherit directly from the Thread class. However, FT threads should call the function defineAsFT, passing its type and callback pointers for all application functions needed, also including a callback for the function which runs the fault-tolerant strategy itself. The static functions executeRB, executeDRB and executeVoting are provided in the Thread class as default implementations for the FT strategies. Stub implementations are provided for application dependent functions.

Besides its better performance, this implementation allows changing the type of a FT thread at run-time, as long as the definition of all application specific and strategy specific functions.

6. Results

Coding and testing was carried out in the Linux environment, using an on-top-of Linux implementation of BOSS. In this configuration, BOSS kernel and the application itself were compiled into a single executable and run as a Linux process with FIFO scheduling and maximum priority. Three Pentium computers, connected by an Ethernet network, were used. The MiddlewareScheduler thread activation period was set to 1 ms and network incoming messages were delivered each 2 ms. Communication was implemented using UDP sockets and broadcast.

The three FT strategies (RB, DRB and NVP) where tested in both implementations using a sorting application. Each variant was implemented in a different sorting algorithm as Bubble Sort, Insertion Sort and Selection Sort. A random array of 2000 elements was generated and published by a Sender thread. In each configuration, a FT thread running RB or DRB strategy or a normal thread sorted this array and sent it to an actuator or a voter thread (for NVP). Faulty conditions were generated by introducing unsorted values after sorting in each variant at compile time. System results were checked by logging all messages and principal function events as thread roles changing in DRB and coordinated voting.

7. Related work

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 [7]. Rate Monotonic Scheduling and Admission Control of threads are performed by both operating systems.

ROAFTS (Real-Time Object-Oriented Adaptive Fault Tolerant Support) is a middleware architecture developed by University of California [8]. It was designed to run over commercial operating systems as UNIX and Windows NT. The middleware supports the RB and DRB strategies, and dynamically switches the units operating mode in response to changes in the resource and application modes. This middleware is applied as a component of the Time-Triggered Message-Triggered Object structuring scheme (TMO) model of computation [9].

Despite having the same goal of this work, these systems do not fit to small-scale embedded systems applications because of its intense resource utilization.

8. Summary and future work

We have presented the work in progress in implementing of fault tolerance support mechanisms for the BOSS embedded real-time operating system using middleware technology. The main goal of the work is adding fault tolerance functionality with minimum complexity and resource commitment in order to satisfy the requirements of high-dependable embedded systems.

Future work will include investigating the application of aspect-oriented programming (AOP) for supporting fault tolerance strategies, improving the customization of the application and its adaptability.

References

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

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

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

[4] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Software Engineering*, vol. SE-1, pp. 220-232, June 1995.

[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," *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," *Proceedings of FTCS-8*, pp. 3-9, Toulouse, France, 1978.

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

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

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