Middleware Fault Tolerance Support for the BOSS Embedded Operating System

F. Afonso[1] , C. Silva[1], S. Montenegro[2] and A. Tavares[1]

[1]Department of Industrial Electronics,
University of Minho, Guimarães, Portugal
{fafonso,csilva,atavares}@dei.uminho.pt

[2]Fraunhofer Institute for Computer Architecture and Software Technology (FIRST),
Berlin, Germany
sergio@first.fhg.de

Abstract *— Critical embedded systems need a dependable operating system and application. Despite all efforts to prevent and remove faults in system development, residual software faults usually persist. Therefore, critical systems need some sort of fault tolerance to deal with these faults and also with hardware faults at operation time.*

This work proposes fault-tolerant support mechanisms for the BOSS embedded operating system, based on the application of proven fault tolerance strategies by middleware control software which transparently delivers the added functionality to the application software. Special attention is taken to complexity control and resource constraints, targeting the needs of the embedded market.

1 Introduction

Real-time operating systems must provide support to applications in order to satisfy the time requirements they are subjected to. Furthermore, both the operating system and the application itself need to be dependable. Dependability involves several attributes[1] like reliability, safety and security, 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 always implies the use of some sort of redundancy. Hardware redundancy is the most used type of redundancy, but several others may be applied, such as software redundancy (several software versions), time redundancy (re-execution) and information redundancy (e.g. correcting codes).

This work provides the BOSS operating system with mechanisms for achieving fault tolerance at the application level, by using fault-tolerant middleware and operating system support.

The remainder of this paper is structured as follows: section 2 introduces the BOSS operating system and its principal features, section 3 covers the principal concepts in fault-tolerant strategies used by this work, section 4 describes the design and section 5 the implementation of these strategies by the middleware, section 6 presents the test configurations and results, section 7 covers the related work and section 8 concludes this paper.

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. Although targeting minimal complexity, no fundamental functionality is missing, as its microkernel provides support for resource management, thread synchronization and communication, input/output and interrupts management. The system is fully preemptive and uses priority based scheduling and round robin for same priority threads.

The BIRD Satellite, designed for early detection of fires, uses BOSS as the multicomputer 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. For instance, several replicas of the same software can run in different nodes and publish the result using the same subject, without knowing of each other. A voter thread may subscribe to that subject and vote on the correct result, as represented in Figure 1. It is also possible to replicate the voter thread and even place a voter thread and a controller thread in the same node. All message communication between threads is taken by the middleware using a broadcast bus interface.

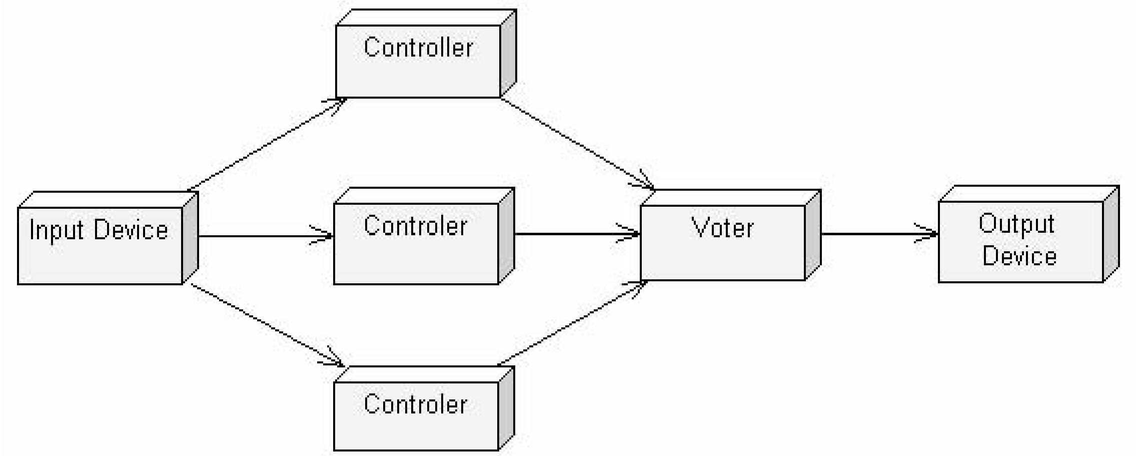


Figure 1: Simple Voting

3 Fault Tolerance Concepts

This work aims to extend BOSS functionality by providing support for application level fault tolerance using several predefined strategies, such as Recovery Blocks (RB)[3][4] and Distributed Recovery Blocks (DRB) [5]. These techniques make the system fault-tolerant to either hardware or software faults. Regarding persistence, faults may be classified as permanent or transient, but transient faults are usually more frequent than permanent ones, especially in the space environment.

An usual mechanism to deal with transient faults is time redundancy or the re-execution of tasks in case of an error detection. Both RB and DRB methods allow re-execution of tasks or blocks, and add an extra possibility: the use of a different version of software, or variant, as the second execution block. The main difference between RB and DRB is the distributed nature of the later, allowing concurrent running of variants in two distinct nodes and a coordination between them to define what node will send the final output. Also, DRB is usually limited to two software variants, while RB has no restriction in the number of variants.

The use of RB or DRB implies the definition of standard application procedures, such as the primary and recovery variants and also procedures for saving the system state (checkpointing) before the first execution and restoring it in case of error, preparing it for running the alternate variant. Futhermore, an acceptance test (AT) must be defined and executed after each variant run. Figure 2 presents the basic algorithm used to implement the RB method with two variants only. In this method, it is possible to use a timer to limit the execution time of each variant, which may be considered as a time AT failure.

The DRB method can be seen as two RB stations running alternate blocks. One computer node is called primary and the other backup or shadow. The primary node runs the procedure A as primary block and procedure B as the recovery one; the shadow node runs procedure B as primary block and procedure A as the recovery one. Only the primary node produces an output. If the shadow node notes the ocurrence of an error in the primary node, either by an AT failure message or by a timeout, it assumes as primary and sends its output. DRB has a clear advantage over RB as it makes use of hardware redundancy besides its intrinsic time and software redundancy inherited from RB.

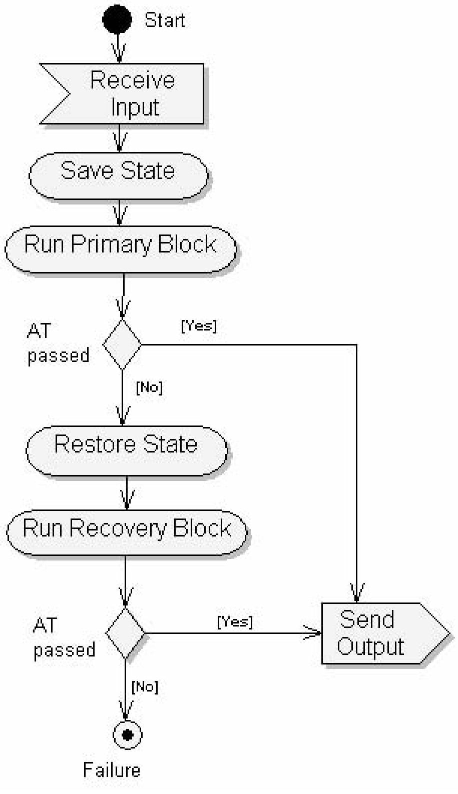


Figure 2: Recovery Blocks (RB) with two variants

4 Fault Tolerance 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 had to be created to control and schedule all fault-tolerant (FT) threads. This thread, called MiddlewareScheduler (MS), runs at the beginning of every clock tick interval and controls the behaviour and execution of each RB or DRB thread. Besides, this thread is also responsible for activating other middleware threads, as the one who reads and delivers the new external incoming messages.

In BOSS, each thread is an object of the Thread Class and has its own stack space and a control block as a data member. The run method defines the main execution code of the thread, and starts running when the thread is scheduled for the first time. A typical thread body is an infinite loop, and the thread activation is dependent either on message reception or on timing requirements. Figure 3 presents an example of thread implementation in BOSS. In this example, the Receiver thread uses an IncommingMessageAdministrator object for reading messages, just like in a mail box. If no messages are queued, the thread is suspended and becomes ready for execution again after a new message has arrived. Only messages with the “subject1” subject are delivered to that mail box by the middleware.

```
class Receiver : public Thread {

  IncommingMessageAdministrator<Msg, 20> inMsg;
  Msg* recMsg;

public:

  void run () {

    incommingMessages.listen("subject1");

    printf("Receiver will wait for 1s...\n");
    suspendFor(1*SECONDS);

    while(1) {
      printf("Receiver waiting...\n");
      recMsg = inMsg.receive();

      printf(".... Got data: %ld %ld %ld \n",
           recMsg->a,  recMsg->b, recMsg->c);
    }
  }
};
```

Figure 3: A BOSS thread example

For the execution of a RB or DRB fault-tolerant strategy, the thread object has to define the implementation of the following procedures:

- primary block;
- recovery block;
- save state;
- restore state;
- acceptance test; and
- send result.

The RB strategy is limited to two variants to be fully compatible, at the application level, with the DRB strategy. The recovery block can be a degraded version of the full functionality provided by the primary block, but it must guarantee the delivering of acceptable results for a short period of time. If only one software variant is available, it is possible to define the recovery block as equal to the primary one. In that case, only time redundancy is achieved and the primary block will be re-executed once if an error is detected.

The diagram in Figure 4 shows the interaction between the MS and RB Threads for the execution of the RB strategy. The operation is started by the application thread upon receiving an input message or after 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 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 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.

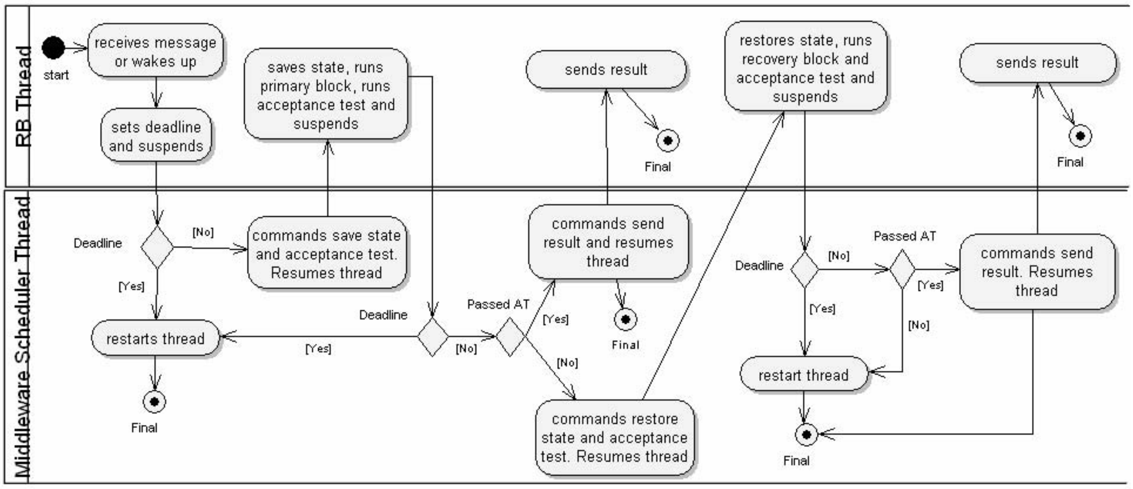

RB Thread
Middleware Scheduler Thread
start
receives message or wakes up
sets deadline and suspends
saves state, runs primary block, runs acceptance test and suspends
sends result
Final
restores state, runs recovery block and acceptance test and suspends
sends result
Final
Deadline
[No]
commands save state and acceptance test. Resumes thread
[Yes]
restarts thread
Final
[Yes]
Deadline
[No]
Passed AT
[Yes]
commands send result and resumes thread
Final
[No]
commands restore state and acceptance test. Resumes thread
Deadline
[No]
Passed AT
[Yes]
commands send result. Resumes thread
[Yes]
[No]
restart thread
Final

Figure 4: RB execution control

The execution of the DRB involves the coordination between two nodes for delivering a unique result to the system. This coordination is carried out by message exchanges between the middleware of both nodes, without any intervention from the DRB threads. The interface from the MS to DRB threads is basically the same, but the control algorithm at the MS thread is augmented by the state information messages that are exchanged with the other partner node and with the role being performed: Primary or Shadow. Figure 5 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 Primary 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 its task 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. Because of the communication delays involved, the shadow deadline should be greater than the primary one.

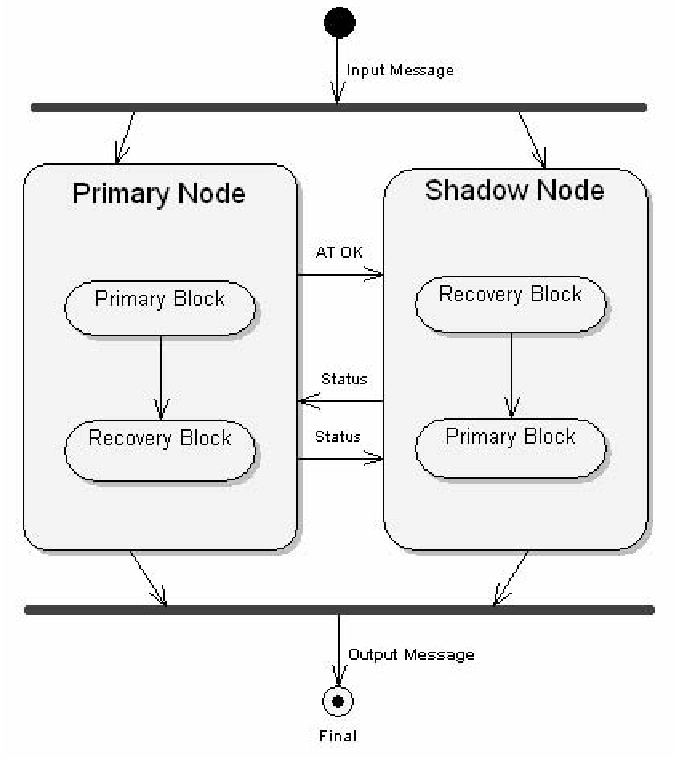


Figure 5: DRB message exchange

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 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 5 as "Status" messages.

Besides these two strategies of fault tolerance, support for the voting operation in N-Modular Redundancy (NMR) or N-Version Programming (NVP) [6] is provided. For this reason, an application voter thread, like the one presented in Figure 1, will implement the following procedures:

- store solution – to save a solution received by the voter;
- find equal solution – to test if two received solution are acceptably equal; and
- send result – to output the voted solution.

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 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.

5 Fault Tolerance Implementation

The initial programming technique used to implement the extra functionality of FT threads was, as would be natural, object-oriented. Three new classes were created, derived from the Thread class, as shown in the UML class diagram in Figure 6.

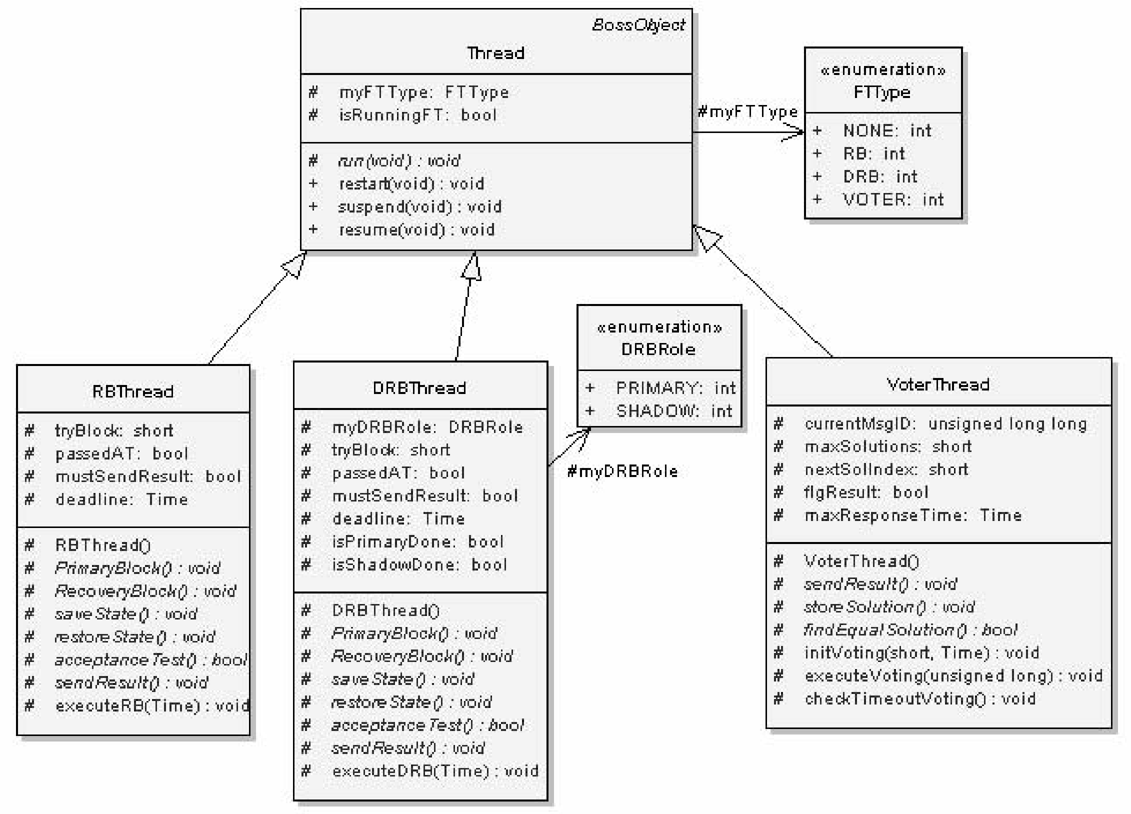
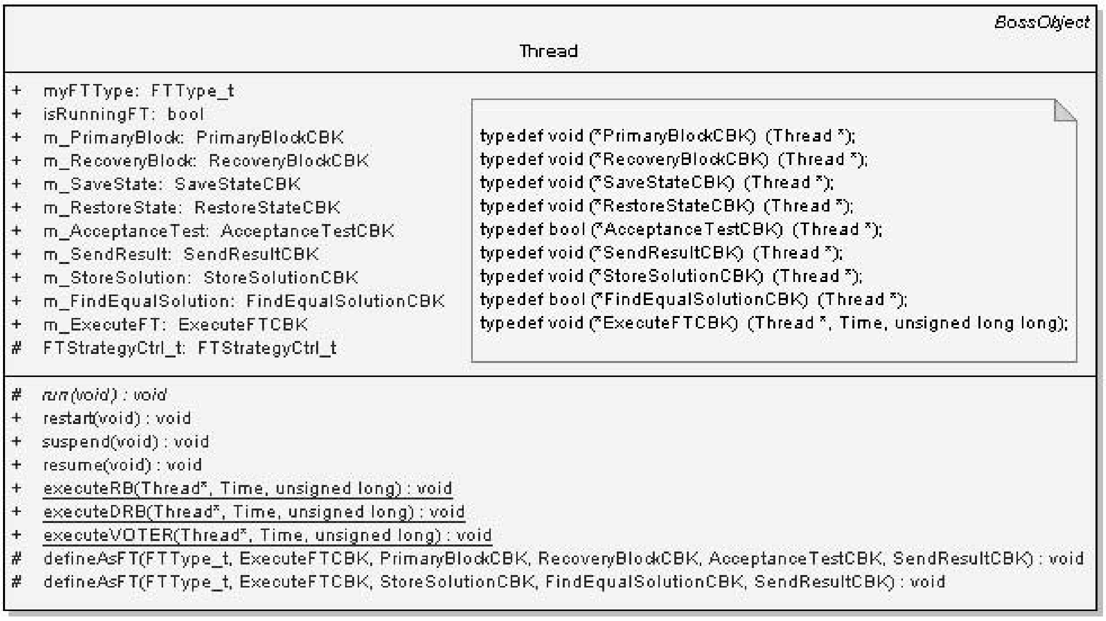


Figure 6: Object-oriented implementation

In this implementation, application threads are defined by single and direct inheritance of classes Thread (for non-FT threads), RBThread, DRBThread or VoterThread. The constructor of these classes sets the enumeration type FTType in class Thread accordingly. Figure 6 only presents FT attributes of the Thread class, and some general operations applied to the control of FT threads. Classes RBThread, DRBThread and VoterThread define application specific procedures as virtual functions and provide an empty (stub) implementation for them. For instance, the saveState function does nothing and the acceptanceTest function returns true. 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. These functions have to be called properly by the application software. In this implementation, no polymorphism was used in order to avoid increasing of the Thread class’ memory footprint.

For performance reasons, a second implementation using callback functions instead of virtual functions was developed. Figure 7 shows a UML diagram of the Thread class in this implementation.



```
+  myFTType: FTType_t
+  isRunningFT: bool
+  m_PrimaryBlock: PrimaryBlockCBK
+  m_RecoveryBlock: RecoveryBlockCBK
+  m_SaveState: SaveStateCBK
+  m_RestoreState: RestoreStateCBK
+  m_AcceptanceTest: AcceptanceTestCBK
+  m_SendResult: SendResultCBK
+  m_StoreSolution: StoreSolutionCBK
+  m_FindEqualSolution: FindEqualSolutionCBK
+  m_ExecuteFT: ExecuteFTCBK
#  FTStrategyCtrl_t: FTStrategyCtrl_t
```

```
typedef void (*PrimaryBlockCBK) (Thread *);
typedef void (*RecoveryBlockCBK) (Thread *);
typedef void (*SaveStateCBK) (Thread *);
typedef void (*RestoreStateCBK) (Thread *);
typedef bool (*AcceptanceTestCBK) (Thread *);
typedef void (*SendResultCBK) (Thread *);
typedef void (*StoreSolutionCBK) (Thread *);
typedef bool (*FindEqualSolutionCBK) (Thread *);
typedef void (*ExecuteFTCBK) (Thread *, Time, unsigned long long);
```

```
#  run(void) : void
+  restart(void) : void
+  suspend(void) : void
+  resume(void) : void
+  executeRB(Thread*, Time, unsigned long) : void
+  executeDRB(Thread*, Time, unsigned long) : void
+  executeVOTER(Thread*, Time, unsigned long) : void
#  defineAsFT(FTType_t, ExecuteFTCBK, PrimaryBlockCBK, RecoveryBlockCBK, AcceptanceTestCBK, SendResultCBK) : void
#  defineAsFT(FTType_t, ExecuteFTCBK, StoreSolutionCBK, FindEqualSolutionCBK, SendResultCBK) : void
```

Figure 7: Callback functions implementation

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 executeVOTER are provided in the Thread class as default implementations for the FT strategies. The structure called FTStragegyCtrl_t keeps the data needed for information exchange between the MiddlewareScheduler and the threads. Stub implementations are provided for application dependent functions, but these are not shown in Figure 7.

Besides its better performance, because it avoids one extra level of indirection in dispatching a virtual function, 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. The BOSS runs in Linux FIFO scheduling with the maximum priority. Two 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.

A sorting application was developed, using different sorting algorithms as variants: Bubble Sort, Insertion Sort and Selection Sort. A random array of 2000 integer elements was generated and published by a Sender thread under the subject “unsorted”. Figure 8 shows the testing configuration for Recovery Blocks. In this configuration, two Receiver threads run on separate nodes as RB threads. The primary and recovery blocks use two of the three sorting algorithms. The operation of the two RB threads is totally independent and each one publishes its results under the “sorted” subject. The Actuator thread subscribes the sorted messages and displays the first 10 elements for checking. Faulty conditions in the Receiver thread were generated by introducing a failure result in the acceptance test function at compile-time, in one or both blocks.

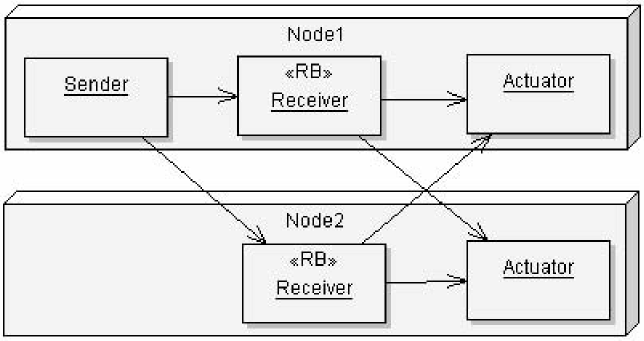


Figure 8: RB testing

Figure 9 shows the configuration and message exchanges in DRB testing. As it can be noted, only the primary Receiver thread sends its output to the Actuator threads.

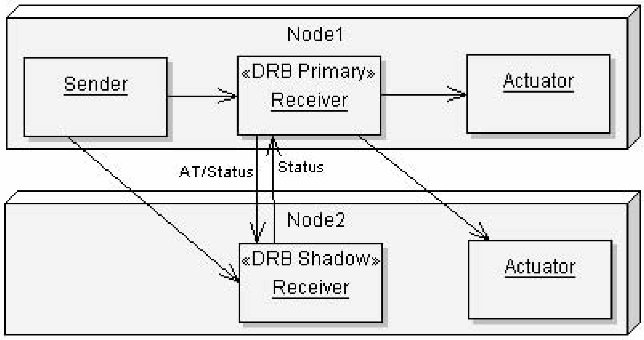


Figure 9: DRB testing

Finally, Figure 10 shows the configuration and message exchanges in the Voter thread testing running a NVP strategy with three variants. Fautly conditions were generated by introducing unsorted integer values after sorting in each variant, at compile time.

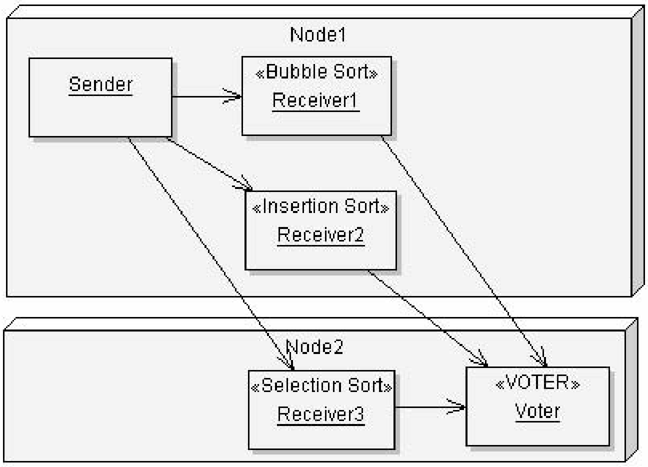


Figure 10: Voter testing

No problems were detected in these configurations. The system results were checked by logging all messages and the principal function events, including timing information. The expected behaviour was observed at all times.

7 Related Work

Several patterns and frameworks for fault tolerance design using object-oriented approaches have been proposed in the last ten years [7][8][9][10]. 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[11] as in [9] and [10]. 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[12]. 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 re-scheduling of the thread before its deadline. This requires that Rate Monotonic Scheduling and Admission Control of threads are performed by both operating systems. In FT-RT-Mach, the checkpointing of state information and its restoration, in case of error, are implemented at the application level, while DEOS implements this capability at the operating system level.

ROAFTS (Real-Time Object-Oriented Adaptive Fault Tolerant Support) is a middleware architecture developed by University of California [13]. It was designed to run over commercial operating systems as UNIX and Windows NT. The middleware 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. This middleware is applied as a component of the TMO (Time-triggered Message-triggered Object structuring scheme) model of computation [14] where the basic units of computation are time-triggered and service methods of real-time distributed objects. When one of these methods is called, the local scheduler assigns a thread to run its computation and schedules it based on its deadline. The system requires a supervisor unit to help detect failures in the working nodes and clock synchronization between nodes.

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.

8 Conclusion and Future Work

We proposed a design and implementation of fault tolerance support mechanisms for the BOSS embedded real-time operating system using modern object-oriented concepts and middleware technology. The main goal of the project was adding fault tolerance functionality with minimum complexity and resource commitment, which is a requirement for high-dependable embedded systems.

Future work will include:

- Implement a schedule policy for fault-tolerant threads, based on its deadlines.
- Establish a protocol for defining a primary voter between many voter threads. This would be useful if only one node must perform an output.
- Execute more rigorous testing using multicomputer/network hardware simulators and also using several microprocessor boards and a real-time control application.
- Develop an aspect-oriented version of the BOSS operating system, including FT support as aspects. Similar versions for logging and thread synchronization have already been developed. Initially static weaving will be used, but experiments with dynamic weaving are planned.

Acknowledgments

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. *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. In *Proceeding of Data Systems on Aerospace -DASIA*, Edinburgh, Scotland, 2005.

[3] J. Horning et al. A Program Structure for Error Detection and Recovery. In *Lecture Notes in Computer Science*, vol. 16, pages 177-187, 1974.

[4] B. Randell and J. Xu. The Evolution of the Recovery Block Concept. In *Software Fault Tolerance*, Michael R. Lyu, editor, Wiley, pages 1-21, 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. In *IEEE Transactions on Computers*, vol. 38, Nº 5, pages 626-636, 1989.

[6] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Proceeding of FTCS-8*, pages 3-9, Toulouse, France, 1978.

[7] K. Tso, et al. A Reuse Framework for Software Fault Tolerance. *In Proceedings of AIAA 10th Computers in Aerospace Conference,* San Antonio, March, 1995.

[8] 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.

[9] R. Duncan and L. Pullum. Object-Oriented Executives and Components for Fault Tolerance. In *IEEE Proceedings of Aerospace Conference*, vol. 6, pages 2849-2855, 2001.

[10] 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*, pages 207-214, 2002.

[11] E. Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

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

[13] 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,* pages 50-57, Washington, D.C., 1998.

[14] K. Kim et al. An Efficient Middleware Architecture Supporting Time-Triggered, Message-Triggered Objects and an NT-based Implementation. In *Proceeding of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 54-63, 1999.